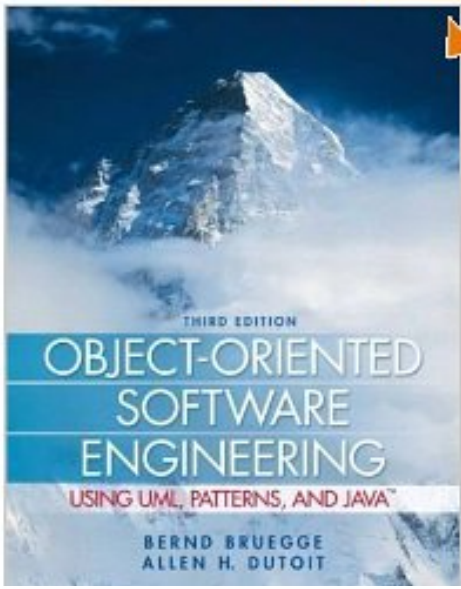
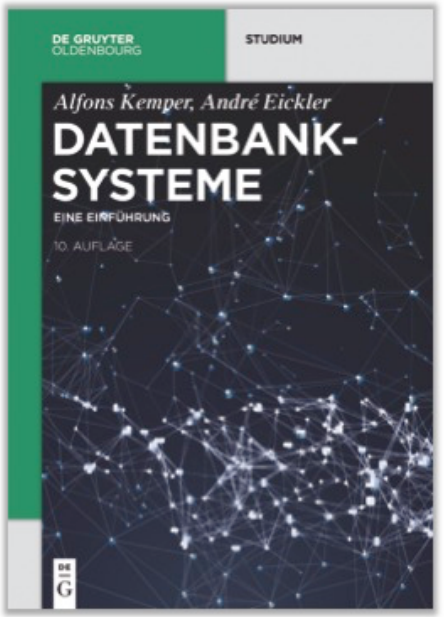
Einführung in die Informatik II für Ingenieurwissenschaften (MSE)

- 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



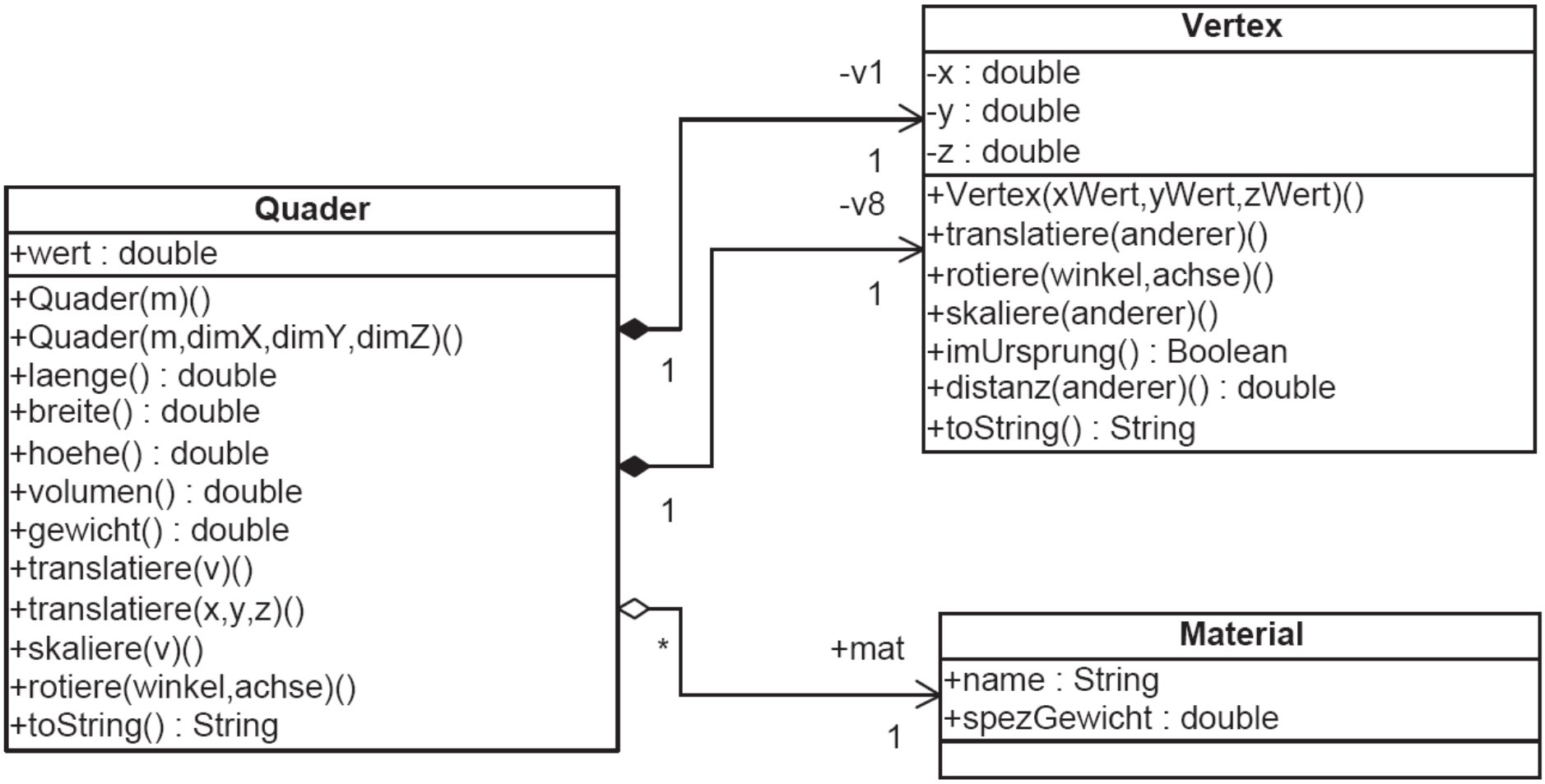
Verhalten der Objekte: Operationen

- Werden in den Klassen definiert
- Werden (i.d.R.) auf einem Objekt aufgerufen
  - Wird das Empfängerobjekt genannt
- Weitere Objekte können „mitspielen“
  - Werden als Parameter übergeben
- Ein Objekt oder ein Wert kann als Rückgabe-Parameter definiert werden
  - Oft werden Operationen aber nichts zurückgeben was als void gekennzeichnet wird

Klassifikation der Operationen

- Konstruktoren
  - Dienen der Initialisierung des Objekts
  - Oft wird in dem Zuge ein ganzes Objektnetz aufgebaut, indem untergeordnete Objekte gleich mit initialisiert werden, indem man im Konstruktor deren Konstruktoren mit aufruft
- Observer/Beobachter
  - Diese Operationen geben den internen Zustand (bzw. einen Teil davon) zurück
  - Haben also immer einen Rückgabe-Parameter
- Mutatoren
  - Ändern den internen Zustand des Objekts
  - Verursachen also Seiteneffekte
  - Haben meist keine Rückgabe: void

Verhalten von Quader- und Vertex-Objekten



...Java

```
1  public class Vertex {
2      double x;
3      double y;
4      double z;
5
6      public Vertex(double xWert, double yWert, double zWert) {
7          this.x = xWert; this.y = yWert; this.z = zWert;
8      }
9
10     public void translatiere(Vertex anderer) {  // Mutator
11         this.x = this.x + anderer.x;   // kürzer: x += anderer.x;
12         this.y = this.y + anderer.y;   //         y += anderer.y;
13         this.z = this.z + anderer.z;   //         z += anderer.z;
14     }
15
16     public void skaliere(Vertex anderer) {
17         // ...
18     }
19
20     public void rotiere(double winkel, char achse) {
21         // ...
22     }
23
24     public boolean imUrsprung() {  // liegt der Wert sehr nahe bei (0,0,0) ?
25         double epsilon = 0.000000001;  // Präzisionsgrenze
26         return ((this.x < epsilon) && (this.x > -epsilon) &&
27                 (this.y < epsilon) && (this.y > -epsilon) &&
28                 (this.z < epsilon) && (this.z > -epsilon));
29     }
30
31     public double distanz(Vertex anderer) { // Distanz zwischen this und anderer
32         double dx, dy, dz;
33         dx = this.x - anderer.x;
34         dy = this.y - anderer.y;
35         dz = this.z - anderer.z;
36         return Math.sqrt(dx * dx + dy * dy + dz * dz);
37     }    // sqrt ist in Math definiert
38
39     public String toString() {
40         return ("x:␣" + this.x + ",␣y:␣" + this.y + ",␣z:␣" + this.z);
41     }
42
43 } // public class Vertex
```

```
[public|private|protected] <ErgebnisTyp> <Name>(<ParameterTypListe>) {
   // Implementierung
}
```

Die einzelnen Bestandteile haben die folgende Bedeutung:

- Der *<Name>* identifiziert die Operation innerhalb der Klasse. Man beachte, dass jede Klasse ihren eigenen Namensraum hat, so dass die gleichnamigen Operationen von *Vertex* bzw. *Quader* sich nicht gegenseitig ins Gehege kommen. Sie sind zwar logisch verwandt, haben aber aus der Sicht des Java-Compilers nichts miteinander zu tun. Anders verhält es sich da bei gleichnamigen Operationen innerhalb derselben Klasse. Auch hierfür haben wir ein Beispiel: Den Konstruktor *Vertex* gibt es zweimal – einmal mit drei Parametern und einmal ohne Parameter. Dies bezeichnen wir als Überladung (engl. *overloading*) der Operation. Es handelt sich hierbei um zwei unterschiedliche Operationen, die vom Compiler anhand der Parameter beim Aufruf auseinander gehalten werden können. Mehr dazu erklären wir in dem Abschnitt 2.6.

- Der *<ErgebnisTyp>* legt den Typ des Rückgabe-Objekts bzw. des Rückgabewerts fest. Falls es keine Rückgabe gibt, wird **void** angegeben.

- Die *<ParameterTypListe>* gibt die Anzahl und die Typen der Parameter an.

- Die Implementierung erfolgt zum Schluss – innerhalb der geschweiften Klammern. Man beachte, dass ein guter objektorientierter Entwurf im Allgemeinen dafür sorgt, dass die einzelnen Operationen sehr einfach und knapp realisiert werden können.

- Die so genannten *access modifier* [**public**|**private**|**protected**] legen die Sichtbarkeit einer Operation fest. Eine öffentliche Operation ist von überall aufrufbar und wird mit dem Schlüsselwort **public** gekennzeichnet. Die privaten Operationen sind nur innerhalb der Klassendefinition aufrufbar – es handelt sich hierbei um Hilfsroutinen zur Implementierung der öffentlichen Operationen. Die mit **protected** angegebenen Operationen können auch in einer Unterklasse verwendet werden. Diese Sichtbarkeitsregeln werden in Abschnitt Abschnitt 2.4 nochmals diskutiert, da sie essentiell

Aufruf der Operationen

```
Vertex meinVertex = new Vertex(1.0,0.0,0.0);;
Vertex translationsVertex = new Vertex(0.0,2.0,2.0);
...
meinVertex.translatiere(translationsVertex);
```

Die Dot-Notation

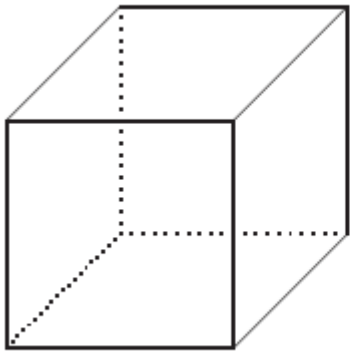
```
meinQuader.v1.translatiere(translationsVertex);
```

Pfadausdrücke (Dot-Notation) mit Operatoren „mitten drin“

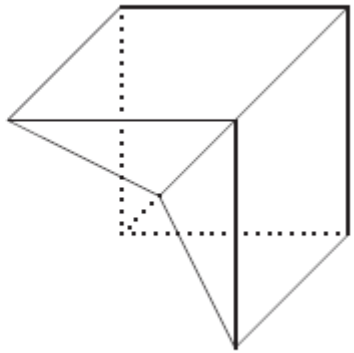
```
public class Person {
    public int alter;
    public Person ehePartner;
    public Person mutter;
    // ...
    public Person schwiegerMutter() {
       return ehePartner.mutter;
    }
    // ...
}
...
Person mickey;
...
mickey.schwiegerMutter();
mickey.schwiegerMutter().ehePartner;
```

Information Hiding: Geheimnisprinzip/Verkapselung

```
   Quader meinKomischerQuader;
   Material gold = new Material();   // neues Material
   gold.name = "Gold";
   ...
   meinKomischerQuader = new Quader(gold); // goldener Quader
   ...
(1)meinKomischerQuader.v1.x = 0.5;
(2)meinKomischerQuader.v1.y = 0.5;
(3)meinKomischerQuader.v1.z = 0.5;
```



before



after

```
1  public class Quader {
2      Vertex v1, v2, v3, v4, v5, v6, v7, v8;
3      Material mat;
4      double wert;
5
6      public Quader(Material m) {  // Konstruktor: Einheitswürfel wird kreiert
7          this.v1 = new Vertex(0.0,0.0,0.0);  this.v2 = new Vertex(1.0,0.0,0.0);
8          this.v3 = new Vertex(1.0,1.0,0.0);  this.v4 = new Vertex(0.0,1.0,0.0);
9          this.v5 = new Vertex(0.0,0.0,1.0);  this.v6 = new Vertex(1.0,0.0,1.0);
10         this.v7 = new Vertex(1.0,1.0,1.0);  this.v8 = new Vertex(0.0,1.0,1.0);
11         this.mat = m;
12         this.wert = 39.99;  // fiktiver Wert
13     }
14     public double laenge() {                // Observer-Funktion
15         return this.v1.distanz(this.v5);
16     }
17     public double breite() {
18         return this.v1.distanz(this.v2);
19     }
20     public double hoehe() {
21         return this.v1.distanz(this.v4);
22     }
23     public double volumen() {
24         return this.laenge() * this.hoehe() * this.breite();
25     }
26     public double gewicht() {
27         return this.volumen() * this.mat.spezGewicht;
28     }
29     public void translatiere(Vertex v) {    // Mutator
30         this.v1.translatiere(v);    this.v2.translatiere(v);
31         this.v3.translatiere(v);    this.v4.translatiere(v);
32         this.v5.translatiere(v);    this.v6.translatiere(v);
33         this.v7.translatiere(v);    this.v8.translatiere(v);
34     }
35     public void translatiere(double xWert, double yWert, double zWert) {
36         this.translatiere(new Vertex(xWert,yWert,zWert));
37                     // delegieren an obige translatiere Op.
38     }
39     public void skaliere(Vertex v) {
40         // ...
41     }
42     public void rotiere(double winkel, char achse) {
43         // ...
44     }
45     public String toString() { // observer
46        return("Quader␣der␣Dimension␣" + this.laenge() + "␣X␣" +
47                                         this.breite() + "␣X␣" + this.hoehe());
48     }
49
50 } // public class Quader
```

Access Modifier beschränken den Zugriff

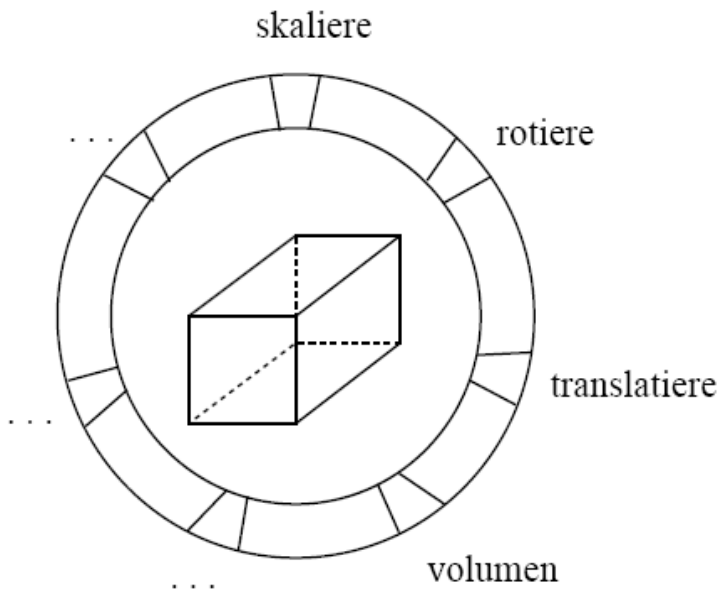
- **public:** Öffentliche Komponenten sind für alle Klienten zugreifbar und werden als *public* angegeben.
- **private:** Die verborgenen Komponenten werden als *private* spezifiziert und sind nur für den Code innerhalb der Klasse sichtbar.
- **protected:** Die mit *protected* angegebenen Komponenten sind wie bei *private* nicht allgemein sichtbar. Im Unterschied zu *private* können Unterklassen aber auf die als *protected* gekennzeichneten Komponenten ihrer Oberklassen zugreifen.
- **package:** Der Default – wenn also nichts angegeben ist – ist der access modifier *package*. Code innerhalb desselben Packages hat Zugriff auf die Komponenten.

Quader-Definition

```
public class Quader {
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;

    public Quader(Material m) {  // Einheitsquader kreieren
        // ...
    }
    public double laenge() {
        // ...
    }
    public double breite() {
        // ...
    }
    public double hoehe() {
        // ...
    }
```

Quader – cont‘d




```
    public double volumen() {
        // ...
    }
    public double gewicht() {
        // ...
    }
    public void translatiere(Vertex v) {
        // ...
    }
    public void translatiere(double xWert, double yWert, double zWert){
        // ...
    }
    public void skaliere(Vertex v) {
        // ...
    }
    public void rotiere(double winkel, char achse) {
        // ...
    }
    public String toString() {
        // ...
    }
} // public class Quader
```

Best Practice: Verbergen von Instanzvariablen

```
class Quader {
    private double wert;
    ...
    public double wert() {
       return this.wert;
    }
}

double wieViel;
Quader meinQuader;
...
wieViel = meinQuader.wert();
```

Später kann die Berechnung geändert werden

Initialisierung eines Objekts

```
public class Quader {
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;

    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.wert = 39.99;  // fiktiver Wert
    }
    // ...
} // public class Quader
```

Initialisierung eines Vertex‘es

```
class Vertex {
    // ...
    public Vertex() {
       this.x = 0.0;  // Initialisierung könnte auch an den
       this.y = 0.0;  // anderen Konstruktor delegiert werden:

       this.z = 0.0;  // this(0.0,0.0,0.0);
    }
    // ...
}
```

```
Quader meinQuader;
Material gold;
gold = new Material();
   ...
meinQuader = new Quader(gold);
   ...
```

Overloading: Mehrere Operationen gleichen Namens

```
public class Quader {
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;
    public Material mat;
    public double wert;

    public Quader(Material m) {  //  Konstruktor: Einheitswürfel
        ... // Implementierung wie vorher


    }

    public Quader(Material m, double dimX, double dimY, double dimZ) {
        this.v1 = new Vertex(0.0,0.0,0.0);
        this.v2 = new Vertex(dimX,0.0,0.0);
        this.v3 = new Vertex(dimX,dimY,0.0);
        this.v4 = new Vertex(0.0,dimY,0.0);
        this.v5 = new Vertex(0.0,0.0,dimZ);
        this.v6 = new Vertex(dimX,0.0,dimZ);
        this.v7 = new Vertex(dimX,dimY,dimZ);
        this.v8 = new Vertex(0.0,dimY,dimZ);
        this.mat = m;
        this.wert = 39.99;  // fiktiver Wert
    }
    // ...
} // end class Quader
```

Aufruf unterscheidet sich entweder in Anzahl oder Typ der Parameter

Die Nutzung wird dann in folgendem Programmfragment illustriert:

```
   Quader meinQuader;
   Quader deinQuader;
   Material gold;
      ...
(1)meinQuader = new Quader(gold);                  // goldener Einheitswürfel
(2)deinQuader = new Quader(gold, 2.0, 3.0, 5.0); // Goldbarren Quader
   ...                                             // der Dimension 2 × 3 × 5
```

Translatiere unterschiedlich aufgerufen ...

```
public class Quader {
    // ...
    public void translatiere(Vertex v) {
        this.v1.translatiere(v);    this.v2.translatiere(v);
        this.v3.translatiere(v);    this.v4.translatiere(v);
        this.v5.translatiere(v);    this.v6.translatiere(v);
        this.v7.translatiere(v);    this.v8.translatiere(v);
    }
    public void translatiere(double x, double y, double z) {
        // delegiere die Arbeit an die andere translatiere-Operation
        this.translatiere(new Vertex(x,y,z));
    // ...
} // public class Quader
```

```
Quader meinQuader = new Quader(...);;
Vertex translatiereVertex = new Vertex(0.0,3.0,2.0)
   ...
meinQuader.translatiere(translatiereVertex);
meinQuader.translatiere(30.0, 2.0, 1.75);
   ...
```

Statische Operationen

```
Vertex p1, p2;
double d;
    ...
d = p1.distanz(p2);
    ...
```

Ungewohnt?

```
    ...
 d = distanz(p1, p2);
    ...
```

besser?

Realisierung ...

```
public class Vertex {
    ...
    public static double distanz(Vertex erster, Vertex zweiter) {
        return erster.distance(zweiter);
    }
}
```

Um dann die *distanz* zwischen zwei Punkten zu berechnen, ruft man die statische Methode wie folgt auf:

```
Vertex a = new Vertex(0.0,0.0,0.0);
Vertex b = new Vertex(3.0,0.0,0.0);
System.out.println(Vertex.distanz(a,b));
```

main() ... als statische Operation zum Testen

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

    public static void main(String args[]) {
       Material eisen = new Material("Eisen", 0.89);
       Quader meinQuader = new Quader(eisen);
       Quader andererQuader = new Quader(eisen);
       andererQuader.translatiere(new Vertex(3.5,2.5,4.5));
       andererQuader.translatiere(2.0,3.0,4.0);
       System.out.println(meinQuader);   // impliziter Aufruf von toString
       System.out.println("meinQuader hat das Gewicht: "
                          + meinQuader.gewicht());
       System.out.println(andererQuader);
       System.out.println("andererQuader hat das Gewicht: "
                          + andererQuader.gewicht());
       Material carbon = new Material("Carbon", 0.75);
       meinQuader.mat = carbon;
       System.out.println("meinQuader hat jetzt das Gewicht: "
                          + meinQuader.gewicht());
    }
```

Nutzung von main

```
javac Quader.java
java Quader
```

Die Ausgabe ist dann wie folgt:

```
Quader der Dimension 1.0 X 1.0 X 1.0
meinQuader hat das Gewicht: 0.89
Quader der Dimension 1.0 X 1.0 X 1.0
andererQuader hat das Gewicht: 0.89
meinQuader hat jetzt das Gewicht: 0.75
```

Parameter-Übergabe

Nun muss man zwei verschiedene Arten der Parameterübergabe unterscheiden: *call by value* und *call by reference*. *Call by value* kommt zum Einsatz, wenn primitive Typen als Argument verwendet werden. In diesem Fall werden die Argumente kopiert. Bei Objekttypen bezeichnet man die Semantik der Parameterübergabe als *call by reference*, da in diesem Fall nur eine Referenz auf das Objekt übergeben wird und keine Kopie des Objekts. Ändert nun die Operation Eigenschaften beim übergebenen Objekt wirkt sich dies auch außerhalb der Operation auf das Objekt aus – es wurde ja gerade keine automatische Kopie erstellt. Möchte man ein übergebenes Objekt nur in der Operation verändern, muss man es selbst kopieren.

Ausnahmen ... abfangen

- Try ... catch

```
public class TesteAusnahme {
    public static void main(String[] args) {
        Material eisen = new Material("Eisen", 0.89);
        Quader meinQuader;
        try {
            meinQuader = new Quader(eisen,0.0,3.0,9.0);
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```