Vorlesungsteil

Einstieg in die Objektorientierung

Objektivität: Alles hat zwei Seiten. Aber erst wenn man erkennt, dass es drei sind, erfasst man die Sache.
Heimito von Doderer

Übersicht zum Vorlesungsinhalt

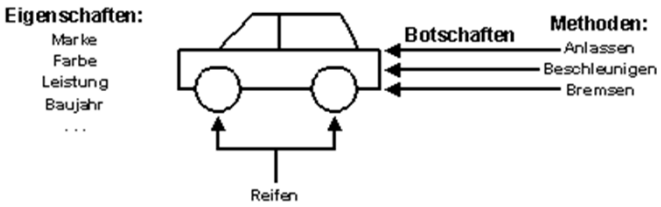
zeitliche Abfolge und Inhalte können variieren

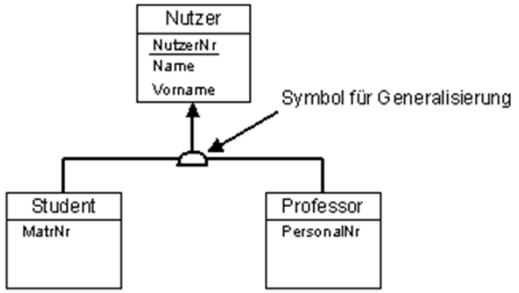
Einstieg in die Objektorientierung

- Projekt: Fußballmanager
- Analysephase der Softwaretechnik
- Abstraktion und Geheimhaltungsprinzip
- Generalisierung/Spezialisierung
- Objekte/Objektreferenzen
- Konstruktoren
- Einfache Vererbung
- Interfaces
- Komplexere Textzerlegung
- Muster: Model-View-Controller

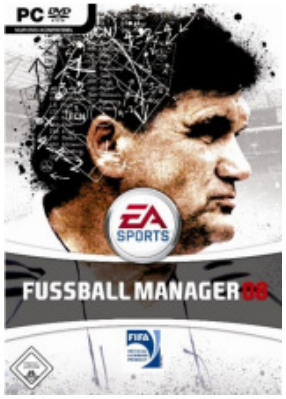
Einstieg in die Objektorientierung

Es gibt viele Standardbeispiele für die Einführung in die Objektorientierung, da wären Autos und Autoteile oder Professoren und Studenten zu nennen:





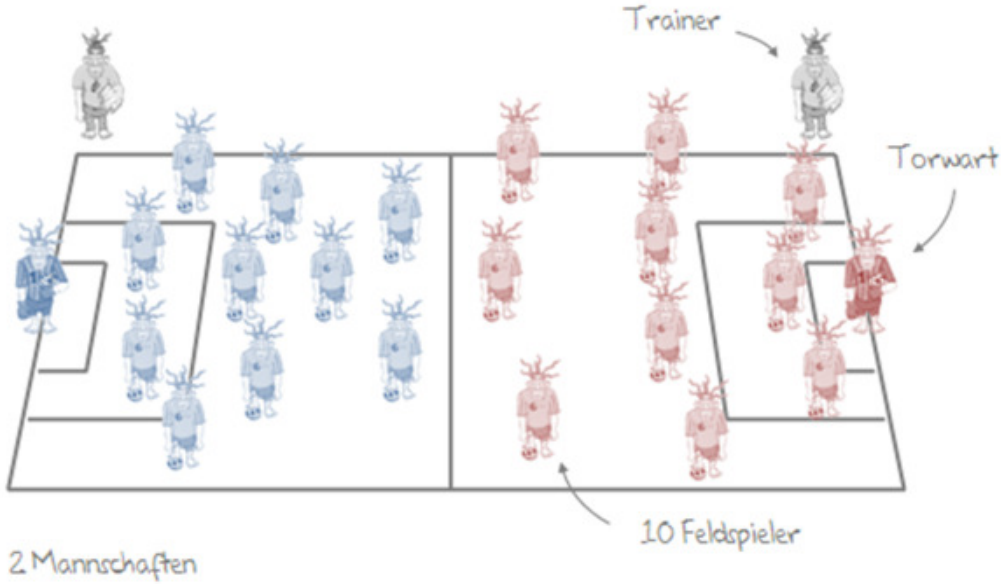
Wir werden Schritt für Schritt einen kleinen Fußballmanager entwickeln, dabei die Motivation für die Verwendung von Vererbungsmechanismen entwickeln und am Ende sogar zwei Mannschaften in einem Freundschaftsspiel gegeneinander antreten lassen.



Vielleicht nicht so professionell, wie der von EA Sports, aber dafür können wir hinterher sagen, dass wir ihn selbst programmiert haben ☺.

Konzept eines einfachen Fußballmanagers I

Ein Fußballspiel findet zwischen zwei Mannschaften statt. Eine Mannschaft besteht jeweils aus einem Trainer, einem Torwart und zehn Feldspielern:

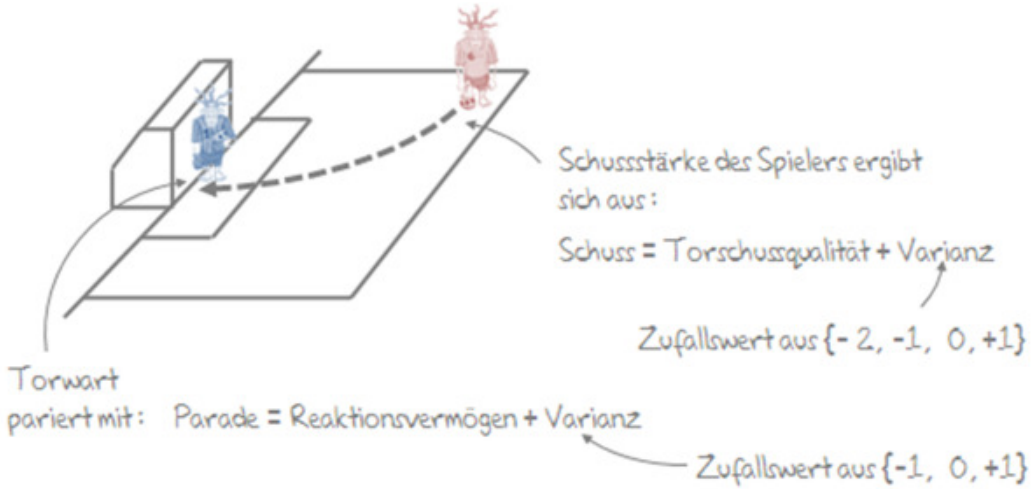


Jeder Spieler hat die Eigenschaften Name, Alter, Spielstärke, Torschussqualität, Motivation und die Anzahl der geschossenen Tore. Zusätzlich hat der Torwart noch ein Reaktionsvermögen. Ein Trainer hat die Eigenschaften Name, Alter und als Besonderheit Erfahrung.

Pro Spielminute soll zufällig, aber in Abhängigkeit der jeweiligen Mannschaftsstärken, eine Torschusstendenz generiert werden. Die Stärke ermittelt sich dabei aus der durchschnittliche Mannschaftsstärke (80%), der durschnittlichen Mannschaftsmotivation (15%) und der Erfahrung des Trainers (5%).

Konzept eines einfachen Fußballmanagers II

Feldspieler können auf das gegnerische Tor schiessen und Torhüter müssen die Schüsse entsprechend parieren:



Ein Spieler, der während des Spiels als Torschütze ausgewählt wurde, schießt den Ball mit der Kraft schuss aufs Tor. Der Schuss ist dabei abhängig von seiner Torschussqualität und einer kleinen Varianz, denn nicht jeder Schuss kann gleich gut sein.

Der gegnerische Torwart pariert den Schuss entsprechend seinem Reaktionsvermögen und ebenfalls einer kleinen Varianz.

Die Spielereigenschaften Spielstärke, Torschussqualität, Motivation und Reaktionsvermögen sollen ganzzahlig sein und im Intervall [0,10] liegen, wobei 0 für schlecht und 10 für weltklasse steht.

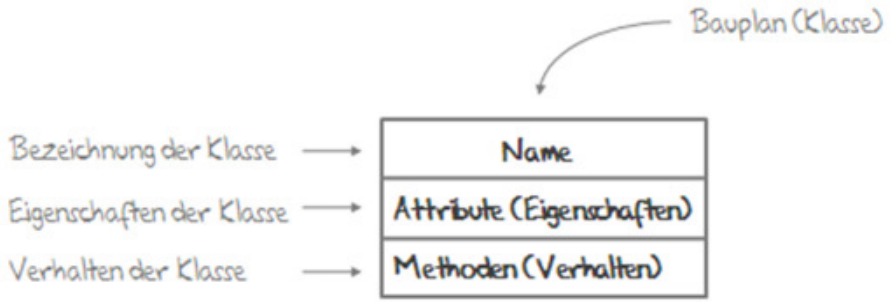
Analysephase

In der Analysephase identifizieren wir zunächst, basierend auf dem vorliegenden Konzept, die unterschiedlich vorhandenen Dinge (Klassen). Ein probates Mittel ist die Identifizierung von Substantiven im Text. Eigenschaften und Tätigkeiten dieser Klassen lassen wir zunächst aussen vor.

Wir identifizieren im Text die folgenden Substantive: Fußballspiel, Mannschaft, Trainer, Spieler, Feldspieler, Torwart, Tor und Schuss.

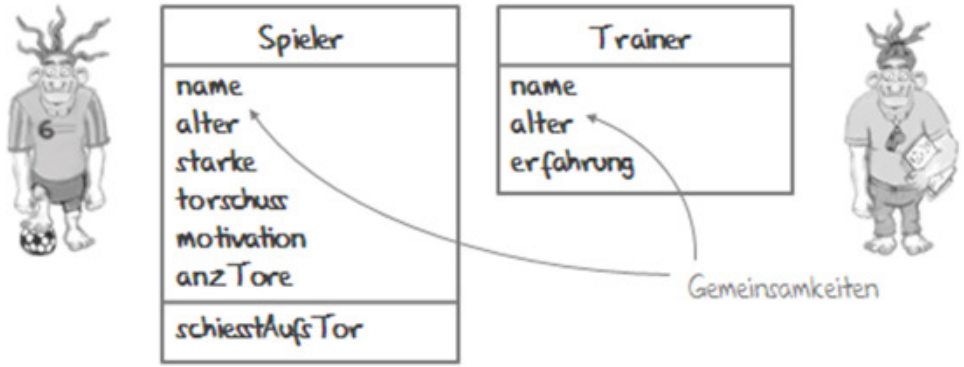
Wir haben beispielsweise elf unterschiedliche Spieler, die sich in den Werten ihrer Eigenschaften und deshalb möglicherweise im Verhalten unterscheiden. Nicht jeder wird den gleichen Torschuss abgeben können. Aber alle haben die gleichen Eigenschaften, was ein Indiz dafür ist, dass es sich um dieselbe Klasse handelt. Eine Klasse in diesem Kontext als Bauplan zu verstehen kann sehr hilfreich sein.

Für die Visualisierung von Klassen mit ihren Attributen (Eigenschaften) und Methoden (Verhalten) bietet uns die UML das Klassendiagramm:



Abstraktion und Prinzip der Geheimhaltung

Zu den Klassen Spieler und Trainer können wir jetzt die im Konzept zum Spiel beschriebenen Attribute und Methoden hinzufügen:



Spieler und Trainer besitzen gemeinsame und unterschiedliche Eigenschaften. Bevor wir mit dem Entwurf der Klassen beginnen, müssen wir zunächst das Konzept der Klassifizierung behandeln. Dabei spielen die Begriffe Generalisierung und Spezialisierung eine entscheidende Rolle.

Generalisierung und Spezialisierung

Unter den beiden Begriffen Generalisierung und Spezialisierung verstehen wir zwei verschiedene Vorgehensweisen, Kategorien und Stammbäume von Dingen zu beschreiben. Wenn wir bei Dingen Gemeinsamkeiten beschreiben und danach kategorisieren, dann nennen wir das eine Generalisierung.

Beispielsweise wurden und werden viele evolutionäre Stammbäume der Pflanzen und Tiere so aufgestellt, dass bei einer großen Übereinstimmung bestimmter biologischer Eigenschaften mit größerer Wahrscheinlichkeit ein gemeinsamer Vorfahre existiert haben muss. Dieser Vorfahre erhält einen Namen und besitzt meistens die Eigenschaften, die die direkten Nachfahren gemeinsam haben. Er ist sozusagen eine Generalisierung seiner Nachfahren. Wir erhalten auf diese Weise einen Stammbaum von den Blättern zu der Wurzel.

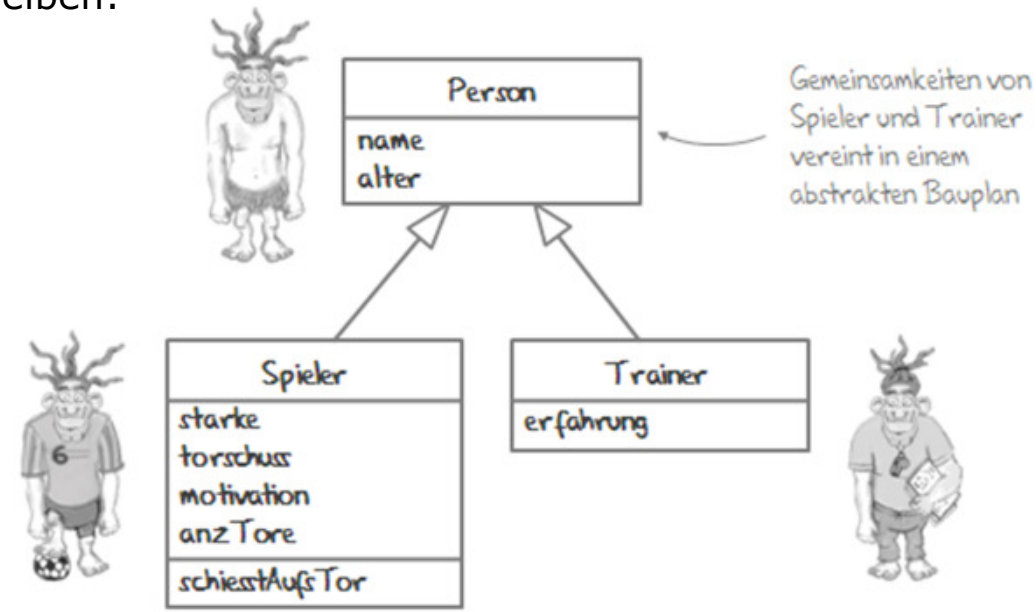
Mit der Spezialisierung beschreiben wir den umgekehrten Weg, aus einem "Ur-Ding" können wir durch zahlreiche Veränderungen der Eigenschaften neue Dinge kreieren, die Eigenschaften übernehmen oder neue entwickeln.

Generalisieren von Spieler und Trainer

Wenn wir uns die Eigenschaften der Spieler und des Trainer anschauen, dann bemerken wir die gemeinsamen Attribute name und alter. Das ist gleich eine gute Gelegenheit, die Vererbung als wichtiges Konzept der Objektorientierung kennenzulernen.

Über Vererbung lassen sich redundante Informationen besser abbilden, das verringert den Arbeitsaufwand und erhöht die Übersichtlichkeit. Zudem können Änderungen, die für alle Beteiligten relevant sind, an einer Stelle vorgenommen werden.

In diesem Fall haben wir zwei gemeinsame Attribute und können diese durch eine eigene Klasse beschreiben:



Implementierungsansatz von Person und Spieler

Wir haben für eine Person einen einfachen Bauplan mit den Eigenschaften name und alter definiert.

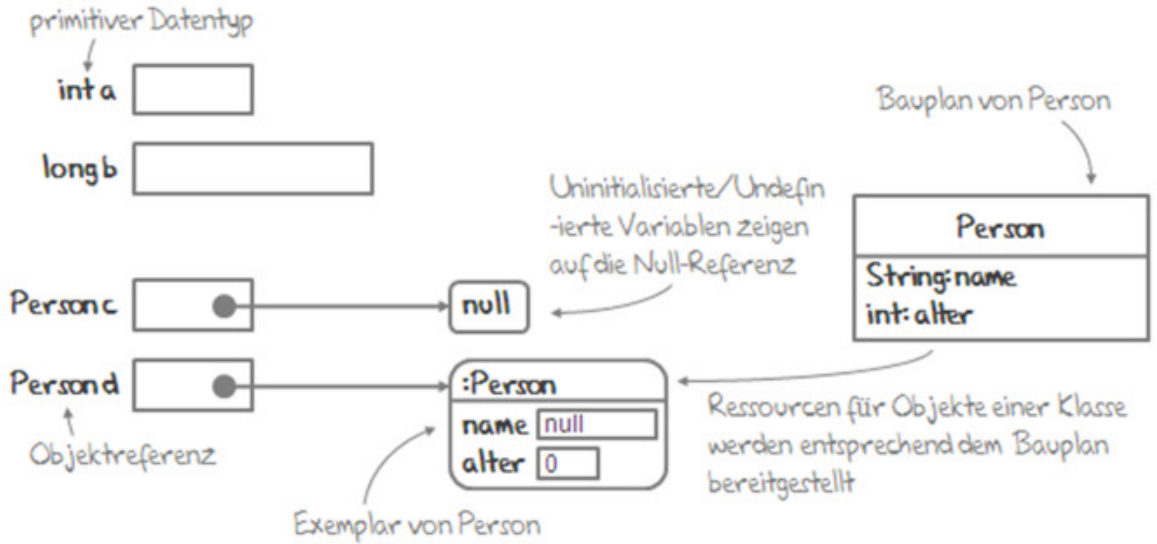
Da Java eine objektorientierte Sprache ist und das Schlüsselwort class sicherlich nicht zufällig gewählt wurde, sieht die Übersetzung dieses Bauplans ziemlich kurz aus:

```
public class Person {
   // Eigenschaften einer Person:
   public String name;
   public byte alter;
}
```

In der Klasse Person müssen wir für die Attribute name und alter mit String und byte noch entsprechende Datentypen festlegen.

Objekte und Objektreferenzen

Bei primitiven Datentypen wird entsprechend dem jeweiligen Speicherbedarf schon bei der Deklaration direkt Speicher zur Verfügung gestellt:



Links oben sehen wir beispielhaft die Deklaration zweier primitiver Datentypen, bei denen gleich der entsprechende Speicher bereitgestellt wird. Mit c haben wir eine mögliche Referenz auf ein Objekt vom Datentyp Person, die noch auf null zeigt.

Person d zeigt auf ein nach dem Bauplan Person erstelltes Objekt. Dabei wurde Speicher für die notwendigen Ressourcen erzeugt und in Person d die Referenz darauf gespeichert

Objekt erzeugen

Hier noch einmal die Klasse Person:

```
public class Person {
   // Eigenschaften einer Person:
   public String name;
   public byte alter;
}
```

Um zu zeigen, dass wir mit Hilfe des Bauplans Person jetzt konkrete Exemplare erzeugen und diese mit unterschiedlichen Werten versehen können, wollen wir in den gleichen Ordner noch eine Klasse TestKlasse erstellen. In der Testklasse gibt es eine main-Methode mit zunächst folgender Zeile:

```
Person fred;
```

Wir deklarieren hier die Variable fred vom Datentyp Person. Es handelt sich dabei um eine sogenannte Objektreferenz, die auf ein Objekt zeigen kann, das nach einem konkreten Bauplan erstellt wurde. Momentan wurde noch kein Objekt erzeugt und daher ist fred noch nicht initialisiert.

Im Gegensatz zu primitiven Datentypen steht jetzt noch kein Speicher bereit in den wir einen Wert speichern können, den müssen wir uns erst schaffen.

Deklaration allein genügt nicht

Sollten wir versuchen, den Inhalt von fred auszugeben, würde das bereits beim Kompilieren einen Fehler verursachen:

```
System.out.println("fred: "+fred);
```

Der Javacompiler teilt uns mit, dass fred noch nicht initialisiert wurde:

```
C:\Java>javac TestKlasse.java
FussballTest.java:7: variable fred might not have been initialized
System.out.println("fred: "+fred);
                           ^
1 error
```

Speicher reservieren

Jetzt wollen wir mit dem Schlüsselwort new nach dem Bauplan Person den notwendigen Speicher reservieren. Der Speicher wird initialisiert und eine Referenz darauf in fred gespeichert:

```
Person fred;
fred = new Person();
```

Deklaration und Speicherreservierung lassen sich syntaktisch auch in einer Zeile zusammenfassen:

<Datentyp> <Name> = new <Datentyp>();

Wir haben mit fred eine Referenz auf ein konkretes Exemplar der Klasse Person erzeugt.

Synonyme für Exemplar sind beispielsweise Objekt oder Instanz, wobei Instanz nur eine schlechte Übersetzung aus dem Englischen ist.

Attribute eines Objekts

Jetzt können wir uns sowohl die Referenz auf dem Speicher als auch die initialisierten Inhalte der Attribute anzeigen lassen:

```
System.out.println("fred: "+fred+" Alter: "+fred.alter+" Name: "+fred.name);
```

Wir sehen, dass über die folgende Syntax

<Referenz>.<Attribut>

auf die jeweiligen Attribute zugegriffen werden kann.

Die Konsole liefert zu unserem Beispiel die folgende Ausgabe:

```
C:\Java>java TestKlasse
fred: Person@19821f Alter: 0 Name: null
```

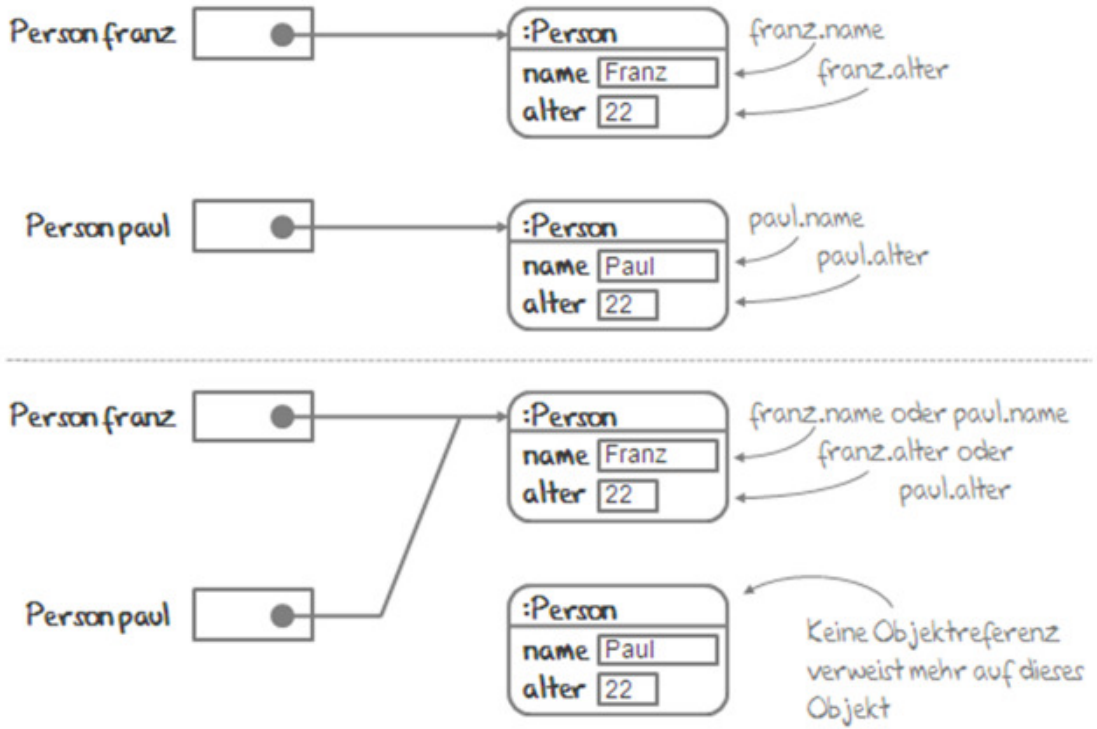
Wir können die Attribute auch verändern und diesen neue Werte zuweisen:

```
fred.alter = 10;
fred.name  = "Fred";
```

Auch hier müssen wir wieder die Objektreferenz angeben, damit wir einzelne Objekte unterscheiden können.

Zugriff auf Attribute

Oben: Zwei Objekte vom Typ Person wurden erstellt. Die entsprechenden Objektreferenzen zeigen auf die Objekte im Speicher.

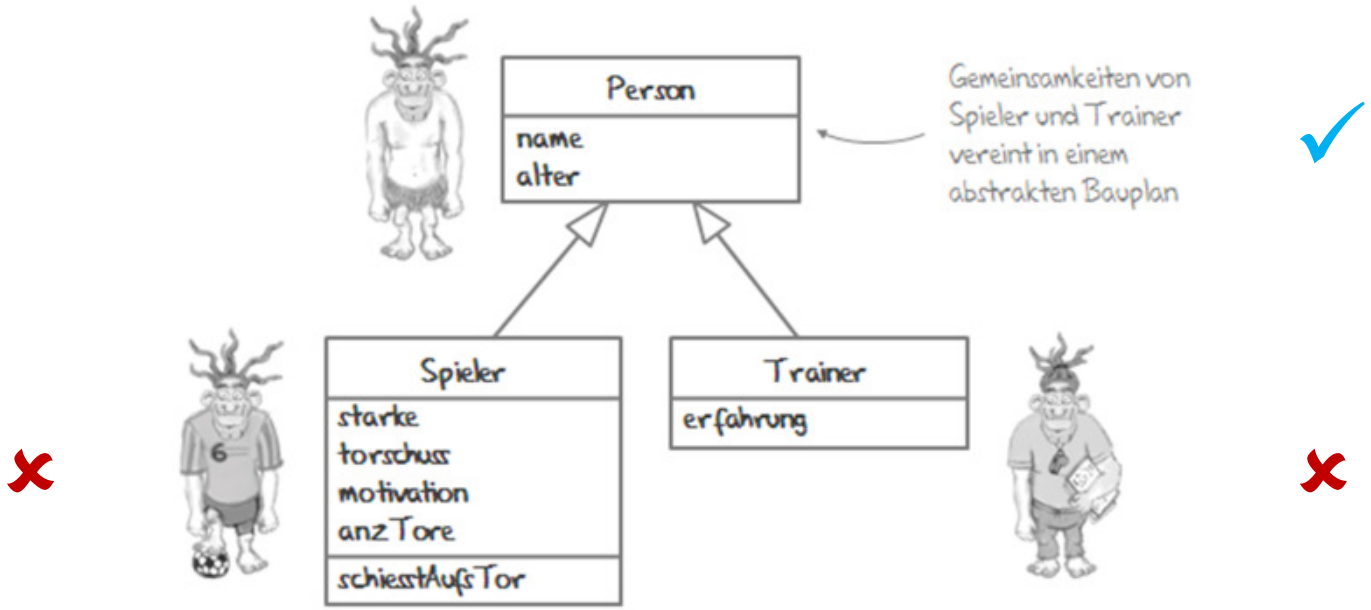


Unten wurde in der Objektreferenz paul der Inhalt von Objektreferenz franz gespeichert, so dass jetzt beide Objektreferenzen auf das gleiche Objekt zeigen.

Live-Coding-Session

Zurück zum Fußballmanager

Person ist bereits fertig, aber das wollten wir realisieren:



Spieler als abgeleitete Klasse von Person

Jetzt wollen wir mit der weiteren Umsetzung in Java fortfahren und die Klasse Spieler von Person ableiten.

Damit werden alle Attribute übernommen, die in der Klasse Person vorkommen. Wir bezeichnen Person dann als Basisklasse (bzw. Oberklasse oder Superklasse) von Spieler und Spieler als abgeleitete Klasse (bzw. Unterklasse oder Subklasse) von Person.

In Java erweitern wir die Definition einer Klasse mit dem Befehl extends und den Namen der Superklasse, von der wir ableiten wollen:

```
public class <Klassenname> extends <Basisklassenname> {
}
```

Das setzen wir gleich für die Klasse Spieler um und leiten von Person ab. Hinzu kommen neue Attribute mit den entsprechenden Datentyen:

```
public class Spieler extends Person {
   // Zusätzliche Eigenschaften eines Spielers:
   public byte staerke;      // von 1 (schlecht) bis 10 (weltklasse)
   public byte torschuss;    // von 1 (schlecht) bis 10 (weltklasse)
   public byte motivation;   // von 1 (schlecht) bis 10 (weltklasse)
   public int  tore;
}
```

Vererbung in Java ist kein Zauber

Das war schon alles!

Jetzt haben wir zwei Klassen Spieler und Person. Alle Attribute der Klasse Person sind nun auch in Spieler enthalten, darüber hinaus hat ein Spieler noch die Attribute staerke, torschuss, motivation und tore.

Mit diesen beiden Klassen haben wir die erste einfache Vererbung realisiert.

Zu viel ist sichtbar

Nehmen wir an, dass diese zwei Klassen von unterschiedlichen Programmierern geschrieben wurden und dass der Programmierer der Klasse Person für die Variable alter nur positive Zahlen zwischen 1 und 100 akzeptieren möchte.

Momentan hat er aber keinen Einfluss auf die Werte, da die Variable alter mit dem Modifizierer public versehen wurde. Das bedeutet, dass jeder, der diese Klasse verwenden möchte auf diese Attribute uneingeschränkt zugreifen kann.

Geheimnisprinzip durch Datenkapselung

Es gibt eine Möglichkeit diese Attribute vor einem direkten Zugriff zu schützen, indem der Modifizierer private verwendet wird.

Jetzt kann diese Variable von außerhalb der Klasse nicht mehr angesprochen werden. Um die beiden Variablen trotzdem auslesen und verändern zu können, schreiben wir jeweils zwei Methoden und geben diesen das Attribut public:

```
public class Person {
   // Eigenschaften einer Person:
   private String name;           // von außen nicht sichtbar
   private byte alter;            // von außen nicht sichtbar

   // get- und set-Methoden:
   public String getName(){       // von außen sichtbar
      return name;
   }

   public void setName(String n){ // von außen sichtbar
      name = n;
   }

   public byte getAlter(){        // von außen sichtbar
      return alter;
   }

   public void setAlter(int a) {  // von außen sichtbar
      alter = (byte)a;
   }
}
```

Mit der get-Methode lesen wir den Inhalt eines Klassenattributs aus und mit der set-Methode verändern wir diesen.

Methoden für Zugriff auf Attribute verwenden

Um beispielsweise die Variablen name und alter wie im ersten Beispiel zu ändern, verwenden wir jetzt die set-Methoden:

```
Person fred = new Person();
fred.setAlter(10);
fred.setName("Fred");
```

Die Methode setAlter kümmert sich um die Typumwandlung zu byte, damit wir bei das nicht bei der Eingabe übernehmen müssen. Sollten wir in der Methode den Datentyp byte erwarten, müssten wir die 10 bei der Übergabe explizit casten:

```
fred.setAlter((byte)10);
```

Der Zugriffsumweg auf die Attribute über die Funktionen, also das Abkapseln der Inhalte von der Umwelt, ist eines der Grundprinzipien der objektorientierten Programmierung und wird als Datenkapselung oder Geheimnisprinzip bezeichnet.

Zum Einen sollen nicht nachvollziehbare Änderungen der Attribute von außen unterbunden und zum Anderen die Implementierungsdetails verborgen werden.

Die Klasse Trainer

Der Trainer ist von Person abgeleitet und verfügt über das zusätzliche Attribut erfahrung.

Unter Einhaltung des Geheimnisprinzips können wir die Klasse Trainer wie folgt ableiten und implementieren:

```
public class Trainer extends Person {
   // Eigenschaften eines Trainers:
   private byte erfahrung;       // von 1 (schlecht) bis 10 (weltklasse)

   // get- und set-Methoden:
   public byte getErfahrung(){
      return erfahrung;
   }

   public void setErfahrung(int e){
      erfahrung = (byte)e;
   }
}
```

Exemplar der Klasse Trainer erzeugen

Durch die Vererbung besitzt die Klasse Trainer, auch wenn wir es hier nicht sehen, alle sichtbaren Methoden von Person. Wenn wir in einer Testklasse beispielsweise die erfahrene Fußballbundestrainerin Silvia Neid (aktuell 47 Jahre alt) als Exemplar der Klasse Trainer erzeugen wollen, dann machen wir das bislang so:

```
Trainer neid = new Trainer();
neid.setName("Silvia Neid");
neid.setAlter(47);
neid.setErfahrung(8);
```

Wir sehen schon, dass die Funktionen setName und setAlter aus der Klasse Person verwendet werden können, das verdanken wir dem Vererbungskonzept.

Komfort durch Konstruktoren

Mit Konstruktoren geht es bedeutend bequemer. Sie sind quasi die Methoden, die bei der Reservierung des Speicherplatzes, also bei der Erzeugung eines Objekts, ausgeführt werden. Wir dürfen hier auch wieder Parameter übergeben, es gibt allerdings keinen Rückgabewert.

Ein Konstruktor hat zunächst die folgende Syntax:

```
public <Klassenname>(<Parameterliste>) {
}
```

Der Torwart kann mit Werten für die drei vorhandenen Attribute name, alter und erfahrung erzeugt werden. Konstruktoren stehen laut Konvention nach den Klassenattributen und vor den Funktionen:

```
public class Trainer extends Person {
   // Klassenattribute
   ...

   // Konstruktor
   public Trainer(String n, int a, int e) {
      ...
   }

   // get- und set-Methoden:
   ...
}
```

Konstruktor in Klasse Person einfügen

Bevor wir den Konstruktor von der Klasse Trainer fertigstellen, sollten wir noch einmal zur Klasse Person zurückkommen und auch dort einen Konstruktor einfügen:

```
public Person(String n, int a) {
   name  = n;
   alter = (byte)a;
}
```

Wenn ein Objekt der Klasse Person mit Hilfe dieses Konstruktors erzeugt wird, dann setzen wir den Namen des Objekts und das Alter entsprechend auf die Werte der Eingabeparameter.

Jetzt ist die Objekterzeugung etwas bequemer:

```
Person fred = new Person("Fred", 10);
```

Konstruktor in Klasse Trainer einfügen

Da Trainer von Person abgeleitet ist, können wir mit dem Schlüsselwort super auf diesen Konstruktor zugreifen. Dazu erweitern wir die Klasse Trainer um diesen Konstruktor:

```
public Trainer(String n, int a, int e) {
   super(n, a);
   erfahrung = (byte)e;
}
```

Die Anweisung super mit den Parametern name und alter ruft den Konstruktor der Klasse auf, von der geerbt wird.

Den ersten Versuch, mit vier Zeilen ein Exemplar der Klasse Trainer zu erzeugen und mit Werten zu füllen, können wir jetzt mit folgender Zeile verkürzt schreiben:

```
Trainer neid = new Trainer("Silvia Neid", 47, 8);
```

Klasse Spieler mit Konstruktor

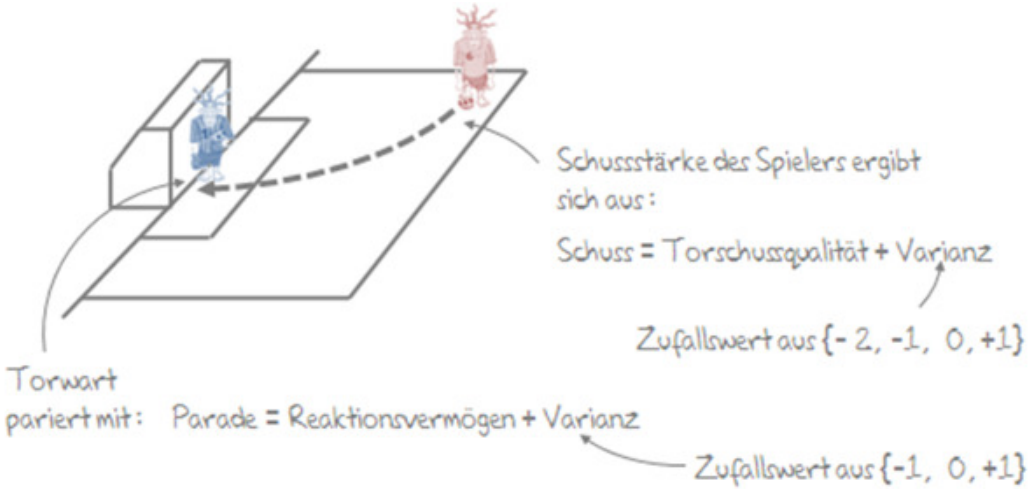
Analog zu Person und Trainer wollen wir jetzt die Klasse Spieler anlegen. Dazu erben wir wieder von Person, legen die Attribute und deren get- und set-Methoden an:

```
public class Spieler extends Person {
   private byte staerke;    // von 1 (schlecht) bis 10 (weltklasse)
   private byte torschuss;  // von 1 (schlecht) bis 10 (weltklasse)
   private byte motivation; // von 1 (schlecht) bis 10 (weltklasse)
   private int  tore;

   // Konstruktor
   public Spieler(String n, int a, int s, int t, int m) {
      super(n, a);
      staerke    = (byte)s;
      torschuss  = (byte)t;
      motivation = (byte)m;
      tore       = 0;
   }

   // get- und set-Methoden:
   public byte getStaerke() {
      return staerke;
   }

   public void setStaerke(int s) {
      staerke = (byte)s;
   }

   // analog: getTorschuss(), setTorschuss(int t), getMotivation(), setMotivation(int m), addTor()
}
```

Aus dem Klassendiagramm entnehmen wir noch, dass die Klasse eine Methode schiesstAufsTor hat.

Auf das Tor schiessen

Wir erinnern uns:



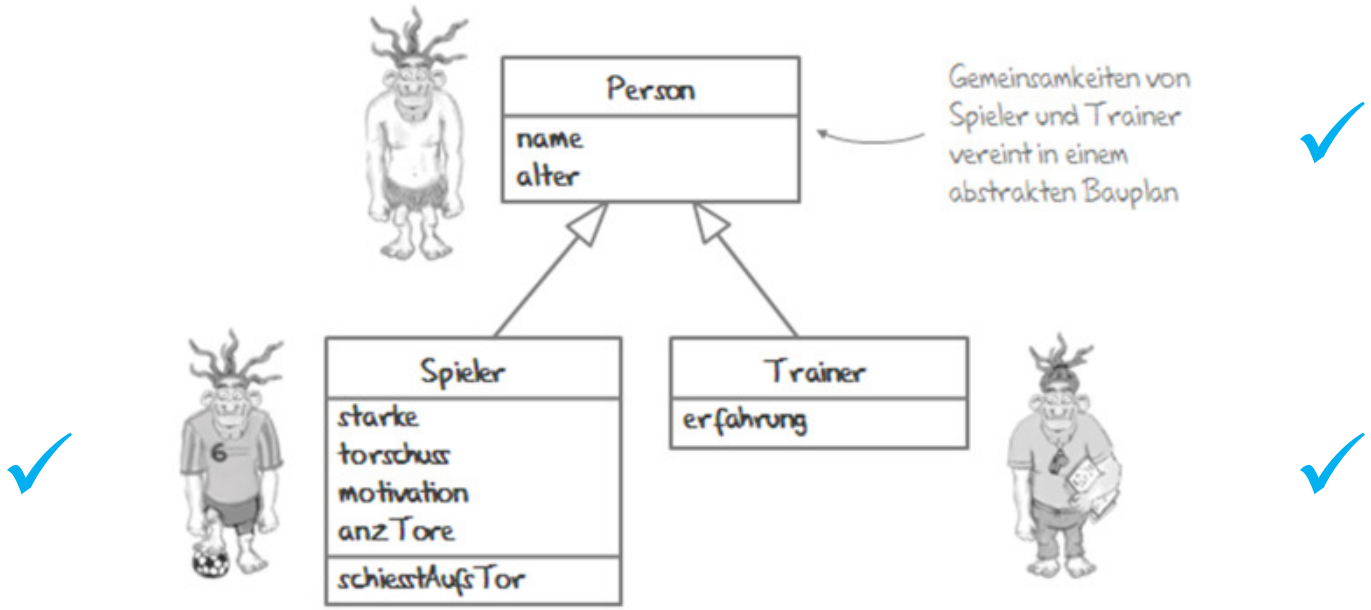
Jetzt wollen wir diese Methode umsetzen:

```
public int schiesstAufsTor() {
   int varianz   = (int)(Math.random()*4) - 2; // [+1,0,-1,-2]
   int aktSchuss = torschuss + varianz;

   // anschließend die Intervallränder 1 und 10 wieder prüfen
   aktSchuss = Math.min(10, aktSchuss);  // oberer Bereich geprüft
   aktSchuss = Math.max( 1, aktSchuss);  // unterer Bereich geprüft

   return aktSchuss;
}
```

In Abhängigkeit zur Torschussqualität des Spielers wird mit einem kleinen Zufallswert (kann um [+1,0,-1,-2] abweichen) ein Schuss aus dem Intervall [1, 10] erzeugt. Anschließend werden die Intervallränder überprüft.

Zurück zum Fußballmanager

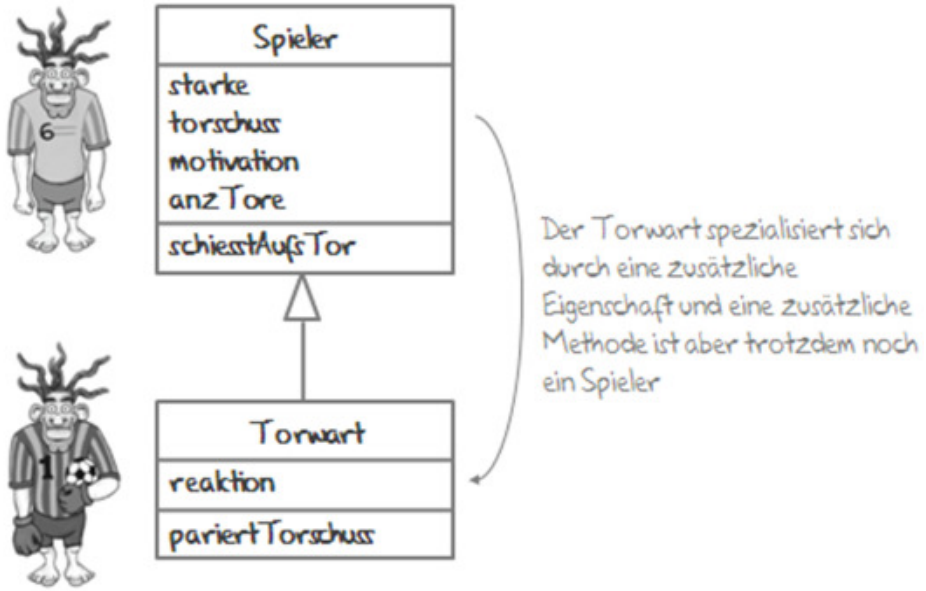
Person, Spieler und Trainer sind bereits fertig:



Was fehlt hier noch?

Spezialisierung des Torwarts

Bei den Spielern hat der Torwart eine Sonderrolle, denn er stellt durch die zusätzliche Eigenschaft Reaktionsvermögen eine Spezialisierung innerhalb der Spieler dar:



Der Torwart ist eine abgeleitete Klasse von Spieler und verfügt über das Attribut reaktion, dass das Reaktionsvermögen darstellen soll.

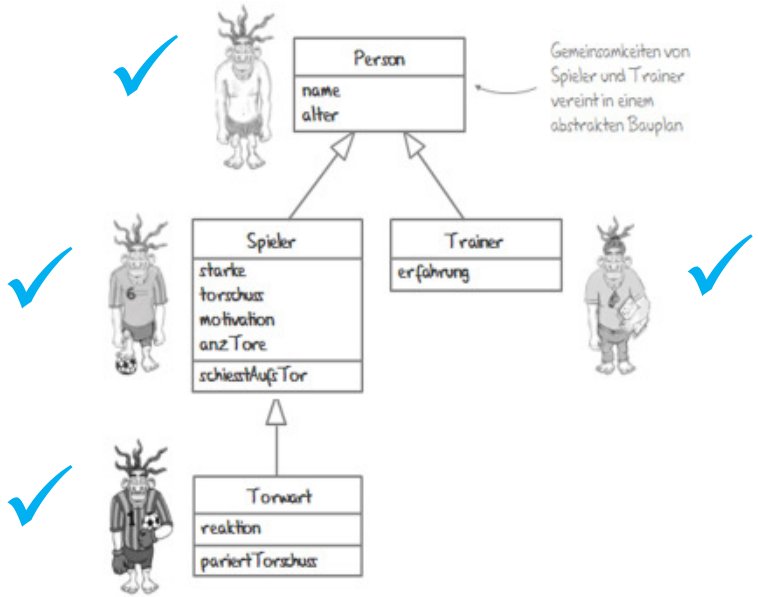
Als zusätzliche Funktion verfügt der Torwart über die Methode pariertTorschuss, die in Abhängigkeit der Torschussstärke und dem Reaktionsvermögen die Entscheidung trifft, ob der Ball gehalten wird

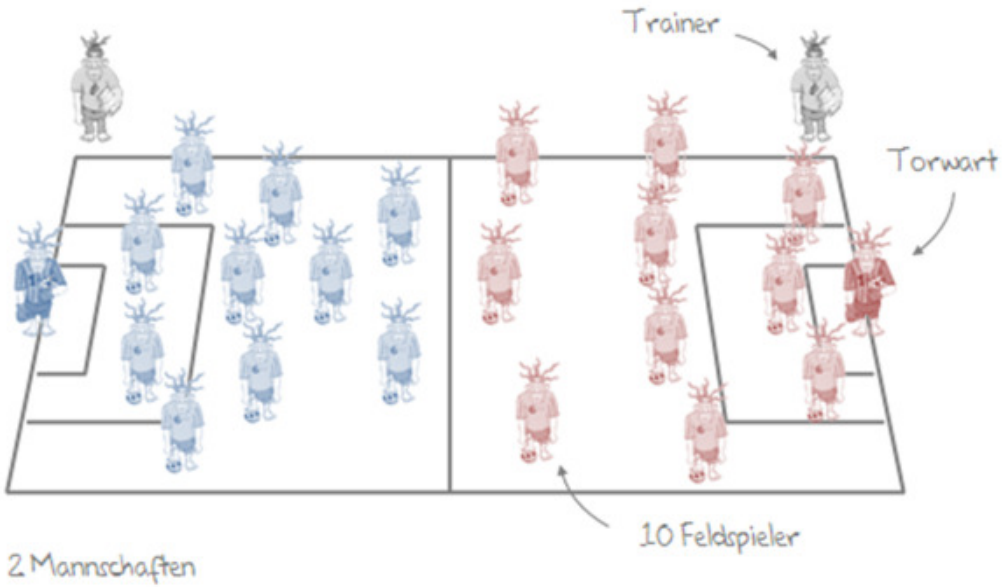
Torwart implementieren

Die Implementierung des Klasse Torwart ist jetzt nicht mehr schwierig:

```
public class Torwart extends Spieler {
   private byte reaktion;        // von 1 (schlecht) bis 10 (weltklasse)

   // Konstruktor
   public Torwart(String n, int a, int s, int t, int m, int r) {
      super(n, a, s, t, m);
      reaktion = (byte)r;
   }

   // get- und set-Methoden:
   public byte getReaktion() {
      return reaktion;
   }

   public void setReaktion(int r) {
      reaktion = (byte)r;
   }

   // Torwartfunktion:
   public boolean pariertTorschuss(int aktSchuss) {
      int varianz     = (int)(Math.random()*3) - 1; // [+1,0,-1]
      int aktReaktion = reaktion + varianz;

      return (aktReaktion >= aktSchuss);
   }
}
```

Spieler, Torwart und Trainer





Die Mannschaft I

Wir haben die Spieler und den Trainer bereits fertiggestellt, jetzt ist es an der Zeit eine Mannschaft zu beschreiben.

Eine Mannschaft ist eine Klasse mit den Eigenschaften name, Trainer, Torwart und Spieler. Es gibt wieder einen Konstruktor und die entsprechenden get-set-Funktionen:

```
public class Mannschaft{
   // Eigenschaften einer Mannschaft:
   private String name;
   private Trainer trainer;
   private Torwart torwart;
   private Spieler[] kader;

   // Konstruktoren
   public Mannschaft(String n, Trainer t, Torwart tw, Spieler[] s){
      name            = n;
      trainer         = t;
      torwart         = tw;
      kader           = s;
   }

   // get- und set-Methoden:
   ...
```

Die Mannschaft II

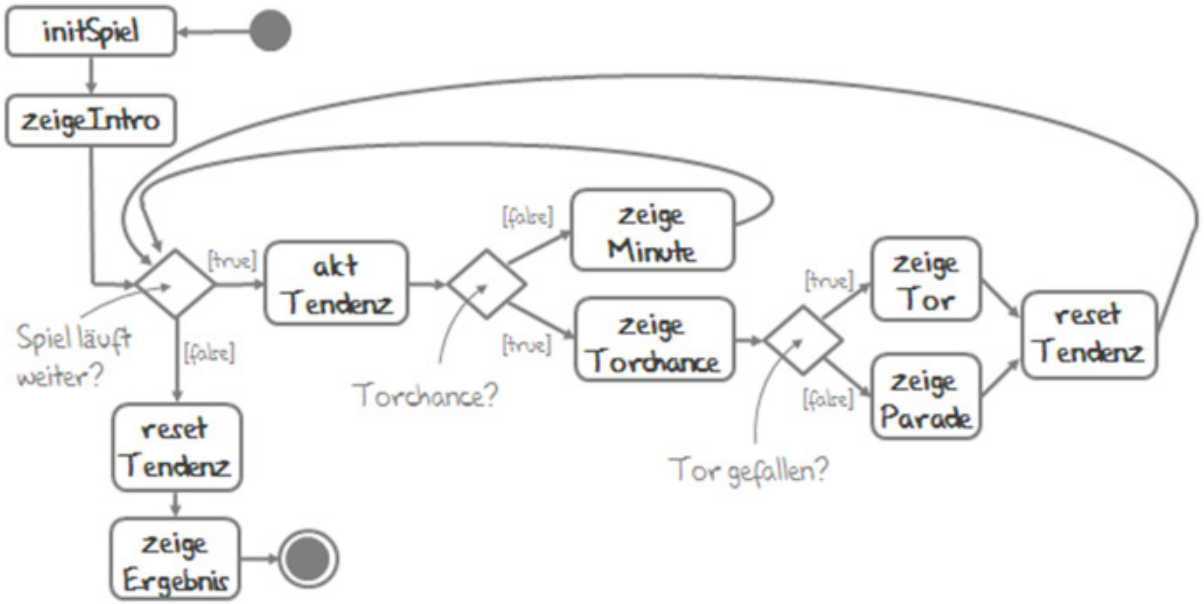
Durchschnittliche Stärke und Motivation werden ermittelt:

```
   ...
   // Mannschaftsfunktionen:
   // liefert die durchschnittliche Mannschaftsstaerke
   public double getStaerke() {
      int summ = torwart.getStaerke();
      for (Spieler s : feldspieler)
         summ += s.getStaerke();
      return summ / 11.0;
   }

   // liefert die durchschnittliche Mannschaftsmotivation
   public double getMotivation() {
      int summ = torwart.getMotivation();
         for (Spieler s : feldspieler)
            summ += s.getMotivation();
      return summ / 11.0;
   }
}
```

Live-Coding-Session

Das Fußball-Freundschaftsspiel - Der Entwurf

Zunächst werden wir uns ein wenig Gedanken zu dem Spielablauf und den notwendigen Klassen machen. Dabei lernen wir gleich ein wichtiges Konzept für den Entwurf von Programmen kennen:



Die Spielparameter werden initialisiert und ein Intro als Vorbereitung zum Spiel ausgegeben. Solange das Spiel läuft, wird die Tendenz zu einer Torchance ermittelt. Schlägt die Tendenz zu Gunsten keiner der beiden Mannschaften, wird die aktuelle Spielminute angezeigt und fortgefahren.

Bei einer aussagekräftigen Tendenz wird eine Torchance angezeigt und falls diese erfolgreich ist, das Tor angezeigt, ansonsten die Parade gezeigt. Nach einer Torchance wird die Tendenz wieder auf den Stardwert zurückgesetzt. Ist das Spiel vorbei, setzen wir ebenfalls die Tendenz zurück und zeigen das Spielergebnis an

Parametrisierte Pausefunktion

Damit wir die einzelnen Spielereignisse mitverfolgen können, werden wir kleine Pausen einbauen. Wir haben bereits ein Beispiel gesehen, mit dem wir Programmpausen realisieren können.

Da wir an mehreren Programmstellen Pausen machen werden und diese möglicherweise unterschiedlich lang, bietet es sich hier an, eine Funktion bereitzustellen:

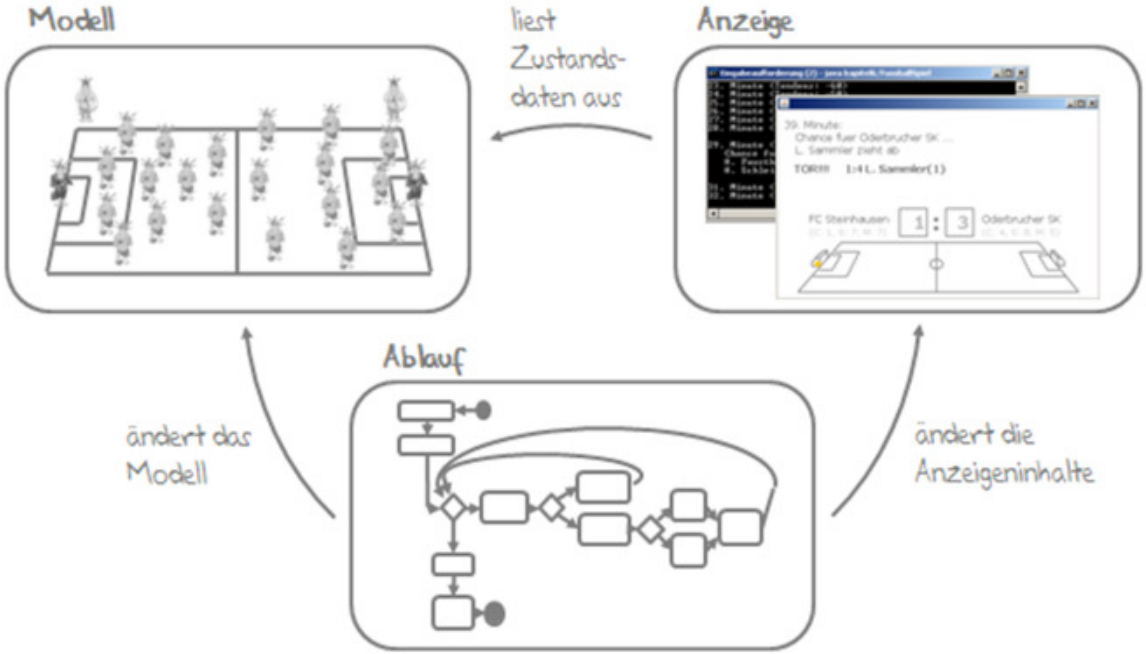
```
public static void warte(int ms) {
   try {
      Thread.sleep(ms);
   } catch(InterruptedException e) {
      e.printStackTrace();
   }
}
```

Modularisierung in Ablauf, Modell und Anzeige

Wir haben viele kleine Programmabschnitte im Aktivitätsdiagramm definiert und haben jetzt eine grobe Vorstellung von der Funktionsweise des Programms. Jetzt müssen wir diese in Klassen unterbringen.

Eine Möglichkeit wäre sicherlich, alle Methoden in einer Klasse Fussballspiel abzulegen, aber aus Entwurfssicht ist das keine gute Lösung.

Eine bessere Lösung ist sicherlich:



Das gewählte Entwurfkonzept ist auf Grund der fehlenden Interaktion mit dem Benutzer eine vereinfachte Variante des Model-View-Controller-Ansatzes (MVC).

Implementierung des Ablaufs I

Fangen wir in top-down-Manier damit an, zunächst den Ablauf zu implementieren und gehen davon aus, dass Anzeige und Modell entsprechende Funktionen bereitstellen:

```
import java.io.FileNotFoundException;

public class Fussballspiel {
   private FussballspielModell   modell  = null;
   private FussballspielAnzeige  anzeige = null;

   public FussballspielStandard() {
      modell  = new FussballspielModell();
      anzeige = new FussballspielAnzeigeKonsole(modell);
   }

   public static void warte(int ms) {
      try {
         Thread.sleep(ms);
      } catch(InterruptedException e) {
         e.printStackTrace();
      }
   }
   ...
```

Implementierung des Ablaufs II

weiter gehts:

```
...
public void starteSpiel(String m1, String m2) throws FileNotFoundException {
   modell.initSpiel(m1, m2);
   anzeige.zeigeIntro();
   warte(1000);

   // gameloop
   while (modell.getSpielLaeuft()) {
      if (modell.aktTendenz()) {
         anzeige.zeigeTorchance();
         warte(1500);
         if (modell.torschussErfolgreich())
            anzeige.zeigeTor();
         else
            anzeige.zeigeParade();
         warte(1500);
         modell.resetTendenz();
      } else
         anzeige.zeigeMinute();
      warte(800);
   }
   warte(1500);
   modell.resetTendenz();
   anzeige.zeigeErgebnis();
}
...
```

Implementierung des Ablaufs III

und schließlich:

```
   ...
   public static void main(String[] args) {
      Fussballspiel fc = new Fussballspiel();

      try {
         fc.starteSpiel("steinhausen.txt", "oderbruch.txt");
      } catch(FileNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

In der main-Funktion wird ein Fussballspiel erstellt. Die Funktion starteSpiel ist das Herzstück dieser Klasse und repräsentiert eine Umsetzung der Programmschleife.

Bedingung für einen Spielstart ist das erfolgreiche Einlesen zweier Mannschaften. Sollte es Probleme beim Einlesen geben, wird ein Fehler geworfen und ausgegeben.

Implementierung des Modells I

In dem Modell wollen wir die aktuellen Spieldaten zum Schreiben und Lesen bereithalten:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FussballspielModell {
   private Mannschaft heim, gast;
   private byte toreHeim, torSchuesseHeim, toreGast, torSchuesseGast, spieldauer, aktZeit, tendenz;
   private final byte TENDENZ_MAX = 10;
   private boolean spielLaeuft;

   private Mannschaft aktMannschaft;
   private Spieler    aktSpieler;
   private Torwart    aktTorwart;

   public void initSpiel(String m1, String m2) throws FileNotFoundException {
      heim = liesMannschaft(m1);
      gast = liesMannschaft(m2);

      toreHeim        = 0;      toreGast        = 0;
      torSchuesseHeim = 0;      torSchuesseGast = 0;
      aktMannschaft   = null;
      aktSpieler      = null;
      aktTorwart      = null;
      spielLaeuft     = false;
      spieldauer      = 0;      tendenz         = 0;

      spielLaeuft     = true;
      // 90 Minuten + mögliche Nachspielzeit
      spieldauer      = (byte)(90 + (byte)(Math.random()*5));
   }
   ...
```

Implementierung des Modells II

Es folgen für alle Attribute entsprechende get-Methoden, die später vom Spielablauf und der Anzeige ausgelesen werden können.

Der Parameter tendenz wird mit 0 initialisiert, pro Minute aktualisiert und repräsentiert die aktuelle Torchancetendenz für beide Mannschaften. Ist der Wert positiv, dann steigt die Wahrscheinlichkeit für die Gastmannschaft. Bei einem negativen Wert, steigt die Wahrscheinlichkeit entsprechend für die Heimmannschaft. Ist der Wert größer TENDENZ_MAX oder kleiner -TENDENZ_MAX, erhält die jeweilige Mannschaft eine Torchance.

Für die visuelle Auswertung liefert die Funktion getTendenz in Anhängigkeit zur Konstanten TENDENZ_MAX die prozentuelle Tendenz:

```
public byte getTendenz() {
   byte val = (byte)(Math.min(TENDENZ_MAX,
                    Math.max(-TENDENZ_MAX, tendenz)));
   return (byte)((val*100.0)/TENDENZ_MAX);
}
```

Da die tendenz zwischenzeitlich ausserhalb der Grenzen liegen kann, werden diese in val überprüft. Nach jeder Torchance wird die tendenz mit resetTendenz wieder auf 0 gesetzt.

Implementierung des Modells III

Beginnen wir jetzt mit dem Einlesen der Mannschaftsdaten. Hier sehen wir beispielsweise den Inhalt der Datei steinhausen.txt:

```
C:\>type steinhausen.txt
FC Steinhausen
Paul Steinwerfer, 39, 7
H. Schleifer, 22, 8, 1, 9, 7
A. Baumfaeller, 23, 9, 5, 9
F. Feuerstein, 25, 8, 2, 7
P. Knochenbrecher, 22, 9, 2, 8
M. Holzkopf, 29, 7, 5, 8
B. Geroellheimer, 26, 9, 8, 9
D. Bogenbauer, 22, 7, 5, 8
B. Schnitzer, 22, 2, 3, 2
L. Schiesser, 21, 7, 8, 9
M. Klotz, 28, 10, 9, 7
O. Mammut, 33, 8, 8, 7
```

Die vorliegende Struktur der Daten ist allerdings etwas komplizierter als unser Einstiegsbeispiel und daher ist es notwendig, die einzelnen Zeilen in ihre Wortbestandteile zu zerlegen.

Einlesen und Zerlegen von Text

```
private Mannschaft liesMannschaft(String dateiName) throws FileNotFoundException {
   Scanner scanner = new Scanner(new File(dateiName));
   String zeile = null;
   int anzZeilen = 0;
   String name  = null;
   Trainer t    = null;
   Torwart tw   = null;
   Spieler[] sp = new Spieler[10];

   while (scanner.hasNextLine()) {
      zeile = scanner.nextLine();
      anzZeilen++;

      if (anzZeilen == 1) {      // Mannschaftsname auslesen
         name = zeile;
         continue;
      }

      // Trainer, Torwart und Spieler auslesen
      String[] worte = zeile.split(", ");
      if (anzZeilen == 2) {                  // Trainer einlesen
         t = new Trainer(worte[0], Integer.parseInt(worte[1]), Integer.parseInt(worte[2]));
      } else if (anzZeilen == 3) {           // Torwart einlesen
         tw = new Torwart(worte[0], Integer.parseInt(worte[1]), Integer.parseInt(worte[2]),
                          Integer.parseInt(worte[3]), Integer.parseInt(worte[4]),
                          Integer.parseInt(worte[5]));
      } else {                               // Spieler einlesen
         sp[anzZeilen-4] = new Spieler(worte[0], Integer.parseInt(worte[1]),
                          Integer.parseInt(worte[2]), Integer.parseInt(worte[3]),
                          Integer.parseInt(worte[4]));
      }
   }
   scanner.close();
   return new Mannschaft(name, t, tw, sp);
}
```

Torschussverhalten

Das Programmverhalten bei einem Torschuss wird durch die Funktion torschussErfolgreich ermittelt:

```
public boolean torschussErfolgreich() {
   if (!aktTorwart.pariertTorschuss(aktSpieler.schiesstAufsTor())) {
      aktSpieler.addTor();
      if (aktMannschaft == heim)
         toreHeim++;
      else
         toreGast++;
      aktZeit+=2;
      return true;
   }
   aktZeit++;
   return false;
}
```

Der aktuell ausgewählte Spieler gibt einen Torschuss ab und der gegnerische Torwart versucht den Torschuss zu parieren. Kann der Torwart den Ball nicht halten, schreiben wir dem Schützen und der Mannschaft ein Tor gut.

Wir berechnen noch pauschal zwei Minuten für die Torfeier samt Aufstellung zum Wiederanstoss. Wird das Tor nicht getroffen, zählen wir nur eine Minute dazu.

Torchancetendenz I

Kommen wir zu der wichtigsten Funktion, die entscheiden wird, zu Gunsten welcher Mannschaft sich die Torchancetendenz verändern wird:

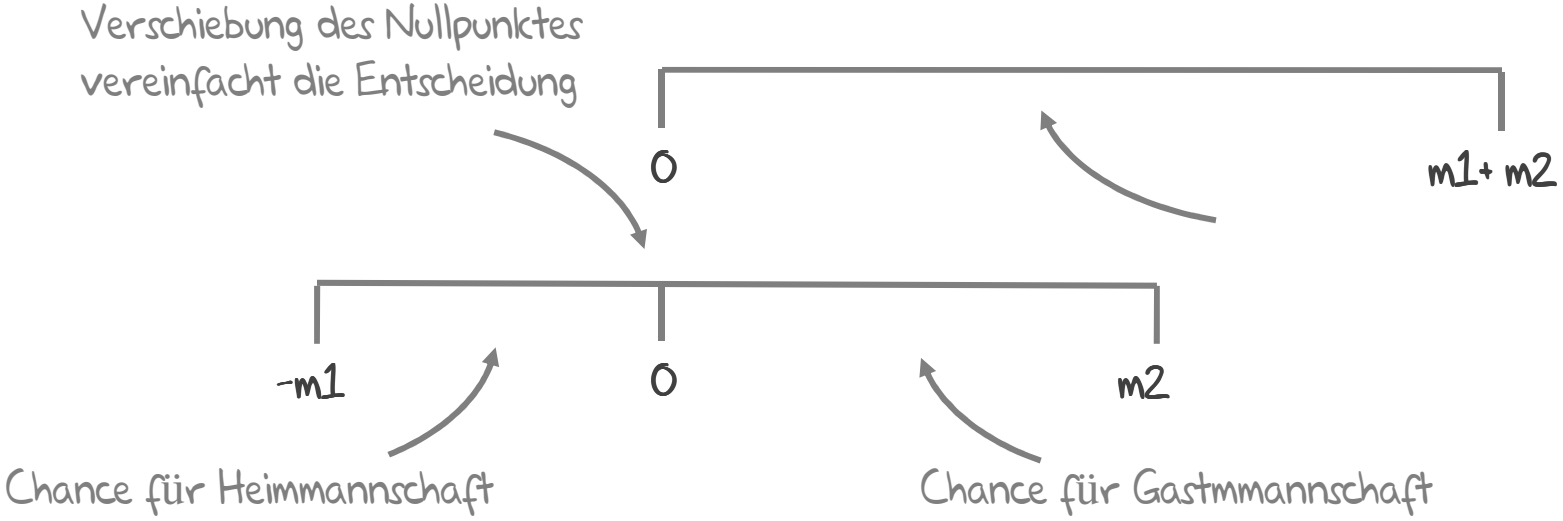
```
public boolean aktTendenz() {
     aktZeit++;
     if (aktZeit >= spieldauer) {
        spielLaeuft = false;
        return false;
     }

     double m1 = 0.8  * heim.getStaerke()    + 0.15 * heim.getMotivation() +
                 0.05 * heim.getTrainer().getErfahrung();
     double m2 = 0.8  * gast.getStaerke()    + 0.15 * gast.getMotivation() +
                 0.05 * gast.getTrainer().getErfahrung();
     tendenz += (byte)(Math.random() * (m1+m2) - m1);

     if (tendenz<=-TENDENZ_MAX) {
        // Heimmannschaft erhält die Torschussmöglichkeit
        aktMannschaft = heim;
        aktSpieler    = heim.getFeldspieler()[(int)(Math.random()*10)];
        aktTorwart    = gast.getTorwart();
        torSchuesseHeim++;
        return true;
     } else if (tendenz>=TENDENZ_MAX) {
        // Gastmannschaft erhält die Torschussmöglichkeit
        aktMannschaft = gast;
        aktSpieler    = gast.getFeldspieler()[(int)(Math.random()*10)];
        aktTorwart    = heim.getTorwart();
        torSchuesseGast++;
        return true;
     }
     return false;
  }
}
```

Torchancetendenz II

Solange wir noch innerhalb der Spielzeit sind, errechnen wir mit m1 und m2 für beiden Mannschaften jeweils einen Wert, der die aktuelle Stärke wiederspiegeln soll.

Dieser ergibt sich zu 80% aus der durchschnittlichen Spielerstärke, zu 15% aus der durchschnittlichen Spielermotivation und zu 5% aus der Erfahrung des Trainers.

Wir ermitteln jetzt eine Zufallszahl aus dem Intervall [0, m1+m2] und verschieben den Nullpunkt anschließend um m1 nach rechts:

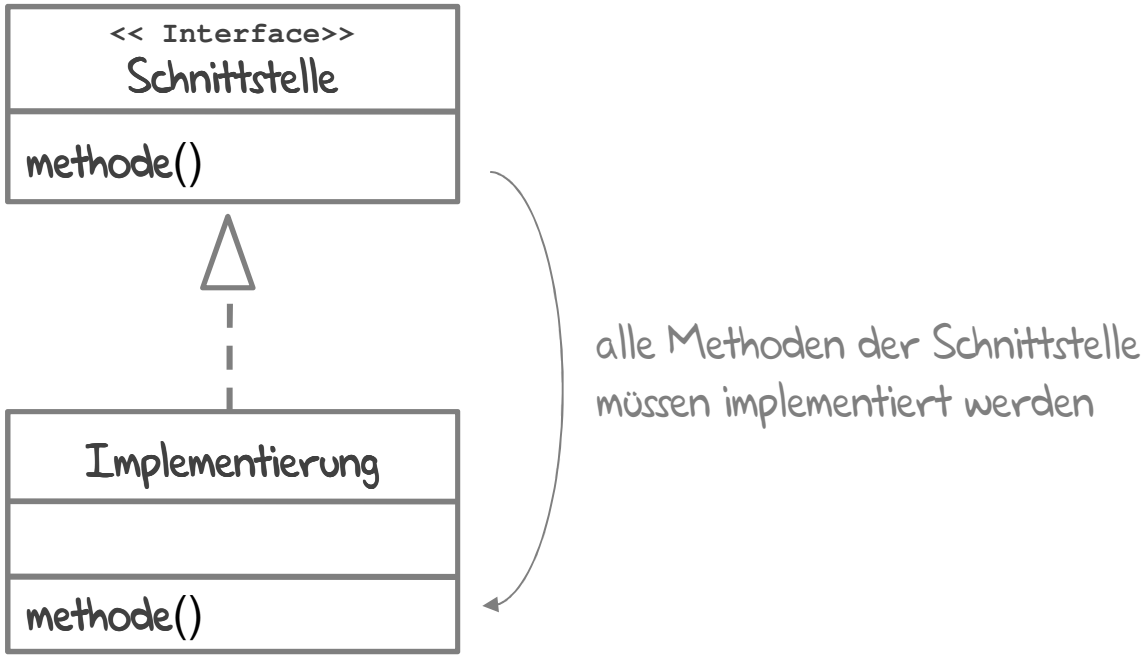


Jetzt genügt es zu überprüfen, ob die Zufallszahl kleiner oder größer 0 ist. Je dominierender eine Mannschaft ist, desto wahrscheinlicher ist das Auftreten der Zufallszahl in dem größeren Intervallbereich.

Konsolenvisualisierung über ein Interface

Vorher wurde erwähnt, dass es sinnvoll ist, die Visualisierung vom Rest des Projekts zu trennen, damit dieser Teil einfacher austauschbar ist. Eine Möglichkeit wäre, dass wir uns auf dem Papier notieren, welche Methoden mit entsprechenden Signaturen eine Anzeige enthalten muss, damit wir diese austauschen können.

Glücklicherweise gibt es in Java den Vererbungsmechanismus Interface, mit dem das sehr einfach umzusetzen ist:



Ein Interface definieren

Wir beschreiben mit dem Schlüsselwort interface eine Sammlung von Methoden, die jeder implementieren muss, der dazu gehören möchte:

```
public interface <Bezeichnung> {
    <Methodensignatur1>;
    <Methodensignatur2>;
    ...
}
```

In der Klasse Fussballspiel haben wir bereits ein paar Methodenaufrufe gesehen, die wir noch implementieren müssen.

Das dazugehörige Interface FussballspielAnzeige sieht entsprechend so aus:

```
public interface FussballspielAnzeige {
   public void zeigeIntro();
   public void zeigeMinute();
   public void zeigeTorchance();
   public void zeigeTor();
   public void zeigeParade();
   public void zeigeErgebnis();
}
```

Jeder Programmierer einer konkreten FussballspielAnzeige muss diese Methoden in seiner Klasse implementieren. Es können mehr dazukommen, es darf aber keine fehlen.

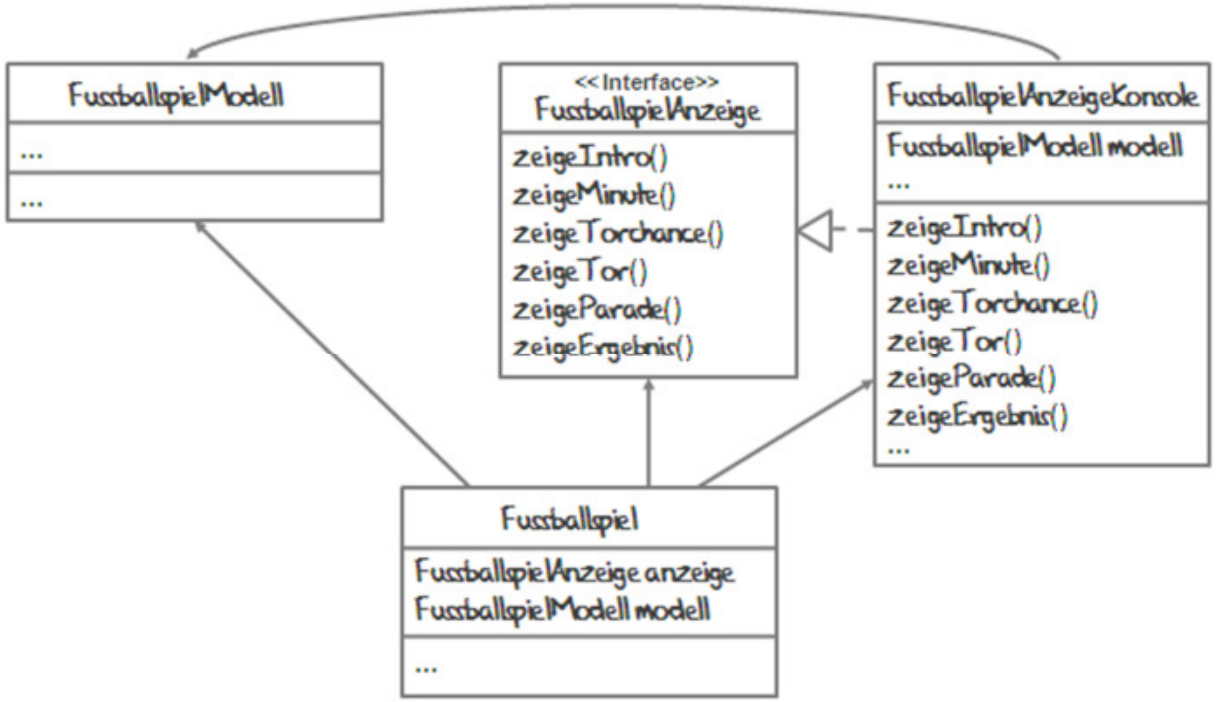
Ein Interface implementieren

Da es sich ebenfalls um einen Vererbungsmechanismus handelt, müssen wir in der neuen Klasse das zu implementierende Interface angeben. Die Syntax mit dem Schlüsselwort implements sieht dafür wie folgt aus:

```
public class <Klassename> implements <Interface> {
    <Methodensignatur1> {
        ...
    }

    <Methodensignatur2> {
        ...
    }
    ...
}
```

MVC mit Interface für Anzeige

Jetzt müssen alle Methoden einen Implementierungsteil haben:



Konsolenanzeige implementieren I

Als erstes Beispiel werden wir gleich das Interface FussballspielAnzeige mit einer einfachen Visualisierung auf der Konsole implementieren:

```
public class FussballspielAnzeigeKonsole implements FussballspielAnzeige {
   private FussballspielModell model;

   public FussballspielAnzeigeKonsole(FussballspielModell m) {
      model = m;
   }

   public void zeigeIntro() {
      System.out.println("-------------------------------------------------");
      System.out.println("Freundschaftspiel startet zwischen: ");
      System.out.println(model.getNameHeim()+" und "+model.getNameGast());
      System.out.println("-------------------------------------------------");
   }

   public void zeigeMinute() {
      System.out.println(model.getAktZeit()+". Minute (Tendenz: "+model.getTendenz()+")");
   }

   public void zeigeTorchance() {
      System.out.println();
      System.out.println(model.getAktZeit()+". Minute (Tendenz: "+model.getTendenz()+"):");
      System.out.println("   Chance fuer "+model.getAktMannschaftName() + " ...");
      System.out.println("   " + model.getAktSpielerName() + " zieht ab");
   }
   ...
```

Konsolenanzeige implementieren II

Im Konstruktor übergeben wir eine Referenz auf das Modell, damit die Anzeige auf die entsprechenden get-Methoden zugreifen kann.

Das Modell kennt die Anzeige allerdings nicht.

```
   public void zeigeTor() {
      System.out.println("   TOR!!!     " + model.getToreHeim() + ":" +
         model.getToreGast() + " " + model.getAktSpielerName() + "("  +
         model.getAktSpielerTore() + ")");
      System.out.println();
   }

   public void zeigeParade() {
      System.out.println("   " + model.getAktTorwartName() + " pariert glanzvoll.");
      System.out.println();
   }

   public void zeigeErgebnis() {
      System.out.println();
      System.out.println("-------------------------------------------------");
      System.out.println("Das Freundschaftsspiel endete:");
      System.out.println(model.getNameHeim()+
         " "+model.getToreHeim()+":"+model.getToreGast()+
         " "+model.getNameGast());
      System.out.println("-------------------------------------------------");
   }
}
```

Freundschaftsspiel FC Steinhausen-Oderbucher SK I

Nach der ganzen Theorie und den vielen Programmzeilen, können wir uns endlich ein wenig zurücklehnen und ein Derby anschauen, dass sich so oder so ähnlich in der Steinzeit zugetragen haben könnte (die Ausgabe wurde in den ausgeglichenen Phasen etwas gekürzt):

```
C:\>java Fussballspiel
--------------------------------------------------
Freundschaftspiel startet zwischen:
FC Steinhausen und Oderbrucher SK
--------------------------------------------------
1. Minute (Tendenz: -30)
2. Minute (Tendenz: -40)
...
18. Minute (Tendenz: 30)

19. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   R. Birkenpech zieht ab
   H. Schleifer pariert glanzvoll.

21. Minute (Tendenz: 0)
...
31. Minute (Tendenz: -80)

32. Minute (Tendenz: -100):
   Chance fuer FC Steinhausen ...
   B. Geroellheimer zieht ab
   T. Faenger pariert glanzvoll.

34. Minute (Tendenz: 60)
...
41. Minute (Tendenz: 30)

42. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   L. Sammler zieht ab
   TOR!!!     0:1 L. Sammler(1)
```

Freundschaftsspiel FC Steinhausen-Oderbucher SK II

Halbzeitpfiff - Ob dem FC Steinhausen der Ausgleich oder sogar der Spielsieg noch gelingt, sehen wir in der zweiten Halbzeit:

```
45. Minute (Tendenz: 30)
...
60. Minute (Tendenz: -90)

61. Minute (Tendenz: -100):
   Chance fuer FC Steinhausen ...
   D. Bogenbauer zieht ab
   T. Faenger pariert glanzvoll.

63. Minute (Tendenz: 70)
...
69. Minute (Tendenz: 90)

70. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   K. Zahnlos zieht ab
   H. Schleifer pariert glanzvoll.

72. Minute (Tendenz: -40)
...
86. Minute (Tendenz: 60)

87. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   K. Zahnlos zieht ab
   TOR!!!     0:2 K. Zahnlos(1)

90. Minute (Tendenz: 60)
91. Minute (Tendenz: 0)
92. Minute (Tendenz: 0)

------------------------------------------------
Das Freundschaftsspiel endete:
FC Steinhausen 0:2 Oderbrucher SK
------------------------------------------------
```

Obwohl es das Spiel der Torhüter hätte werden können, trafen die Derbyspezialisten L. Sammler und K. Zahnlos der doch etwas stärkeren Gastmannschaft und sicherten damit den Sieg.

Live-Coding-Session

Ausblick

Unser Konzept ermöglicht es uns, später die Ausgabe (wenn wir Kenntnisse zu visuellen Ausgabe erlangt haben) einfach auszutauschen: