Objektorientierte Programmierung mit Java und UML

1. Grundlagen der objektorientierten Programmierung

2. Die Klasse
   - 2.1 Modellierung einer Klasse
   - 2.2 Realisierung
     - 2.2.1 Attribute
     - 2.2.2 Methoden
   - 2.3 Die Applikation (Arbeit mit einer Klasse)
   - 2.4 Übungen
   - 2.5 Get- und set-Methoden und Datenkapselung
   - 2.6 Übungen

3. Realisierung eines einfachen Anwendungsfalls (Fallbeispiel 1)
   - 3.1 Beschreibung der Fallsituation
   - 3.2 Modellierung
     - 3.2.1 Use Cases
     - 3.2.2 Klassendiagramm
   - 3.3 Realisierung und Applikation
   - 3.4 Erweiterungen
   - 3.4.1 Der Konstruktor
   - 3.4.2 Die toString-Methode
   - 3.5 Endfassung
   - 3.6 Mehrere Objekte einer Klasse in einem Array verwalten
   - 3.7 Übungen

4. Realisierung einer Anwendung mit mehreren Klassen (Fallbeispiel 2)
   - 4.1 Beschreibung der Fallsituation
   - 4.2 Analysephase
   - 4.3 Designphase
   - 4.4 Codierungsphase
   - 4.5 Erweiterung der Fallsituation – Vererbung
   - 4.6 Übungen

5. Arrays und Collections als Container für Beziehungen zwischen Klassen
   - 5.1 Verweisattribute als Array
   - 5.2 Collections am Beispiel ArrayList

6. UML-Diagramme
   - 6.1 Statische Sicht
     - 6.1.1 Das Anwendungsfalldiagramm (Use-case-Diagramm)
     - 6.1.2 Klassendiagramme und Erweiterungen
     - 6.1.3 Das Objektdiagramm
   - 6.2 Dynamischer Blick auf einen Anwendungsfall
     - 6.2.1 Das Aktivitätsdiagramm
     - 6.2.2 Das Interaktionsdiagramm
   - 6.3 Weitere Diagramme
   - 6.4 Übungen

7. Das objektorientierte Vorgehensmodell

8. Datenbankzugriff unter Java
9. Wiederholung - Grundbegriffe der Objektorientierten Programmierung

1. Grundlagen der objektorientierten Programmierung

Die Entwicklung komplexer moderner Softwaresysteme ist eine Aufgabe, die nur arbeitsteilig im Team vollzogen werden kann und die oft eine lange Zeit in Anspruch nimmt. Damit eine solche Aufgabe bewältigt werden kann, ist es notwendig, diese in mehrere Komponenten zu zerlegen. Man spricht von **Modularisierung**.

Modularisierte Programme erleichtern die Übersicht, lassen sich leichter warten und neuen Erfordernissen anpassen. Den Schlüssel dafür bietet die **objektorientierte Programmierung (OOP)**.
Die Notwendigkeit für objektorientierte Programmierung ergibt sich auch dadurch, dass Programme heute auf verschiedenen Plattformen laufen sollen, dass von verschiedenen Programmen auf gemeinsame Daten zugegriffen wird, dass Veränderungen von Programmen mit möglichst geringem Aufwand vollzogen werden können.

Noch stärker als bei der prozeduralen und strukturierten Programmierung gilt für die OOP, dass das zu entwickelnde System vorher genau geplant werden muss. Bis das Programm entsteht, vergeht eine lange Entwicklungszeit, die für die Analyse, die Strukturierung und die Modellierung verwendet wird.

Die OOP ist einerseits einer Weiterentwicklung der prozeduralen Programmierung, indem sie mit dem Konzept der **Klasse** einen neuen umfassenden Datentyp zur Verfügung stellt, andererseits gibt es aber auch völlig neue Prinzipien, die die prozeduralen Programmierung nicht zur Verfügung stellt. Die drei Basisprinzipien der OOP sind:

- **Kapselung**
- **Vererbung**
- **Polymorphie**.

Diese drei Prinzipien werden in den folgenden Kapiteln näher erläutert.

Bei der **prozeduralen** Programmierung stehen der logische Programmablauf im Vordergrund, bei der **objektorientierten** Programmierung stehen die **Daten** im Vordergrund.

Objekte mit gleichen Eigenschaften (Attributen) und gleichen Fähigkeiten (Methoden) werden zu Klassen zusammengefasst.

Eine Klasse beinhaltet die Eigenschaften (= Daten) sowie die Methoden, die mit diesen Daten arbeiten.

Beispiel: Ein Auto ist ein **Objekt**, das bestimmte **Eigenschaften und Methoden** besitzt. Diese werden zunächst in einer **Klassendefinition** genau beschrieben. Unser Auto soll zunächst einmal durch die **Eigenschaften**

- Kennzeichen
- Marke
- Kilometerstand
- Tankinhalt

beschrieben werden. Außerdem soll es die beiden **Methoden**

- Fahren
- Tanken

durchführen können

Dadurch, dass jetzt definiert wurde, welche Eigenschaften das Auto besitzt und welche Funktionen es ausführen kann, steht allerdings noch kein Auto vor der Tür. Dazu muss erst eine **Instanz** dieser Klasse erzeugt werden. **Eine Instanz ist dann das eigentliche Objekt**, von dem beliebig viele andere erschaffen werden können. Instanzen können z.B. jetzt *Annes Polo* und *Franks Golf* sein

Attribute und Methoden

Die Eigenschaften einer Klasse werden auch deren **Attribute** genannt. Zu einer kompletten Klassendefinition gehören auch Angaben darüber, welche Werte diese Attribute annehmen können. Da Attribute nichts anderes sind als Variablen geschieht dies durch die Angabe des Datentyps (int, float, char ...)
Alle Instanzen besitzen die gleichen Attribute, allerdings in verschiedenen Ausprägungen (z.B. Farbe rot oder gelb).

Als **Methoden** werden die Funktionen bezeichnet, die in ein Objekt eingebunden sind. In den Methoden müssen die Funktionen der Klasse genau beschrieben werden. Für das Beschleunigen etwa müsste man alles vom Niederdrücken des Gaspedals, über das Öffnen der Benzinzufuhr bis zur Veränderung der Vergasereinstellung programmieren. Alle Instanzen der gleichen Klasse benutzen dieselben Funktionen

Kapselung

Unter Kapselung versteht man, dass Attribute und Methoden in einem Objekt zusammengefasst werden und ihrem Umfeld nur als geschlossene Einheit zur Verfügung gestellt werden. Welches Attribut und welche Methode nach außen sichtbar sind, kann der Programmierer festlegen, indem er sie entweder als **private** oder **public** deklariert.
Alle Deklaration im private-Bereich sind **gekapselt**, d.h. sie können außerhalb der Klasse nicht angesprochen und verändert werden. Die Vereinbarungen im public-Bereich sind öffentlich, d.h. sie sind auch außerhalb der Klasse erreichbar. Die Datenstruktur soll ja von niemanden beeinflusst werden können, die Methoden aber sollen von unterschiedlichen Anwendungen benutzt werden können.

Vererbung

Vererbung ist eines der wichtigsten Prinzipien der OOP. Man versteht darunter, dass von bereits bestehenden Klassen neue Klassen abgeleitet werden können, die zunächst die gleichen Attribute und Methoden besitzen wie ihre Vorgänger. Sie können aber mit zusätzlichen Elementen ausgestattet werden. Diejenige Klasse, von der eine neue **Unterklasse** abgeleitet wird, nennt man **Oberklasse** oder **Basisklasse**.
Von der Basisklasse Auto ließ sich beispielsweise eine Unterklasse **LKW** bilden, die als zusätzlichen Attribute die *Ladekapazitä*t sowie als neue Methoden *Beladen* und *Entladen* hätte.

Polymorphie (Überladen von Funktionen)

Als Polymorphie bezeichnet man die Fähigkeit verschiedener Unterklassen ein und derselben Oberklasse, auf die gleiche Botschaft unterschiedlich zu reagieren. Das bedeutet, dass z. B. die Unterklasse LKW die gleiche Methode *Fahren* besitzt wie die Oberklasse Auto, die aber bei beiden unterschiedlich ausgeführt wird.
Dazu muss die vererbte Methode *Fahren* der Oberklasse durch eine veränderte Methode *Beschleunigen* überschrieben werden.

Überladen

Unter dem Überladen einer Methode versteht man, dass es innerhalb einer Klasse mehrere Varianten der gleichen Methode geben kann. Diese haben den gleichen Namen, unterscheiden sich aber durch ihre Argumente (Übergabeparameter).
So wäre es beispielsweise denkbar, dass die Methode *Hupen* in zwei Varianten existiert. Wird sie ohne Übergabewert aufgerufen, ertönt ein gewöhnliches Hupsignal, werden dieser Methode aber die Variable *Dauer* übergeben, ertönt ein Hupsignal von unterschiedlicher Dauer.

UML (Unified Modelling Language)

UML ist eine Modellierungssprache für objektorientierte Vorgehensmodelle. In der UML gibt es verschiedene Arten von Diagrammen, mit denen objektorientierte Programme in verschiedenen Phasen und aus unterschiedlichen Sichten modelliert werden.
Grundsätzlich kann man zwischen statischen und dynamischen UML-Darstellungen unterscheiden. **Statische Diagramme** beschreiben den Zustand eines Systems. Das wichtigste Diagramm ist das **Klassendiagramm**.
Dynamische Diagramme beschreiben das zeitliche und logische Verhalten eines Systems oder die Aktivitäten innerhalb eines Systems. Wichtige Diagrammarten sind dafür das **Interaktionsdiagramm** und das **Aktivitätsdiagramm**.

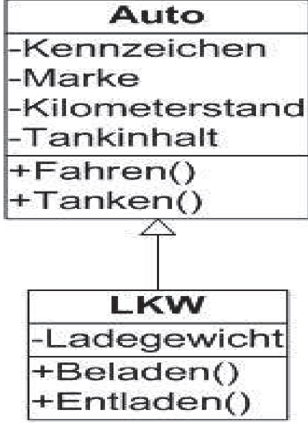
UML wird laufend weiterentwickelt und ist von der OMG standardisiert worden.
Die derzeit aktuelle Version ist die Version 2.1.2 (Nov. 07).

Da die UML mittlerweile einen sehr umfangreichen Notationsaufwand benötigt, werden wir im Rahmen des Unterrichts lediglich einige Diagramme der UML verwenden und den Notationsaufwand auf das Nötige begrenzen. In dieser eingeschränkten Form ist die UML ein sehr nützliches Hilfsmittel bei der Analyse und Modellierung objektorientierter Systeme.

Umfangreiche Möglichkeiten zur graphischen Darstellung von UML-Diagrammen bietet das Programm **Visio** von Microsoft. Einfache Klassendiagrammzeichner sind in vielen Java-Editoren integriert.

Darstellung eines Klassendiagramms nach UML

Im UML-Klassendiagramm wird unser Auto-Beispiel wie folgt dargestellt:



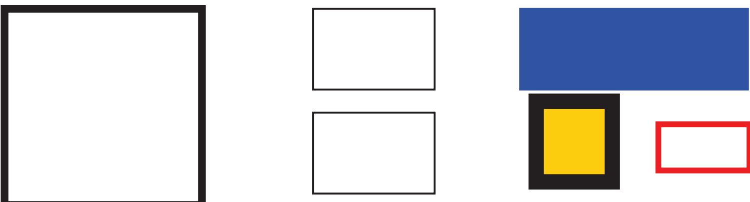
Wiederholungsfragen

1. **Nennen Sie Gründe, die zur Entwicklung der objektorientierten Programmierung geführt haben.**
2. **Durch welche Merkmale unterscheidet sie die OOP von der prozeduralen Programmierung?**
3. **Welche zwei Arten von Informationen gehören zur Definition einer Klasse?**
4. **Wie bezeichnet man die einzelnen Objekte, die von einer Klasse gebildet werden.**
5. **Erläutern Sie den Begriff Kapselung.**
6. **Was bedeutet Vererbung in der objektorientierten Programmierung?**
7. **Nennen Sie jeweils drei Attribute und Methoden einer fiktiven Klasse *Flugzeug* und zeichnen Sie die Klassendefinition in der UML-Darstellung.**

2. Die Klasse

Objekte mit gleichen Eigenschaften werden zu Klassen zusammengefasst.
Eine Klasse ist der Plan nach dem konkrete Objekte konstruiert werden.

Diesen Zusammenhang wollen wir uns am Beispiel einer Klasse mit der Bezeichnung **Rechteck** verdeutlichen.



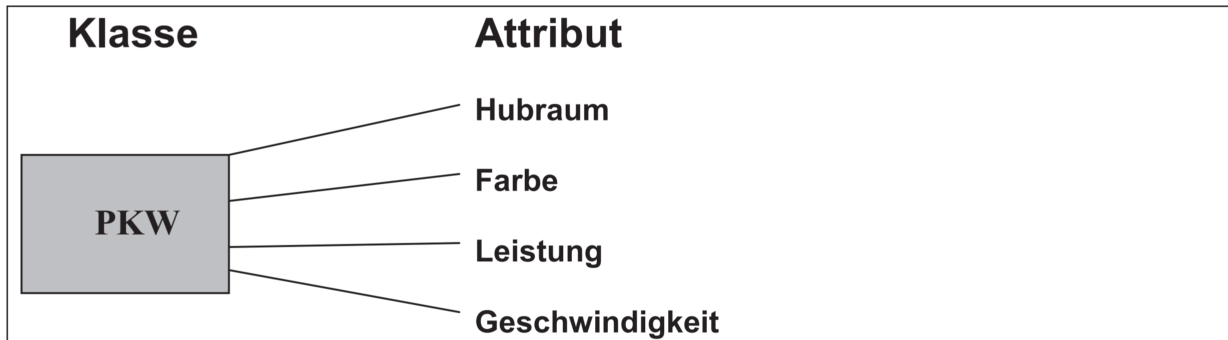
In obigem Schaubild sehen wir eine Anzahl von Rechtecken, die alle die gleichen Attribute, jedoch unterschiedliche Attributwerte aufweisen.
Alle Rechtecke haben Gemeinsamkeiten. Sie haben alle vier Seiten, von denen jeweils zwei gegenüberliegende Seiten parallel und gleich lang sind. Jeder Winkel im Rechteck beträgt 90 Grad.
Die Attribute (Seitenlängen, Farbe, Linienstärke, Linienfarbe) sind aber bei jedem Rechteck anders ausgeprägt.

Die einzelnen, oben abgebildeten Rechtecke sind **Objekte** mit unterschiedlichen Attributausprägungen, die alle zur gemeinsamen Klasse **Rechteck** gehören.
Wir sagen auch: Die Rechtecke sind **Instanzen** oder **Objekte** der Klasse Rechteck.

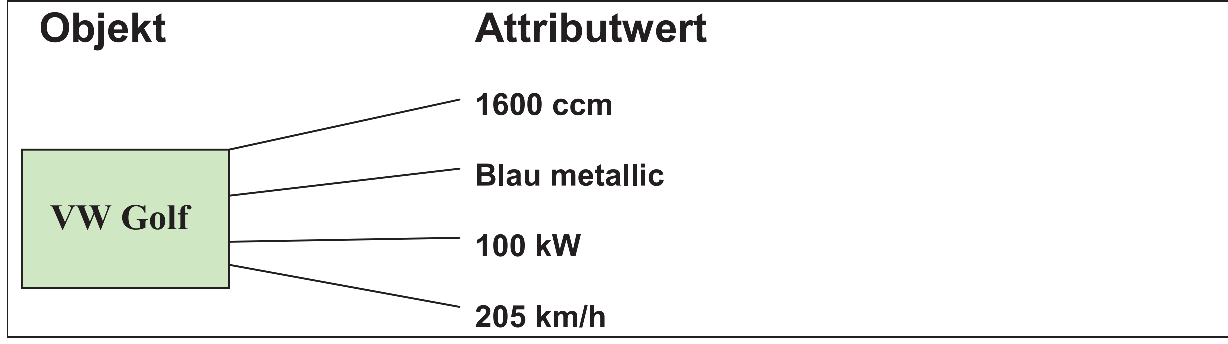
Für andere geometrische Formen wie Dreiecke, Kreise usw. lassen sich ähnliche Beziehungen bilden.

Beispiel für Objekte und Klassen:

Nehmen wir an, alle PKWs bilden eine Klasse mit folgenden Attributen:



Eine **Instanz (Objekt)** der Klasse PKW könnte folgendermaßen aussehen:



Auf diese Art lassen sich also beliebig viele Objekte der Klasse PKW erzeugen.

2.1 Modellierung einer Klasse

Anhand des Auto-Beispiels wollen wir lernen, wie man das Design einer Klasse entwirft, in einem UML-Klassendiagramm darstellt und wie man das Diagramm in Java-Quellcode umsetzt. Dann werden wir Objekte dieser Klasse erzeugen und mit einer Applikation auf die Klasse zugreifen.

Als erstes legen wir fest, welche Eigenschaften (**Attribute**) ein Auto haben soll: Wir beschränken uns dabei auf die Attribute ***kennzeichen***, ***marke***, ***kilometerstand*** und ***tankinhalt***.
Für jedes Attribut legen wir den Datentyp fest und erhalten folgende Übersicht:

Datentyp	**Bezeichnung**
String	kennzeichen
String	marke
int	kilometerstand
double	tankinhalt

Jetzt legen wir fest, welche Fähigkeiten (Methoden) unser Auto haben soll. Wir beschränken uns auf die beiden Methoden **fahren** und **tanken**
Fahren bedeutet, dass sich der Kilometerstand des Fahrzeuges erhöht, Tanken bedeutet, dass der Tankinhalt des Autos erhöht wird.

(Es ist Java-Konvention, alle Attributsbezeichner und Methodennamen mit kleinen Buchstaben zu beginnen)

Nachdem wir uns über Aufbau und Funktion der Klasse einig sind, zeichnen wir das **Klassendiagramm.**

Auto
kennzeichen: String **marke: String** **kilometerstand: int** **tankinhalt: double**
fahren(int) **tanken(double)**

Das Klassendiagramm ist die wichtigste Diagrammart in der UML. Es beschreibt den Aufbau einer Klasse mit Attributen und Methoden. Es stellt sozusagen den Bauplan dar für die Objekte, die dann von dieser Klasse erzeugt werden. Folgende Regeln sollten für die Erstellung eines Klassendiagramms beachtet werden:

Das Klassendiagramm besteht aus einem Rechteck, welches in drei Bereiche unterteilt ist.
In der **oberen Zeile** steht der **Klassenname** zentriert und fett.
In der **zweiten Zeile** werden die **Attribute** aufgelistet. Das Minuszeichen vor der Attributsbezeichnung bedeutet, dass das Attribut gekapselt ist und mit dem Schlüsselwort **private** versehen ist (siehe Kapitel Kapselung). Wir werden das in Zukunft immer so handhaben.
Außerdem ist es möglich, den Datentyp des Attributes hinzuzufügen. Z.B. *–kilometerstand ; int*
In der **dritten Zeile** stehen die **Methoden**. Diese sind mit dem Schlüsselwort **public** versehen, was an dem Pluszeichen zu erkennen ist. Methoden können mit Übergabe- oder Rückgabewert noch näher beschrieben werden.
Zur Darstellung von Klassendiagrammen mit mehreren Klassen und deren Beziehungen zueinander, können Attribute und Methoden weggelassen werden.

2.2 Realisierung

Nachdem der Aufbau der Klasse durch das Klassendiagramm modelliert wurde geht es daran, die Klasse Auto in Java umzusetzen. Dazu erzeugen wir eine neue Java-Klasse mit der Bezeichung Auto.

2.2.1 Attribute

Der **Java-Quellcode** für unser Auto sieht zunächst folgendermaßen aus:

```
public class Auto {
  //Attribute
  String kennzeichen;
  String marke;
  int kilometerstand;
  double tankinhalt;
}
```

Nach dem Klassenkopf mit dem Namen der Klasse werden die Attribute mit ihrem Datentyp einzeln aufgelistet. (Klassenbezeichnungen beginnen mit Großbuchstaben, Attribute mit Kleinbuchstaben).
Damit ist die Klasse zunächst funktionsfähig und kann nun erfolgreich kompiliert werden.

2.2.2 Methoden

Unser Auto verfügt jetzt über Attribute aber noch nicht über Methoden. Methoden werden in die Klassendefinition eingebaut und sehen aus wie Prozeduren. Nach Realiserung der Methoden ***fahren*** und ***tanken*** sieht unsere Klasse wie folgt aus:

```
public class Auto {
  //Attribute
  String kennzeichen;
  String marke;
  int kilometerstand;
  double tankinhalt;
  //Methoden
  public void fahren(int km)
  {
        this.kilometerstand=this.kilometerstand+km;
  }
  public void tanken(int l)
  {
        this.tankinhalt=this.tankinhalt+l;
  }
}
```

Zum Aufbau einer Methode siehe Kapitel 11 im strukturierten Teil des Skripts. Die Methode ***fahren*** funktioniert nun wie folgt:
Der Methodenkopf zeigt an, dass der Methode ein Integerwert übergeben wird, der die gefahrenen Kilometer repräsentiert. Im Rumpf der Methode wird nun das Attribut kilometerstand um den übergebenen Wert erhöht. Das Schlüsselwort **this** welches durch einen Punkt mit dem Attribut verbunden ist, verweist darauf, dass der Wert des Attributes für jedes Auto unterschiedlich sein kann.

Die Klassendatei wird als eigene Datei unter der gleichen Bezeichnung wie der Klassenname gespeichert und kompiliert. Sie enthält keine main-Prozedur, da sie nicht ausführbar ist.

Mit dem Java-Editor kann man die Klasse über die Option *UML – Neue Klasse* interaktiv erstellen, ohne dass wir den Quellcode selbst schreiben müssen. Lediglich die beiden Methoden ***fahren*** und ***tanken*** müssen wir im Quellcode ergänzen.

Die Klassendatei **auto.java** wird nun kompiliert aber nicht ausgeführt, da eine Klasse keine ausführbare Datei ist (keine main-Methode).

2.3 Die Applikation (Arbeit mit einer Klasse)

Wir haben jetzt zwar den Plan, wie ein Auto aussehen soll, es existiert aber noch kein konkretes Auto. Im nächsten Schritt erstellen wir eine Applikation **autofahren**, die ein Auto-Objekt erzeugt und alle Attribute besetzt.
Zunächst überlegen wir uns, welche Eigenschaften unser Auto haben soll, d.h. welche Ausprägungen die Attribute der Klasse haben sollen.

Zur Veranschaulichung dient hier ein weiteres UML-Diagramm, das **Objektdiagramm**. Das Objektdiagramm ähnelt dem Klassendiagramm, es werden allerdings die konkreten Ausprägungen der Attribute für ein Objekt benannt.

In der ersten Zeile steht die Objektbezeichnung gefolgt von der Klassenbezeichnung fett und unterstrichen
In der zweiten Zeile folgen die Attributbezeichnungen und die konkreten Ausprägungen der Attribute

Objektdiagramm

meinAuto : Auto	
kennzeichen	GI-AB 123
marke	Opel
kilometerstand	25600
tankinhalt	30

Um nun von unserer Klasse reale Objekte zu erzeugen, müssen wir eine **Java-Applikation** schreiben.
Die entscheidende Zeile zur Erzeugung eines Objektes lautet: **auto meinAuto = new auto( );**
Das neue Objekt wird nun unter der Bezeichnung meinAuto angesprochen. Diese Bezeichung ist beliebig.

```
// autofahren
public class autofahren
{
  public static void main(String argv [ ])
  {
      auto meinAuto = new auto();
      meinAuto.kennzeichen="GI-AB 123";
      meinAuto.marke="Opel";
      meinAuto.kilometerstand=25600;
      meinAuto.tankinhalt=30;
  }
}
```

Mit den Anweisungen die der Objekterzeugung folgen, werden die Attribute mit den entsprechenden Werten initialisiert.

Wenn wir für ein Objekt eine Methode aufrufen wollen, so muss die Objektbezeichnung immer der Methodenbezeichnung durch einen Punkt getrennt vorangestellt werden.

Je nachdem, ob die Methode Übergabeparameter enthält oder Rückgabewerte liefert, müssen Werte in die Klammern des Methodenaufrufs geschrieben werden oder der Rückgabewert des Methodenaufrufes in der Applikation aufgefangen werden. Die Methode ***tanken*** rufen wir wie folgt auf:

```
meinAuto.tanken(25);                    // Wir tanken 25 Liter
```

entsprechend die Methode ***fahren***

```
meinAuto.fahren(170);                   // Wir fahren 170 km
```

Wir können nun die geänderten Werte für den tankinhalt und den kilometerstand zur Kontrolle wieder am Bildschirm ausgeben. Dazu folgende Anweisungen:

```
System.out.println("Neuer Tankinhalt: "+meinAuto.tankinhalt);
System.out.println("Neuer Kilometerstand: "+meinAuto.kilometerstand);
```

Es sollte folgende Bildschirmausgabe erscheinen:

```
Neuer Tankinhalt: 55.0
Neuer Kilometerstand: 25770
```

2.4 Übungen

1. Wir benötigen eine Klasse **Person** mit den Attributen **name**, **vorname** und **alter**. Spezielle Methoden werden nicht benötigt. Erzeugen Sie diese Klasse und dazu eine Applikation. Erzeugen Sie zwei unterschiedliche Objekte und lassen sie deren Attributwerte am Bildschirm ausgeben.

2. Wir benötigen eine Klasse **Rechteck** mit den Attributen **seiteA** und **seiteB** und einer Methode ***showFlaeche***, mit der die Fläche des Rechtecks berechnet werden soll. Zeichnen Sie das Klassendiagramm, codieren Sie die Klasse und erstellen Sie eine Applikation in der die Fläche eines Objekts der Klasse Rechteck am Bildschirm angezeigt wird.

3. Codieren Sie anhand des Klassendiagrammes die Klasse und eine Applikation, die die Ergebnisse der beiden Methoden am Bildschirm ausgibt.

Kreis
radius; double
berFlaeche: double berUmfang: double

4. Es soll eine Klasse **Artikel** zur Erfassung der Artikeldaten eines Unternehmens erstellt werden. Es werden die Attribute, **artikelnr** – int, **artbez** – String, **preis** – double, **bestand** – int benötigt. Außerdem eine Methode ***berWert***, die aus preis und bestand den Gesamtwert berechnet. Codieren Sie die Klasse, erzeugen Sie mindestens 2 Instanzen dieser Klasse und lassen sie den Gesamtwert aller Artikel anzeigen.

2.5 Get- und set-Methoden und Datenkapselung

Wie bereits gesagt wurde, ist die Datenkapselung ein wesentliches Merkmal der objektorientierten Programmierung. Das bedeutet, dass Attribute und Methoden einer Klasse eine Einheit darstellen und das Attribute vor Manipulationen geschützt sein sollen. Diese wird über das Schlüsselwort **private** erreicht. Attribute, die gekapselt werden sollen, werden mit dem Schlüsselwort **private** versehen.
Damit sind sie vor jeglichem Zugriff geschützt. Eine Veränderung solcher Attribute ist nur noch mit einer speziellen Methode möglich, die das Schlüsselwort **public** besitzen muss. Für jedes Attribut, welches den Status **private** besitzt sollte es je eine Methoden geben, die das Attribut mit einem Wert besetzen kann und eine Methode, die den aktuellen Wert eines Attributes zurückliefert. Diese Methoden werden get- und set-Methode genannt.
Wir werden in weiteren Verlauf alle Attribute einer Klasse kapseln. Die get- und set-Methoden für jedes Attribut müssen nicht im Klassendiagramm dargestellt werden, da dieses sonst zu unübersichtlich werden würde. Die meisten Case-Tools zum Design und zur Codierung einer Klasse erzeugen get- und set-Methoden automatisch.
Nach Einführung der Datenkapselung sieht unsere Klasse nun wie folgt aus:

```
public class Auto {
 //Attribute
 private String kennzeichen;
 private String marke;
 private int kilometerstand;
 private double tankinhalt;
 //Methoden
 public void fahren(int km)
 {
        this.kilometerstand=this.kilometerstand+km;
 }
 public void tanken(int l)
 {
        this.tankinhalt=this.tankinhalt+l;
 }
 public void setKennzeichen(String k)
 { this.kennzeichen = k;
 }
 public String getKennzeichen( )
 { return this.kennzeichen;
 }
 public void setMarke(String m)
 { this.marke = m;
 }
 public String getMarke( )
 { return this.marke;
 }
 public void setKilometerstand(int k)
 { this.kilometerstand = k;
 }
 public int getKilometerstand( )
 { return this.kilometerstand;
 }
 public void setTankinhalt(double l)
 { this.tankinhalt = l;
 }
 public double getTankinhalt( )
 { return this.tankinhalt;
 }
}
```

Im Klassendiagramm werden Attribute die private sind mit einem – gekennzeichnet und public-Methoden mit einem +

```
Auto
-kennzeichen
-marke
-kilometerstand
-tankinhalt
+fahren()
+tanken()
```

2.6 Übungen

1. Verwenden Sie die Klasse **Artikel** aus Aufgabe 4 von Kapitel 2.4. Versehen Sie alle Attribute mit dem Schlüsselwort **private**. Welche Fehlermeldung erhalten Sie, wenn Sie das Attribut **artnr** des Objektes a1 mit der Anweisung `System.out.println("Artikelnr.: "+a1.artnr);` ausgeben wollen?
   Wie können Sie das Problem lösen?

2. Welche Änderungen ergeben sich in nebenstehendem Klassendiagramm, wenn die Eigenschaften **private** und **public** mit dargestellt werden sollen?

Kreis
radius; double
berFlaeche: double berUmfang: double

3. Codieren Sie die Klasse Person (Aufg. 1, Kapitel 2.4) neu, indem alle Attribute gekapselt werden.
   Wie sieht der Aufruf in der Applikation aus, mit der das Attribut **vorname** des Objektes **p1** am Bildschirm angezeigt wird?

Zur Wiederholung

a) Wie werden Attribute in einer Klasse angelegt:

Zum Anlegen eines Attributes in einer Klasse werden üblicherweise 3 Schlüsselwörter verwendet:

Zugriffsrecht, Datentyp und Attributsbezeichung

Beispiel: `private int breite;`

Zugriffsrecht: Für Attribute gibt es die Zugriffsrechte privat, public und protected.

- private: (-) Wegen der Datenkapselung sollten Attribute grundsätzlich als private gekennzeichnet werden. Sie sind dann außerhalb der Klasse nicht mehr direkt ansprechbar, sondern müssen über get- und set-Methoden angesprochen werden

- public (+) Mit public gekennzeichnete Attribute können überall genutzt werden. Dieses Zugriffsrecht wird weniger für Attribute als für Methoden oder Klassen verwendet.

- protected (#) Auf Attribute und Methoden, die mit protected gekennzeichnet sind, kann innerhalb desselben Paketes zugegriffen werden. Derartige Attribute und Methoden werden an alle Subklassen weitervererbt und sind dort zugänglich. Attribute einer Basisklasse, die in einer Methode einer abgeleiteten Klasse verwendet werden, müssen protected sein.

Datentyp: Die gängigen Datentypen sind String, int, double, char oder Klassenbezeichner

Attributsbezeichnungen: Die Bezeichnung eines Attributes ist frei wählbar. Java-Konvention ist es, generell mit kleinen Buchstaben zu beginnen und bei zusammengesetzten Wörtern die Anfangsbuchstaben des folgenden Wortes groß zu schreiben: Bsp.: **zimmerBreite**.

b) Wie ist eine Methode aufgebaut:

Jede Methode besteht aus einem Methodenkopf, einem Paar geschweifter Klammern und dem Methodenrumpf.

Allgemein: ***Zugriffsrecht Rückgabedatentyp Methodenbezeichnung (Übergabeparameterliste)***

```
{
        Methodenrumpf (Quellcode)
}
```

Beispiel:

```
public int berFläche(int a, int b)
{
        int f;                          oder kürzer
        f = a * b;                      return a*b;
        return f;
}
```

Zugriffsrecht: Methoden sollten in der Regel mit **public** gekennzeichnet sein.

Rückgabedatentyp: Wenn eine Methode mit **return** ein Ergebnis zurückliefert, so ist hier der Datentyp des zurückgelieferten Wertes anzugeben. Wenn es kein return gibt, so ist der Rückgabedatentyp **void**. Eine Methode kann immer nur einen Wert zurückgeben.

Methodenbezeichnung: frei wählbar, Java-Konventionen beachten

Übergabeparameter: Es können beliebig viele Parameter übergeben werden, die mit Datentyp und Bezeichnung angegeben werden müssen. Die Bezeichnung ist innerhalb der Methode frei wählbar und muss nicht mit der Variablenbezeichnung beim Methodenaufruf übereinstimmen (call by value). Die richtige Reihenfolge muss aber unbedingt eingehalten werden.

Beispiel: obiges Beispiel kann in der Applikation wie folgt aufgerufen werden:

`r1.berFläche(zahl1, zahl2)` oder auch `r1.berFlächet(5, 7);`

(r1 steht für die Bezeichnung des aufrufenden Objekts)

3. Realisierung eines einfachen Fallbeispiels

Nachdem wir nun die Grundlagen des Klassenkonzepts kennen gelernt haben, wollen wir anhand von zwei Fallbeispielen im Kapitel 3 und 4 zeigen, wie man einfache objektorientierte Anwendungssysteme entwickelt. Dabei wollen wir auf objektorientierte Vorgehensmodelle eingehen und weitere Anwendungsmöglichkeiten der UML kennenlernen

3.1 Beschreibung der Fallsituation

Auslöser einer objektorientierten Anwendungsentwicklung ist häufig eine Fallsituation, die in Form eines Textdokumentes oder eines Befragungsergebnisses vorliegt. Dieses liefert die Ausgangslage für die Modellierung des Anwendungssystems

Peter Merz hat sich eine neue Wohnung gemietet. Vor dem Einzug ist noch viel zu tun.
Der Fußboden muss neu verlegt werden und die Wände müssen gestrichen werden. Für die Berechnung der notwendigen Materialmengen muss Peter Merz die Grundfläche jedes einzelnen Zimmers sowie die Wandfläche jedes Zimmers berechnen. Wir wollen diese Aufgabe mit Hilfe eines objektorientierten Java-Programms lösen

Anhand der vorliegenden Aussagen werden die zu realisierenden Anwendungsfälle sowie die benötigten Klassen herausgefunden. Dieses gehört zur **Analysephase** der Softwareentwicklung.

3.2 Modellierung

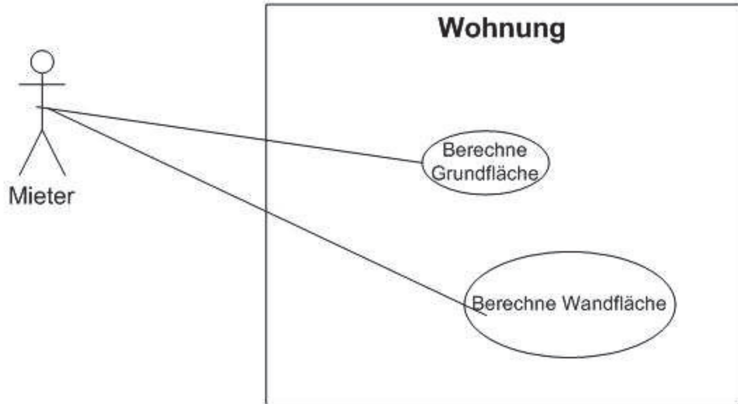
Anhand der Ergebnisse der Analysephase wird versucht, dass künftige System mit Hilfe der UML-Diagramm Use-case-Diagramm und Klassendiagramm zu entwerfen. Dieses gehört zur **Designphase** der Softwareentwicklung.

3.2.1 Use Cases

Auslöser unserer Aufgabenstellung ist die Absicht zwei **Anwendungsfälle** zu realisieren:

1. Berechnen der Grundfläche eines Zimmers
2. Berechnen der Wandfläche eines Zimmers

In der UML gibt es für die Darstellung der Anwendungsfälle einen speziellen Diagrammtyp, das sog. **Anwendungsfalldiagramm** oder auch **Use-Case-Diagramm**.
Anwendungsfalldiagramme werden zur Vereinfachung der Kommunikation zwischen Entwickler und zukünftigen Nutzer bzw. Kunde erstellt. Sie sind vor allem bei der Festlegung der benötigten Kriterien des zukünftigen Systems hilfreich. Somit treffen Anwendungsfalldiagramme eine Aussage, *was* zu tun ist, aber nicht *wie* das erreicht wird.
Anwendungsfälle werden durch **Ellipsen,** die den Namen des Anwendungsfalles tragen und einer Menge von beteiligten Objekten (**Akteuren**) dargestellt. Zu jedem Anwendungsfall gibt es eine Beschreibung in Textform. Die entsprechenden Anwendungsfälle und Akteure sind durch **Linien** miteinander verbunden. Akteure können durch Strichmännchen dargestellt werden. Die **Systemgrenze** wird durch einen Rahmen um die Anwendungsfälle symbolisiert.
Für unserer Fallbeispiel könnte das Anwendungsfalldiagramm so aussehen:

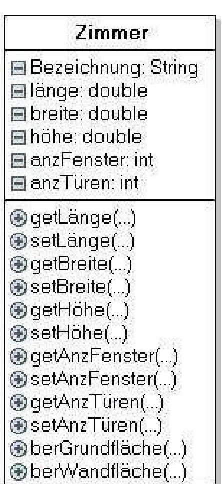


Das Anwendungsfalldiagramm ist sehr einfach und anschaulich und stellt den Ausgangspunkt für die Realisierung des Systems dar.

(Zum Use-case-Diagramm siehe auch Kapitel 9.3)

3.2.1 Das Klassendiagramm

Aus der Beschreibung der Fallsituation ergibt sich, dass wir mit einer einzigen Klasse auskommen können, da Peters Wohnung aus den Zimmern Wohnzimmer, Schlafzimmer, Küche, Bad und Flur besteht. Alle Zimmer haben eine rechteckige Grundfläche und haben eine unterschiedliche Zahl an Fenstern und Türen. Es liegt daher nahe, eine Klasse **Zimmer** zu modellieren.
Wir wollen folgende Attribute verwenden: **Länge**, **Breite** und **Höhe** des Zimmers, die **Anzahl Türen** und die **Anzahl Fenster**. Länge, Breite und Höhe haben den Datentyp double, die Zahl der Türen und Fenster sind Integerwerte. Außerdem benötigen wir ein Attribut **Bezeichnung** als String-Wert, welches uns anzeigt, um welches Zimmer es sich handelt. Es ergibt sich folgendes Klassendiagramm:



Die wichtigen beiden Methoden der Klasse sind die Methoden ***berGrundfläche*** und ***berWandfläche***, mit der die Grundfläche und die Wandflächen berechnet werden sollen
Außerdem sind hier im Klassendiagramm alle get- und set-Methoden mit aufgeführt. Dies ist normalerweise nicht üblich bzw. notwendig, da sich bei einer großen Anzahl von Attributen das Klassendiagramm nur unnötig aufblähen würde.
Für die Attribute sind die Datentypen ersichtlich, bei den Methoden sind in dieser Darstellung die Übergabe und Rückgabeparameter nicht ersichtlich. Die Ausführlichkeit des Klassendiagramms hängt von dem jeweiligen case-Tool ab, welches zum Design der Klasse verwendet wird.

3.3 Realisierung und Applikation

Der Quellcode für die **Klasse** sieht folgendermaßen aus:

Zur Verkürzung wird hier lediglich die get- und set-Methode für das Attribut **länge** dargestellt.

Die Methode ***berGrundfläche*** soll die Grundfläche eines Zimmer berechnen. Sie benötigt dazu die beiden Attribute **länge** und **breite**. Die Methode enthält keine Übergabeparameter, da die Werte für die Länge und die Breite schon bei der Erzeugung des Objekts mit einer set-Methode festgelegt wurden. Das Ergebnis der Berechnung **länge * breite** wird mit **return** zurückgeliefert.

Die Methode ***beWandfläche*** soll die Fläche aller Zimmerwände berechnen. Sie benötigt dazu die Attribute **länge, breite** und **höhe**. Außerdem sind die Flächen für die Türen und die Fenster abzuziehen.
Aus Vereinfachungsgründen wollen wir eine Tür mit den Maßen 1 m x 2 m und ein Fenster mit den Maßen 1 m * 0,8 m ansetzen.
Die Methode enthält ebenfalls keine Übergabeparameter.

```
public class Zimmer {

  // Anfang Variablen
  private String Bezeichnung;
  private double länge;
  private double breite;
  private double höhe;
  private int anzFenster;
  private int anzTüren;
  // Ende Variablen

  // Anfang Ereignisprozeduren

  public double getLänge() {
    return länge;
  }

  public void setLänge(double länge) {
    this.länge = länge;
  }


  public double berGrundfläche()
  {
     return this.breite*this.länge;
  }
  public double berWandfläche()
  {
    double w,w1,w2,tf,ff;
    w1=this.länge*this.höhe*2;
    w2=this.breite*this.höhe*2;
    tf=this.anzTüren*1*2;
    ff=this.anzFenster*1*0.8;
    w=w1+w2-(tf+ff);
    return w;
  }
  public String getBezeichnung() {
    return Bezeichnung;
  }

  public void setBezeichnung(String Bezeichnung) {
    this.Bezeichnung = Bezeichnung;
  }

  // Ende Ereignisprozeduren
}
```

Wir nennen unser **Applikation** ‚`wohnen`' und erzeugen zunächst ein Objekt z1, welches das Wohnzimmer sein soll. Das folgende Objektdiagramm zeigt die Ausgangsdaten:

z1 : Zimmer	
Bezeichnung	Wohnzimmer
länge	5
breite	4
höhe	2,4
anzFenster	3
anzTüren	2

```
public class wohnen
{
  public static void main(String argv[])
  {
    Zimmer z1= new Zimmer( );
    z1.setLänge(5);
    z1.setBreite(4);
    z1.setHöhe(2.4);
    z1.setAnzFenster(3);
    z1.setAnzTüren(2);
```

Der Quellcode links erzeugt ein Objekt mit der internen Bezeichnung z1 und initialisiert alle Attribute mit einer set-Methode.

```
//wir rufen die Methoden zur Berechnung der Grundfläche und der Wandfläche auf und lassen den
//Rückgabewert am Bildschirm anzeigen

    System.out.println("Grundfläche: "+z1.berGrundfläche());
    System.out.println("Wandfläche: "+z1.berWandfläche());
  }
}
```

Auf die gleiche Weise müssen wir nun die übrigen Objekte für die anderen Zimmer erzeugen und initalisieren.

3.4 Erweiterungen

3.4.1 Der Konstruktor

Attribute für alle Objekte mit einer set-Methode zu initialisieren ist mitunter recht mühsam. Einfacher wäre es, wenn es möglich wäre, direkt bei der Erzeugung eines neuen Objektes alle Attribute mit einer einzigen Anweisung zu initialisieren. Dieses ermöglicht ein **Konstruktor**.
Eine Konstruktor ist eine Methode, die ein neues Objekt einer Klasse erzeugt. Der Aufruf
`Zimmer z1 = new Zimmer( ) ;` ist ein solcher Konstruktor. Wir erkennen an dem Klammerpaar, dass es sich um eine Methode handelt. Ein Konstruktor ist eine Methode, die die gleiche Bezeichnung trägt wie die Klasse. Ein Konstruktor, der keine Übergabeparameter aufweist ist der sog. **Standardkonstruktor**.
Dieser muss nicht explizit programmiert werden, sondern wird vom Compiler beim Erzeugen eines Objekts als Standardmethode aufgerufen.
Wenn wir einen Konstruktor verwenden möchten, der etwas mehr tut, als ein Objekt lediglich zu erzeugen, so müssen wir uns einen eigenen Konstruktor programmieren und diesen in der Klasse implementieren.

Konstruktor als Methode der Klasse Zimmer:

```
public Zimmer(String z,double l, double b, double h, int f, int t)
 {
   this.Bezeichnung=z;
   this.länge=l;
   this.breite=b;
   this.höhe=h;
   this.anzFenster=f;
   this.anzTüren=t;
 }
```

this: Der Wert eines Attributes gehört immer zu einem bestimmten Objekt der Klasse. In der Applikation wird dies immer durch die Objektbezeichnung gefolgt von einem Punkt kenntlich gemacht. Da wir in der Klasse das konkrete Objekt nicht kennen, verwendet man den Parameter **this** als Platzhalter für das konkrete Objekt, welches in der Applikation gemeint ist. **This** kann innerhalb der Klasse vor einem Attribut oder einer Methode stehen.

So wird der Konstruktor in der Applikation aufgerufen:

```
Zimmer z1= new Zimmer("Wohnzimmer",5,4,2.40,3,2);
```

Es ist darauf zu achten, dass Reihenfolge und Datentypen der Übergabeparameter in der Applikation und in der Klasse genau übereinstimmen.

Achtung: Man kann mehrere unterschiedliche Konstruktoren definieren. Wenn ein eigener Konstruktor geschrieben wurde, wird der Standardkonstruktor dadurch überschrieben. Er existiert dann nicht mehr. Wenn man jetzt ein Objekt erzeugen möchte ohne Attribute zu initialisieren, so muss der **Standardkonstruktor** erst wieder explizit codiert werden. Das sieht einfach so aus:

```
public Zimmer( ) {
}
```

3.4.2. Die toString-Methode

Im Rahmen der Dateiverarbeitung im vorhergehenden Teil des Skripts wurde bereits die toString-Methode als Möglichkeit, Daten in eine Datei zu schreiben, erwähnt. Die toString-Methode kann dazu verwendet werden, die Attributwerte eines Objektes wieder auszugeben, ohne die jeweilige get-Methode für die Attribute verwenden zu müssen. Dafür bilden wir einen Ausgabestring, der unsere gewünschte Bildschirmausgabe enthält. Der Aufruf der toString-Methode ist dann denkbar einfach:
Es wird lediglich die Objektinstanz innerhalb der Ausgabeanweisung übergeben.
Zuvor muss in der Klasse die toString-Methode in der Klasse implementiert werden. Für unser Beispiel wie folgt:

```
public String toString( )
{
   String ausgabe = "Zimmer: "+this.Bezeichnung+", Länge: "+this.länge+", Breite: "+this.breite;
   return ausgabe;
}
```

Der Aufruf in der Applikation erfolgt dann so

```
System.out.println(z1);
```

Und erzeugt folgende Ausgabe:
Zimmer: Wohnzimmer, Länge: 5, Breite: 4

3.5 Endfassung

Die entgültige Applikation mit Ausgabe der Daten über die to-String-Methode und Ausgabe aller Flächen sieht so aus:

```
public class wohnen
{
  public static void main(String argv[])
  {
    Zimmer z1= new Zimmer("Wohnzimmer",5,4,2.40,3,2);
    Zimmer z2= new Zimmer("Schlafzimmer",4,4,2.40,2,1);
    Zimmer z3= new Zimmer("Küche",3,3,2.40,1,1);

    System.out.println(""+z1.getBezeichnung()+" Grundfläche: "+z1.berGrundfläche());
    System.out.println(""+z1.getBezeichnung()+" Wandfläche: "+z1.berWandfläche());
    System.out.println(""+z2.getBezeichnung()+" Grundfläche: "+z2.berGrundfläche());
    System.out.println(""+z2.getBezeichnung()+" Wandfläche: "+z2.berWandfläche());
    System.out.println(""+z3.getBezeichnung()+" Grundfläche: "+z3.berGrundfläche());
    System.out.println(""+z3.getBezeichnung()+" Wandfläche: "+z3.berWandfläche());
  }
}
```

erzeugt folgende Ausgabe:

```
Wohnzimmer Grundfläche: 20.0
Wohnzimmer Wandfläche: 36.800000000000004
Schlafzimmer Grundfläche: 16.0
Schlafzimmer Wandfläche: 34.8
Küche Grundfläche: 9.0
Küche Wandfläche: 25.999999999999996
```

(Um die unregelmäßige Anzahl Nachkommastellen zu vermeiden, müssten wir eine Ausgabeformatierung vornehmen, wie in Kapitel 16 des Teils zur strukturierten Programmierung beschrieben).

3.6 Mehrere Objekte einer Klasse in einem Array verwalten

Normalerweise existiert von einer Klasse eine Vielzahl von Objekten. So wie eine Wohnung mehrere Zimmer hat, so hat eine Klasse Kunden möglicherweise mehrere Hundert Objekte. Eine Auswertung aller Objekte in der obigen Form ist nun nicht mehr zu leisten. Wir können nicht Hunderte von System.out-Anweisungen schreiben, um alle Attributwerte auszugeben. Dieses wird durch Wiederholungskonstrukte, wie for-, while- oder do while-Schleifen erledigt. Dazu ist es aber nötig, dass man die einzelnen Objekte der Klasse über einen **Index** ansprechen kann.
Die einfachste Lösung ist es, die Objekte in einem Array zu speichern.

Beispiel: Wir wollen alle Objekte der Klasse Zimmer in einem Array speichern:

1. Wir erzeugen ein Array vom Datentyp der Klasse, die wir verarbeiten wollen

```
Zimmer [] z = new Zimmer[5];
```
Dieses Array kann 5 Zimmerobjekte aufnehmen.

2. Wir erzeugen Objekte und speichern diese im Array

```
z[0] = new Zimmer("Wohnzimmer",5,4,2.40,3,2);
z[1] = new Zimmer("Schlafzimmer",4,4,2.40,2,1);   usw.
```

3. Die Verarbeitung, Auswertung usw. geschieht über eine Schleife.

In Kapitel 5.2 werden wir noch sehen, dass es auch noch komfortablere und flexiblere Methoden gibt, eine große Zahl von Objekten zu verwalten.

Wenn die Objekte einer Klasse dauerhaft gespeichert werden sollen, müssen die Daten in einer Datenbank abgelegt werden. Eine Darstellung zur Speicherung in einer Datenbank liefert das Kapitel 12 (Dateiverarbeitung) des vorhergehenden Skripts.

3.7 Übungen

1. Benötigt wird eine Klasse **Kunde** mit den Attributen **kundennummer**(int), **anrede**(String) und **name**(String).
   Schreiben Sie diese Klasse mit get- und set-Methoden, einem Konstruktor und einer to-String-Methode.
   Erzeugen Sie zwei Kundenobjekte in einer Applikation, initialisieren Sie diese mit dem Konstruktor und lassen alle Werte über die to-String-Methode wieder ausgeben.
   Verändern Sie einige Werte (set-Methode) und lassen sie sich die veränderten Werte wieder anzeigen (get-Methode).

2. Benötigt wird eine Klasse **Artikel** mit den Attributen **artnr(**int), **artbez**(String), **bestand**(int) und **preis**(double).
   Schreiben Sie diese Klasse mit get- und set-Methoden, einem Konstruktor und einer to-String-Methode.
   Speichern Sie alle erzeugten Objekte in einem Array und lassen Sie alle Werte über die to-String-Methode wieder ausgeben. Berechnen Sie den Gesamtwert der Artikel mit einem Wiederholungskonstrukt

4. Realisierung einer Anwendung mit mehreren Klassen (Fallbeispiel 2)

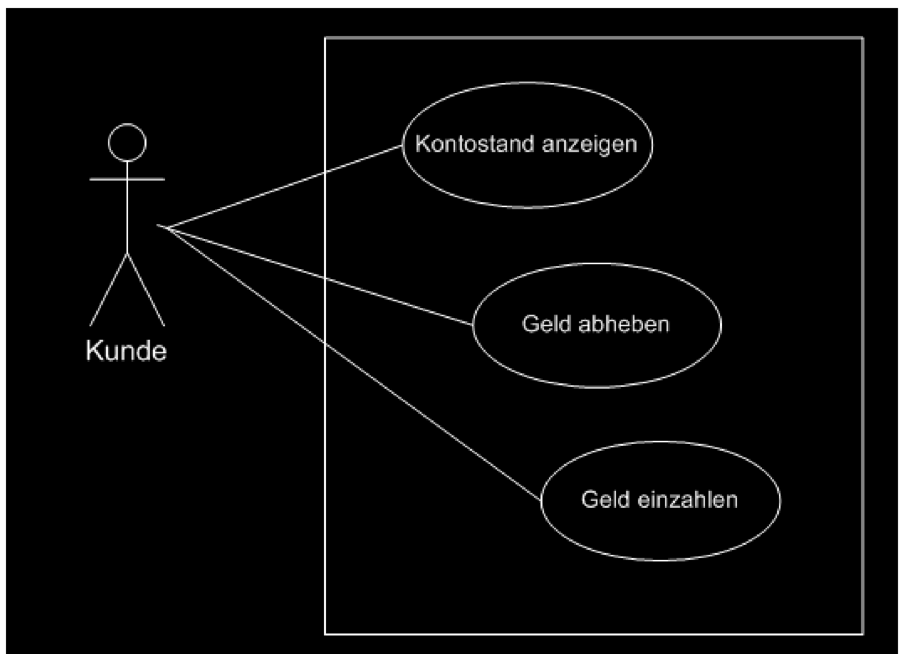
In praktischen Anwendungsfällen der OOP hat man es normalerweise nicht nur mit einer einzigen Klasse zu tun, sondern mit mehreren Klassen zwischen denen Beziehungen bestehen.
In der folgenden Situation realisieren wir einen Anwendungsfall, der aus mehreren Klassen besteht. Wir wollen uns an die Vorgehensweise in der objektorientierten Softwareentwicklung halten und die Phasen **Analyse – Design – Codierung** durchlaufen.

4.1 Beschreibung der Fallsituation

Fallsituation:
Eine Bank hat Kunden und Konten. Jeder Kunde hat genau ein Konto. Die Kunden wollen ihren Kontostand abfragen, Geld abheben und Geld einzahlen können.

4.2 Analysephase

Die zu realisierenden Anwendungsfälle sind: ***Kontostand abfragen***, ***Geld abheben*** und ***Geld einzahlen***.
Wir stellen dies in einem **Use-case-Diagramm** dar:



4.3 Die Designphase

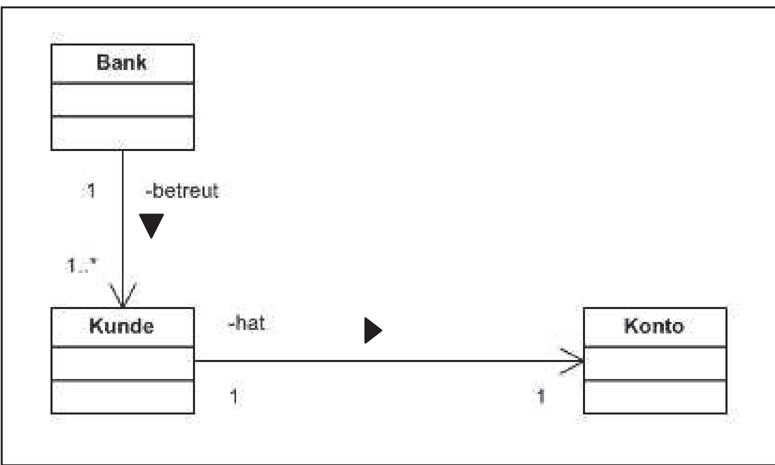
Anhand der kurzen Fallbeschreibung identifizieren wir folgende Klassen: **Bank**, **Kunde** und **Konto**.
Wir stellen dies in einem **Klassendiagramm** dar:

Zwischen den genannten Klassen **Bank**, **Kunde** und **Konto** bestehen logische und mengenmäßige Beziehungen:

Assoziationen: Bezeichnung der Beziehung und der Beziehungsrichtung: z.B. Bank betreut Kunden oder Kunde hat Konto

Kardinalität: Mengenmäßige Beziehung zwischen Klassen: z.B. Eine Bank betreut eine unbekannte Anzahl (aber mindestens 1) Kunden.

Daraus ergibt sich folgendes Klassendiagramm:



Zur besseren Übersichtlichkeit wurden hier keine Attribute und Methoden in der Klasse dargestellt.

Die Pfeilspitze der Assoziationslinien gibt die **Richtung** der Assoziation an. In diesem Fall spricht man von einer **gerichteten** Assoziation. Wenn die Assoziationslinie keinen Pfeil hat, ist die Assoziation **ungerichtet**, d.h. die Assoziationsrichtung geht in beide Richtungen. Die Art der Assoziation wird durch den **Assoziationsnamen** ausgedrückt (betreut, hat).
(In der korrekten UML-Syntax sollte die Leserichtung durch eine ausgefüllte Pfeilspitze hinter dem Assoziationsnamen dargestellt werden.Oft findet man die Pfeilspitze aber auch am Ende der Linie.)
Für die Klasse Kunde werden die beiden Attribute **kdnr** und **name** benötigt. Für die Klasse Konto die Attribute **ktonr**, **ktoStand** und **pin**.

Die Methode **kontostandAbfragen** kann durch eine einfache get-Methode realisiert werden.
Bei den Methoden **einzahlen** und **auszahlen** wird jeweils der aktuelle Wert des Attributes **ktoStand** erhöht bzw. vermindert. Evtl. ist darauf zu achten, dass durch die Auszahlung ein bestimmtes Dispositionslimit nicht überschritten werden darf. Dieses wollen wir aber zunächst außer acht lassen.

4.4 Die Codierungsphase

Hier die Klassen **Kunde** und **Konto.** Zur Vereinfachung verzichten wir auf die Klasse Bank

```
public class Kunde {
  // Anfang Variablen

   private int kdnr;
   private String name;
   private Konto meinKonto;   //Verweisattribut

  // Ende Variablen

   public Kunde(int k, String n)
   {
     this.kdnr=k;
     this.name=n;
   }

  // Anfang Methoden
  public void setKdnr(int k)
  {
    this.kdnr=k;
  }
  public int getKdnr() {
    return kdnr;
  }
  public void setName(String n)
  {
    this.name=n;
  }
  public String getName() {
    return name;
  }
  public Konto getMeinKonto() {
    return meinKonto;
  }
  public void setMeinKonto(Konto meinKonto) {
    this.meinKonto = meinKonto;
  }
   // Ende Methoden
}
```

```
public class Konto {

  // Anfang Variablen
  private int ktonr;
  private double ktoStand;

  public Konto(int k, double s)
  {
    this.ktonr=k;
    this.ktoStand=s;
  }
  public double getKtoStand() {
    return ktoStand;
  }

  public void setKtoStand(double ktoStand) {
    this.ktoStand = ktoStand;
  }

  public int getKtonr() {
    return ktonr;
  }

  public void setKtonr(int ktonr) {
    this.ktonr = ktonr;
  }
  public void auszahlen(double b)
  {
    this.ktoStand=this.ktoStand-b;
  }
  public void einzahlen(double b)
  {
    this.ktoStand=this.ktoStand+b;
  }
  // Ende Methoden
}
```

Im Klassendiagramm stehen diese beiden Klassen zunächst noch ohne Beziehung nebeneinander.

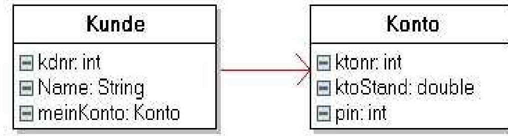
4,4.1 Assoziationen zwischen Klassen herstellen

Die Assoziation, dass ein Kunde ein Konto hat wird durch Einfügen eines **Verweisattributes** in der Klasse Kunde realisiert. Diese Attribut muss auf das dem Kunden zugehörige Kontoobjekt zeigen und muss daher den Datentyp **Konto** haben. Dieses Attribut kann folgendermaßen in der Klasse **Kunde** codiert werden:

```
private Konto meinKonto;
```

(Die Attributsbezeichnung meinKonto ist beliebig)

Jetzt wird die Assoziation im Klassendiagramm sichtbar.



In der Applikation wird das Attribut **meinKonto** erst dann initialisiert, wenn das dazu gehörige Kontoobjekt exisitert.

```
public class bank
{
  public static void main(String argv[])
  {
     Kunde k1=new Kunde(123,"Meier");
     Konto kt1 = new Konto(4711,3500.00,111);
     k1.setMeinKonto(kt1);
  }
}
```

Es wird hier ein Konstruktor verwendet, um Objekte zu erzeugen. Die Anweisung **k1.setMeinKonto(kt1);** weist dem Kunden sein Konto zu und erzeugt damit die Assoziation.
Für das Verweisattribut muss es eine get- und set-Methode geben.

4.4.2 Zugriff auf Klassenmethoden über das Verweisattribut:

Um die genannten Anwendungsfälle **kontoStandAbfrage**, **einzahlen** und **auszahlen** zu realisieren müssen wir folgende Überlegungen anstellen: Auslöser für einen dieser Anwendungsfälle ist jeweils der Kunde (siehe use-case-Diagramm). D.h. die Klasse Konto kann nicht selbst einzahlen oder auszahlen.
Dies bedeutet, dass der Kunde über das Verweisattribut auf sein Konto zugreifen muss. Da das Verweisattribut gekapselt ist, geschieht dies über die get-Methode.

Mit dem Verweisattribut kann nun über den Kunden direkt auf sein Konto zugegriffen werden, ohne die Klasse Konto direkt anzusprechen. Wir wollen den Kontostand des Kunden Meier am Bildschirm anzeigen:

```
System.out.println("Kunde "+k1.getName( )+" Kontostand: "+k1.getMeinKonto( ).getKtoStand( ) );
```

Im zweiten Aufruf benötigen wir zwei Punkte, da wir dem Methodenaufruf zwei Objekte voranstellen:

- **k1** ist das Kundenobjekt
- **getMeinKonto()** ist die Methode, die das dem Kunden zugehörige Kontoobjekt aufruft.
- **getKtoStand()** ist eine Methode der Klasse Konto.

4.4.2 Die Applikation

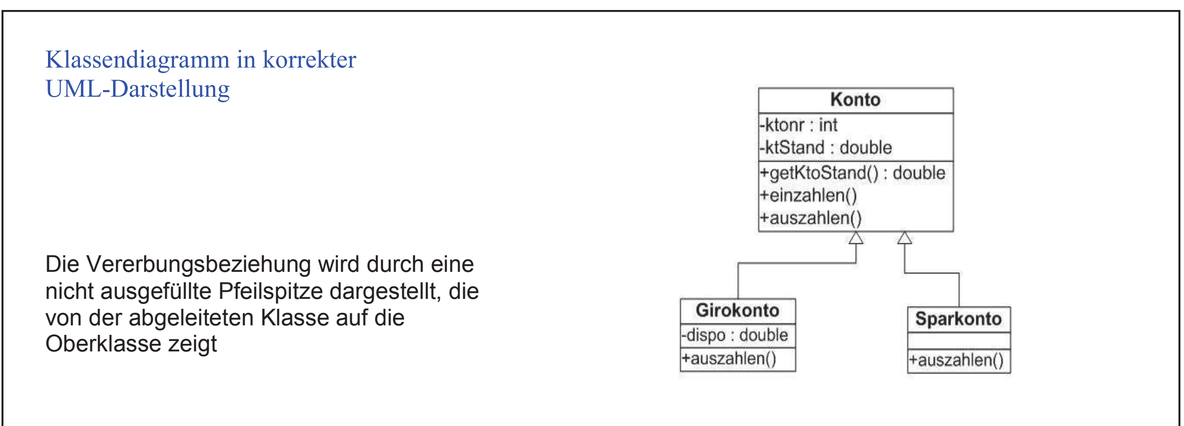
Die drei Anwendungsfälle werden nachfolgend über eine kleine Menüstruktur, die über ein switch-case-Konstrukt gelöst wird realisiert. Selbstverständlich ließen sich auch die Ein- und Auszahlungen über eine Tastatureingabe komfortabler regeln:

```
import java.io.*;
public class bank
{
 public static void main(String argv[]) throws IOException
 {
   Kunde k1=new Kunde(123,"Meier");
   Konto kt1 = new Konto(4711,3500.00);
   k1.setMeinKonto(kt1);
   int eingabe;
   do
   {
    System.out.println("\n1 - Kontostand anzeigen");
    System.out.println("2 - Einzahlen");
    System.out.println("3 - Auszahlen");
    System.out.println("0 - Beenden\n");
    eingabe=Console_IO.IO_int("Eingabe: ");
    switch(eingabe)
    {
      case 1: System.out.println("Kunde "+k1.getName()+" Kontostand: "+k1.getMeinKonto().getKtoStand());
              break;
      case 2: k1.getMeinKonto().einzahlen(150); break;
      case 3: k1.getMeinKonto().auszahlen(900.95); break;
      case 0: break;
    }
   }
   while(eingabe!=0);
 }
}
```

4.5 Erweiterung der Fallsituation - Vererbung

Wir verändern unser Fallbeispiel derart, dass wir jetzt zwischen Girokonten und Sparkonten unterscheiden wollen. Jeder Kunde kann ein **Girokonto** und ein **Sparkonto** haben. Girokonten und Sparkonten haben nun aber mehr Gemeinsamkeiten als Unterschiede. So haben beide eine Kontonummer und einen Kontostand. Der Unterschied könnte darin bestehen, dass das Girokonto einen Dispositionskredit aufweist und das Sparkonto eine höhere Guthabenverzinsung.

Hier bietet sich das Prinzip der **Vererbung** an. Bei der Vererbung erbt eine abgeleitete Klasse (Unterklasse) sämtliche Eigenschaften einer **Oberklasse** und kann zusätzlich eigene Merkmale aufweisen.
Vererbung wird in der UML durch eine nicht ausgefüllte Pfeilspitze dargestellt. Die folgende Abbildung zeigt die korrekte Darstellung für das Klassendiagramm:



Innerhalb einer Klassenhierarchie kann es gleiche Methoden mit gleichem Aufbau geben. Die Methode **auszahlen** erscheint hier in allen drei Klassen. Es wird jeweils die Methode aufgerufen, die zu der Klasse gehört, dessen Objekt diese Methode aufruft.
Man bezeichnet, die Möglichkeit, gleiche Methodenbezeichnungen innerhalb einer Vererbungshierarchie zu verwenden als **Polymorphie**.

Methoden mit gleicher Bezeichnung können andere Methoden **überladen** oder **überschreiben**. Als **Überladen** bezeichnet man es, wenn Methoden gleichen Namens eine unterschiedliche Parameterliste haben. Der Compiler erkennt dann anhand der Parameter, welche Methode angesprochen werden soll. Als **Überschreiben** bezeichnet man es, wenn aus einer Klassenhierarchie ersichtlich ist, welche Methode angesprochen wird, obwohl sowohl Methodenbezeichnung als auch Parameterliste identisch sind.

Der Vorteil des Überschreibens von Methoden liegt darin, dass man keine unterschiedlichen Bezeichnungen für gleiche Funktionalitäten benötigt. Die Klasse **Girokonto** ist von der Basisklasse **Konto** abgleitet. Beide Klassen haben eine Methode **auszahlen**, die aber in der Subklasse Girokonto eine andere Funktionalität besitzt, wie in der Klasse Konto.

4.5.1 Realisierung der Vererbungsbeziehung

Eine Vererbungsbeziehung wird im Kopf der Klasse durch das Schlüsselwort **extends** angezeigt.

```
public class Girokonto extends Konto {

  // Anfang Variablen
  private double dispo;
```

4.5.2 Zugriff auf Attribute der Basisklasse

Attribute der Basisklasse, die mit private deklariert wurden sind in den abgeleiteten Klassen nur über ihre get- und set-Methoden ansprechbar. Dieses ist innerhalb einer Klassenhierarchie nicht unbedingt sinnvoll, da es häufig vorkommt, dass Methoden der abgeleiteten Klassen Attribute der Basisklasse verwenden. Aus diesem Grund ist es sinnvoll, die Attribute in den Basisklassen mit dem Schlüsselwort **protected** zu deklarieren. Das Symbol dafür ist das hash-Zeichen (#). Damit sind die Attribute innerhalb der Klassenhierarchie verfügbar.

4.5.3 Konstruktoren und Vererbung

Ein Konstruktor einer Basisklasse kann nicht vererbt werden. Die abgeleitete Klasse muss einen eigenen Konstruktor implementieren. Attribute der Basisklasse, die im Konstruktor der abgeleiteten Klasse verwendet werden, müssen in der Basisklasse als **protected** deklariert sein.
Wenn die abgeleitete Klasse über einen eigenen Konstruktor verfügt, muss in der Basisklasse ein Standardkonstruktor vorhanden sein. Ohne diesen Standardkonstruktor kann kein Objekt der abgeleiteten Klasse erzeugt werden.
Hat die Basisklasse einen eigenen Konstruktor implementiert (wie im Quellcodeauszug unten), so wird dadurch der Standardkonstruktor überschrieben. Dieser muss dann explizit wieder definiert werden.

4.5.4 Arbeiten mit super( )

Mit dem Aufruf von **super( )** kann man von einer abgeleiteten Klasse aus auf eine überschriebene Methode der Basisklasse zugreifen. Dieses wird sehr häufig beim der Verwendung von Konstruktoren verwendet. Wenn ein neues Objekt einer abgeleiteten Klasse mit einem Konstruktor erzeugt werden soll, so muss der Konstruktor in der abgeleiteten Klasse komplett neu geschrieben werden. Da er aber meist so ähnlich aufgebaut ist, wie der Konstruktor der Basisklasse, wäre es sinnvoll, wenn der Konstruktor der Basisklasse mit verwendet werden könnte. Dies ist mit dem Aufruf von **super( )** möglich.

Basisklasse

```
public class Konto {

  protected int ktnr;
  protected double ktStand;

  public Konto()   { }

  public Konto(int n, double b)
  {
    this.ktnr=n;
    this.ktStand=b;
  }
}
```

abgeleitete Klasse

```
public class Girokonto extends Konto {

  private double dispo;

  public Girokonto(int a, double b, double d)
  {
    super(a,b);           //Aufruf Konstruktor
    this.dispo=d;
  }
```

4.6 Übungen

1. Eine Schule besteht aus Schülern und Lehrern. Fassen Sie wichtige Eigenschaften in einer Oberklasse Person zusammen und bilden Sie durch Vererbung zwei Unterklassen Schüler und Lehrer. Von den Schülern werden wieder zwei Klassen Vollzeitschüler und Teilzeitschüler abgeleitet. Zeichnen Sie ein Klassendiagramm

2. Eine Schule beschäftigt genau 25 Lehrer. Jeder Lehrer unterrichtet in bis zu 5 Klassen. In einer Klasse sind maximal 25 Schüler. Erstellen Sie ein Klassendiagramm und bilden Sie geeignete Assoziationen und Kardinalitäten für die genannten Klassen jeweils in zwei Richtungen.

4. Eine Automobilfirma hat zwei Werke. In jedem Werk wird ein bestimmter Fahrzeugtyp hergestellt. Für die Fahrzeuge werden die Attribute *Produktname*, *Preis* und *Verkaufsmenge* erfasst. Für die Werke werden die Attribute *Standort* und *Werksumsatz* sowie die Methode *Werksumsatz_ermitteln* erfasst. Die Firma hat einen *Namen* und verfügt über die Methode *Gesamtumsatz_ ermitteln*.
   a) Erstellen Sie ein Klassendiagramm mit den Assoziationen.
   b) Der Firmeninhaber möchte den Gesamtumsatz ermitteln. Erstellen Sie ein Interaktionsdiagramm für diese Anwendung.

5. Ein Zug besteht aus Waggons. Jeder Waggon hat genau sechs Abteile. In einem Abteil können bis zu sechs Fahrgäste sitzen.

   Stellen Sie in einem Klassendiagramm die Assoziationen und die Kardinalitäten dar.

6. Kursverwaltung

Ein Fortbildungsinstitut möchte seine Software zur Kursverwaltung auf die objektorientierte Programmierung umstellen. Zu diesem Zweck soll zunächst folgender Sachverhalt als Klassendiagramm modelliert werden:

Für die Teilnehmer eines Kurses werden der Name, die Anschrift und der Status (beschäftigt, Schüler/Student bzw. arbeitslos) gespeichert. Jeder Teilnehmer kann sich für ein oder mehrere Kurse anmelden. Für jeden Kurs werden dessen Nummer, die Bezeichnung, das Datum sowie die Kursgebühr gespeichert. An einem Kurs können nicht mehr als 20 Teilnehmer teilnehmen.
Jeder Kurs wird von einem Kursleiter angeboten. Ein Kursleiter kann mehrere Kurse anbieten. Für den Kursleiter werden Name und Firma gespeichert. Ein Teilnehmer kann nicht gleichzeitig Kursleiter sein.
Jeder Teilnehmer hat genau ein Konto. Im Konto werden Kontonummer, und die bezahlte Kursgebühr gespeichert.

Erstellen Sie das Klassendiagramm mit den Klassenbezeichnungen, den Attributen und den Assoziationen Get- und set-Methoden für die Attribute müssen nicht dargestellt werden.

5. Arrays und Collections als Container

5.1 Verweisattribute als Array

Wenn ein Kunde mehrere Konten haben kann, dann gibt es mehrere Verweisattribute, die in einem Array gespeichert werden können.

Die Syntax für das Anlegen dieses Arrays lautet für unser Beispiel:

```
public class Kunde
 {
   private int kdnr;
   private String Name;
   private Konto meineKonten[ ];

   public Kunde(int k, String n)                    //Konstruktor
   {
      this.kdnr=k;
      this.Name=n;
      meineKonten = new Konto[3];     //damit können Verweise auf 3 Konten erzeugt werden.
   }
  public void setMeineKonten(Konto k,int i)
   {
     this.meineKonten[i]=k;
   }
   public Konto getMeineKonten(int i)
   {
     return this.meineKonten[i];
   }
}
```

In der Applikation sieht das so aus:

```
public class bank
{
  public static void main(String argv[])
  {
     Kunde k1=new Kunde(123,"Meier");
     Konto kt1 = new Konto(4711,3500.00,111);
     Konto kt2 = new Konto(815,500.00,222);
      k1.setMeinKonto(kt1);
      k1.setMeineKonten(kt1,0);
      k2.setMeineKonten(kt2,1);
      //Ausgabe von Kontendaten über den Kunden
System.out.println("Kunde "+k1.getName( )+" Kontostand: "k1.getMeineKonten(0).getKtoStand());  }}
```

Es erscheint folgende Ausgabe:

Kunde Meier Kontostand: 3500.00

5.2 Collections am Beispiel ArrayList

Wenn größere Datenmengen des gleichen Typs benötigt werden, so wurden diese bisher in einem Array gespeichert. Auf diese Art kann man bequem auch mehrere Objekte einer Klasse verwalten. Der Nachteil der Arbeit mit Arrays ist nur, dass man schon bei der Erzeugung wissen muss, wie viele Elemente man benötigt.
Nehmen wir an, wir wollen eine Kundendatei erstellen und die Objekte der Klasse Kunden in einem Array speichern. Da sich das Unternehmen aber noch im Aufbau befindet, kann die Zahl der Kunden sehr schnell wachsen.

Zur Lösung dieses Problems bietet Java die so genannten **Collection-Klassen** an. Diese sind eine Sammlung verschiedener Klassen, die eine beliebige Anzahl von Objekten speichern können.
Wir werden hier ein Beispiel für die Klasse **ArrayList** vorstellen.

Beispiel: Es wird eine ArrayList angelegt, in der beliebig viele Objekte der Klasse Kunden gespeichert werden können. Die Klasse Kunde besteht aus den Attributen kdnr (Kundennummer) und kdName (Kundenname). Neben den get- und set-Methoden für die Attribute gibt es eine toString-Methode für die Datenausgabe sowie einen Konstruktor.

Für die Verwendung der Klasse ArrayList wird die Bibliothek **java.util** eingebunden.
Für die Arbeit mit der ArrayList stehen uns die folgenden Grundfunktionen zur Verfügung:

Anweisung	**Bedeutung**
Import java.util.ArrayList;	Importiert die Klasse ArrrayList
ArrayList kliste = new ArrayList( );	Anlegen einer neuen ArrayList mit dem Namen kliste
ArrayList<Kunde> kliste = new ArrayList<Kunde>()	Anlegen einer neuen ArrayList mit dem Namen kliste, die nur Daten vom Typ Kunde aufnehmen kann.
Kunde k = new Kunde(100, “Meier“); kliste.add(k);	Erzeugt ein Objekt der Klasse Kunde Fügt das Objekt der Liste zu.
kliste.remove(2)	Entfernt den dritten Eintrag des Arrays. Beachte, dass das Array ab dem Index 0 gezählt wird.
kliste.get(1)	Liest das Element an der Stelle 2
kliste.size( )	Liefert die Anzahl der Elemente des Arrays
kliste.contains(“Zeichenkette“)	Es wird geprüft, ob eine bestimmte Zeichenkette in der Liste enthalten ist.

Das folgende Beispiel zeigt die Verwendung einer ArrayList. Zunächst wird die Klasse Kunde dargestellt, anschließend werden in einer Applikation einige der obigen Anweisungen angewendet.

```
public class Kunde {

  private int kdnr;
  private String name;

  public Kunde(int p,String n)
  {
    this.kdnr=p;
    this.name=n;
  }
  public int getKdnr() {
    return kdnr;
  }
public void setKdnr(int kdn) {
    this.kdnr = kdn;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String toString()
  {
    String ausgabe=""+this.kdsnr+" "+this.name;
    return ausgabe;
  }
}
```

```
1  import java.util.ArrayList;

2  public class Kundenverwaltung {

3  public static void main(String[] args)
4  {
5  ArrayList<Kunde> kListe = new ArrayList<Kunde>();   //generische Liste nur für
                                                        //Kundenobjekte
6  Kunde k1 = new Kunde(100,"Meier");
7  kliste.add(k1);
8  Kunde k2 = new Kunde(101,"Berger");
9  kliste.add(k2);
10 System.out.println("Größe der Liste: "+kliste.size());
11 System.out.println("Eintrag an der Stelle 2: "+kliste.get(1));
12 System.out.println("Ein Attribut ausgeben: "+kliste.get(0).getName());
13 //Ein Attribut ändern
14 kliste.get(1).setName("Neu");
15 //Alle Werte mit for ausgeben
16 for(int k=0; k<kliste.size(); k++)
17 {
18   System.out.println(kliste.get(k));
19 }
20 //noch mehr Objekte mit for erzeugen  und alle wieder ausgeben
21 for(int m=2; m<5; m++)
22 {
23   Kunde k = new Kunde(199,"Test");
24   kliste.add(k);
25 }
26 for(int k=0; k < kliste.size(); k++)
27 {
28   System.out.println(kliste.get(k));
29 }
30 // Objekt löschen und alles wieder ausgeben
31 kliste.remove(3);
32 System.out.println("Element 4 gelöscht");
33 for(int k=0; k<kliste.size(); k++)
34 {
35   System.out.println(kliste.get(k));
36 }
37 }
38 }
```

Erläuterungen:
Zeile 1: Import für Java-Collections-Klasse ArrayList
Zeile 5: Erzeugen einer ArrayList mit dem Namen kliste, die Daten vom Typ Kunde aufnimmt
Zeile 7: Der Liste wird das Objekt k1 der Klasse Kunde hinzugefügt.
Zeile 10: Die Anzahl der Elemente, die die Liste enthält wird ausgegeben.
Zeile 11: Das Element mit dem Index 1 (Position 2) wird ausgelesen.und mit der toString-Methode angezeigt.
Zeile 12: Ein bestimmtes Attribut des 1. Elementes der Liste wird angezeigt.
Zeile 31: Das Element mit dem Index 3 (Position 4) wird gelöscht.

5.2.1 Der Iterator

Der Iterator ermöglicht es, alle Elemente einer Liste kontrolliert zu durchlaufen.
Der Iterator liefert folgende Methoden, die ebenfalls im Paket java.util verfügbar sind:

hasNext( ) liefert true, wenn noch mindestens ein Element in der Liste steht
next( ) liefert das jeweils nächste Element der Liste
remove( ) entfernt das zuletzt mit next angesprochene Element aus der Liste

In obigen Beispiel wird die Liste in den Zeilen 16 bis 19 mit einer for-Schleife abgearbeitet. Mit dem Iterator-Konzept ist der Zugriff auf eine Liste noch effizienter.
Wir können obige for-Schleife durch folgende Anweisungen ersetzen:

```
Iterator it = kliste.iterator( );          //erzeugt einen Iterator mit der Bezeichnung it für die Liste
while(it.hasNext() )                       //so lange noch weitere Elemente vorkommen
{
   System.out.println(it.next() );         //Ausgabe des zuletzt aufgerufenen Objekts
}
```

Mit it.remove( ) wird das zuletzt mit it.next( ) aufgerufene Objekt gelöscht.

6. UML-Diagramme

Bei den Diagrammarten unterscheiden wir zwischen statischen Diagrammen, die den Zustand eines Systems beschreiben und dynamischen Diagrammen, die Abläufe in ihrer logischen und zeitlichen Reihenfolge darstellen.

6.1 Statische Sicht

6.1.1 Das Anwendungsfalldiagramm (Use- case - Diagram)

Das Anwendungsfalldiagramm ist ein *Verhaltensdiagramm*. Es zeigt eine bestimmte Sicht auf das *erwartete* Verhalten eines Systems und wird deshalb für die Spezifikation der Anforderungen an ein System eingesetzt. Anwendungsfalldiagramme beschreiben die Beziehungen zwischen einer Gruppe von Anwendungsfällen und den teilnehmenden Akteuren.

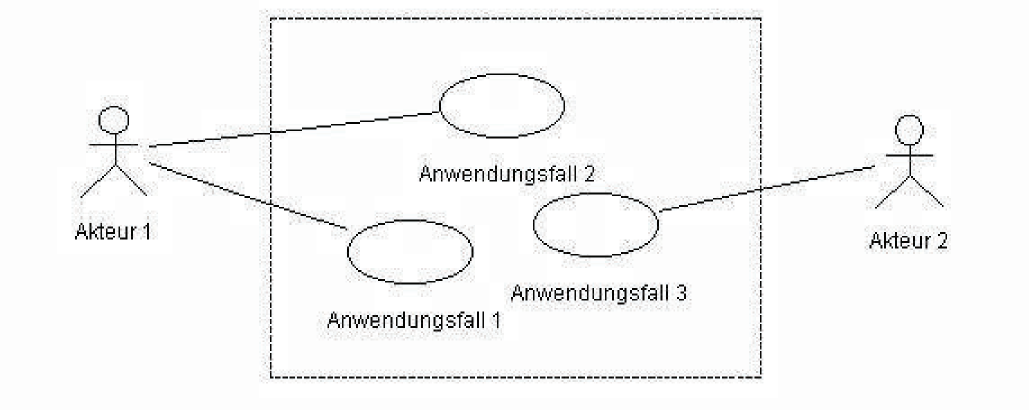
Dabei ist zu beachten, dass ein Anwendungsfalldiagramm nicht das Systemdesign widerspiegelt und damit keine Aussage über die Systeminterna trifft. Anwendungsfalldiagramme werden zur Vereinfachung der Kommunikation zwischen Entwickler und zukünftigen Nutzer bzw. Kunde erstellt. Sie sind vor allem bei der Festlegung der benötigten Kriterien des zukünftigen Systems hilfreich. Somit treffen Anwendungsfalldiagramme eine Aussage, *was* zu tun ist, aber nicht *wie* das erreicht wird.

Anwendungsfälle werden durch **Ellipsen** die den Namen des Anwendungsfalles tragen und einer Menge von beteiligten Objekten (**Akteuren**) dargestellt. Zu jedem Anwendungsfall gibt es eine Beschreibung in Textform. Die entsprechenden Anwendungsfälle und Akteure sind durch **Linie**n miteinander verbunden. Akteure können durch Strichmännchen dargestellt werden. Die **Systemgrenze** wird durch einen Rahmen um die Anwendungsfälle symbolisiert. **Include-Beziehungen** bestehen zwischen Anwendungsfällen, die einen anderen Anwendungsfall beinhalten. Diese werden durch gestrichelte Linien dargestellt.

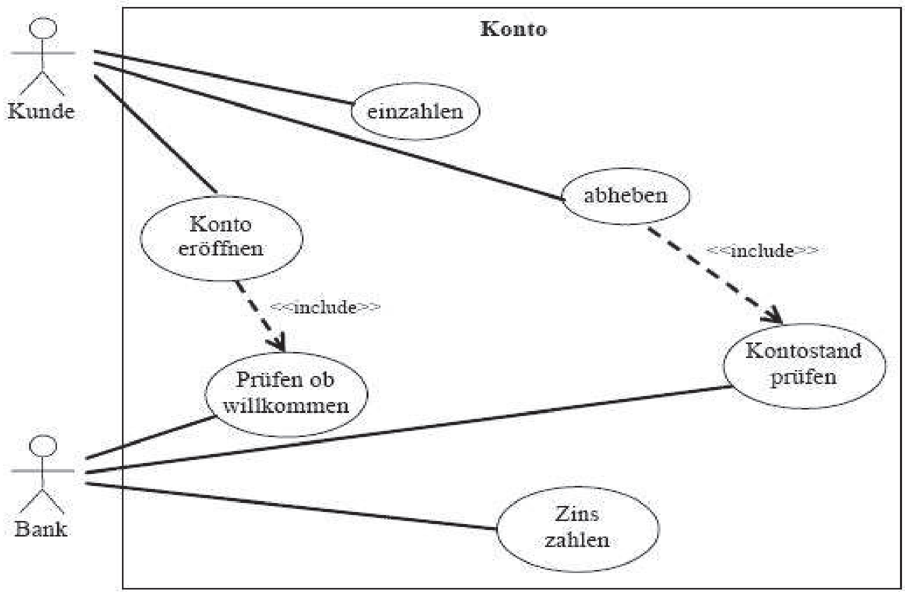
Das use case Diagramm beinhaltet folgende Elemente:

System	Der Rahmen stellt das Anwendungssystem dar, innerhalb dessen der Anwendungsfall realisiert wird.
Anwendungsfall1	Ein Anwendungsfall ist ein Teilproblem, welches innerhalb eines Anwendungssystems gelöst werden soll. Der Anwendungsfall wird immer von einem oder mehreren Akteuren oder anderen Anwendungsfällen ausgelöst
Akteur1	Der Akteur kann eine Person oder ein anderes Anwendungssystem sein, welches einen Anwendungsfall auslöst
——————▶	Die Assoziation besteht verbindet Akteure und Anwendungsfälle
- - - - - - - - ▶	include – oder extend-Beziehung. Eine include-Beziehung besteht zwischen einem Anwendungsfall, der einen anderen Anwendungsfall beinhaltet. Eine extends-Beziehung besteht, wenn eine Anwendungsfall durch einen anderen Anwendungsfall erweitert wird, wenn eine bestimmte Bedingung eintritt.

Allgemeines Beispiel

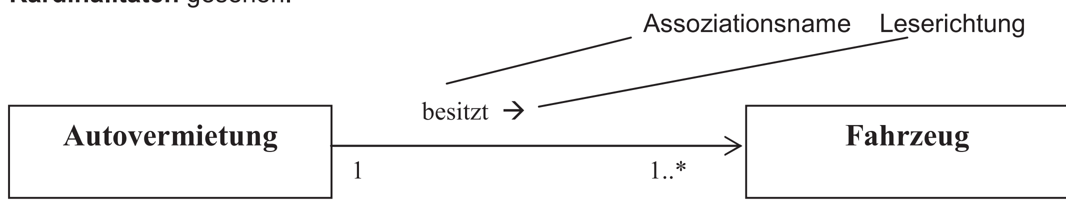


Beispiel für Anwendungsfälle in einer Bank



6.1.2 Klassendiagramm und Erweiterungen

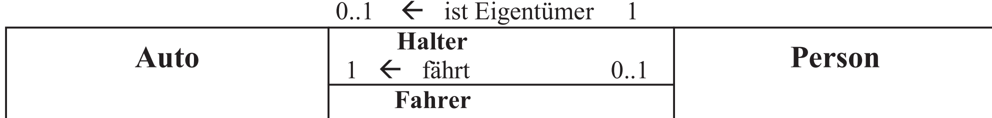
Im Kapitel 4.3 haben wir schon den Aufbau eines Klassendiagramms mit **Assoziationen** und **Kardinalitäten** gesehen.



Kardinalität sagt aus, wie viele Objekte einer Klasse existieren können.

Kardinalität	Bedeutung
1	Genau 1
0...1	1 oder kein
*	Beliebig viele
1..*	1 bis beliebig viele
5..14	5 bis 14
6,7	6 oder 7

Rollenname: Die Beziehungen zwischen zwei Klassen können unterschiedlich sein

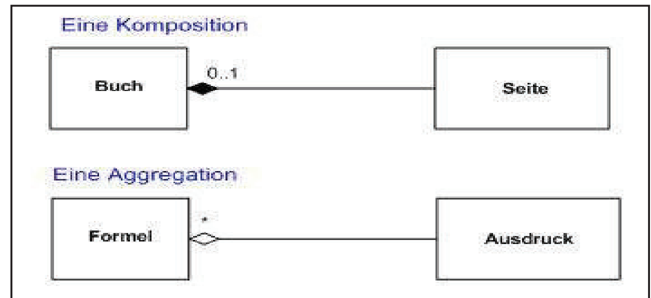


Neben der Vererbungsbeziehung, die durch die nicht ausgefüllten Pfeilspitzen dargestellt wird (siehe Kapitel 4.5), gibt es noch die Beziehung **Aggregation und Komposition**

Eine Aggregation beschreibt eine Ganzes – Teil Beziehung, wobei die Teile auch eigenständig existieren können.
Die Aggregation wird durch einen Pfeil mit einer nicht ausgefüllten Raute dargestellt.

Eine Komposition ist eine strenge Ganzes – Teil Beziehung. Die Lebensdauer des Teilobjektes richtet sich nach der Lebensdauer des Aggregationsobjektes

Die Komposition wird durch einen Pfeil mit einer ausgefüllten Raute dargestellt.



weitere Beispiele:

Aggregation: Kurs ◇——— Teilnehmer — Teilnehmer sind zwar Teil eines Kurses, können aber auch unabhängig vom Kurs existieren.

Komposition: Firma ◆——— Abteilung — Die Lebensdauer der Abteilung ist abhängig von der Lebensdauer der Firma

6.1.3 Objektdiagramme

Das Objektdiagramm dient dazu, konkrete Klassenausprägungen (Objekte oder Instanzen) darzustellen. Dabei werden die Objektbezeichnung und die Attributwerte genannt. Methoden werden nicht dargestellt. Das Objektdiagramm dient eher dazu, anhand eines Objektes beispielhaft dessen Aufbau konkret darzustellen.

In der ersten Zeile steht die Objektbezeichnung gefolgt von der Klassenbezeichnung **fett** und **unterstrichen**
In der zweiten Zeile folgen die Attributbezeichnungen und die konkreten Ausprägungen der Attribute

Objektdiagramm

meinAuto : Auto	
kennzeichen	GI-AB 123
marke	Opel
kilometerstand	25600
tankinhalt	30

6.2 Dynamischer Blick auf einen Anwendungsfall

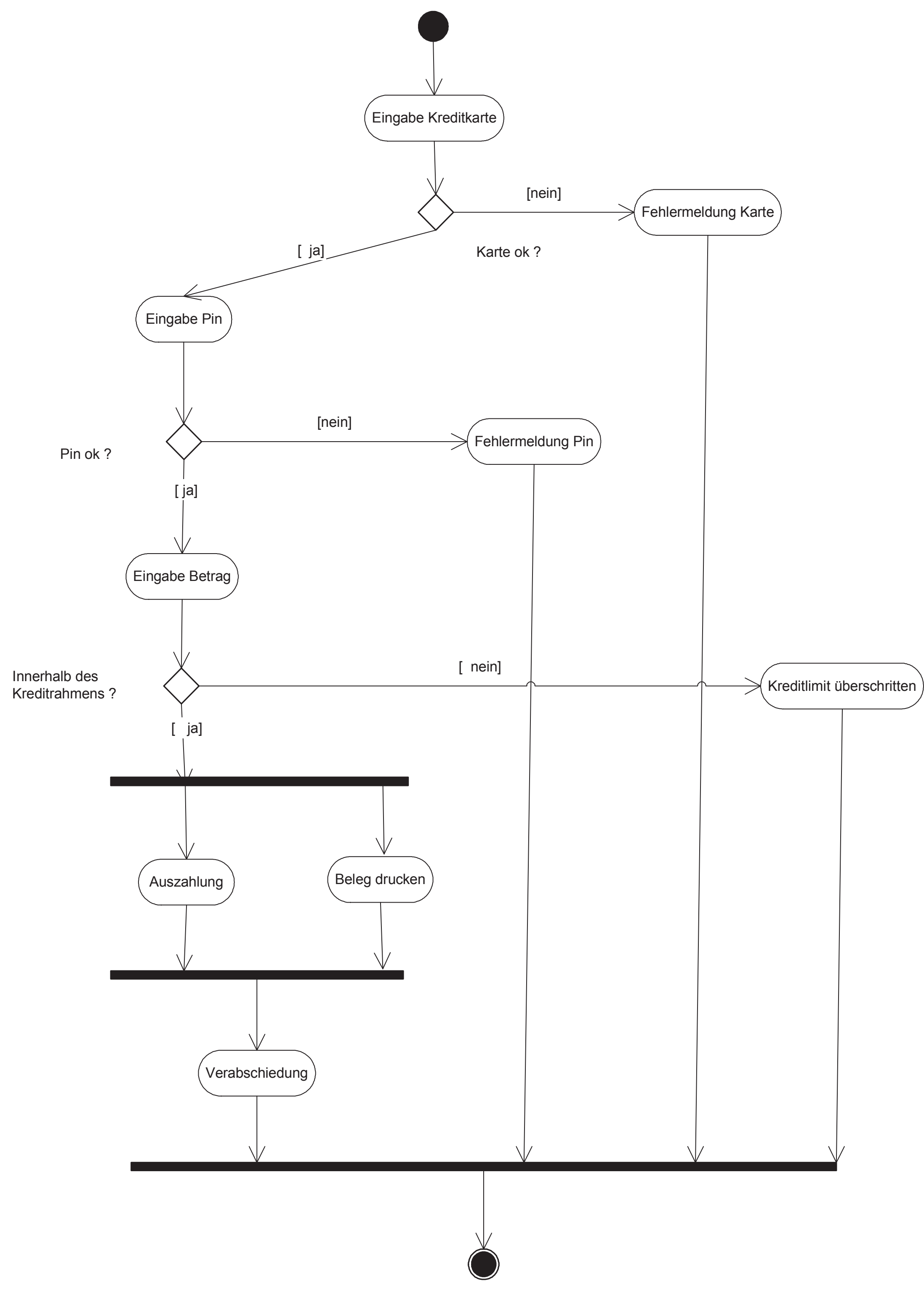
Use-case-Diagramme, Klassendiagramme und Objektdiagramme beschreiben den Zustand (**statische Sicht**) eines Systems. Der eigentliche Ablauf in zeitlicher und logischer Sicht ist nicht erkennbar.
Aus der strukturierten Programmierung kennen wir das **Struktogramm** als eine Darstellungsform, welche uns den logischen (und zeitlichen) Ablauf einer Anwendung vor Augen führt. Solche Darstellungen stellen die **dynamische Sicht** auf ein System dar.
In der UML gibt es verschiedene Diagrammarten für diese Sichtweise. Hier sollen das **Aktivitätsdiagramm** und das **Interaktionsdiagramm** vorgestellt werden.

6.2.1 Das Aktivitätsdiagramm

Das Aktivitätsdiagramm stellt den Ablauf des zu entwickelnden Programms dar. Es entspricht etwa der Programmdarstellung durch ein Struktogramm oder einen Programmablaufplan in der strukturierten Programmierung. Ein Aktivitätsdiagramm enthält sechs unterschiedliche Elemente:

	Startpunkt einer Aktivität
	Endpunkt einer Aktivität
Aktivität	Beschreibt die Aktion, die vom Programm auszuführen ist
	Entscheidung. In Abhängigkeit von einer Bedingung können verschiedene Wege beschritten werden
	Gabelung. Eine Aktivität gabelt sich in zwei weitere Aktivitäten auf.
	Zusammenführung. Zwei Aktivitäten werden zu einer Aktivität zusammengeführt.

Beispiel: Ein Bankkunde möchte an einem Geldautomat Geld abheben. Zunächst wird die Gültigkeit seiner Kreditkarte geprüft. Danach gibt er seine Pin ein. Ist die Pin gültig, kann er einen Betrag eingeben. Falls sein Kontostand innerhalb des Kreditrahmens liegt, wird der Betrag ausgezahlt und gleichzeitig ein Beleg gedruckt, andernfalls wird die Auszahlung verweigert.

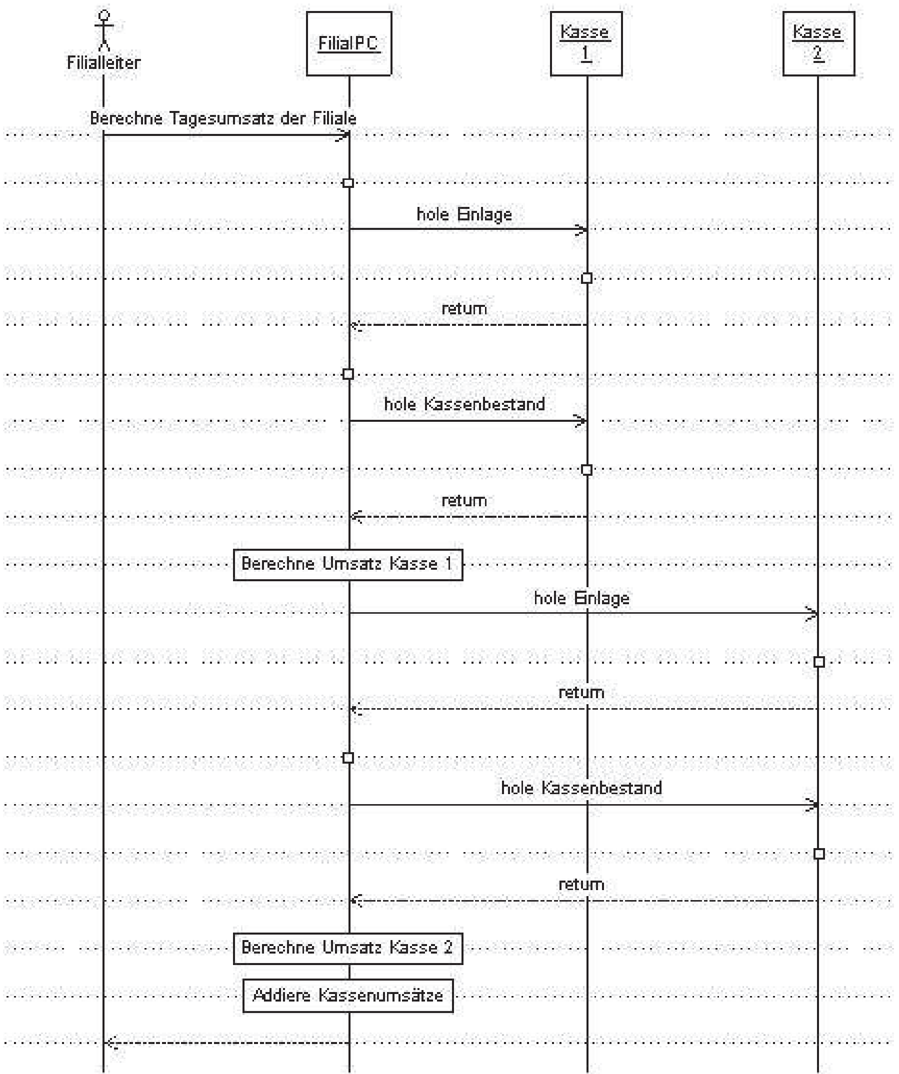


6.2.2 Das Interaktionsdiagramm (auch: Sequenzdiagramm)

Im Sequenzdiagramm wird der Nachrichtenaustausch (**Botschaften**) zwischen den Objekten in zeitlicher Reihenfolge dargestellt. In den Spalten werden die angesprochenen Objekte dargestellt, in den Zeilen die Reihenfolge der Aktivitäten, die sich hier als Nachricht darstellen. Rechtecke bezeichnen Rechenzeiten des Computers und können näher erläutert werden. Im Gegensatz zum Aktivitätsdiagramm bezieht das Interaktionsdiagramm die beteiligten Objekte mit in das Diagramm ein. Ein Sequenzdiagramm besteht aus folgenden Grundelementen:

Element	Beschreibung
Auslösendes Objekt	Das auslösende Objekt, der Akteur, der den Anwendungsfall auslöst
Objekt1	Ein beteiligtes Objekt, welches eine Botschaft übermittelt oder eine Botschaft erhält. Die gestrichelte Linie ist die Lebenslinie des Objekts.
Botschaft	Eine Botschaft, die mittels eines Methodenaufrufes übermittelt wird. Es kann die Methodenbezeichnung sowie die Übergabeparameter genannt werden.
Rückgabebotschaft	Der Antwort auf eine Botschaft. Die Antwort kann auch als Text auf der Linie stehen.
	Stellt den Zeitraum dar, der zwischen der Botschaft und der Rückmeldung liegt. Symbolisiert die Rechnerzeit in der die Daten verarbeitet werden. Innerhalb des Kästchens kann auch eine kurze Beschreibung des Vorgangs stehen.

Beispiel: Der Filialleiter eines Supermarktes ruft abends von seinem PC aus die Tagesumsätze der beiden Kassen ab und ermittelt den täglichen Filialumsatz. Das Objekt FilialPC gehört zur Klasse Filiale und besitzt die Methode ***BerecheTagesumatzFiliale***. Diese Methode ruft die Methoden ***getEinlage*** und ***getKassenbestand*** bei den einzelnen Kassenobjekten auf, die jeweils die Einlage bzw. den Kassenbestand als Rückgabeparameter liefern. Die Methode ***BerecheTagesumatzFiliale*** berechnet die Kassenumsätze, addiert diese auf und liefert dann das Ergebnis zurück.



Umsetzung des Interaktionsdiagramms in Quellcode

Das Interaktionsdiagramm zeigt dem Programmierer, welche Objekte beteiligt sind und welche Methoden in welcher Reihenfolge aufgerufen werden. Damit sollte er in der Lage sein, den Anwendungsfall zu programmieren. Es sind die Klassen **Filiale** und **Kasse** beteiligt. Die Klasse **Filiale** hat eine Methode ***BercheneTagesumsatz***, die Klasse **Kasse** hat die Methoden ***getEinlage*** und ***getKassenbestand***.

In der Applikation wird die Methode ***BerechneTagesumsatz*** für das Objekt **FilialPC** der Klasse **Filiale** aufgerufen:

```
public class filialeAbrechnen
{
 public static void main(String argv[])
 {
   ...
   ...
   System.out.println("Umsatz der Filiale: "+FilialPC.BerechneTagesumsatz( ));
  } }
```

Methode der Klasse Filiale:

```
public class filiale
{
        ..
        ..
        public double BerechneTagesumsatz( )
        {
                double e,k,fumsatz ;
                e=Kasse1.getEinlage( );
                k=Kasse2.getKassenbestand( );
                fumsatz=k-e;
                e=Kasse2.getEinlage( );
                k=Kasse2.getKassenbestand( );
                fumsatz=fumsatz+(k-e);
                return fumsatz;
        }
}
```

Anmerkung:

Sinnvoller wäre es hier gewesen, in der Klasse Filiale Verweisattribute auf die zugehörigen Kassen anzulegen. Da eine Filiale mehrere Kassen haben kann, würde man hier ein Array des Datentyps kasse anlegen. (siehe Kapitel 5.3)

z.B.:

```
private kasse[ ] meineKassen = new kasse [10];
```

dann weiter in der Methode:

```
e=meineKassen[0].getEinlage( );
k=meineKassen[0].getKassenbestand( );
usw.
```

Methoden der Klasse Kasse:

```
public class filiale
{
        private double einlage;
        private double kassenbestand ;
        public double getEinlage( )
        {
                return einlage;
        }
        public double getKassenbestand( )
        {
                return kassenbestand;
        }
}
```

6.3 Weitere Diagrammarten

Auf die folgenden der insgesamt 13 Diagrammarten wird in diesem Skript nicht näher eingegangen:

- Zustandsdiagramme
- Kollaborationsdiagramme (Kommunikationsdiagramm)
- Komponentendiagramme
- Verteilungsdiagramme
- Paketdiagramme

6.4 Übungen

1. Eine Autovermietung vermietet Fahrzeuge an Kunden. Eine Vermietung hat jeweils eine Buchung zur Folge.

Autovermietung		**Fahrzeug**
Kunde		**Buchung**

Aufgabe: Zeichen Sie Assoziationen und Kardinalitäten ein.

2. Eine Supermarktkette verwaltet von der Zentrale (Z1) aus 2 Filialen (F1 und F2).
   Die Filiale 1 hat 3 Kassen (k1, k2 und k3), die Filiale 2 hat 2 Kassen (k4, k5).
   Über den Zentrale sollen die Tagesumsätze aller Filialen ermittelt werden.
   Die Kassen speichern die Kassenbezeichnung, die morgendliche Bargeldeinlage und den aktuellen Kassenbestand. Die Filialen werden mit Name und Tageseinnahme erfasst.
   a) Erstellen Sie das Klassendiagramm
   b) Erstellen Sie ein Interaktionsdiagramm für die Ermittlung der Tagesumsätze durch die Zentrale.

3. Eine Studentin möchte ein Buch aus der Bibliothek ausleihen
   - Sie gibt bei der Ausleihe Lesernummer, Autor und Titel des Buches an.
   - Die Ausleihe prüft über den Buchbestand, ob das Buch in der Bibliothek geführt wird.
   - Wenn das Buch existiert, wird im zweiten Schritt versucht, dieses auszuleihen.
   - Es wird jetzt geprüft, ob ein Exemplar vorhanden ist.
   - Sollte in Exemplar vorliegen, wird es mit der Lesernummer der Studentin ausgeliehen.
   - Die Studentin erhält von der Ausleihe das Buch.

   a) Erstellen Sie ein Klassendiagramm
   b) Erstellen Sie ein Aktivitätsdiagramm

4. In einem Unternehmen soll die Auftragsbearbeitung mit einem neuen Programm erledigt werden. In einem Klassendiagramm soll zunächst die Struktur des Systems modelliert werden. Die Grundlage dazu liefern folgende Informationen:

Wenn ein Kunde einen Auftrag erteilt, werden die einzelnen Auftragspositionen erfasst.
Bei jeder Auftragsposition wird anhand der Produktdaten die Lieferfähigkeit überprüft.
Nach Abschluss der Auftragserfassung wird eine Rechnung generiert.

Aufträge werden mit Auftragsnummer, Datum und Auftragssumme erfasst.
Für Kunden werden die Kundennummer, Name und Anschrift gespeichert. Auftragspositionen
Beinhalten die Positionsnummer, Artikelnummer und Menge.
Die Produktdaten speichern Artikelnummer, Artikelbezeichnung, Bestand und Preis.
Rechnungen beinhalten die Rechnungsnummer und das Rechnungsdatum.

a) Erstellen Sie zunächst das **Klassendiagramm.**
b) Wie werden die **Assoziationen** zwischen den Klassen realisiert? Zwischen welchen Klassen sehen Sie eine Kompositionsbeziehung?
c) Stellen Sie den Anwendungsfall „Auftragsstatistik erstellen“ in einem **Interaktionsdiagramm** dar. Als Ergebnis der Auftragsstatistik soll täglich eine Liste mit den Aufträgen des Tages erstellt werden nach folgendem Muster:

Auftragsnummer	Kundennummer	Kundenname	Auftragssumme
080703001	2007	Meier KG	559,50
080703002	2003	Rinn und Keil	1300,00
080703003	2013	Loose GmbH	712,50
080703004	2003	Rinn und Keil	79,95

d) Jedes Produkt wird von einem Lieferanten geliefert. Ergänzen Sie das Klassendiagramm um eine Klasse Lieferant. Für Lieferanten werden die Attribute Lieferantennummer, Name und Anschrift gespeichert.
e) Welche weiteren Anwendungsfälle lassen sich aus dem Klassendiagramm ableiten. Erstellen Sie dazu ein Anwendungsfalldiagramm.

7. Das objektorientierte Vorgehensmodell

Für Anwendungssysteme, die objektorientiert gelöst werden und deren Entwicklungsphasen durch UML-Diagramme unterstützt werden sollen, bietet sich das **Phasenmodell** als Vorgehensweise an. Dabei können folgende Entwicklungsphasen unterschieden werden:

1. Problemstellung (Kundenauftrag)

2. Analyse

Analyse des Geschäftsprozesses Darstellung mit **EPK** und/oder **Anwendungsfalldiagramm**.

3. Design (Entwurf)

Entwurf der Klassen mit Attributen und Methoden	**Klassendiagramm**
dynamisches Verhalten der Klassen, Ablauflogik	**Sequenz- und Aktivitätsdiagramme auch Struktogramme u. a.**

Häufig wird das sog. Prototyping oder RAD (Rapid-Analyse-Design) benutzt, um dem Auftraggeber schon in einem frühen Stadium ein grobes Modell der Problemlösung vorzustellen. Anhand des Prototyps können evt. Abweichende Vorstellung noch rechtzeitig korrigiert werden bevor die Systementwicklung schon zu weit fortgeschritten ist.

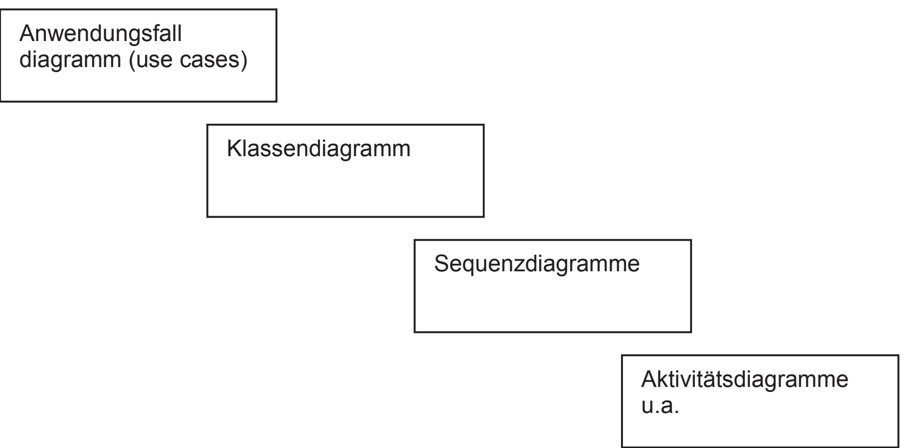
4. Codierung

5. Test

6. Implementierung

Modellierung mit UML

In den einzelnen Phasen des Vorgehensmodells können UML-Diagramme eingesetzt werden, wie sie im Skript dargestellt wurden. Zunächst wird das zu entwickelnde System durch eine Sammlung informeller Aussagen beschrieben. Dies kann durch verwendete Formulare ergänzt werden. Daraus können folgende Diagramme abgeleitet werden:



Eine detaillierter Darstellung der unterschiedlichen Vorgehensmodelle bei der Systementwicklung finden Sie in der Powerpoint-Präsentation **ae2007.ppt**.

8. Datenbankzugriff unter Java

Im folgenden Kapitel wird in Kürze dargestellt, wie man mit Java auf SQL-Datenbanken zugreifen kann. Einfache Anwendungen lassen sich unter DOS realisieren, komfortablere Verarbeitung sollte man mit einer GUI programmieren.

Wenn Java auf eine Datenbank zugreifen will, wird das Paket JDBC benötigt. Darin ist der notwendige JDBC-ODBC Treiber enthalten, mit dem man auf die meisten Datenbanken zugreifen kann. (Es gibt insgesamt 4 Klassen für Datenbanktreiber (Class 1-4)). Der Treiber-Manager im JDBC regelt den Zugriff von Java auf die Datenbank. Die Art der Datenbank spielt dabei für den Java-Programmierer keine Rolle.

Damit für die Kommunikation mit einer Access-Datenbank Access nicht mehr benötigt wird, muss die Datenbank im ODBC-Manager angemeldet werden.

1. ODBC-Treiber registrieren
   Dazu aufrufen: **Systemsteuerung – Verwaltung – ODBC-Datenquellen Benutzer-DSN – Hinzufügen** . Dann Microsoft-Access-Treiber auswählen.
   Der Datenquelle einen beliebigen Namen geben, unter dem sie später in Java angesprochen wird. Z.B. **mydb**. Dann **Auwählen** und die entsprechende Datenbank auswählen. Alles mit OK bestätigen.

2. Das Java-Programm schreiben:
   import java.sql.;*
   Laden des Datenbanktreibers:
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

   Exception mit try und catch abfangen

3. **Verbindung zur Datenbank herstellen**.
   Dazu wir ein Object der Klasse Connection benötigt.

   Connection conn;
   conn=DriverManager.getConnection("jdbc:odbc:mydb");

   allgemein: DriverManager.getConnection(url, user, password)
   url ist immer wie folgt aufgebaut: jdbc:odbc:Datenquelle

4. **Eine SQL-Anfrage absetzen**
   Um Anfragen zu stellen, muss ein Objekt der Klasse **Statement** erzeugt werden.

   Statement stm = conn.CreateStatement();

   Das Ergbenis der Anfrage wird einem Objekt des Interface ResultSet zugewiesen:
   Die wichtigsten Methoden des Statement-Objekts sind die Methoden **executeQuery** und **execute Update**. Die Methode executeQuery gibt ein Objekt vom Typ **ResultSet** zurück.

   *ResultSet rs=stm.executeQuery("Select * from Artikel");* oder
   ResultSet rs=stm.executeQuery("Select Artnr, Artbez from Artikel WHERE Artnr >100");

   Artikel sei hier eine Tabelle aus der Datenbank, die unter mydb angesprochen wurde.

5. **Das Ergebnis der SQL-Anfrage auswerten.**

   Die Ergebnistabelle wurde in dem resultSet-Objekt **rs** gespeichert und kann nun ausgewertet werden.
   Zum Auswerten kann der Datenbankzeiger verwendet werden. Es gibt:
   rs.next( ); nächster Datensatz
   rs.previous( ) vorheriger Datensatz
   rs.first( ) erster Datensatz
   rs.last( ) letzter Datensatz

   Ausgabe: Gesamte Tabelle mit der **getString-Methode**:
   while(rs.next( ))
   System.out.println(rs.getString(Artnr)+“ “+rs.getString(Artbez));

   Parameter der getString-Methode ist entweder die Spaltenüberschrift oder der Spaltenindex, z.B.*(rs.getString(1).*
   Bei der get-String-Methode wird das Ergebnis in einen String umgewandelt:
   Für andere Datentypen gibt es andere Methoden wie:
   getInt( ); getDouble( ), getDate( ), getBoolean( ) usw.

6. **Update einer Datenbank mit der Methode executeUpdate**

   int us = stm.executeUpdate(“Update arikel set Preis=59.50 where Artnr=104”)

   Diese Anweisung gibt kein Ergebnis zurück, welches direkt abgefragt werden kann.

Beispiel:

```
import java.sql.*;   //SQL-Treiber laden
public class jdb100
{
  public static void main(String[] args)
  {
        try   {
           //Installation des odbc-Treibers
           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
           System.out.println("Treiber geladen");
        }
        catch (ClassNotFoundException e)
        {
           System.out.println("Treiber nicht gefunden");
        }
        try   {
          Connection conn;
          conn = DriverManager.getConnection("jdbc:odbc:mydb");
          System.out.println("Datenbank gefunden");
          //eine Abfrage auf die tabelle KUNDEN erzeugen
          Statement stm=conn.createStatement();
          ResultSet rs = stm.executeQuery("Select * From Artikel");  //eine Tabelle aus der Datenbank
          while(rs.next())
          {
           System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
          }
       }
        catch (SQLException ex)
        {
          System.out.println("Fehler");
        }
  } }
```

9. Wiederholung - Grundbegriffe der Objektorientierten Programmierung

Generalisierung
Die Kunst, gemeinsame Strukturen von Objekten zu erkennen und diese in so genannten Superklassen zu verallgemeinern

Vererbung
Das Prinzip, nach dem sich Eigenschaften von der Superklasse automatisch auf die Subklassen übertragen

Objekt oder Instanz
Ist die konkrete Ausprägung einer Klassendefinition

Methoden
Die Fähigkeiten, die die Objekte einer Klasse besitzen, um z.B. die Attributsausprägungen der Klasse zu verändern.

Assoziation
Bezeichnet die Beziehung, die zwischen zwei Klassen besteht.

Kapselung
Das Verbinden von Variablen (Attributen) und Methoden in einem Objekt. Der interne Aufbau eines Objekts wird vor den Benutzern versteckt, um dessen Programme unabhängig vor Änderungen im internen Klassenaufbau zu machen und unbefugten Zugriff auf Attribute zu verhindern.

Polymorphie
Ein Mechanismus, der das individuelle Gestalten von Methoden innerhalb einer Klassenhierarchie erlaubt, wobei die Methoden den gleichen Namen tragen.

Klasse
Darin werden Objekte des gleichen Typs zusammengefasst.

Komposition
Aus mehreren Objekten wird eine neue Gesamtheit gebildet, wobei die Objekte unabhängig voneinander betrachtet keine sinnvolle Funktion haben.

Attribut
Ein Merkmal, das zur Beschreibung einer Klasse gehört.

Aggregation
Objekte werden miteinander zu einem sinnvollen Ganzen verbunden. Die einzelnen Objekte können aber auch unabhängig voneinander existieren.

Kardinalität
Beschreibt die mengenmäßige Beziehung zwischen zwei Klassen

Botschaft
Eine Nachricht, die einer Methode übergeben wird oder die eine Methode zurückliefert.

Konstruktor
Bereits beim Erzeugen eines neuen Objekts werden Attribute initialisiert.

Anhang

Software

▪ Java-Hamster-Modell:	www.java-hamster-modell.de
▪ Javaeditor von G. Röhner	http://lernen.bildung.hessen.de/informatik/javaeditor/
▪ NetBeans Java Entwicklung	http://www.netbeans.org/
▪ Java-Developement Kit	http://www.oracle.com
▪ Struktogrammeditor	www.strukted.de
▪ Microsoft Visio	Für alle Arten von Diagrammen. Testversion bei http://office.microsoft.com/de-de/visio/
▪ Mind Maps erstellen	http://schule.bildung.hessen.de/sponsor/mind-manager/hinweise

Literatur

Deck, Neuendorf: Java-Grundkurs für Wirtschaftsinformatiker, Vieweg 2007
Rau, Karl-Heinz, Objektorientierte Systementwicklung, Vieweg 2007
Geers, Pellatz, Wagner, Wirtschaftsinformatik für Berufliche Gymnasien

Links

Auf meiner Seite **www.pellatz.de** gibt es dieses Skript, aktuelle Informationen zum Stand des Unterrichts sowie weitere nützliche Links.