Einführung zur Aufgabengruppe 0

Alle Literaturhinweise beziehen sich auf die Vorlesung
Modellierung und Programmierung 1

1 Einige Gesichtspunkte zur Softwareentwicklung

1.1 Strukturierung im Kleinen (s. Kapitel Anweisungen)

Edsger W. Dijkstra [1930-2002], ein niederländischer Informatiker, führte 1968 in einem Artikel „*Go To Statement Considered Harmful*“ aus: „*die Qualität eines Programms sei umgekehrt proportional zu der Anzahl der darin enthaltenen* ***goto*** *- Sprünge*“.

D-Diagramm

1. Eine einfache Aktion ist ein **D-Diagramm**. **Anweisung**
2. Wenn ***A*** und ***B*** **D-Diagramme** sind und ***c*** eine ***Bedingung*** ist, so auch:
   A B **Anweisungssequenz**
   if c then A end **Auswahlanweisungen**
   if c then A else B end
   while c do A end **Schleifenanweisung**
3. Nichts sonst ist ein **D-Diagramm**.

C- und Java-Anweisungen: **Ausdrucksanweisungen**
zusammengesetzte Anweisungen
Schleifenanweisungen
Auswahlanweisungen
strukturbezogene Sprunganweisungen

Einfache Aktionen:

Ausdrucksanweisungen *Ausdruck***;**

Ablaufsteuerung:

zusammengesetzte Anweisungen **{** *Anweisung Anweisung ... Anweisung* **}**

Auswahlanweisungen:

if(*Bedingung* **)** *Anweisung*

if(*Bedingung* **)** *Anweisung* **else** *Anweisung*

```
switch(Ausdruck ){
  case Konstante: Anweisung Anweisung ...
  case Konstante: Anweisung Anweisung ...
  ...
  default: Anweisung Anweisung ...
}
```

Schleifenanweisungen:

while(*Bedingung* **)** *Anweisung*

do *Anweisung* **while(** *Bedingung* **);**

for(*Ausdruck 1***;** *Ausdruck 2***;** *Ausdruck 3***)** *Anweisung*

strukturbezogene Sprunganweisungen:

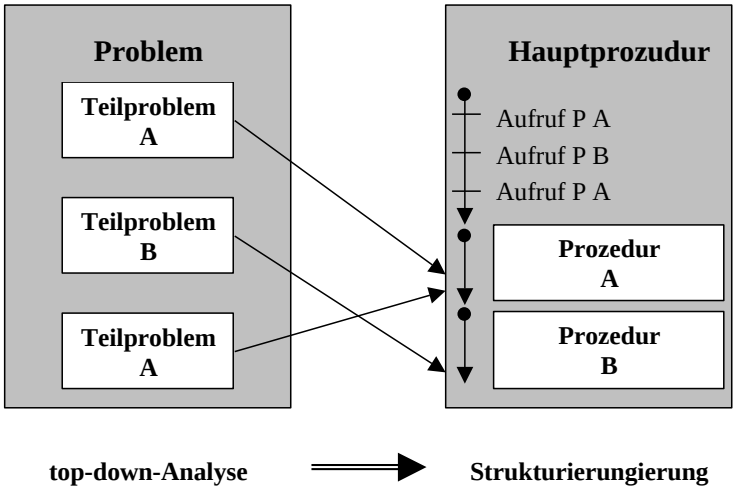
continue;
break;
return;

1.2 Prozedurorientierte Programmierung (s. Kapitel Methoden)

⇒ **Prozeduren dienen der Strukturierung eines Programms,**

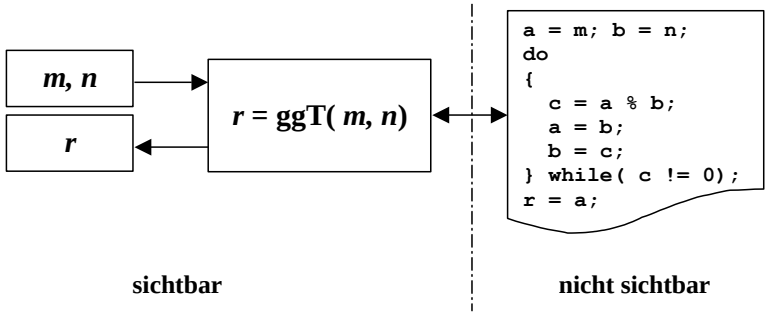
1. **der Ausgliederung wiederkehrender Berechnungen und**
2. **der Zerlegung komplexer Probleme in kleinere.**

Zielgerichteter Entwurf durch schrittweise Verfeinerung (Niklaus Wirth 1971 - Pascal)



Die Gesamtheit aller Prozeduren, zusammen mit der Vorschrift ihres Zusammenwirkens in einer Hauptprozedur, realisiert eine Lösung zu einem Problem. Der Zugriff auf Prozeduren ist über seine Schnittstellen geregelt. Die Integration in andere Programmteile erfordert keine Kenntnisse über den Aufbau der Prozedur.

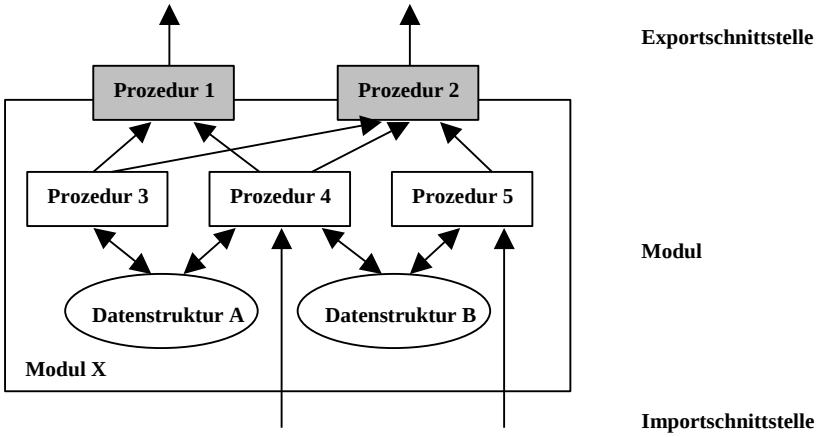
Schnittstelle einer Prozedur:
Prozedurname, Argumente (Eingabewerte) und Resultate (Ausgabewerte)



Pascal: Prozeduren, Funktionen
C: Funktionen
Java: Methoden

1.3 Strukturierung im Großen (s. Kapitel OOP / Klassen)

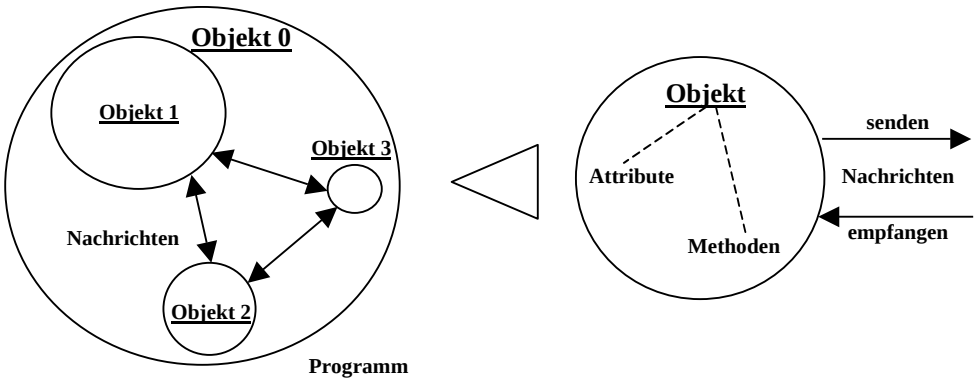
***Modulare Programmierung** (Adele Goldberg 1972/80 - Smalltalk)*



Ein Modul ist eine Sammlung (Kapselung) von Algorithmen und Datenstrukturen zur Bearbeitung einer in sich abgeschlossenen Aufgabe. Die Integration eines Moduls in ein Programmsystem wird über seine Schnittstellen geregelt und erfordert keine Kenntnisse des Aufbaus und der konkreten Realisierung der gekapselten Datenstrukturen und Algorithmen.

Schnittstelle eines Moduls:
Importschnittstelle (Nachfragen nach Dienstleistungen) und
Exportschnittstelle (Anbieten von Dienstleistungen)

Objektorientierte Programmierung (OOP)



Objektorientierte Programmierung gehört zur modularen Programmierung:
Die Module sind Objekte. Ein objektorientiertes Programm ist ein System miteinander kommunizierender Objekte.

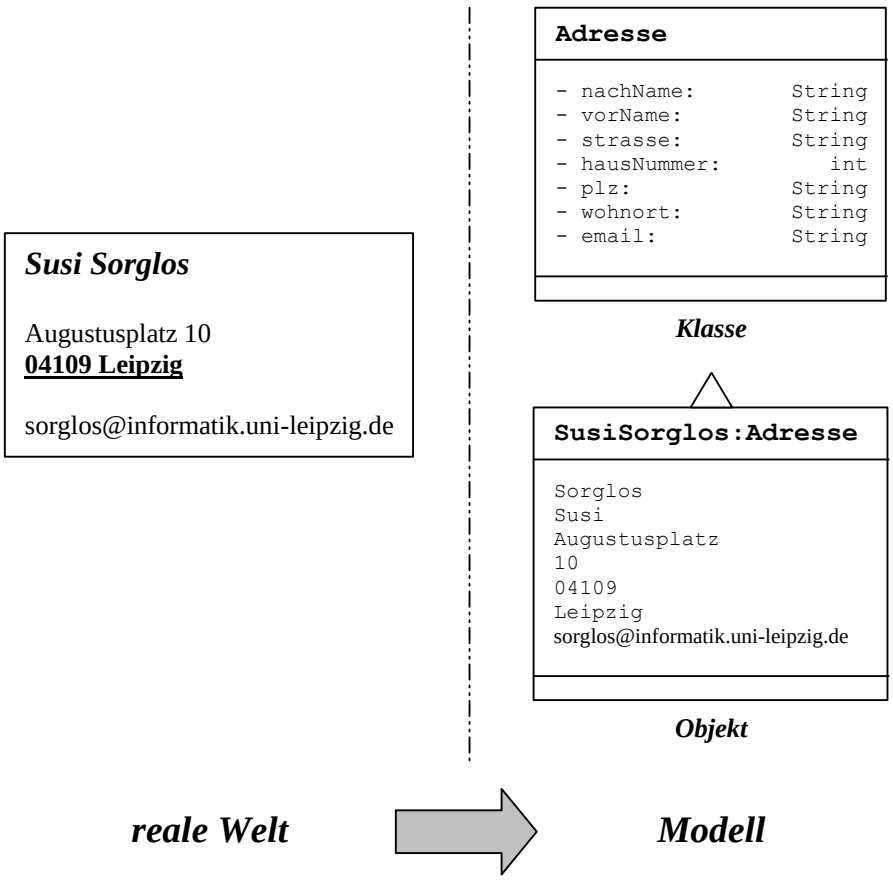
Schnittstelle eines Objektes:
Importschnittstelle (Empfangen von Nachrichten)
Exportschnittstelle (Senden von Nachrichten)

Programm als Kooperation von Objekten (Konzepte der OOP)

- **Objekte** sind die **Datengrundbausteine** (*Module*), aufgebaut aus **Attributen** (*Daten*) und **Methoden** (*Anweisungen*).
- *Objekte* kommunizieren miteinander über **Nachrichten**.
- **Klassen** sind die Datentypen (*Datenabstraktion*) der Objekte. Sie fassen Objekte mit gleichem Aufbau zusammen und dienen der **Datenkapselung**.
- *Klassen* sind *hierarchisch* aufgebaut. **Vererbungsmechanismen** gestattet die ***Weiterentwicklung*** bereits existierender Klassen. Aus Klassen können **Subklassen** und **Superklassen** abgeleitet werden.
- **Polymorphismus** und **dynamisches Binden** bilden zusammen mit bereits vorhandenen **Klassenbibliotheken** die Basis für die Wiederverwendbarkeit von Softwarebausteinen.

Java Platform, All Classes: **http://download.oracle.com/javase/7/docs/api/**

1.4 Modellierung (s. Kapitel Modellieren)



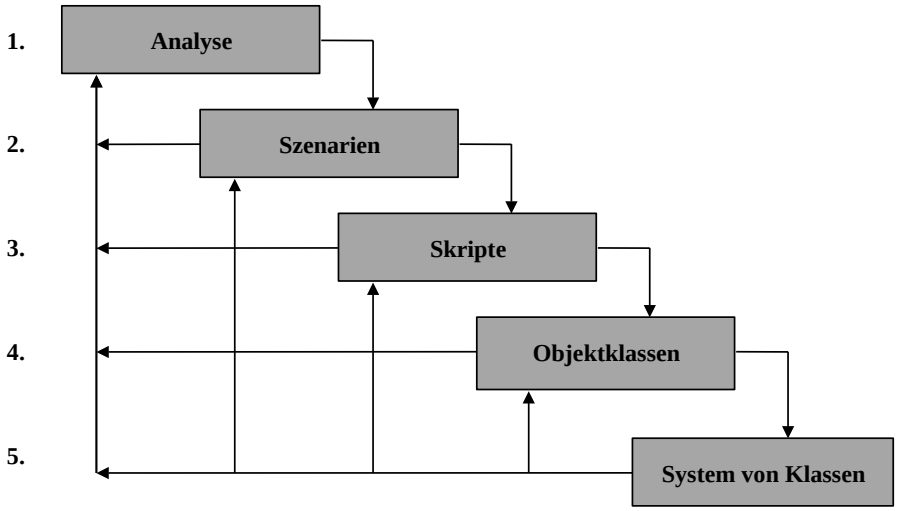
Einfache Probleme

Kompositioneller Entwurf:
Unterstreichen der Substantive als potentielle Kandidaten für Klassen.

Komplexe Probleme

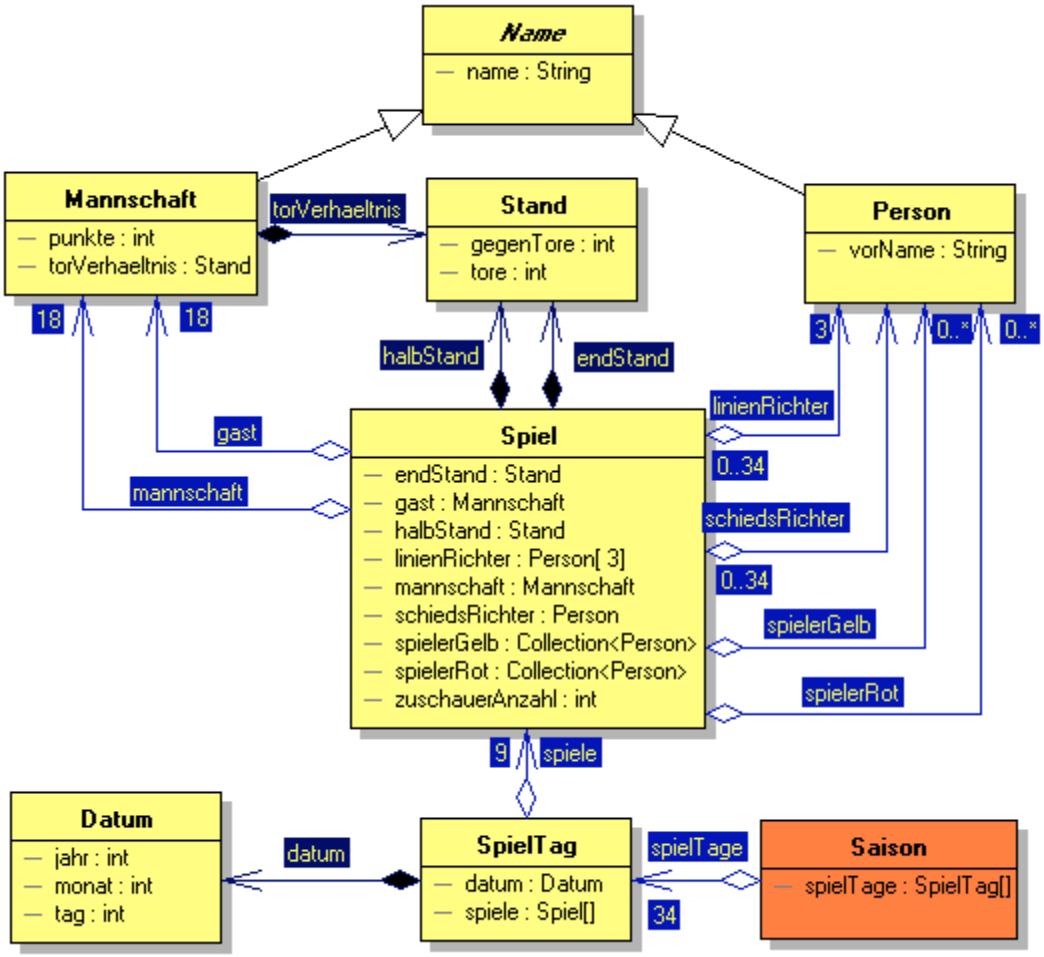
Objektorientierter Entwurf:

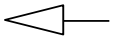
Schrittweises Vorgehen in sogenannten Modellierungsphasen (Wasserfallmodell).

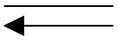


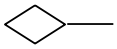
Wasserfallmodell

UML (Unified Modeling Language)



Vererbung: 
Objekte, die durch Spezialisierung oder Generalisierung aus anderen Objekten hervorgehen. Alle Attribute und Methoden werden *vererbt*.
Beispiel: Boot und Paddelboot.

Assoziation: 
Objekte, die miteinander in Beziehung treten. Auf Grund der Assoziationen können dann Objekte dieser Klassen *miteinander kommunizieren*.
Beispiel: Ein Hund und sein Herrchen.

Aggregation: 
Objekte, die *lose* miteinander in *Verbindung* treten.
Beispiel: In einem Raum befinden sich mehrere Möbelstücke. Diese existieren unabhängig vom Raum. Sie können auch in einem anderen Raum verbracht werden (Umzug).

Komposition: 
Die Teilobjekte sind *existenzabhängig* vom aggregierten Objekt.
Beispiel: Ein Gebäude besteht aus mehreren Räumen. Ein Raum kann nicht ohne Gebäude existieren.

Multiplizitäten
Die Multiplizität an den Beziehungen zwischen den Objekten gibt an, *wie viele* Objekte des einen Typs mit Objekten des anderen Typs verbunden sein können oder müssen.

Beispiele

1	*genau* 1 Objekte	Die Anfangsklasse einer *Komposition* hat stets diese Multiplizität. Sie kann deshalb weggelassen werden.
18	*genau* 18 Objekte	
0..1	*kein* oder 1 Objekt	Hat die Anfangsklasse einer *Aggregation* diese Multiplizität, so kann sie weggelassen werden.
1..4	1 bis *höchstes* 4 Objekte	
0..*	*beliebig viele* Objekte	

Material

UML-Klassendiagramm **http://timpt.de/topic95.html**

Dia - Programm für Diagrammerstellung

Windows **http://dia-installer.de/index.html.de**
Linux **http://projects.gnome.org/dia/**

1.5 Faustregeln bei der Entwicklung objektorientierter Projekte

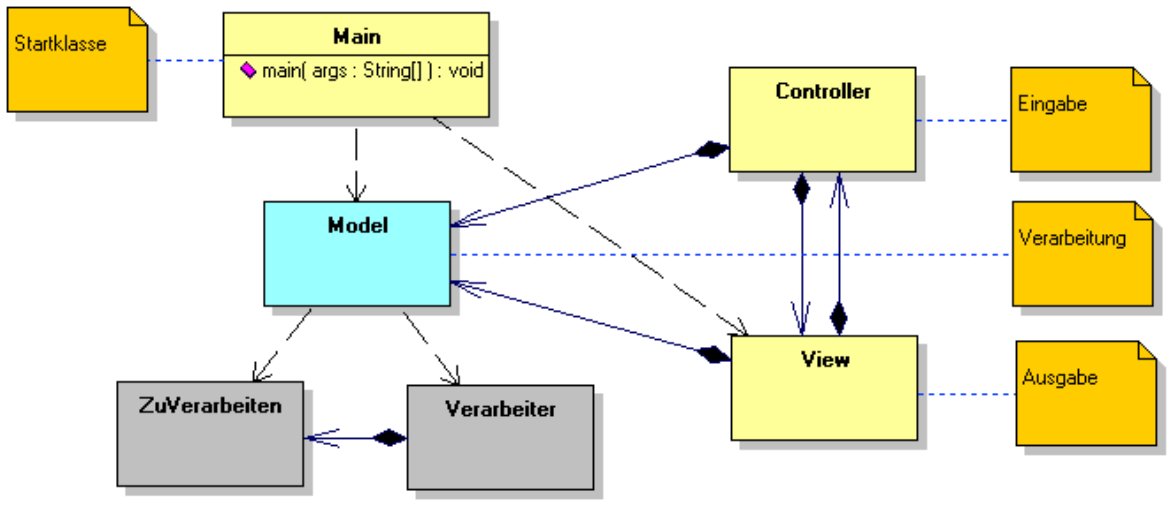
Namensgebuung (Camel-Case-Notation)

- ⇨ **Namen beschreiben den Inhalt: *"Sprechende" Namen*.**
- ⇨ ***Klassennamen* beginnen mit Großbuchstaben: `HalloWeltApplikation`.**
- ⇨ ***Variablen-, Objekt-, Attribut-* und *Methodennamen* beginnen mit Kleinbuchstaben: `lampe`.**
- ⇨ **Bei *zusammengesetzten Namen* beginnen weitere Wörter jeweils mit einem Großbuchstaben: `setLampe`**
- ⇨ ***Konstanten* werden mit Großbuchstaben bezeichnet: `MAX`.**
- ⇨ **Bei *zusammengesetzten* Konstanten werden weitere Worte jeweils durch einen Unterstrich getrennt: `MAX_VALUE`.**

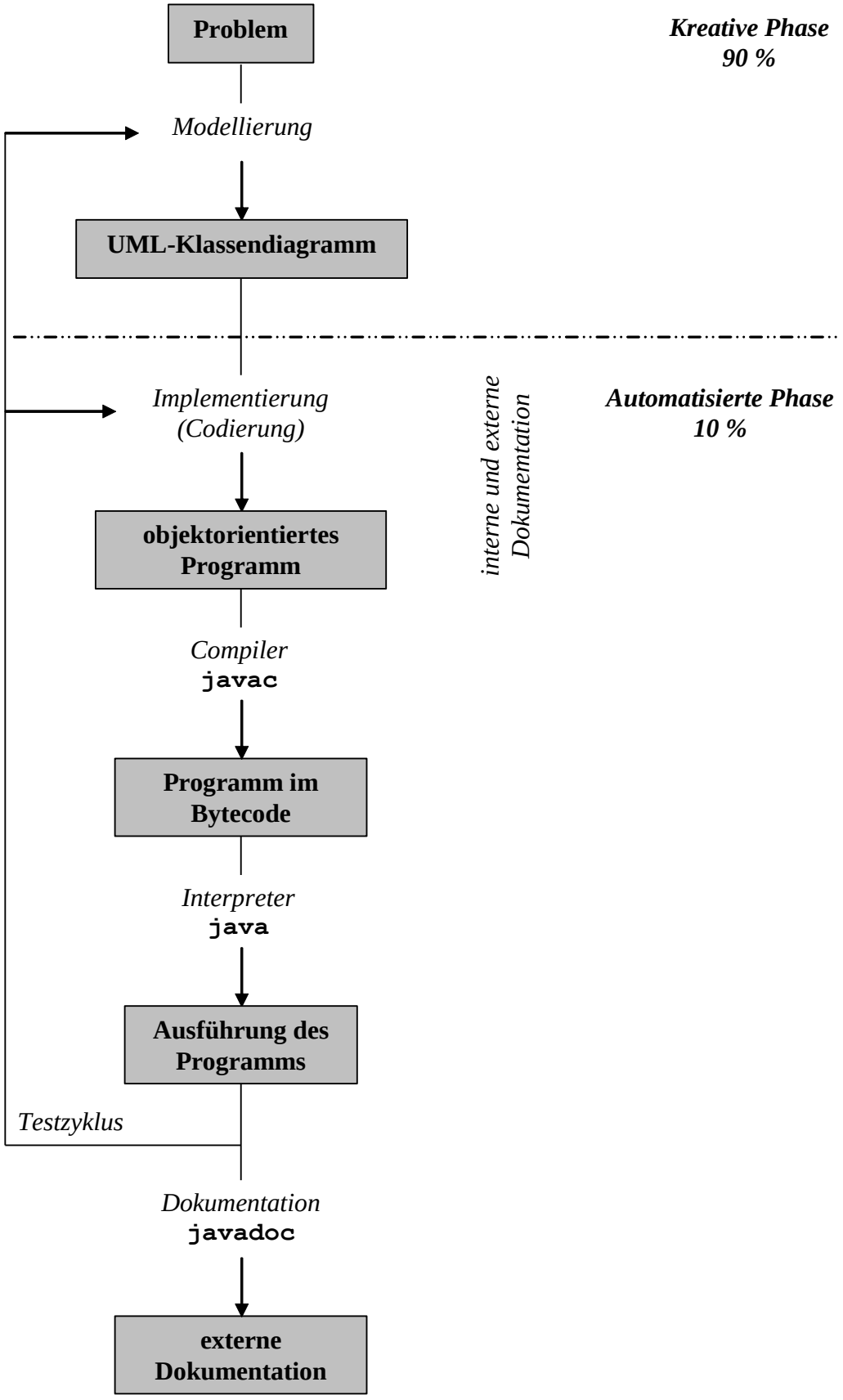
Modellierung und Programmentwicklung

- ⇨ ***Ein-* und *Ausgaben* sind im Hinblick auf grafische Oberflächen *nur* in der `main`-Methode oder in speziell dafür vorgesehene Klassen enthalten.**
- ⇨ ***Kleine Klassen* sind durch Generalisierung bzw. Spezialisierung zu modellieren, dabei ist auf Wiederverwendbarkeit ist zu achten.**
- ⇨ ***Jede Klasse, jedes Attribut* und *jede Methode* in ihr sind mit externen Kommentaren /** ... */ für die Dokumentation zu versehen.**
- ⇨ ***Methoden* sind zu strukturieren, *nicht länger* als eine A4-Seite. Ansonsten sind sie in kleinere Methoden aufzuteilen. Sie sind mit internen Kommentaren // ... bzw. /* ... */ zu versehen.**

Grobstruktur eines Programms



1.6 Vom Problem zum Programm



2 Beispiel „Der einarmige Bandit“

2.1 Aufgabenstellung

Der einarmige Bandit

Der einarmige Bandit ist ein Spielautomat. Ein Spiel mit ihm soll simuliert werden.

Aufgabenstellung

Schreiben Sie mittels der Klasse **`Wuerfel`** aus dem Netz ein Simulationsprogramm:

Als Anfangkapital besitzt der Automat 200 Euro. Als Taschengeld hat der Spieler 300 Euro. Der Einsatz wird festgelegt. Drei Zahlen zwischen 0 und 9 werden gewürfelt. Sind alle drei Zahlen gleich, wird das Vierfache, sind zwei Zahlen gleich, das Doppelte des Einsatzes zurückgezahlt, sonst wird der Einsatz vom Kapital abgezogen.

Das Spiel endet, falls der Automat oder der Spieler pleite ist.

Eingabe

Einsatz in jeder Spielrunde.

Ausgabe

Gewürfelte Zahlen, vorhandene Kapital des Spielers und des Automaten nach jeder Spielrunde.

Abbruch

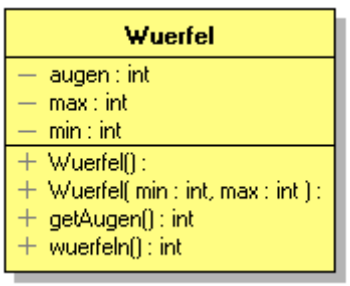
Das Programm bricht ab, falls der Nutzer kein neues Spiel mehr wünscht.

Hinweise zum Programm

Erläutern Sie zu Beginn dem Nutzer die Spielregeln. Beachten Sie beim Einsatz, dass das vorhandene Kapital nicht überschritten wird. Speichen Sie die drei gewürfelten Zahlen in ein statisches Feld. Werten Sie nach Abschluss jedes Spiels dieses aus: Geben Sie den Sieger an.

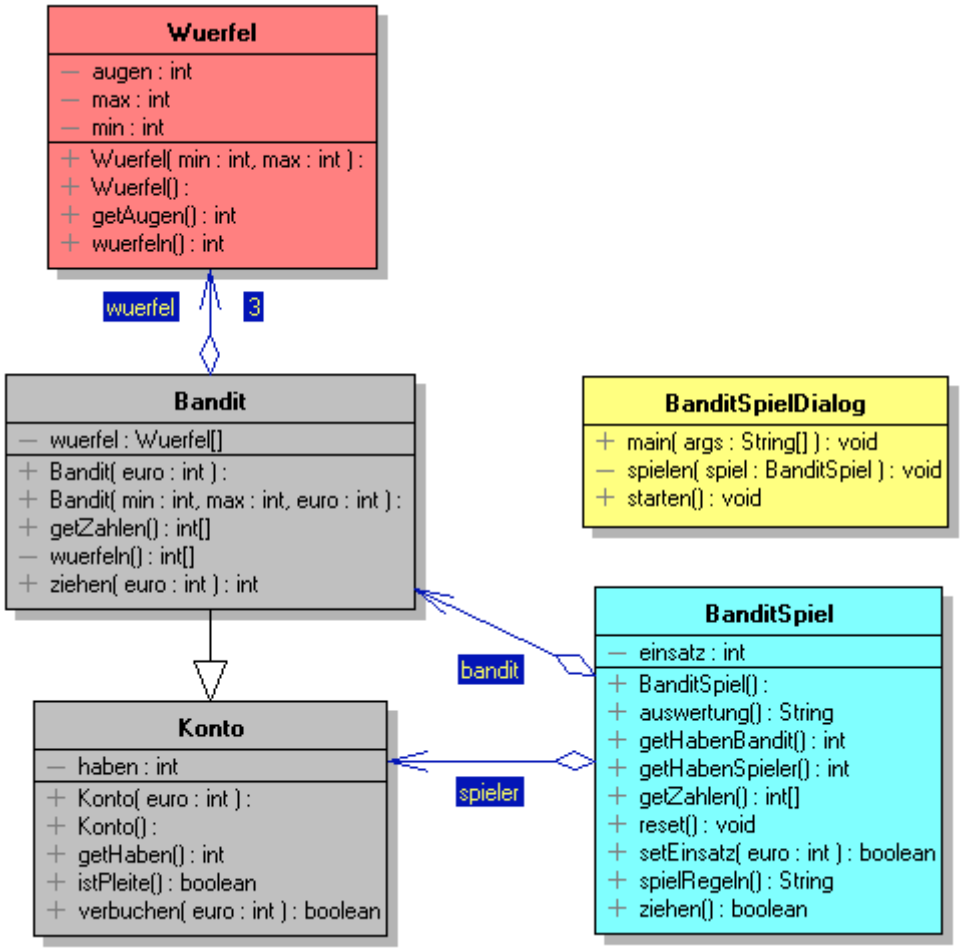
Wie hoch ist die Wahrscheinlichkeit, dass 2 Zahlen bzw. alle 3 Zahlen gleich sind?

Klasse `Wuerfel`



Modellierung und Programmierung **Klassendiagramm, Dokumentation, *Wuerfel.zip***

2.2 Modellierung



2.3 Implementierung

Konto.java
Bandit.java
BanditSpiel.java
BanditSpielDialog.java

2.4 Testphase

Testen der Nutzereingaben:

- *Korrekte* Eingaben, werden als erstes ausgetestet.
- Es wird *mehr* Einsatz als vorhandenes Kapital eingegeben.
- Es wird ein *negativer* Einsatz eingegeben.

Testen des Abbruchs

- *Abbruch bei Pleite des Spielers*, ist die Regel.
- *Abbruch bei Pleite des Automaten*: Dazu sollte man als >>ersten<< Einsatz mindestens 200 Euro eingeben. Die Wahrscheinlichkeit für zwei gleiche Zahlen liegt bei 28%, so dass man bei genügend vielen Spielstarts auch einmal gewinnen kann.

Wahrscheinlichkeiten
Anzahl der möglichen Ereignisse ist 1000, davon haben 10 Ereignisse drei gleiche Zahlen und 270 Ereignisse genau zwei gleiche Zahlen (280 mindestens zwei gleichen Zahlen).

2.5 Dokumentation

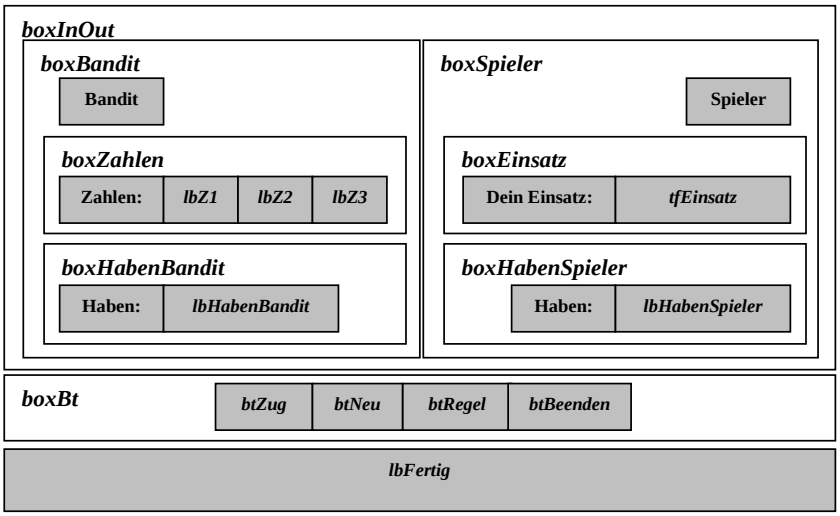
Einrichten eines Verzeichnisses für die Dokumentation, z.B. `Doc`. Wechsel in das Verzeichnis und Aufruf von `javadoc`:

```
javadoc -private ../*.java
```

Modellierung und Programmierung **Klassendiagramm, Dokumentation, *Bandit.zip***

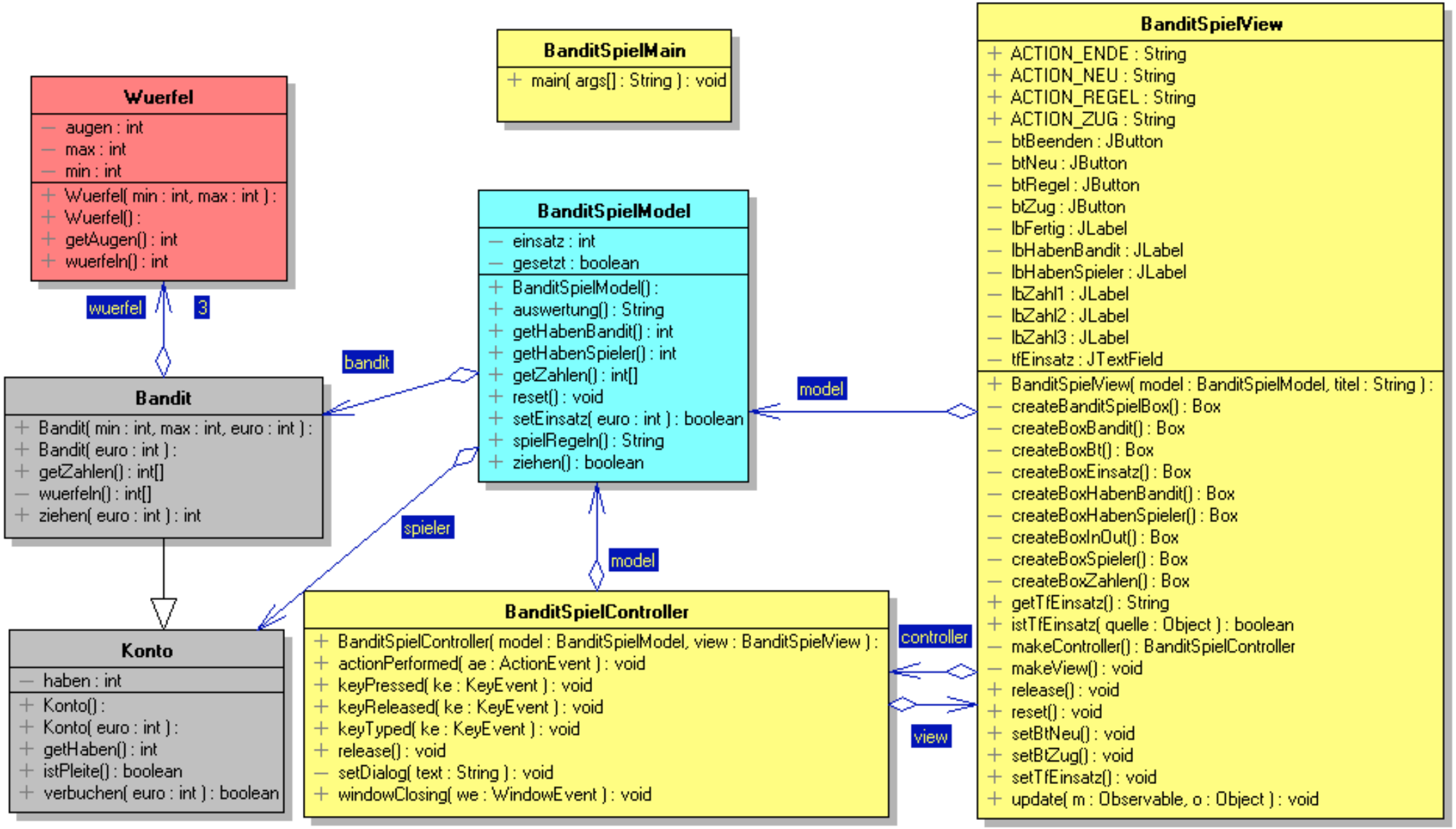
3 Zusatz: Benutzerschnittstelle zum Programm

Aufteilung des Fensters unter Verwendung von Boxen als Container (`BoxLayout`):



Modellierung und Programmierung mit MVC-Architektur

Klassendiagramm, **Dokumentation**, ***BanditMVC.zip***


BanditSpielMain
+ main(args[] : String) : void
Wuerfel
- augen : int
- max : int
- min : int
+ Wuerfel(min : int, max : int) :
+ Wuerfel() :
+ getAugen() : int
+ wuerfeln() : int
wuerfel
3
BanditSpielModel
- einsatz : int
- gesetzt : boolean
+ BanditSpielModel() :
+ auswertung() : String
+ getHabenBandit() : int
+ getHabenSpieler() : int
+ getZahlen() : int[]
+ reset() : void
+ setEinsatz(euro : int) : boolean
+ spielRegeln() : String
+ ziehen() : boolean
BanditSpielView
+ ACTION_ENDE : String
+ ACTION_NEU : String
+ ACTION_REGEL : String
+ ACTION_ZUG : String
- btBeenden : JButton
- btNeu : JButton
- btRegel : JButton
- btZug : JButton
- lbFertig : JLabel
- lbHabenBandit : JLabel
- lbHabenSpieler : JLabel
- lbZahl1 : JLabel
- lbZahl2 : JLabel
- lbZahl3 : JLabel
- tfEinsatz : JTextField
+ BanditSpielView(model : BanditSpielModel, titel : String) :
- createBanditSpielBox() : Box
- createBoxBandit() : Box
- createBoxBt() : Box
- createBoxEinsatz() : Box
- createBoxHabenBandit() : Box
- createBoxHabenSpieler() : Box
- createBoxInOut() : Box
- createBoxSpieler() : Box
- createBoxZahlen() : Box
+ getTfEinsatz() : String
+ istTfEinsatz(quelle : Object) : boolean
- makeController() : BanditSpielController
- makeView() : void
+ release() : void
+ reset() : void
+ setBtNeu() : void
+ setBtZug() : void
+ setTfEinsatz() : void
+ update(m : Observable, o : Object) : void
bandit
model
Bandit
+ Bandit(min : int, max : int, euro : int) :
+ Bandit(euro : int) :
+ getZahlen() : int[]
- wuerfeln() : int[]
+ ziehen(euro : int) : int
spieler
model
BanditSpielController
+ BanditSpielController(model : BanditSpielModel, view : BanditSpielView) :
+ actionPerformed(ae : ActionEvent) : void
+ keyPressed(ke : KeyEvent) : void
+ keyReleased(ke : KeyEvent) : void
+ keyTyped(ke : KeyEvent) : void
+ release() : void
- setDialog(text : String) : void
+ windowClosing(we : WindowEvent) : void
controller
view
Konto
- haben : int
+ Konto() :
+ Konto(euro : int) :
+ getHaben() : int
+ istPleite() : boolean
+ verbuchen(euro : int) : boolean