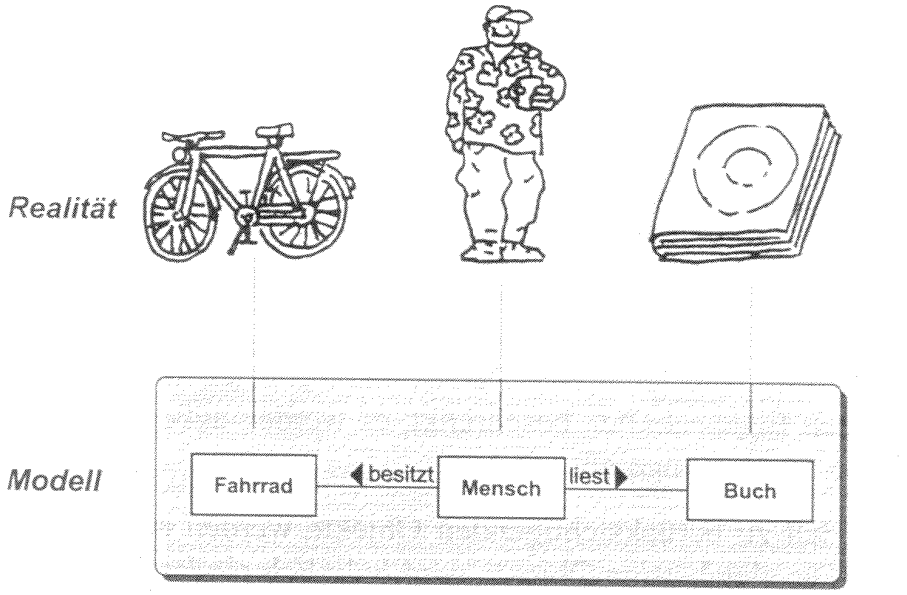
Grundkonzepte der Objektorientierten Programmierung

Die Objektorientierung sieht die in der realen Welt vorkommenden Gegenstände als Objekte an.



Prinzip:

Komplexität durch abstrakte Modelle vereinfachen

Beispiele für Objekte:

Fahrrad, Mensch, Buch, Telefon,
Versicherungspolice, Überweisung

Grundkonzepte der Objektorientierten Programmierung

- **Objekt**
  Gegenstand der Erkenntnis und Wahrnehmung, des Denkens und Handelns.
- **Klasse**
  Bezeichnung für eine Menge von Objekten, die durch bestimmte Eigenschaften ausgezeichnet sind.
- **Attribut**
  Eigenschaft, Kennzeichen.
- **Operation**
  Durchführung einer bestehenden Vorschrift.
- **Vererbung**
  Strukturierung von Eigenschaften, Wiederverwendung von Eigenschaften.
- **Botschaft**
  Die Kommunikation von Objekten untereinander geschieht durch den Austausch von Botschaften.
- **Polymorphismus**
  „Vielgestaltigkeit“
  Eine Operation kann sich (in unterschiedlichen Klassen) unterschiedlich verhalten.

weitere Konzepte der Objektorientierten Programmierung

- **Zusicherungen**
  Bedingungen, Voraussetzungen und Regeln, die ein Objekt erfüllen muss.

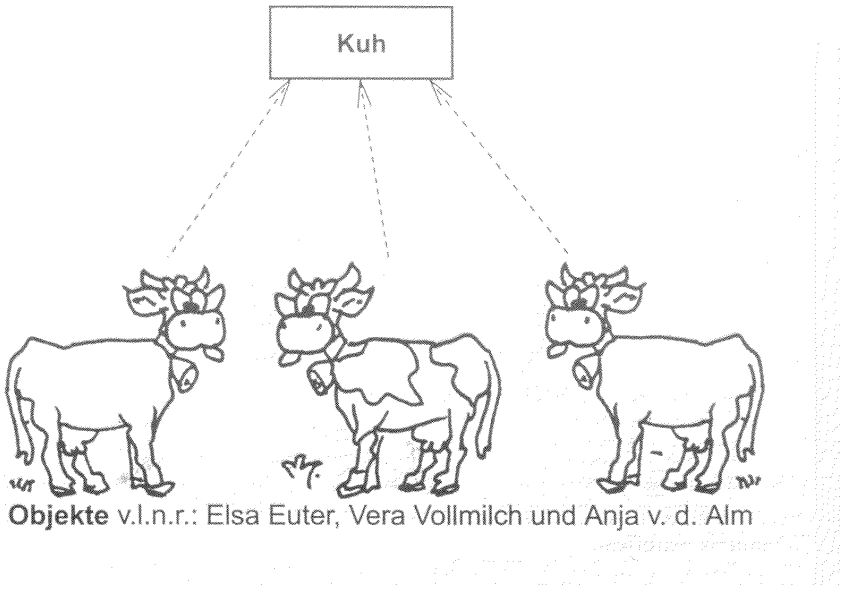
- **Assoziation**
  Eine Beziehung zwischen verschiedenen Objekten einer oder mehrerer Klassen.

- **Aggregation**
  Eine gerichtete Assoziation zwischen den Objekten verschiedener Klassen.

- **Komposition**
  Eine Aggregation mit existenzabhängigen Teilen.

Klassen und Objekte

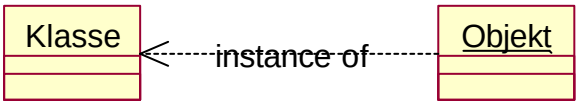
Klasse = Bauplan gleichartiger Objekte



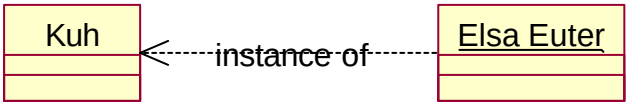
Objekte = Exemplare einer Klasse

Notation für Klassen und Objekte

Darstellung der Objekt-Klassen-Beziehung



„Elsa Euter ist ein Exemplar der Klasse Kuh.“



Attribute und Operationen

Eigenschaften von Klassen und Objekten

- **Attribute**
  die Struktur der Objekte: ihre Bestandteile und die in ihnen enthaltenen Informationen bzw. Daten.

- **Operationen**
  das Verhalten der Objekte.

- **Zusicherungen**
  die Bedingungen, Voraussetzungen und Regeln, die die Objekte erfüllen müssen.

Beispiel:

Eigenschaften eines **Kreis**es, den wir auf dem Bildschirm darstellen wollen:

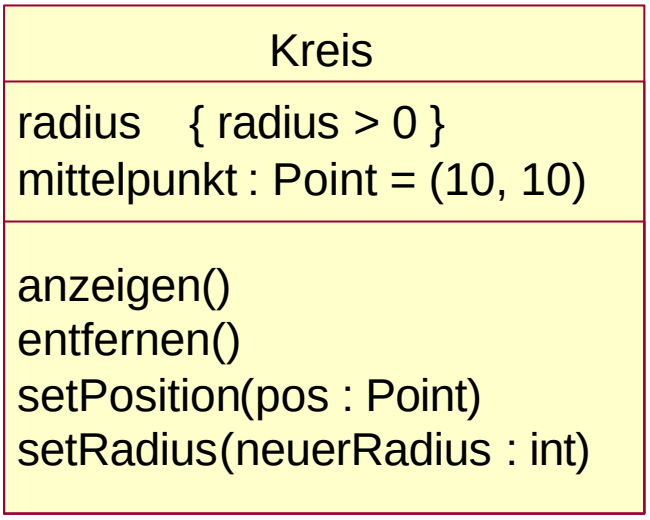
- **Attribute**
  sein Radius und seine Position auf dem Bildschirm.

- **Operationen**
  die Möglichkeit, ihn anzuzeigen, ihn zu entfernen, zu verschieben und den Radius zu verändern.

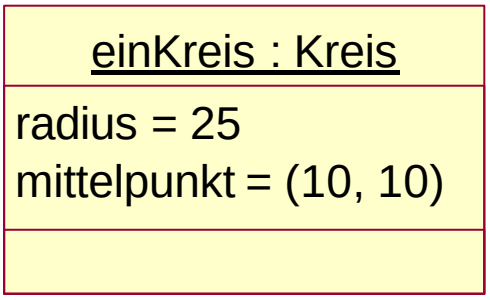
- **Zusicherungen**
  der Radius darf nicht negativ und nicht Null sein.

Attribute und Operationen

Klasse



Objekt



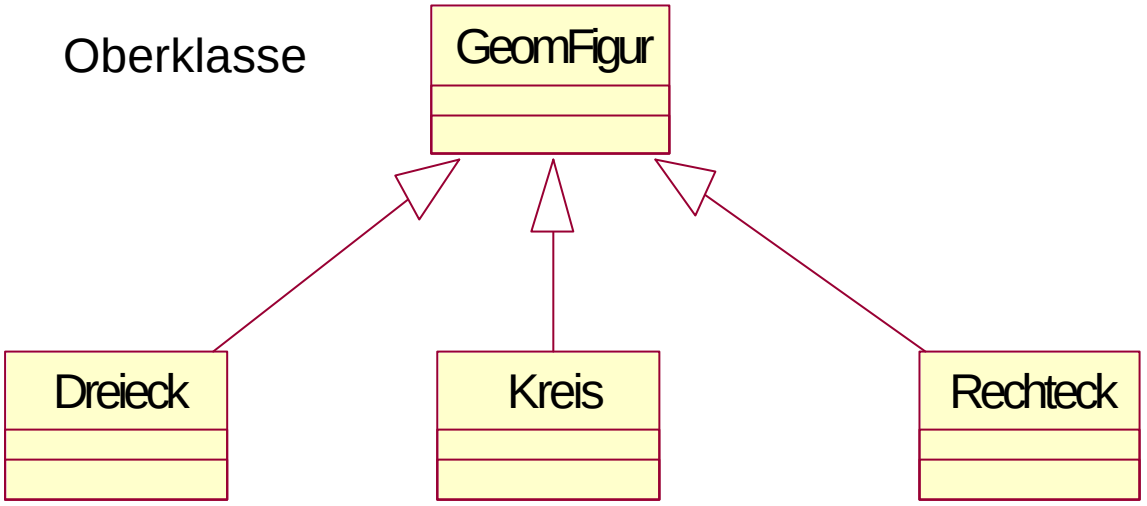
Attribute und Operationen

Java-Code für die Klasse `Kreis` (vereinfacht)

```
class Kreis
{
  int   radius;
  Point mittelpunkt;

  public void setRadius(int neuerRadius)
  {
    if(neuerRadius > 0)      // Zusicherung
    {
      radius = neuerRadius;
      ...
    }
  }

  public void setPosition(Point pos)
  { ... }

  public void anzeigen()
  { ... }

  public void entfernen()
  { ... }
}
```

Vererbung

Vererbung = Eigenschaften wiederverwenden



Vererbungshierarchie:

Oberklasse = Klasse, die Eigenschaften weitergibt

Unterklasse = Klasse, die etwas erbt

Prinzip der **Generalisierung** bzw. **Spezialisierung**

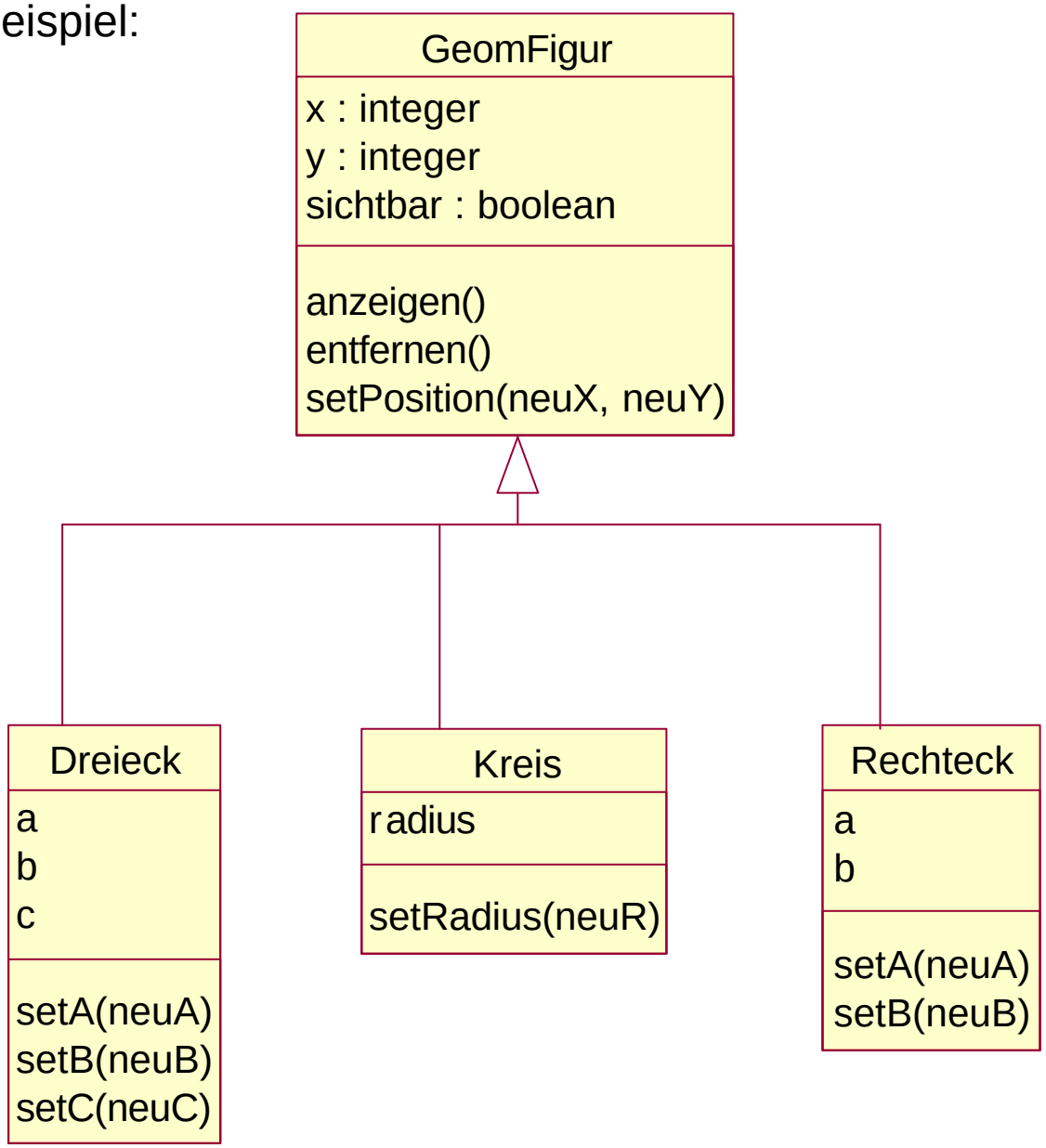
Ist-Ein-Beziehung: ein Kreis **ist eine** geometrische Figur.

Vererbung

Die Unterklasse erbt die Eigenschaften der Oberklasse.

Einzelheiten dürfen spezialisiert und neue Eigenschaften dürfen hinzugefügt werden.

Beispiel:



Eigenschaften hierarchisch strukturieren

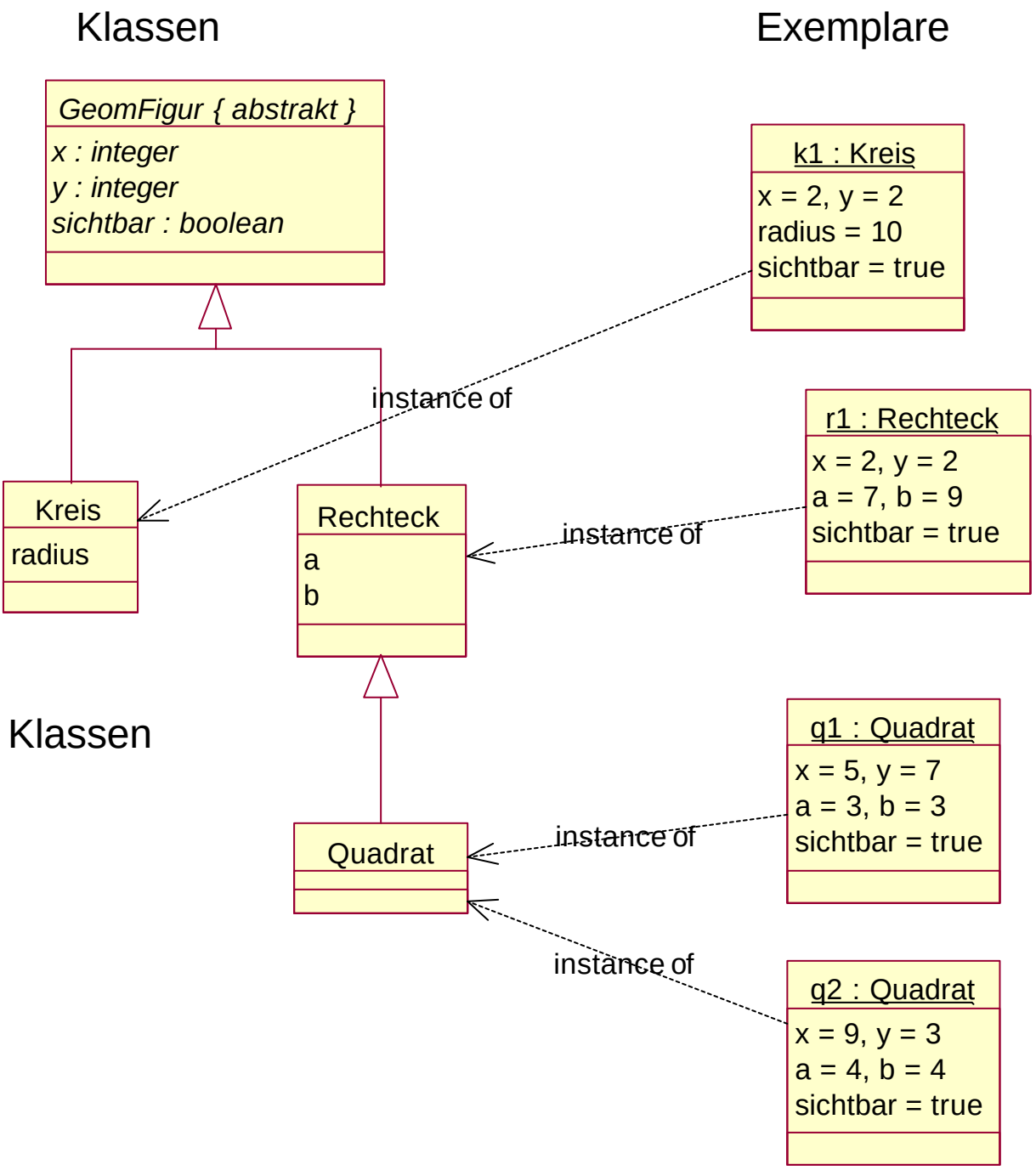
Vererbung

Java-Code für die Klassen `GeomFigur` und `Rechteck`

```
class GeomFigur
{
  int x, y;
  boolean sichtbar;

  public void anzeigen() { ... }

  public void entfernen() { ... }

  public void setPosition(int neuX, int neuY)
  {
    if(sichtbar) {
      entfernen();
      x = neuX;
      y = neuY;
      anzeigen();
    } else {
      x = neuX;
      y = neuY;
    }
  }
}

class Rechteck extends GeomFigur
{
  int a, b;    // Kanten

  public void setA(int neuA)
  { if(neuA > 0) a = neuA; }

  public void setB(int neuB)
  { if(neuB > 0) b = neuB; }
}
```

Abstrakte Klassen

Klassen, von denen keine konkreten Exemplare erzeugt werden können, d.h. von denen es grundsätzlich keine Objekte geben wird, bezeichnet man als **abstrakte Klassen**.



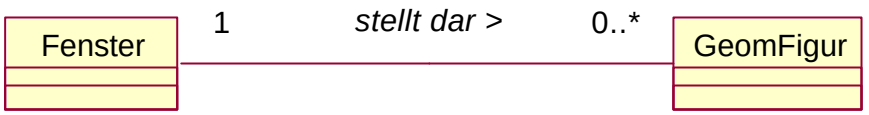
Abstrakte Klassen

Java-Code für die Klassen `GeomFigur` und `Rechteck`

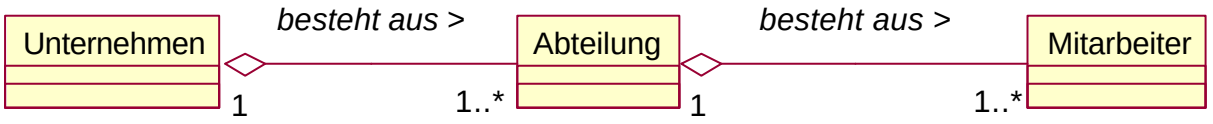
```
abstract class GeomFigur
{
  int x, y;
  boolean sichtbar;

  public abstract void anzeigen();
  public abstract void entfernen();

  public void setPosition(int neuX, int neuY)
  {
    // wie oben
  }
}

class Rechteck extends GeomFigur
{
  int a, b;    // Kanten

  public void anzeigen() { ... }

  public void entfernen() { ... }

  public void setA(int neuA)
  { if(neuA > 0) a = neuA; }

  public void setB(int neuB)
  { if(neuB > 0) b = neuB; }
}
```

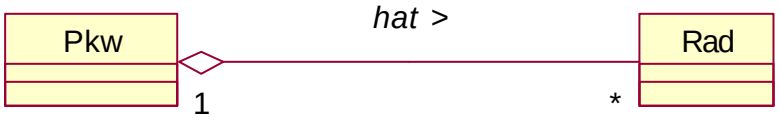
Assoziationen und Aggregationen

Eine **Assoziation** ist eine Beziehung zwischen verschiedenen Objekten einer oder mehrerer Klassen.



Eine **Aggregation** ist eine gerichtete Assoziation zwischen den Objekten verschiedener Klassen.





Hat-Beziehung:

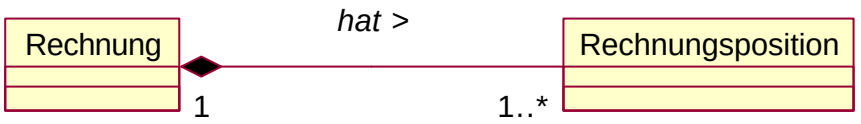
ein Unternehmen **hat** Abteilungen, ein Auto **hat** Räder

Aggregation und Komposition

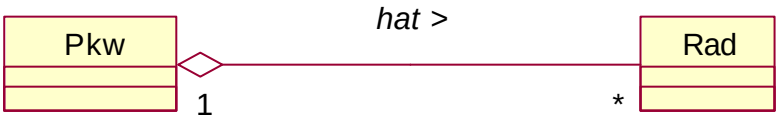
Eine **Komposition** ist eine Aggregation mit existenzabhängigen Teilen.

Wenn das Ganze (z.B. eine Rechnung) gelöscht werden soll, so werden auch alle existenzabhängigen Teile (z.B. ihre Rechnungspositionen) mitgelöscht.

Komposition

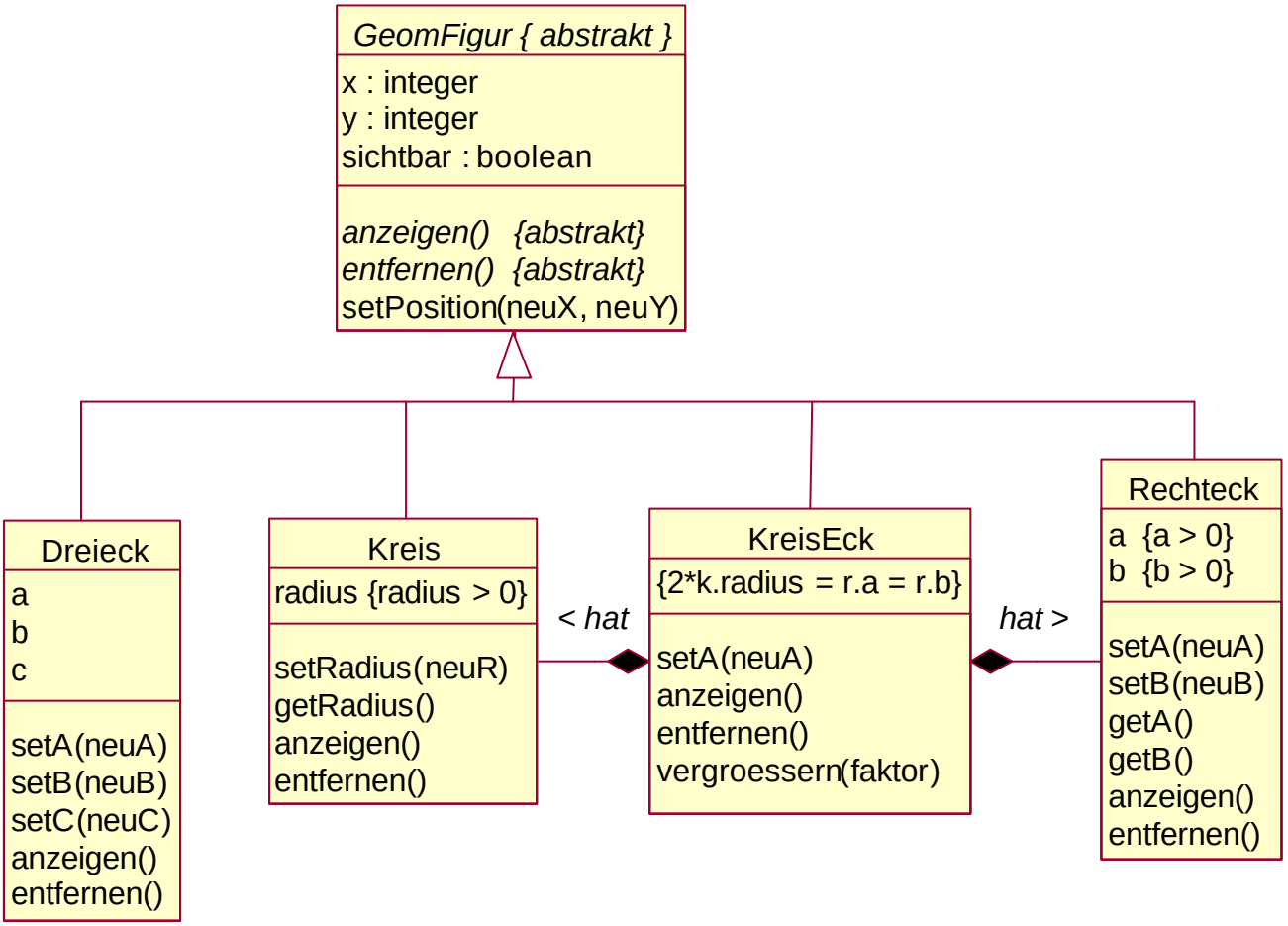


Aggregation



Aggregation und Komposition

Beispiel:



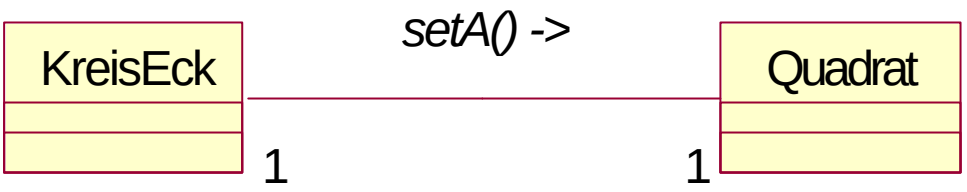
Aggregation und Komposition

Java-Code für die Klasse `Kreiseck`

```
class KreisEck extends GeomFigur
{
  Kreis    k;
  Rechteck r;

  public void anzeigen()
  {
    k.anzeigen();
    r.anzeigen();
  }

  public void entfernen()
  {
    k.entfernen();
    r.entfernen();
  }

  public void setA(int neuA)
  {
    k.setRadius(neuA / 2);
    r.setA(neuA);
    r.setB(neuA);
  }

  public void vergroessern(float faktor)
  {
    setA(r.getA() * faktor);
  }
}
```

Nachrichtenaustausch

Die Kommunikation der Objekte untereinander geschieht durch **Nachrichten** (auch **Botschaften** genannt).



Eine Nachricht kann von einem Objekt nur dann interpretiert werden, wenn es eine dazu passende Operation besitzt.

Beispiel:

```
class KreisEck extends GeomFigur
{
  Kreis    k;
  Rechteck r;

  ...

  public void setA(int neuA)
  {
    k.setRadius(neuA / 2);
    r.setA(neuA);
    r.setB(neuA);
  }

  public void vergroessern(float faktor)
  {
    setA(r.getA() * faktor);
  }
}
```

Polymorphismus

Eine Operation kann sich in unterschiedlichen Klassen unterschiedlich verhalten.

Statischer Polymorphismus

Operatoren wie `+`, `-`, `*`, `/` sind sowohl auf Festpunkt- als auch auf Gleitpunktzahlen anwendbar.

Bei der Übersetzung wird entschieden, welche Operation angewandt wird.

Ein weiterer Aspekt ist die Überladung gleichnamiger Operationen.

Beispiel:

```
class Uhrzeit {
  public
    void setzeZeit(char zeit[8]) {...}
  public
    void setzeZeit(int h, int m, int s) {...}
}
...

Uhrzeit einWecker = new Uhrzeit();

einWecker.setzeZeit(17, 1, 0);
einWecker.setzeZeit("11:55:00");
```

Polymorphismus

Dynamischer Polymorphismus

Voraussetzung ist die **späte** oder **dynamische Bindung**.

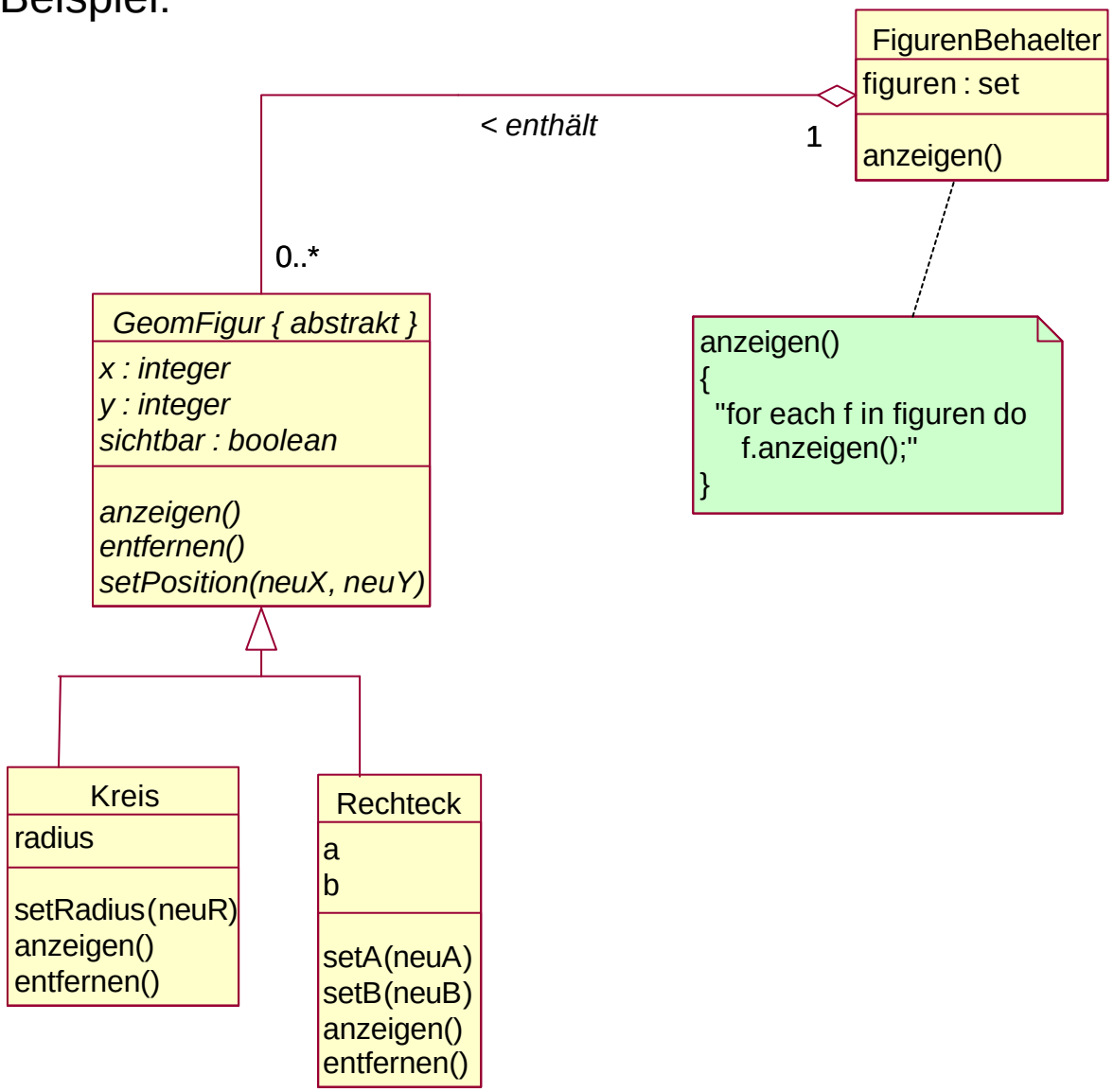
Bindung: Der Zeitpunkt im Leben eines Programms, an dem der Aufrufer einer Operation die Speicheradresse dieser Operation erhält.

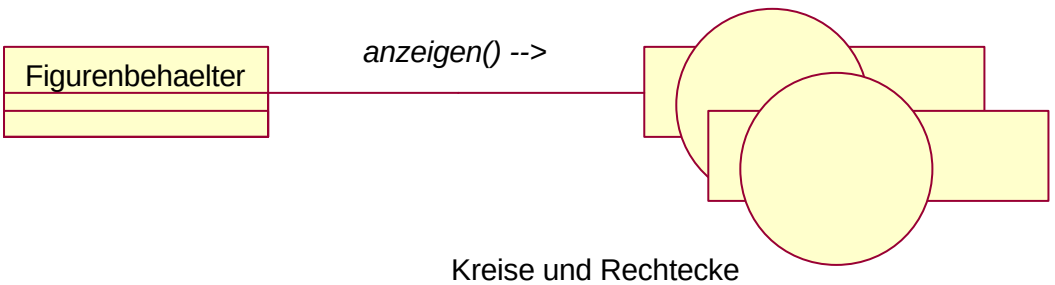
Üblicherweise geschieht dies **zur Übersetzungszeit**. Man spricht dann von **früher** oder **statischer Bindung**.

Bei der **dynamischen Bindung** wird die Adresse der Operation erst ermittelt, wenn der Aufruf stattfindet, d.h. **zur Laufzeit** des Programms.

Polymorphismus

Beispiel:





Kreise und Rechtecke