3. Konzepte der objektorientierten Programmierung

3.1 Basiskonzepte

3.2 Generalisierung / Spezialisierung

3.3 Aggregation

3.4 Assoziation

3.5 Nachrichten

3.6 Polymorphismus

3.1 Basiskonzepte

Objekt

Ein Objekt repräsentiert ein Ding, eine Person, einen Vorgang, einen Zusammenhang, ein Ereignis o. ä. aus der Domäne der Anwendung. Ein Objekt

- hat einen Zustand
- weist ein wohldefiniertes Verhalten auf
- hat eine eindeutige Identität.

Ein Objekt kapselt seine Daten als **Attribute** und bietet Interaktionen als **Dienste** an.

Ein Objekt gehört genau einer Klasse an, es kann nicht während seiner Lebenszeit seine Klassenzugehörigkeit ändern.

Klasse

Eine **Klasse** spezifiziert die gemeinsamen Eigenschaften und das gemeinsame Verhalten einer Sammlung von Objekten.

Eine Klasse beschreibt auch, wie die Erzeugung neuer Objekte dieser Klasse vonstatten geht. Sie definiert dazu einen **Konstruktor**.

Zu einer Klasse kann es eine variable Anzahl von Objekten geben (auch null).

Eine Klassenbeschreibung umfasst

- das **Interface** (die öffentliche Schnittstelle)
- die Implementierungen (der **Methoden** und **Klasseneigenschaften**).

Zum Interface gehören insbesondere alle Methoden, die öffentlich aufgerufen werden können.

Attribut

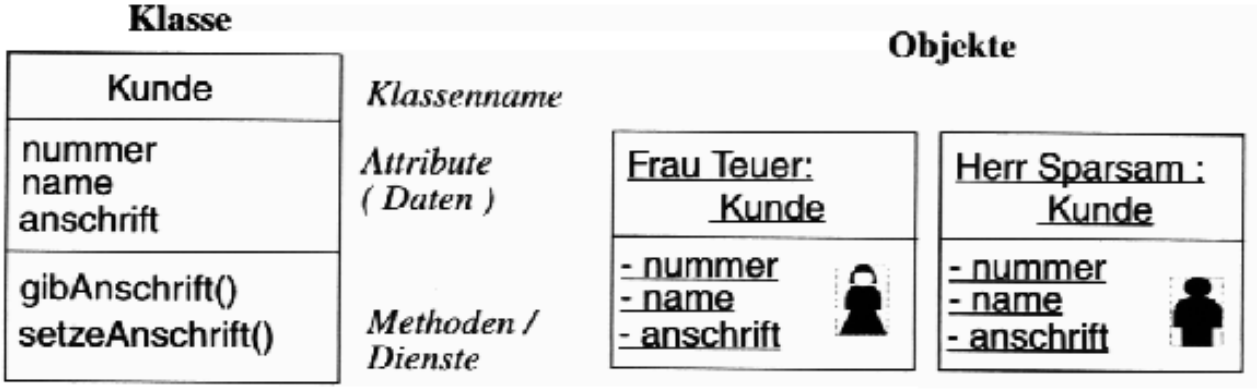
Attribute bezeichnen Eigenschaften von Objekten, die in Form von Daten gespeichert werden. Die Daten sollen in der Regel nicht öffentlich, sondern in der Klasse gekapselt sein.

In Java sind für Attribute auch die Bezeichnungen *Membervariable* oder *Instanzvariable* gebräuchlich.

Methode

Eine **Methode** bezeichnet eine Operation, die in einem Objekt ausgeführt wird. Die Begriffe *Methode*, *Operation* und *Dienst* sind synonym.

Beispiel für eine Klasse und ihre Objekte:



Beispiel: Klasse Kunde in Java

```
public class Kunde {

    // Konstruktor
    public Kunde() { ... }

    // Interface
    public String gibAnschrift() { ... }
    public void setzeAnschrift(String n, String a) { ... }

    // Instanzvariablen
    private int nummer;
    private String name;
    private String anschrift;
}
```

Abstrakte Klasse und Methoden

Eine **abstrakte Methode** bezeichnet eine Methode, für die nur die Aufrufschnittstelle fest gelegt wird, ohne ihre Implementierung zu definieren.

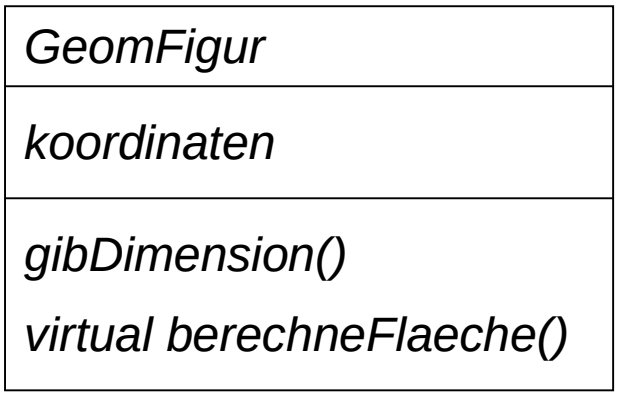
Abstrakte Klassen legen nur ein gemeinsames Konzept fest. Es ist dann die Aufgabe der von der abstrakten Klasse abgeleiteten Subklassen Ausprägungen des Konzepts zu spezifieren.

Von abstrakten Klassen können keine Objekte erzeugt werden.

Java bietet mit der **Interfacedeklaration** einen Spezialfall einer abstrakten Klasse (keine Instanzvariablen, keine Methodenimplementierungen).

Beispiel: Abstrakte Klasse und Methoden

Definition eines gemeinsamen Konzepts für zweidimensionale geometrische Figuren:



[**UML**: Abstrakte Klassen werden in kursiver Schrift dargestellt]

Umsetzung in Java:

```
public abstract class GeomFigur {
    ...
    public final int gibDimension()  { return 2; }
    public abstract double berechneFlaeche();

    protected ArrayList koordinaten;
}
```

Virtuelle Methoden

Eine **virtuelle Methode** eröffnet die Möglichkeit zum dynamischen Binden (auch spätes Binden, „late binding“). Diese ist insbesondere im Zusammenhang mit der Vererbung interessant. Wir werden später noch darauf zurück kommen.

Beim **frühen Binden** wird die Referenz zwischen Methodenaufruf und Methodendefinition bereits zur Übersetzungszeit aufgelöst. Beim **späten Binden** geschieht dies erst zur Laufzeit; erst dann muss bekannt sein, welches Objekt die aufgerufene Methode definiert.

Klassenvariable und Klassenfunktionen

Nicht nur Objekt-Instanzen, sondern auch Klassen selbst können Methoden und Variable haben. Sie sind auch dann schon definiert, wenn es noch gar keine Objektinstanz der Klasse gibt. Man spricht von Klassenvariablen und Klassenfunktionen.

Ein wichtiges Beispiel für eine Klassenfunktion ist ein **Konstruktor**. Er wird beim Erzeugen einer neuen Objektinstanz der Klasse aufgerufen.

Klassenvariable sind den globalen Variablen in herkömmlichen Programmiersprachen ähnlich.

Klassenfunktionen können nicht auf die Attribute einer Objektinstanz zugreifen, aber sie können auf Klassenvariable zugreifen. Ein Beispiel für ein Klassenvariable ist ein Zähler, der die Anzahl der gerade existierenden Objekte der Klasse angibt.

3.2 Generalisierung / Spezialisierung

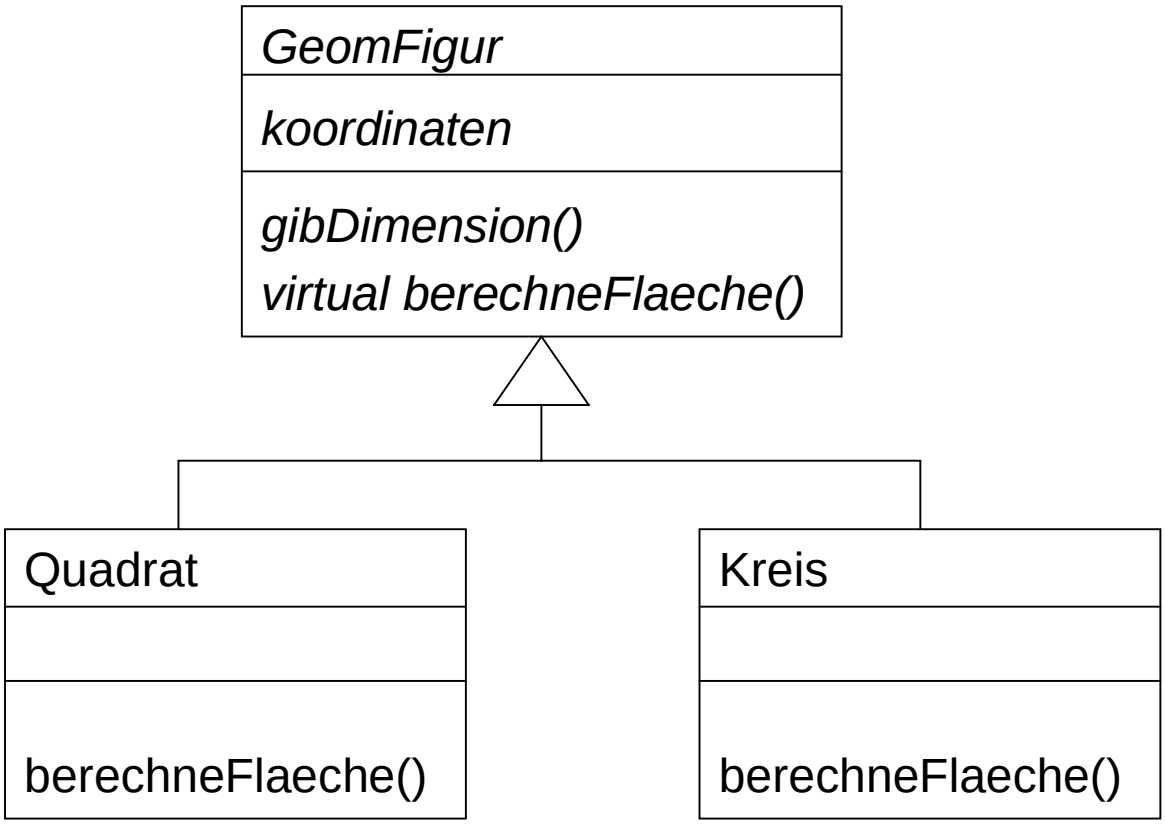
Prozeß der Klassenbildung ist ein Abstraktionsvorgang:

1. Spezialisierung: Aus bestehenden Klassen können konkretere **Unterklassen (Subklassen)** gebildet werden.

2. Generalisierung: Eigenschaften bestehender Klassen können in einer **Oberklasse (Superklasse)** zusammengefaßt werden.

Dieses Konzept wird auch unter dem Begriff **Vererbung** zusammengefasst.
Generell sollte Vererbung nur da verwendet werden, wo semantisch wirklich eine Spezialisierung vorliegt („ist ein“, „ist Spezialfall von“, „ist Erweiterung von“).

Vererbung (allgemein)



[**UML**: Vererbung wird durch ein Dreieck gekennzeichnet.]

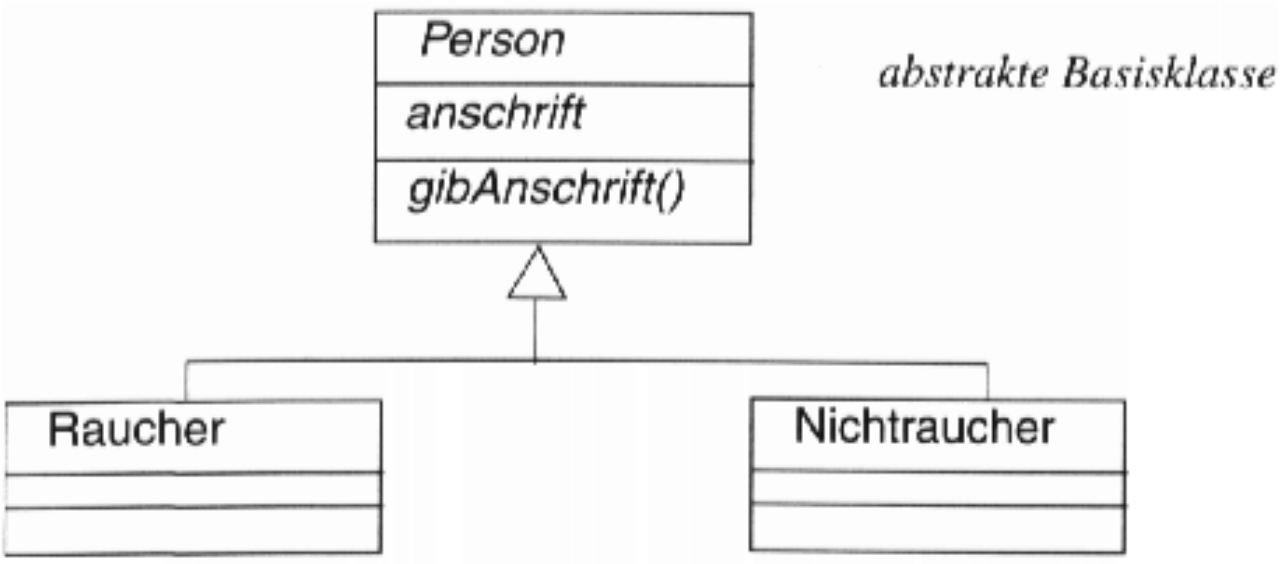
Die Klassen Quadrat und Kreis „erben“ das Attribut *koordinaten* sowie die Funktion *gibDimension().*

Sie redefinieren die virtuelle Funktion *berechneFlaeche().*

Vererbung (abstrakte Basisklasse)

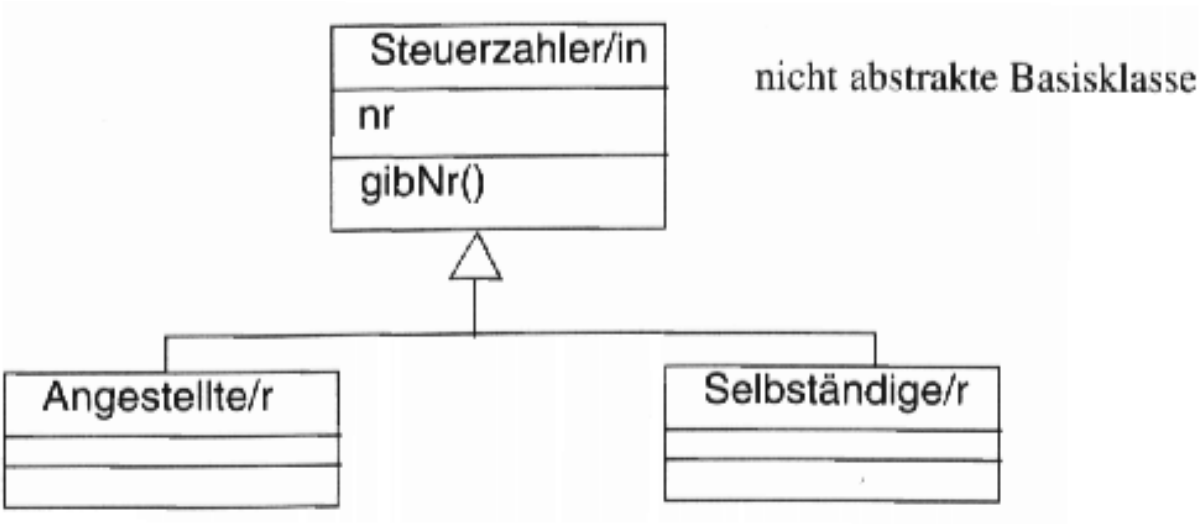
Eine Obermenge von Objekten (Personen) ist vollständig durch ihre spezialisierten Untermengen (Raucher, Nichtraucher) abgedeckt.

Es gibt also keine Objekte der Obermenge selbst, sondern nur Objekte in den Untermengen.



Vererbung (reguläre Basisklasse)

Eine Obermenge von Objekten (Steuerzahler) ist nicht vollständig durch ihre spezialisierten Untermengen (Angestellte, Selbständige) abgedeckt.



3.3 Aggregation und Komposition

Die **Aggregation** setzt einzelne Objekte miteinander in Beziehung und stellt eine Rangordnung unter ihnen her, die sich durch **„ist Teil von“** und **„besteht aus“** beschreiben läßt.

Die Anzahl der Objekte, die miteinander in Beziehung stehen wird durch die Angabe von **Kardinalitäten** gegeben.

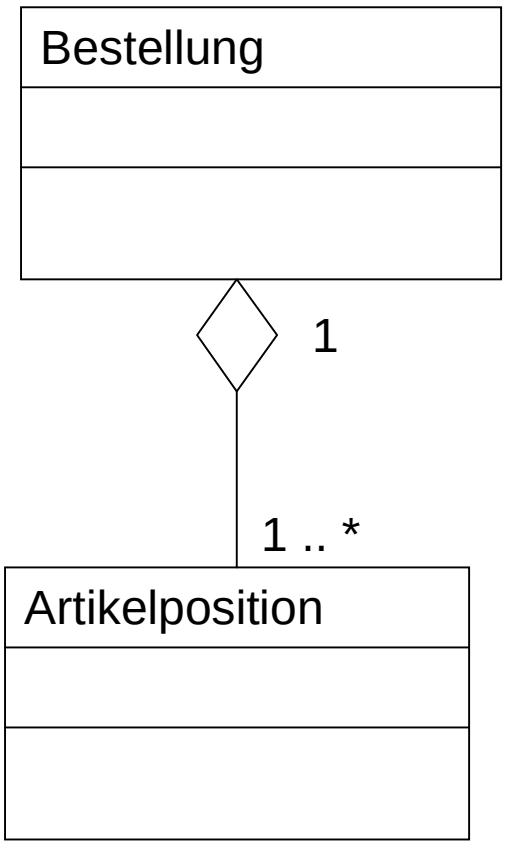
Bei der **Komposition** darf ein Teilobjekt nur zu genau einem Ganzen gehören und ist existentiell von ihm abhängig, d.h. eine Destruktion des Ganzen löscht auch den abhängigen Teil.

Kardinalitäten:

exakt eins:	1
optional:	0..1
null oder mehr:	0..*
eins oder mehr:	1..*
viele:	*

Beispiel: Aggregation

Eine Bestellung besteht aus mindestens einer Artikelposition. Eine Artikelposition gehört zu genau einer Bestellung.



[**UML**: Die Aggregation wird über eine Raute dargestellt.]

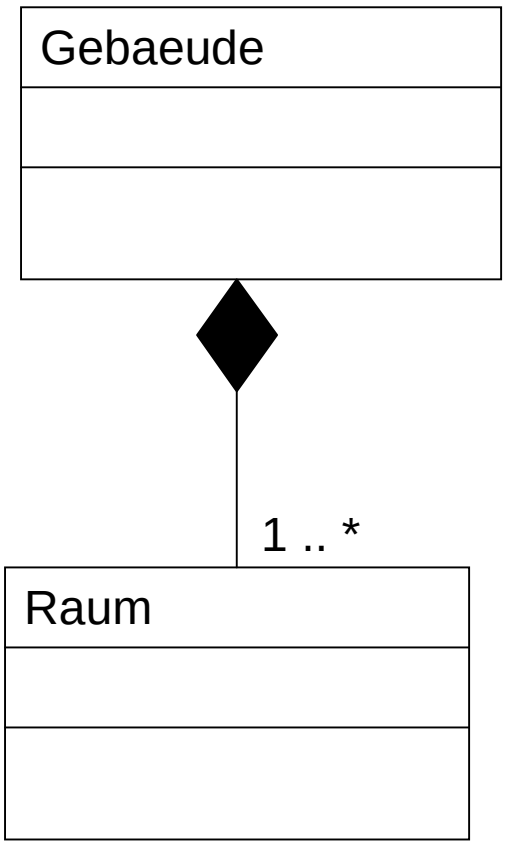
Beispiel: Aggregation - Umsetzung in Java

```
// Artikelposition.java
public class Artikelposition {
...
}

// Bestellung.java
public class Bestellung {
    ...
    // beliebig viele Artikelpositionen
    private Artikelposition[] posten;
    ...
}
```

Beispiel: Komposition

Eine Gebäude beinhaltet mehrere Räume. Jeder Raum gehört zu genau einem Gebäude und ist von ihm existentiell abhängig.



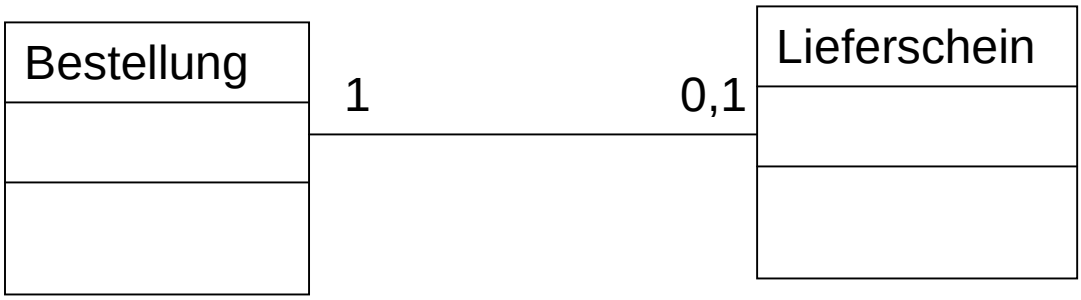
[**UML**: Die Komposition wird über eine ausgefüllte Raute dargestellt.]

3.5 Assoziation

Die **Assoziation** ist die Verallgemeinerung der Aggregation. Sie stellt eine gleichrangige Beziehung zwischen Objekten her.

Die Anzahl der Objekte, die miteinander in Beziehung stehen wird wiederum durch die Angabe von **Kardinalitäten** gegeben.

Beispiel:
Zu jeder Bestellung gehört maximal ein Lieferschein.



[**UML**: Die Assoziation wird über eine verbindende Linie dargestellt.]

Beispiel: Assoziation - Umsetzung in Java

```
// Bestellung.java
public class Bestellung {
    ...
    // kann auch „null“ sein
    private Lieferschein lieferschein;
    ...
}

// Lieferschein.java
public class Lieferschein {
    ...
    // bestellung != null muss sichergestellt werden
    private Bestellung bestellung;
    ...
}
```

3.6 Nachrichten

Die **Interaktion** von Objekten erfolgt über Nachrichtenaustausch.

Eine Nachricht ist eine Aufforderung eines (sendenden) Objekts an ein (empfangendes) Objekt, eine Methode auszuführen.

Ablauf:

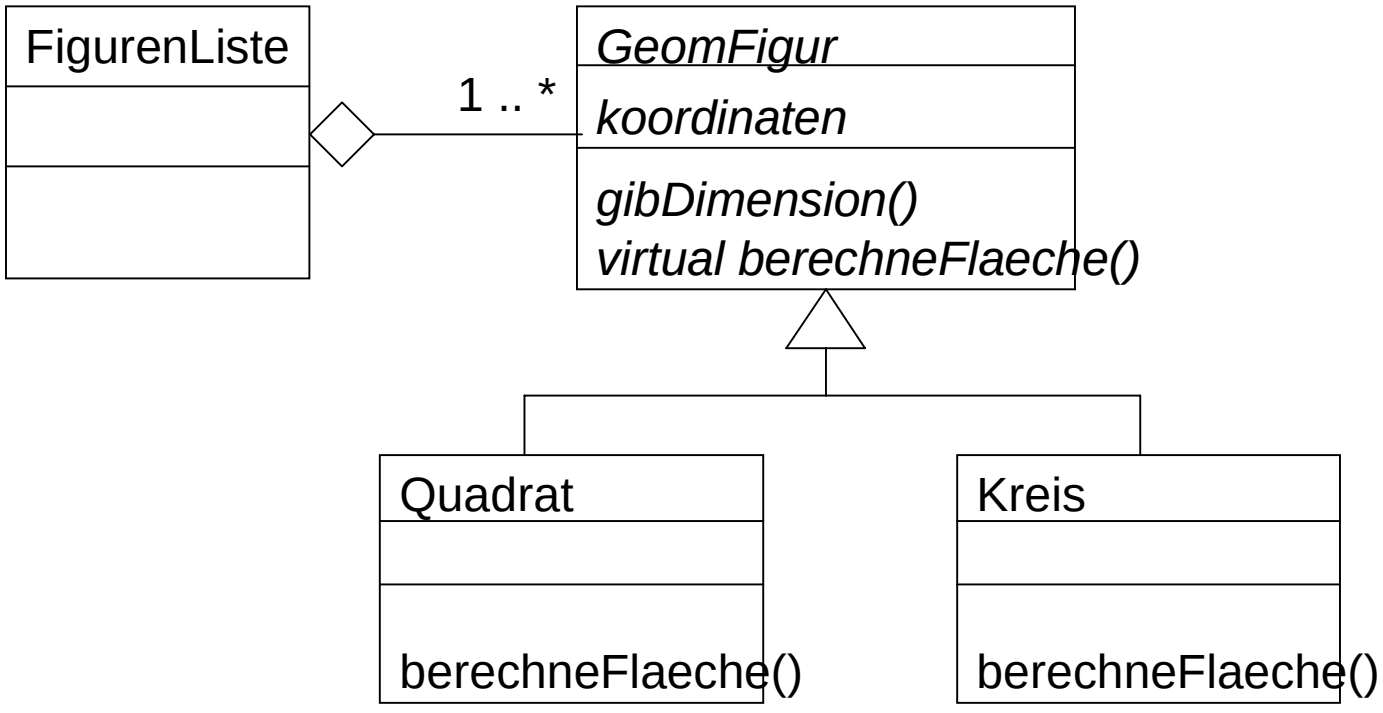
1. Ein Senderobjekt schickt eine Nachricht an ein Empfängerobjekt.
2. Das Empfängerobjekt führt eine seiner Methoden aus.
3. Das Empfängerobjekt schickt ggf. ein Ergebnis an das Senderobjekt zurück.

3.7 Polymorphismus (Vielgestaltigkeit)

In bestimmten Situationen ist es notwendig, dass Objekte unterschiedlicher Klassen einer Klassenhierarchie auf die gleiche Nachricht verschieden reagieren.

Beispiel:

Ausgabe der Fläche aller Figuren in einer Liste.



Beispiel: Umsetzung in Java (1)

```
// Quadrat.java
public class Quadrat extends GeomFigur {
    ...
    public double berechneFlaeche() { ... } // a*a
}
// Kreis.java
public class Kreis extends GeomFigur {
    ...
    public double berechneFlaeche() { ... } // PI*r*r
}

// FigurenListe.java
public class FigurenListe {
    ...
    public void setzeFigur(GeomFigur f);
    public GeomFigur gibFigur(int i);
    public int gibGroesse();
    ...
    private GeomFigur[] figur;
    ...
}
```

Beispiel: Umsetzung in Java (2)

```
// main-Funktion
static void main(String[] args)
{
    FigurenListe liste;

    Quadrat q = new Quadrat(5); // Seitenlaenge
    Kreis k      = new Kreis(4); // Radius

    liste.setzeFigur(q);
    liste.setzeFigur(k);

    for (int i = 0; i < liste.gibGroesse(); i++) {
       // hier wird fuer jedes Objekt die selbe
       // Funktion aufgerufen (fruehes Binden)
       System.out.println(figur.gibDimesion());

        GeomFigur figur = liste.gibFigur(i);
        // hier wird fuer jedes Objekt seine spezifische
        // Funktion aufgerufen (spaetes Binden)
        System.out.println(figur.berechneFlaeche());
    }
}
```