Objektorientierte Modellierung

Die Entwicklung guter Software gelingt nur unter Beachtung von Qualitätsmerkmalen und Einsatz entsprechender Werkzeuge und Methoden. Die Bedeutung der Qualitätsmerkmale hat sich im Laufe der Zeit geändert. Programme, die auf den Rechnern der 1. und 2. Generation laufen sollten, mussten eine gute Speicher- und Zeiteffizienz aufweisen. Dies wurde oft durch trickreiche und kunstfertige Programmierung erreicht. Die Entwicklung besserer Hardware erlaubte es, komplexere Anwendungen anzugehen, welche nicht mehr von einer einzelnen Person bearbeitet werden konnten. Mit den benutzten Entwicklungsmethoden wurden die Programme immer fehlerhafter, was um 1965 zur *Softwarekrise* führte.

Die Qualitätsmerkmale *Zuverlässigkeit, Korrektheit* und *Robustheit* rückten ins Zentrum des Interesses. Erreicht werden sollen diese Qualitätsmerkmale durch die *strukturierte Programmierung* auf der Basis des Top-Down-Prinzips und der Methode der schrittweisen Verfeinerung.

Wenn Programme immer größer werden, lassen sich Fehler auch bei strukturierter Programmierung nicht vermeiden. Daher ist es wichtig, dass man auftretende Fehler rasch analysieren und beheben kann. Dies gelingt dann gut, wenn die Software das Qualitätsmerkmal der *Wartbarkeit* (ab 1970) erfüllt. Die Wartbarkeit wird durch das Modulkonzept, das *Geheimnisprinzip* und die *Kapselung* unterstützt.

Eine weitere defensive Maßnahme zur Erhöhung der Zuverlässigkeit besteht in der Wiederverwendung bewährter Module. Dies setzt allerdings voraus, dass Softwaremodule so entworfen werden, dass sie dem Qualitätsmerkmal der *Wiederverwendbarkeit* entsprechen. Wieder verwendbare Module müssen kombinierbar, verständlich, stetig erweiterbar und geschützt sein. Hier greift die *objektorientierte Programmierung*, die mit Klassen und Objekten, sowie Vererbung und Polymorphismus geeignete Konzepte und Methoden zur Realisierung der Wiederverwendbarkeit bereitstellt.

Der objektorientierte Ansatz hat für die heutige Softwareentwicklung eine sehr hohe Bedeutung, ähnlich wie die strukturierte Programmierung in den vergangenen Jahrzehnten. Er setzt auf den bislang entwickelten Konzepten des Software-Engineerings auf und entwickelt sie konsequent weiter. Die strukturierte Programmierung bleibt dabei als wichtige Methode erhalten, sozusagen als Grundvoraussetzung für eine ordentliche Programmierung. Die objektorientierte Softwareentwicklung erlaubt es, Software übersichtlicher zu strukturieren und sie aus wieder verwendbaren Bausteinen zu konstruieren.

Klassen und Objekte

Objektorientierung ist eigentlich etwas ganz Normales, was täglich benutzt wird, um mit der Komplexität der Umwelt zurechtzukommen. Wir betrachten die Welt als eine Menge von Objekten, die zueinander in Beziehung stehen und gegebenenfalls miteinander kommunizieren. Geht man beispielsweise im Supermarkt einkaufen, so wird man sich für die meisten Menschen nur insoweit interessieren, dass man sich mit den Einkaufswagen nicht gegenseitig behindert und in der Warteschlange vor einer Theke nicht vorgedrängelt wird. Man nimmt die Menschen nur als Ganzes - als Objekt - wahr und interessiert sich nicht für die Person. Die Marktleitung sieht in den Menschen im wesentlichen Kunden. Dabei wird ebenfalls von den Eigenheiten der einzelnen Individuen abstrahiert und auf das Wesentliche reduziert: ein Kunde ist ein Mensch der im Supermarkt einkauft. Der Begriff Kunde ist daher eine Abstraktion für einen einkaufenden Menschen. In objektorientierter Terminologie spricht man von der Klasse Kunde.

Die wichtigsten Bestandteile eines objektorientierten Programms sind *Objekte* und *Klassen*. *Objekte* sind Elemente, die in einem Anwendungsbereich von Bedeutung sind. In einem Textverarbeitungsprogramm beispielsweise könnte ein Objekt einen bestimmten Absatz repräsentieren. In einer Finanzbuchhaltung ist die Bilanz des Monats November ein mögliches Objekt und bei einem Auftragsverwaltungssystem wird man es mit Objekten wie Kunden und Aufträgen zu tun haben.

Aus der Sicht des Benutzers stellen Objekte bestimmte Dienstleistungen und Informationen zur Verfügung. Ein Aufzug beispielsweise besitzt die Funktionalität, auf- und abwärts zu fahren, sowie

Informationen über die augenblickliche Position oder anzufahrende Stockwerke. Aus der Sicht des Programmierers sind Objekte Funktionseinheiten eines Programms, die zusammenarbeiten, um eine gewünschte Funktionalität zur Verfügung zu stellen.

Eine *Klasse* entsteht durch die Abstraktion von den Details gleichartiger Objekte und beschreibt die Eigenschaften und Struktur einer Menge gleichartiger Objekte. Gleichartige Objekte werden also durch eine gemeinsame Klasse zusammengefasst und beschrieben. Die Objekte sind *Exemplare* dieser gemeinsamen Klasse. Die Möglichkeit, von einer Menge ähnlicher Objekte Gemeinsamkeiten zu abstrahieren und in einer Klasse allgemein zu beschreiben, ist ein markantes Merkmal objektorientierter Systeme.

Eine Klasse ist die Beschreibung gleichartiger Objekte.

Attribute und Methoden

Die gleichberechtigte Betrachtung von Daten und Funktionen innerhalb der objektorientierten Programmierung kommt durch den Aufbau von Objekten zur Geltung. Jedes Objekt besteht aus Attributen und Methoden.

Die Attribute stellen den Datenbereich eines Objekts dar. In der zugehörigen Klasse wird die Datenstruktur dieses Datenbereichs beschrieben. Als Attribute der Klasse *Kunde* könnte man beispielsweise Name, Adresse und Bonität verwenden. Objekte dieser Klasse werden dann durch bestimmte Wertbelegung charakterisiert. Diese Wertbelegung legt den Zustand von Objekten fest.

Methoden stellen die Operationen dar, mit denen Objekte manipuliert werden können. Sie kommen in Form von Prozeduren und Funktionen vor. Über Prozeduren kann der Zustand von Objekten, charakterisiert durch die Attributwerte, geändert werden. Mit Funktionen lässt sich Auskunft über den Zustand von Objekten einholen.

Modellierung von Autos

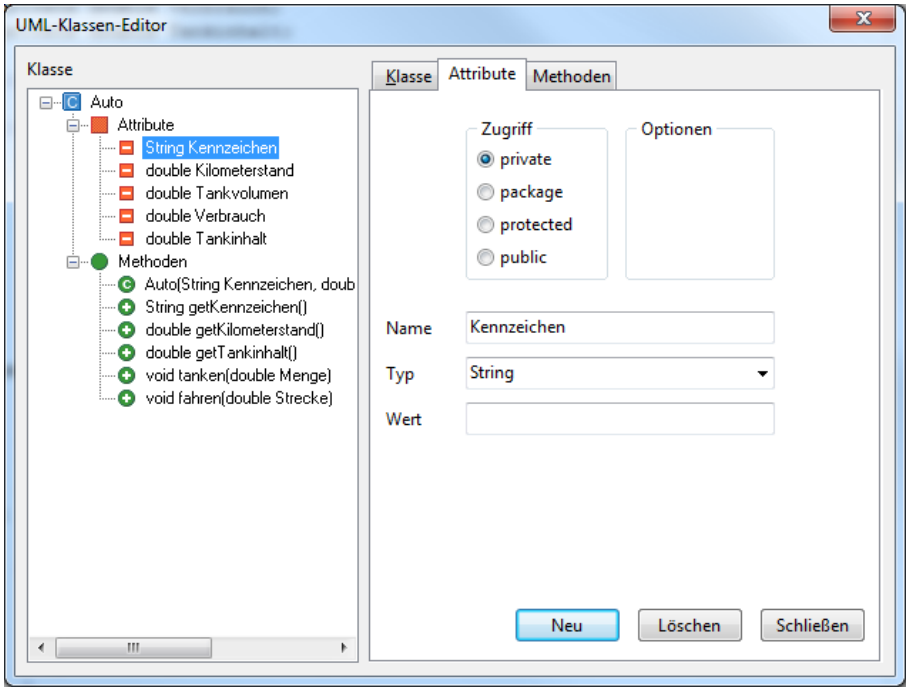
Autos soll modelliert werden. Modellieren heißt von den konkreten Details eines Autos zu abstrahieren und eine vereinfachte Beschreibung der Wirklichkeit zu finden. Wie diese Beschreibung aussieht hängt natürlich auch davon ab, zu welchem Zweck modelliert wird. In einer Simulation sollen die Tank- und Kilometerstandsanzeige von Autos modelliert werden. Die anzuzeigenden Werte werden durch tanken und fahren beeinflusst. In unser Auto-Modell gehen daher die Attribute Kennzeichen, Kilometerstand, Tankvolumen, Verbrauch und Tankinhalt.

Die Modellierung wird im UML-Klassen-Editor vorgenommen. Dort werden die erforderlichen Attribute und Methoden (Prozeduren, Funktionen und Konstruktoren) angelegt. Automatisch wird dabei der entsprechende Quelltext für die Auto-Klasse erzeugt.

Im Editor müssen dann die Methoden ausprogrammiert werden. Das kann wie folgt aussehen:

```
public class Auto {
  // Anfang Attribute

  private String Kennzeichen;
  private double Kilometerstand;
  private double Tankvolumen;
  private double Verbrauch;
  private double Tankinhalt;
  // Ende Attribute

  // Anfang Methoden
  public void Auto(String Kennzeichen, double Tankvolumen, double Verbrauch) {
    this.Kennzeichen = Kennzeichen;
    this.Tankvolumen = Tankvolumen;
    this.Verbrauch = Verbrauch;
    Tankinhalt = 0;
```

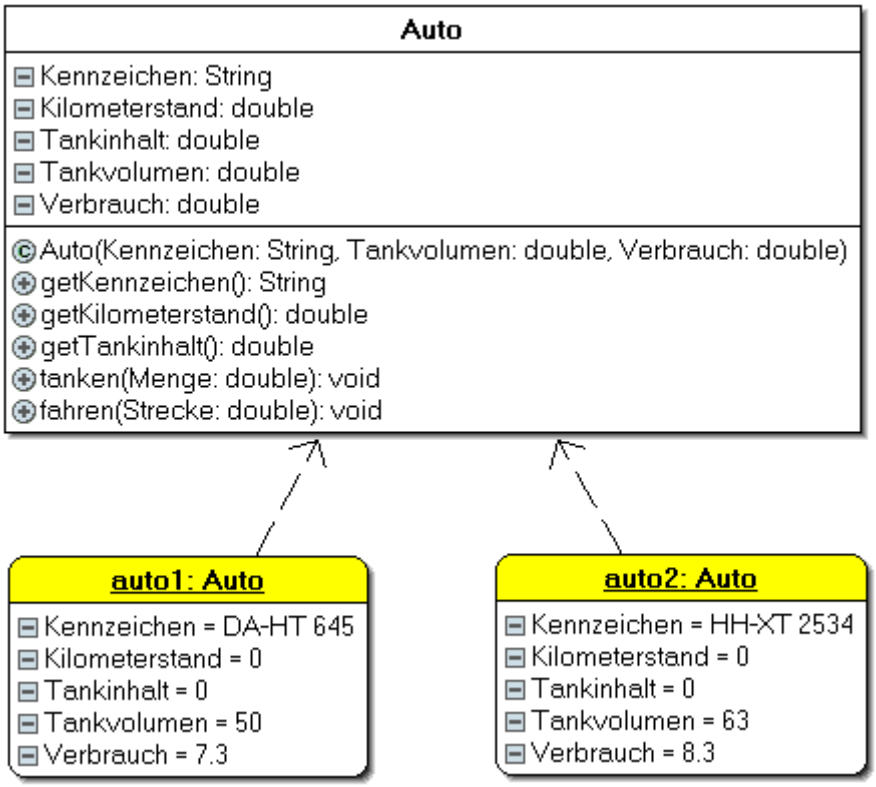
```
    Kilometerstand = 0;
  }

  public String getKennzeichen() {
    return Kennzeichen;
  }

  public double getKilometerstand() {
    return Kilometerstand;
  }

  public double getTankinhalt() {
    return Tankinhalt;
  }

  public void tanken(double Menge) {
    // zu implementieren
  }

  public void fahren(double Strecke) {
    // zu implementieren
  }


  // Ende Methoden
}
```



Auto-Objekte

Über die Klasse werden allgemeine Eigenschaften von Autos modelliert. Es wird allerdings keine Aussage über irgendein konkretes Auto gemacht. Dazu brauchen wir das Konzept des *Objekts.*

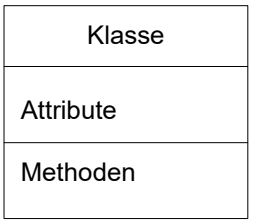
Ein Objekt ist ein individuelles *Exemplar* einer Klasse.



Objekte können im UML-Fenster über das Kontextmenü der Klasse erstellt werden. Dazu wählt man einen Konstruktor aus und gibt die erforderlichen Parameterwerte ein.

UML-Darstellung von Klassen

Klassen werden als Rechtecke dargestellt. Der Klassenname steht im Singular und zentriert im Kopf des Rechtecks. Darunter - durch eine Strecke abgeteilt - werden die Attribute linksbündig aufgeführt, ganz unten stehen die Methoden der Klasse.



UML-Darstellung von Objekten

Objekte werden als oben abgerundete Rechtecke dargestellt. Der Objektname steht im Singular und zentriert im Kopf des Rechtecks, bei Bedarf folgt mit Doppelpunkt abgetrennt der Klassenname. Im Unterschied zur Darstellung einer Klasse ist der Objektname unterstrichen. Bei Objekten gibt man nur die Attribute und zwar zusammen mit ihren Werten an. Die Methoden werden nicht angegeben, denn sie sind für alle Objekte einer Klasse gleich und ergeben sich aus der Klasse.

Prozeduren

Die Auto-Klasse enthält die beiden Prozeduren *tanken* und *fahren*. Die Prozedur *tanken(double Menge)* soll unter Beachtung des Tankvolumens den Tankinhalt um die als Parameter angegebene Menge erhöhen.

Die Prozedur *fahren(double Strecke)* soll im Modell eine Autofahrt simulieren. Der Kilometerstand nimmt also um die zu fahrende Strecke zu und der Tankinhalt um die für die Strecke erforderliche Kraftstoffmenge ab. Dabei ist vom durchschnittlichen Verbrauch auszugehen.

Reicht der Tankinhalt aber für die zu fahrende Strecke nicht aus, so soll das Auto so weit fahren, bis der Tank leer ist.

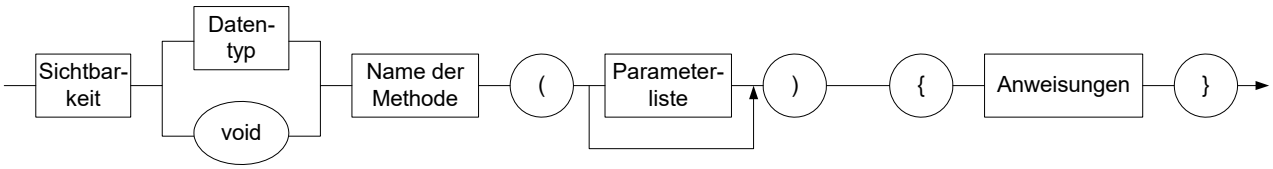
```
public void fahren(double Strecke) {
  double VerbrauchFuerStrecke = (Strecke * Verbrauch) / 100;

  if (VerbrauchFuerStrecke > Tankinhalt) {
    Strecke = Tankinhalt / Verbrauch * 100;
    Kilometerstand = Kilometerstand + Strecke;
    Tankinhalt = 0;
  } else {
    Kilometerstand = Kilometerstand + Strecke;
    Tankinhalt = Tankinhalt - VerbrauchFuerStrecke;
  }
}
```

Aufgaben

1. Analysiere und Implementiere die Prozedur *fahren* in deiner Auto-Klasse.
2. Implementiere die Prozedur *tanken* in der Auto-Klasse.

Syntaxdiagramme für Prozeduren und Funktionen

Das folgende *Syntaxdiagramm* legt fest, wie Methoden (Prozeduren und Funktionen) geschrieben werden:



Als Sichtbarkeit wird meist *public* bzw. *private* benutzt. Prozeduren werden anschließend mit *void* gekennzeichnet, bei Funktionen gibt man den Datentyp des Funktionswerts an. Es folgen der Name der Methode und in runden Klammern die Liste der Parameter, die auch leer sein kann. In den geschweiften Klammern stehen die Anweisungen, die in der Methode ausgeführt werden.

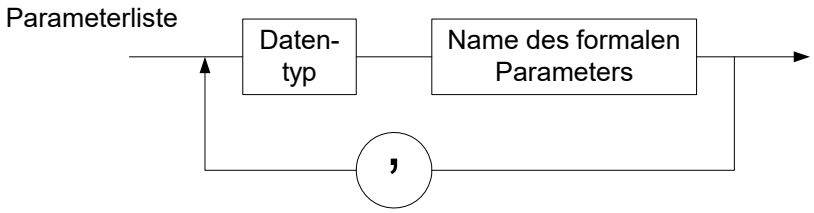
Beispiel:

```
public double maxStrecke() {
  return Tankinhalt / Verbrauch;
}
```

Bei Funktionen muss mit einer *return*-Anweisung der Funktionswert zurückgegeben werden.

Parameter

Methoden werden durch den Einsatz von Parametern flexibel nutzbar. Bei der Deklaration einer Methode benutzt man sogenannte *formale* Parameter. Die Parameterliste wird nach folgendem Syntaxdiagramm aufgebaut:




```
public void tanken(double Menge) {
  Tankinhalt = Tankinhalt + Menge;
  if (Tankinhalt > Tankvolumen)
    Tankinhalt = Tankvolumen;
}
```

Beim Benutzen einer Methode werden anstelle der formalen Parameter *aktuelle* Parameterwerte eingesetzt. Dies können konstante Werte, Variablen oder auch komplexe Ausdrücke sein.

Beispiele:
```
Auto1.tanken(20.4);
Auto2.tanken(Auto2.getTankvolumen() – Auto2.getTankinhalt());
```

Aufgabe
Vergleiche die beiden Syntaxdiagramme mit dem jeweils angegebenen Beispiel.
Erläutere dann den Zusammenhang zwischen Syntaxdiagramm und Beispiel.

Konsolenprogramm für Autos

Nachdem man alle Methoden implementiert hat, kann man im UML-Fenster Auto-Objekte erzeugen und die Methoden testen. Im unteren Fenster Auto.uml werden die Anweisungen protokolliert. Diese Anweisungen kann man in einem Konsolenprogramm mittels copy&paste benutzen:

```
public class AutoCon {

  public static void main(String[] args) {
    Auto auto1 = new Auto("DA-HT 645", 50, 7.3);
    Auto auto2 = new Auto("HH-XT 2534", 70, 8.7);
    auto1.tanken(30);
    auto2.tanken(40);
    auto1.fahren(250);
    auto2.fahren(500);
    System.out.println("Auto 1 - Tankinhalt: " + auto1.getTankinhalt());
    System.out.println("Auto 1 - Kilometerstand: " + auto1.getKilometerstand());
    System.out.println("Auto 2 - Tankinhalt: " + auto2.getTankinhalt());
    System.out.println("Auto 2 - Kilometerstand: " + auto2.getKilometerstand());
  }
}
```

GUI-Programm für ein Auto

Man kann auch ein Programm mit grafischer Benutzungsoberfläche entwerfen.

Aufgabe
Entwerfe das GUI-Formular am PC.

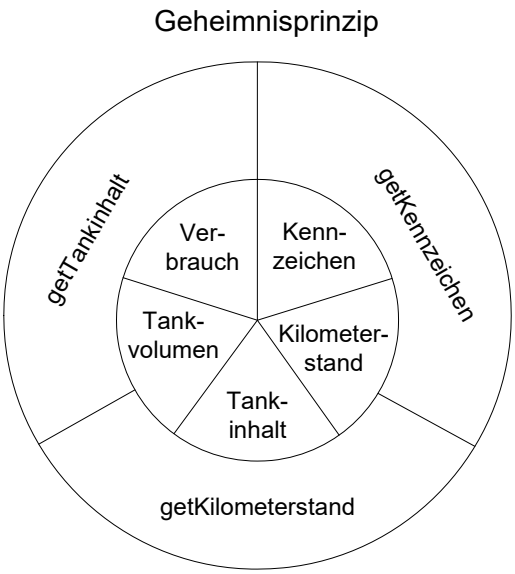
Im Bereich *Attribute* ergänzt man dann ein Auto:

```
  Auto auto1 = new Auto("DA-HT 950", 47, 6.7);
```

Dessen Daten werden nach jeder Operation durch die Methode *anzeigen* im GUI-Formular angezeigt.

```
  public void anzeigen() {
    tfKennzeichen.setText(auto1.getKennzeichen());
    nfKilometerstand.setDouble(auto1.getKilometerstand(), 1);
    nfTankinhalt.setDouble(auto1.getTankinhalt(), 1);
  }
```

Die Methode zum Anzeigen benutzt die get-Methoden für die Attribute, die man sich beim Anlegen einer neuen Klasse automatisiert erzeugen lassen kann. Über die Sichtbarkeit *private* und die *get-Methoden* wird das ***Geheimnisprinzip*** realisiert, nachdem die Objekte selbst für ihre Attribute verantwortlich sind und nur über Methoden auf diese Attribute zugegriffen werden kann.

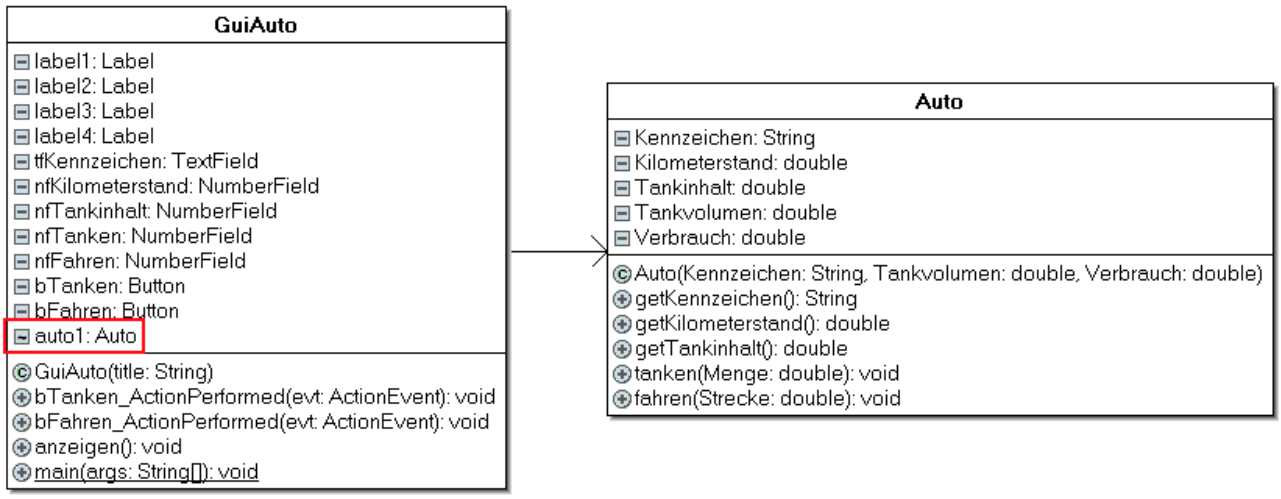


Aufgabe
Implementiere das GUI-Programm.

Assoziation

Das GUI-Programm besteht aus den beiden Klassen *GuiAuto* und *Auto*. Die Klasse Auto stellt das sogenannte ***Fachkonzept*** dar. In dieser Klasse werden die eigentlichen Fachinhalte behandelt. Das Fachkonzept kann in Konsolen- oder GUI-Programmen verwendet werden.

Die Beziehung zwischen den beiden Klassen wird als ***Assoziation*** bezeichnet und durch einen gerichteten Pfeil dargestellt. Er ist so zu interpretieren, dass man von der Klasse GuiAuto auf das Auto zugreifen kann, die Klasse Auto aber keine Ahnung hat von welcher Klasse sie benutzt wird. Die Assoziation wird dadurch implementiert, dass die Klasse GuiAuto das zusätzliche Attribut auto1 vom Typ Auto erhält.

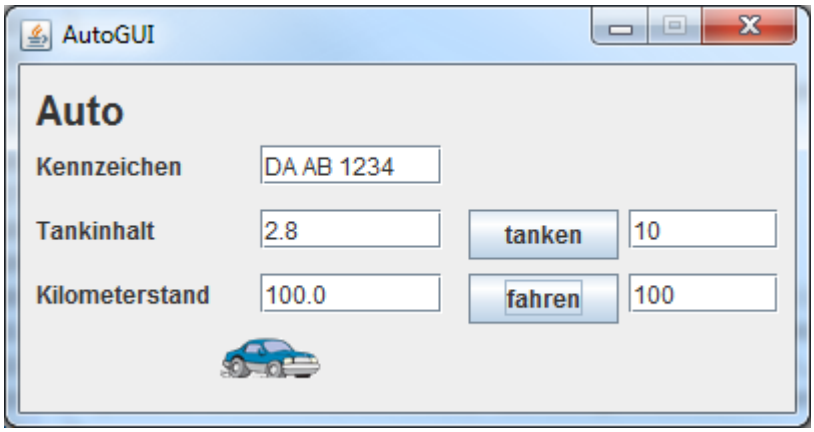


Simulation des Fahrens

Programme sind dann anschaulicher, wenn ablaufende Vorgänge visualisiert werden. Wir ergänzen daher auf dem Formular eine JLabel-Komponente und geben ihr den Namen *jlAuto*. Dann weisen wir ihr im Attribut Icon das Bild eines Autos zu.

Die Prozedur *anzeigen* ergänzen wir dann um die Positionierung Bildes in Abhängigkeit von der gefahrenen Strecke.

```
jlAuto.setLocation((int) auto1.getKilometerstand(), jlAuto.getY());
```



Aufgabe

Realisiere die Simulation des Fahrens mit dem Bild eines Autos.

Erweiterung des Fachkonzepts

Das Fachkonzept kann aus mehreren Klassen bestehen. Als Beispiel betrachten wir die Modellierung einer Fahrschule.

Zusätzlich zur bekannten Klasse *Auto* kommt hier noch die Klasse *Fahrschueler* vor, in der die für die Simulation relevanten Attribute eines Fahrschülers modelliert werden.

Aufgaben
Analysiere und beschreibe die Klasse Fahrschueler.
Modelliere sie im Java-Editor.

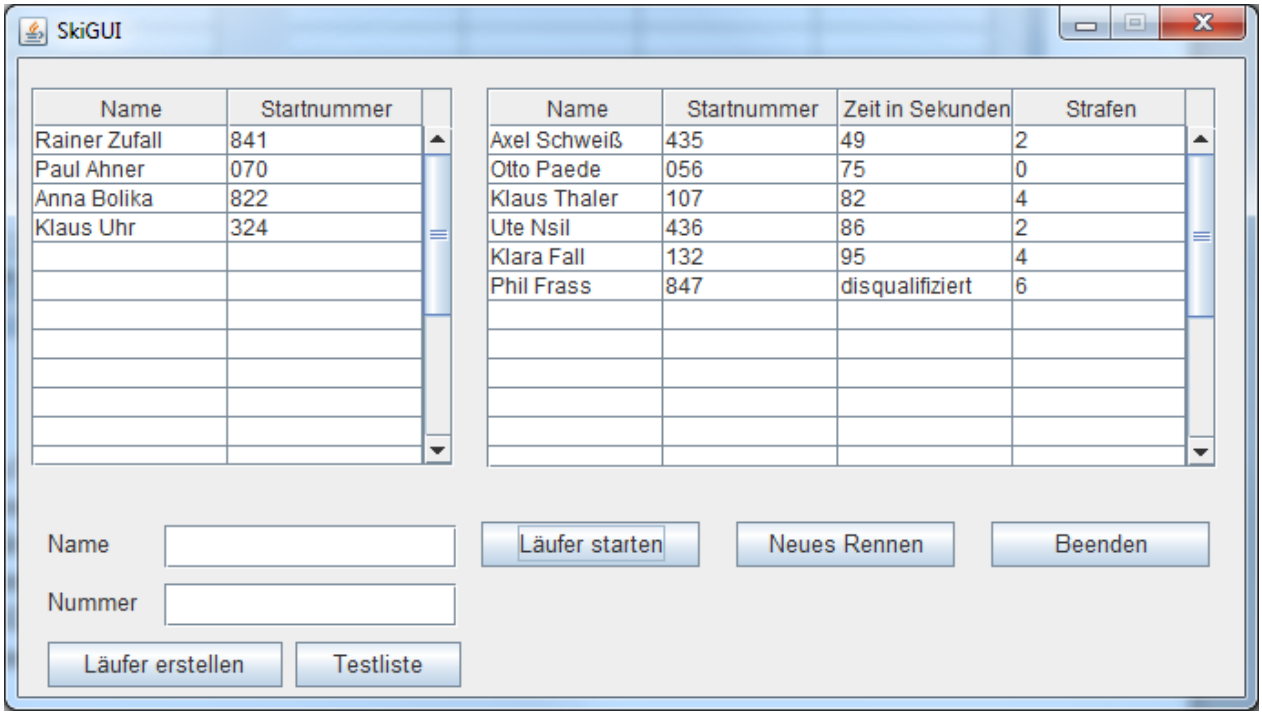
Erläutere diese beiden Methoden:

```
public boolean aussteigen() {
  if (dasAuto != null) {
    dasAuto = null;
    return false;
  } else
    return true;
}

public boolean einsteigen(Auto einAuto) {
  if (hatFührerschein()) {
    this.dasAuto = einAuto;
    return true;
  } else
    return false;
}
```

Implementiere die Klasse Fahrschueler.

Skirennen

Schaut man sich in einer Sportsendung ein Skirennen an, so wird bei der Zieldurchfahrt sofort die Platzierung der Läuferin bzw. des Läufers oder auch die aktuelle Ergebnisliste angezeigt. Die Zeitnahme erfolgt automatisch und gemäß der erzielten Zeit wird die Ergebnisliste ergänzt. Wir gestalten ein Simulationsprogramm, um die dabei auftretenden informatischen Fragestellungen zu erarbeiten.



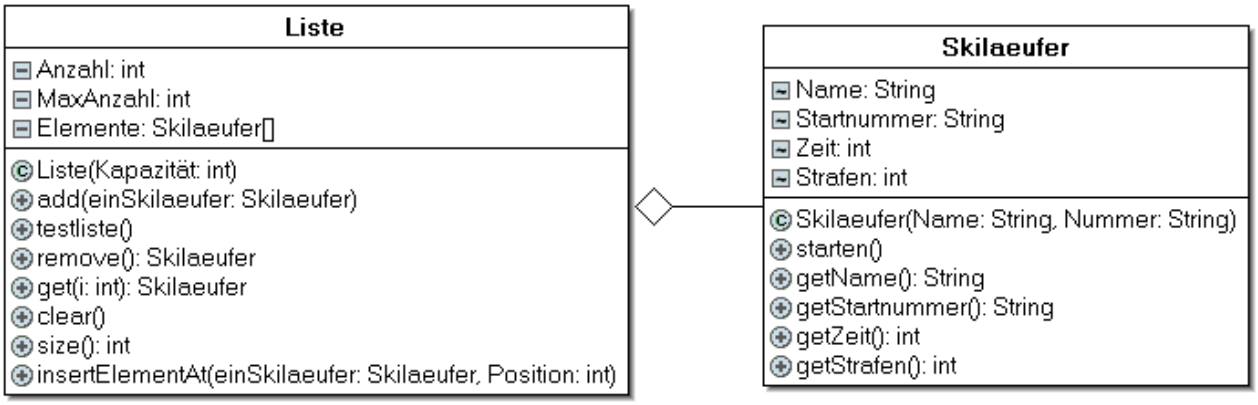
Aufgaben

a) Simuliere ein Skirennen mit dem fertigen Programm.
b) Analysiere, welche Objekte verwendet werden.
c) Modelliere für die Objekte geeignete Klassen.
d) Implementiere die Klassen und erprobe dein Modell interaktiv im UML-Fenster.
e) Entwirf eine GUI für das Simulationsprogramm.
f) Implementiere das Simulationsprogramm.

Im Programm Skirennen kommen als Objekte Skiläufer vor. Die Klasse Skiläufer hat Name, Startnummer, ZeitInSekunden und Strafen als Attribute. Name und Startnummer werden als Parameter für den Konstruktor gebraucht. get-Methoden sind für alle Attribute nötig. Als weitere Methode wird *starten* gebraucht. In dieser Methode werden Zeit und Strafen zufallsabhängig ermittelt.

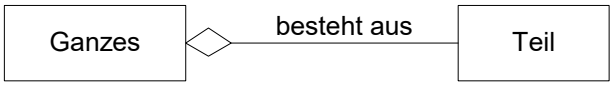
Nicht so leicht zu erkennen sind die beiden Listen-Objekte, links die Starterliste und rechts die Zieleinlaufliste. Eine Liste verwaltet die darin befindlichen Listenelemente. In unserem Beispiel sind dies die Skiläufer. Zur Modellierung einer Liste können wir ein Feld für Skiläufer benutzen. Im Konstruktor kann man die gewünschte Kapazität der Liste angeben und im Attribut *Anzahl* merkt man sich die aktuelle Anzahl der Elemente. Wir brauchen eine Prozedur *add(Skilauefer einSkilaufer)*, um einen Skiläufer in die Liste aufzunehmen und eine Funktion *remove()* , die den ersten Skiläufer aus der Liste entnimmt. Zusätzlich braucht man eine Funktion *size()*, die die aktuelle Länge der Liste liefert und eine Funktion *get(int i)*, welche den i-ten Skiläufer liefert. Da sich in der Zieleinlaufliste die Reihenfolge sich aus der Laufzeit ergibt, sehen wir noch eine Methode *insertElementAt(Skilaeufer einSkiLaeufer, int Position)* vor.

Als Klassendiagramm ergibt sich:



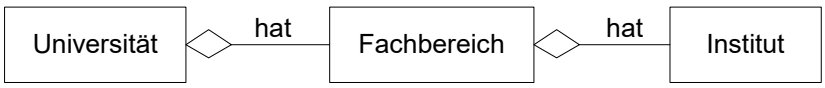
Aggregation

Ein Spezialfall der Assoziation ist die *Aggregation*, bei der die beteiligten Klassen in einer Ganzes-Teile-Beziehung (hat-Beziehung) stehen. Eine Aggregation bildet also die Zusammensetzung eines Objekts aus einer Menge gleichartiger Einzelteile ab. Dabei übernimmt die Aggregatklasse stellvertretend Aufgaben für die untergeordnete Klasse.



In der UML-Darstellung wird die Aggregatklasse mit einer Raute versehen. Die Raute symbolisiert das Behälterobjekt, in dem die Teile gesammelt werden.

Die Teil-Klasse kann selbst wieder eine Aggregatklasse sein:



Zur Implementierung einer Aggregation muss die Aggregatklasse um ein Attribut ergänzt werden, in dem mehrere Objekte gespeichert werden können. In Java benutzt man ein Feld oder einen Vektor.

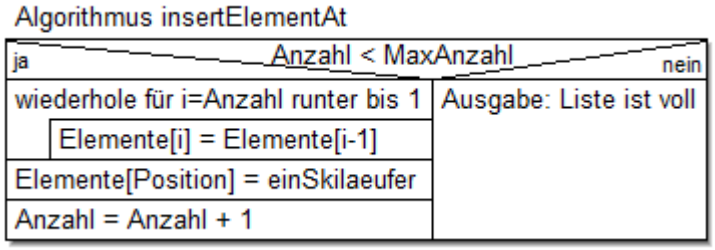
Aufgaben

a) Analysiere und vervollständige die Funktion *remove*

```
public Skilaeufer remove() {
  if (Anzahl > 0) {
    Skilaeufer einSkilaeufer = Elemente[0];
    for (int i = 1; i < Anzahl; i++)
      Elemente[i-1] = Elemente[i];
    ... // zu implementieren
  } else {
    System.out.println("Liste ist leer");
    return null;
  }
}
```

b) Die Prozedur *clear* soll alle Elemente der Liste löschen. Implementiere *clear* mit Hilfe von *remove*.

a) Verwende diesen Algorithmus für die Prozedur insertElementAt



b) Implementiere die Methode *add(Skilaeufer einSkilaeufer).* Benutze *add* zur Implementierung der Methode *testliste.* Beispiel: *add(new Skilaeufer("Axel Schweiß", "435")*

c) Im GUI-Programm werden zur Anzeige der Listen JTable-Komponenten benutzt. Da nach jedem Start eines Skiläufers die Listen neu angezeigt werden müssen, modellieren wir eine Methode *ListenAnzeigen*, die beide Listen anzeigt. Im folgenden Codeausschnitt bezeichnet jTStart die JTable-Komponente zur Anzeige der Starterliste.

```
public void ListenAnzeigen() {
  // Daten aus dem Fachkonzept in die GUI übernehmen

  // Starterliste
  for (int i = 0; i < Starterliste.size(); i++) {
    Skilaeufer einSkilaeufer = Starterliste.get(i);
    jTStart.setValueAt(einSkilaeufer.getName(), i, 0);
    jTStart.setValueAt(einSkilaeufer.getStartnummer(), i, 1);
  }
  for (int i = Starterliste.size(); i < jTStart.getRowCount(); i++) {
    jTStart.setValueAt("", i, 0);
    jTStart.setValueAt("", i, 1);
  }

  // Ergebnisliste
  . . . // zu implementieren
```

GUI-Programm des Skirennens

Im Programm Fahrschule haben wir zwei Autos im GUI-Programm erzeugt. Beim Skirennen brauchen wir analog dazu zwei Listen, die Starterliste und die Ergebnisliste:

```
private Liste Starterliste = new Liste(20);
private Liste Ergebnisliste = new Liste(20);
```

Testliste

Am einfachsten kann man ein Rennen simulieren, wenn man eine Testliste erzeugt.

```
public void ErzeugeTestliste() {
  Starterliste.clear();
  Starterliste.testliste();
  Ergebnisliste.clear();
  ListenAnzeigen();
}
```

Läufer erstellen

Die Testliste kann durch einen einzelnen Läufer ergänzt werden. Man liest die Name und Nummer ein. Wenn beide nicht leer sind, erzeugt man mit *neiuerSkiläufer = **new** Skiläufer(Name, Nummer)* einen neuen Skiläufer und fügt in mit *Starterliste.add(neuerSkiläufer)* der Starterliste hinzu.

Läufer starten

Als einfachste Variante kann man den startenden Läufer aus der Startliste entnehmen, ihn laufen lassen und dann ungeachtet der Zeit in die Ergebnisliste einfügen. Die Ergebnisliste soll aber nach den Zeiten geordnet sein. Daher muss man erst einmal die Einfügestelle suchen. Analysiere dazu:

```
public int getInsertPosition(Skilaeufer einSkilauefer) {
  int i = 0;
  Skilaeufer TempLaeufer = get(i);
  while (TempLaeufer != null && TempLaeufer.Zeit <= einSkilauefer.Zeit) {
    i++;
    TempLaeufer = get(i);
  }
  return i;
}
```

Benutzer dann *insertElementAt* , um den gestarteten Skiläufer an der richtigen Stelle in die Ergebnisliste einzufügen.

Neues Rennen

Um ein neues Rennen durchzuführen, leert man die Ergebnisliste und ruft in der Starterliste die Testliste auf.

Optionale Erweiterung

Zum zweiten Rennen soll die Starterliste umgekehrt zur Ergebnisliste des ersten Laufs aufgestellt werden, d. h. die schnellsten Läufer starten zuletzt. Konzipiere einen Algorithmus und setze ihn in eine Methode um, die aus der Ergebnisliste des 1. Laufs die Starterliste des 2. Laufs bildet.

Textdateien

Programme können Daten wie z. B. Spielstände in Textdateien abspeichern oder von dort aus einlesen. Textdateien enthalten im Gegensatz zu Textdokumenten nur unformatierten Text. Der Text selbst ist zeilenweise strukturiert. Das Ende einer Zeile wird durch Steuerzeichen festgelegt. Unter DOS und Windows werden dazu die zwei Steuerzeichen *CarriageReturn* (CR, Wagenrücklauf, #13, \r) und *Linefeed* (LF, neue Zeile, 10, \n) benutzt, unter Unix wird nur das CarriageReturn und beim Mac nur Linefeed verwendet. Im ASCII-Code haben diese Steuerzeichen die Codes 13 bzw. 10, in Java gibt man sie mit \r und \n an.

Java bietet für die Arbeit mit Textdateien die Klassen FileWriter und FileReader im IO-Package an. Beim Aufruf des Konstruktors gibt man den Dateinamen an. Er öffnet dann die angegebene Datei zum Schreiben bzw. Lesen. Dabei gilt es zu beachten, dass der Verzeichnis-Separator von Unix der Slash / und nicht der Backslash \ von DOS und Windows benutzt wird. Da die Ein- und Ausgabe von Dateien grundsätzlich zu Laufzeitfehlern führen kann (volle Festplatte, falscher Pfad, fehlende Rechte, ...) müssen die Ein-/Ausgabeoperationen in einem try-catch-Block stehen. Nachdem die Textdatei gelesen oder geschrieben ist, muss man sie mittels close wieder schließen. Mit der Methode write kann man zwar einen ganzen String ausgeben, aber die entsprechende Methode read liest nur ein einziges int-Zeichen ein, das mittels Typcasting (in Klammern vorangestellter Typ) als char-Zeichen interpretiert werden kann.

```
import java.io.*;

public class ReaderWriter {

  public static void main(String[] args) {
    try {
      FileWriter myFileWriter = new FileWriter("c:/temp/test.txt");
      myFileWriter.write("Hallo,\r\n");
      myFileWriter.write("ich gehöre in\r\n");
      myFileWriter.write("eine Textdatei!\r\n");
      myFileWriter.close();
    } catch (IOException e) {
      System.out.println("Fehler: Konnte Datei nicht erstellen!");
    }

    try {
      FileReader myFileReader = new FileReader("c:/temp/test.txt");
      while (myFileReader.ready()) {
        int Zeichen = myFileReader.read();
        System.out.print((char) Zeichen);
      }
      myFileReader.close();
    } catch (IOException e) {
      System.out.println("Fehler: Konnte Datei nicht öffnen!");
    }
  }
}
```

Komfortabler sind die Klassen *BufferedReader* und *BufferedWriter*. Sie puffern mittels eines internen Speichers die Eingabe und Ausgabe ab, was zu weniger Schreib/Lesevorgängen führt und damit die Performance erhöht. Der erhöhte Komfort rührt daher, dass BufferedWriter ein explizites betriebssystemunabhängiges *newLine*() enthält und BufferedReader ganze Zeilen mittles *readLine*() lesen kann.

Die Konstruktoren von BufferedReader und BufferedWriter werden nicht mit den Dateinamen aufgerufen, sondern mit Reader/Writer-Objekten, also für unseren Bedarf mit FileReader- und FileWriter-Objekten.

```
import java.io.*;

public class BufferedReaderWriter {

  public static void main(String[] args) {
    try {
      BufferedWriter myBufferedWriter =
        new BufferedWriter(new FileWriter("c:/temp/test.txt"));
      myBufferedWriter.write("Hallo,"); myBufferedWriter.newLine();
      // jetzt mit \n statt newLine()
      myBufferedWriter.write("ich gehöre in\n");
      myBufferedWriter.write("eine Textdatei!\n");
      myBufferedWriter.close();
    } catch (IOException e) {
      System.out.println("Fehler: Konnte Datei nicht erstellen!");
    }

    try {
      BufferedReader myBufferedReader =
        new BufferedReader(new FileReader("c:/temp/test.txt"));
      while (myBufferedReader.ready()) {
        String Zeile = myBufferedReader.readLine();
        System.out.println(Zeile);
      }
      myBufferedReader.close();
    } catch (IOException e) {
      System.out.println(" Fehler: Konnte Datei nicht öffnen!");
    }
  }
}
```

Erkennung von Wörtern

Zum Erkennen der Wörter in einer Zeile kann man einen StringTokenizer einsetzen. Er steckt im Paket java.util weswegen man den *import java.util.*;* braucht.

```
StringTokenizer st = new StringTokenizer("Dies ist ein Test");
while (st.hasMoreTokens()) {
  System.out.println(st.nextToken());
}
```

Man kann somit die Skiläufer in einer Textdatei speichern und sie daraus wieder einlesen.

Auf der Swing2-Registerkarte stehen Dialoge zur Auswahl von Dateien zur Verfügung.