

Objektorientierte Softwareentwicklung (mit SEMI-OOS)

© 2007-2011 Robert Pütterich

Stand: März 2011

1. Vier Phasen der Softwareentwicklung

Analyse

Bei der Analyse steht die Frage im Vordergrund, WAS die Software leisten soll.

Zur Darstellung der Inhalte werden verschiedene Modelle verwendet. Zu einem sehr verbreiteten Standard hat sich die UML (Unified Modeling Language) entwickelt.

Entwurf (Design)

Diese Phase stellt die Frage in den Mittelpunkt, WIE die Problemlösung konkret erreicht werden soll, d.h. es ist festzulegen, wie beispielsweise die technische Umsetzung erfolgen soll bzw. welche Klassen und Algorithmen nötig sind, um die Aufgabe zu bewältigen. Die Darstellung der konkreten Lösungswege erfolgt wiederum mit Modellen.

Implementierung

Umsetzung der Modelle der Entwurfsphase beispielsweise in ein Programm der gewählten Programmiersprache.

Entwicklung / Evaluation

Aus Erfahrungen mit dem System werden Verbesserungsvorschläge erarbeitet und umgesetzt.

2. Die betriebssystemunabhängige Programmiersprache Java

Compiler verschiedener Sprachen wie C++, Pascal oder Basic arbeiteten bisher nach dem Prinzip, dass sie das Programm in höherer Programmiersprache in einen Maschinencode übersetzten, den der Computer dann abarbeiten kann. In der Regel steht dieser Maschinencode in einer Datei mit der Endung „.exe“ und funktioniert nur unter dem Betriebssystem, für das er erzeugt wurde.

Soll nun beispielsweise im Internet ein Programm wie ein Chatfenster zur Verfügung gestellt werden, ist nicht bekannt, welches Betriebssystem der Benutzer hat, d.h. das im ersten Absatz beschriebene Prinzip funktioniert nicht.

Auf diesem Problem basiert die Idee, welche mit Java realisiert wurde: Der Java-Compiler erzeugt aus dem Programm in der höheren Programmiersprache Java einen Zwischencode, den sogenannten Bytecode, der in Dateien mit der Endung „.class“ abgelegt wird. Dieser Bytecode kann von einem Java-Interpreter, den jeder Benutzer passend zu seinem Betriebssystem kostenlos aus dem Internet beziehen und installieren kann, ausgeführt werden.

Bei Java existiert zudem die Möglichkeit, alle „.class“-Dateien zu einer mit dem Java-Interpreter ausführbaren Java-Archiv-Datei (Endung „.jar“) zusammen zu fassen und zu komprimieren. Die Kompression entspricht dem „zip“-Format, sodass „.jar“-Dateien auch mit einem entsprechenden „zip“-Programm geöffnet werden können.

3. Objektorientierung

Allgemeines

Die Objektorientierung ist eine Sichtweise, die hilft, Objekte dieser Welt nach einem bestimmten Muster zu schematisieren. Dabei wird jedem Objekt ein bestimmter Typ (Klasse) zugeordnet.

Beispiel: Die Objekte Frau Müller und Herr Meier gehören dem Typ (der Klasse) Mensch an. Sie sind also Objekte (oder Instanzen) der Klasse Mensch.

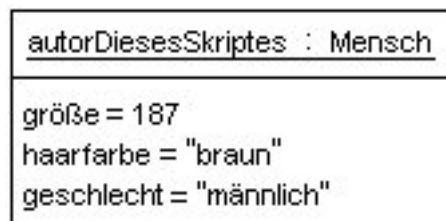
Oft lassen sich auch speziellere Klassen finden: So könnte Frau Müller als ein Objekt der Klasse Frau und Herr Meier als ein Objekt der Klasse Mann gesehen werden.

Klassen werden beschrieben, indem man Attribute (Eigenschaften) und Methoden (mögliche Handlungen) angibt, wobei eine schrittweise beschriebene Handlung als Algorithmus bezeichnet wird.

So könnten für die Klasse Mensch sicherlich die Attribute gröÙe, haarfarbe und geschlecht und die Methoden gehen(), sprechen() und essen() gefunden werden.
Das UML-Modell dieser Klasse sieht folgendermaßen aus:

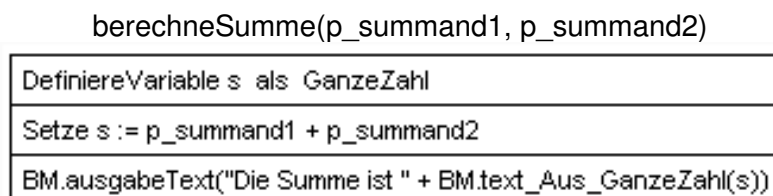


Ein Objekt einer Klasse besitzt zu jedem Attribut einen individuellen Attributwert.
Eine bestimmte Person, wie beispielsweise der Autor dieses Skriptes, besitzt die Attributwerte 1,87m, braun und männlich zu den oben genannten Attributen gröÙe, haarfarbe und geschlecht.
Attributwerte sind Momentaufnahmen, die sich mit der Zeit ändern können.
Das Objekt autorDiesesSkriptes wird in UML wie folgt modelliert:



Im oberen Rechteck wird dabei der Objektname mit Angabe der Klasse unterstrichen und zu jedem Attribut der individuelle Attributwert angegeben. Die Methoden werden beim Objektdiagramm in der Regel weg gelassen, da sie ja bei jedem Objekt äquivalent zur Angabe bei der Klasse sind.

Bei Methoden ist es möglich, Parameter in den runden Klammern anzugeben.
Ein Beispiel hierfür wäre die Methode berechneSumme(p_summand1, p_summand2) mit den Parametern p_summand1 und p_summand2, welche aus zwei ganzen Zahlen die Summe berechnen soll. Um Verwechslungen mit anderen Variablen oder Attributen zu vermeiden, wird empfohlen, jede Parameterbezeichnung mit „p_“ zu beginnen.
Somit könnten in einem Programm dann beispielsweise die Methodenaufrufe **berechneSumme(12, 14)** oder auch **berechneSumme(zahl1, zahl2)**, wenn zahl1 und zahl2 entsprechende Variablen sind, erfolgen. Im ersten Fall ist p_summand1 = 12 und p_summand2 = 14, im zweiten Fall ist p_summand1 = zahl1 und p_summand2 = zahl2. Der nachfolgend aufgeführte Algorithmus (Notation gemäß SEMI-OOS-Syntax, siehe folgende Kapitel dieses Skriptes) der Methode berechnet dann aus den Parametern p_summand1 und p_summand2 die Summe und gibt sie aus:



Die Methode berechneSumme(p_summand1, p_summand2) könnte aber auch so gestaltet werden, dass sie einen Wert zurück gibt, der dann beispielsweise gleich einer Variable gesetzt wird. Eine entsprechende Anweisung könnte dann **Setze summe := berechneSumme(5, 8)** lauten.

Bei diesem Aufruf ist p_summand1 gleich 5 und p_summand2 gleich 8. Die Methode berechnet daraus die Summe und gibt den Wert 13 zurück, der dann gleich der Variablen summe gesetzt wird. Der Algorithmus würde dann wie folgt aussehen:

berechneSumme(p_summand1, p_summand2) mit Rückgabewert

DefiniereVariable s als GanzeZahl
Setze s := p_summand1 + p_summand2
methodeBeendenMitRückgabe(s)

Anmerkung: Die lokale Variable s ist nur innerhalb dieser Methode gültig und nicht zu verwechseln mit der Variable summe beim Aufruf **Setze summe := berechneSumme(5, 8)**.

Eine Methode muss aber keinen Wert zurückgeben. Die Methode ausgabeDerZahlenVon1Bis10() soll beispielsweise lediglich die Zahlen von 1 bis 10 ausgeben, eine Wertrückgabe erscheint nicht sinnvoll.

Konventionen zur Schreibweise

Klassennamen werden immer groß geschrieben, Methoden, Attribute und Parameter beginnen mit einem kleinen Buchstaben.

Setzt sich die Bezeichnung aus mehreren Worten zusammen, so wird alles zusammen geschrieben und ab dem 2. Wort die Teilwörter groß.

Beispiele:

Klassen: Mensch, LandInEuropa

Attribute: gröÙe, haarfarbe, anzahlDerEinwohnerInMillionen

Methoden: gehen(), sprechen(), insAutoSteigen(), demokratischeWahlDurchführen()

Vererbung

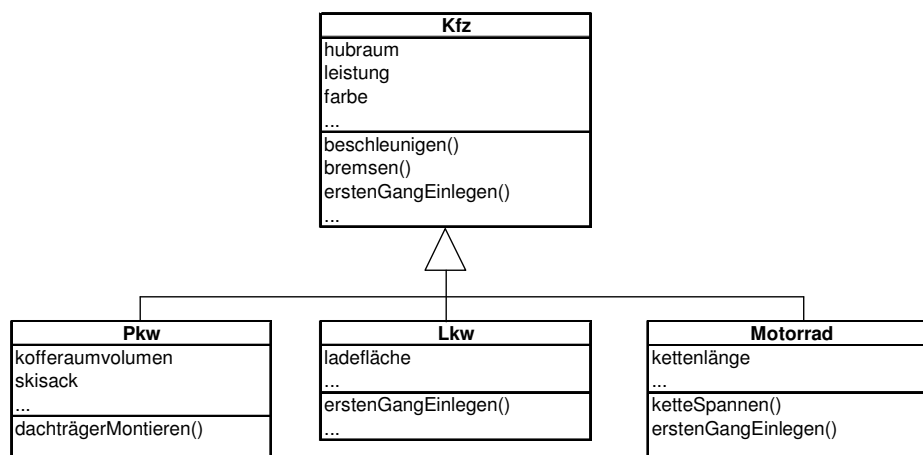
Übernimmt eine Klasse (Unterklasse) alle Attribute und Methoden einer anderen Klasse (Oberklasse), so nennt man dies Vererbung.

Methoden der Oberklasse können dabei überschrieben werden (Polymorphismus: „gleiche Methodenbezeichnung, unterschiedliche Handlung“).

In der Regel ist die Unterklasse eine Spezialisierung der Oberklasse bzw. die Oberklasse eine Generalisierung der Unterklasse.

Beispiel: Oberklasse Kfz, Unterklassen Pkw, Lkw und Motorrad

UML-Darstellung mit Attributen und Methoden



4. Softwareentwicklung mit SEMI-OOS

Allgemeines

Für die Softwareentwicklung hat sich die objektorientierte Sichtweise als sehr praktikabel erwiesen, sodass sie heute eine nicht wegzudenkende Rolle spielt. Sie ermöglicht die Programmierung in sehr realitätsnahen Strukturen und in kleinen überschaubaren Einheiten, welche wieder verwendet und durch die Möglichkeit der Vererbung erweitert werden können.

Grundsätzliche Vorgehensweise

Zunächst werden Klassen (d.h. deren Attribute und Methoden), die als Baupläne für Objekte dienen, definiert. Zu jeder Methode ist zudem ein Algorithmus festzulegen, der die einzelnen Arbeitsschritte, die beim Ausführen einer Methode abzuarbeiten sind, fixiert.

Gemäß den definierten Klassen können dann Objekte erzeugt (instantiiert) werden. Aus dem Zusammenspiel der Objekte mit gegenseitigen Methodenaufrufen ergibt sich der Programmablauf.

Bei der Erstellung objektorientierter Software sind die im vorhergehenden Abschnitt genannten Definitionen in einer objektorientierten Programmiersprache wie z.B. Java zu formulieren, wobei darauf geachtet werden muss, dass die Syntax (Schreibweise) der Programmiersprache exakt eingehalten wird.

Da dies gerade im Unterricht mit großen Schwierigkeiten verbunden ist, wurde SEMI-OOS so konstruiert, dass lediglich die inhaltliche Struktur (Semantik) des Programms einzugeben ist, die Formulierung in der Programmiersprache erfolgt dann automatisch.

Bei Oberflächen, wie sie in der Softwareentwicklung heutzutage üblich sind, ergibt sich der Programmablauf in der Regel ereignisgesteuert. Wird beispielsweise eine Befehlsschaltfläche gedrückt (Ereignis!), so hat dies die Ausführung einer bestimmten Methode zur Folge.

Prinzipiell gibt es eine Vielzahl von Ereignissen, die eintreten können. Neben dem Drücken einer Befehlsschaltfläche sind beispielsweise auch das Drücken einer bestimmten Maustaste oder Mausebewegungen Ereignisse, welche Methodenaufrufe zur Folge haben können.

Im Rahmen der didaktischen Reduktion ist bei SEMI-OOS nur das Ereignis „Drücken einer Befehlsschaltfläche“ zugelassen.

Datentypen

Attributen, Parametern, lokalen Variablen, die innerhalb einer Methode definiert werden können, und Methoden mit Rückgabewert muss ein bestimmter Datentyp zugewiesen werden.

Wir unterscheiden zwischen „primitiven“ Datentypen und Referenzdatentypen.

Unter SEMI-OOS sind folgende „primitive“ Datentypen verwendbar:

Text

Wahrheitswert wahr / falsch

GanzeZahl liegt im Intervall $[-2^{31}; 2^{31}-1]$ wobei 2^{31} : ca 2,147 Mrd.

GanzeZahlGroß liegt im Intervall $[-2^{63}; 2^{63}-1]$

Kommazahl ca. 15 Stellen Genauigkeit

Referenzdatentypen verweisen auf Objekte, ihr Datentyp ist die Klasse des Objektes. Aus diesem Grund ist jede Klasse, nachdem sie gespeichert wurde, auch als Datentyp verfügbar. Die Bezeichnung einer Referenzvariable könnte als Objektname gesehen werden, so wie der Name einer Person einen bestimmten Menschen, also ein Objekt der Klasse Mensch, referenziert.

Beispiel:

Der Datentyp einer Referenzvariable `autorDiesesSkriptes` ist `Mensch`, `autorDiesesSkriptes` verweist auf mich, ein Objekt der Klasse `Mensch`. Damit wäre eine Anweisung **`autorDiesesSkriptes.sprechen()`**, welche bei mir die Methode `sprechen()` ausführen würde, möglich.

5. Praktische Beispiele mit SEMI-OOS (Ein- und Ausgabe über die Konsole)

Bitte zuerst unter „Einstellungen“ auf „Einsteiger“ und „Deutsch“ stellen.

5.1 Konto

Projekt Konto.semi

Wir definieren eine Klasse Konto mit dem Attribut kontostand und den Methoden einzahlen() und auszahlen().

Nach dem Start von SEMI-OOS erscheint die Oberfläche zur Eingabe der Klassenbestandteile, die auch über die Befehlschaltfläche „Klasse“ der Steuerungskonsole im oberen Bereich erreichbar ist.

Hier geben wir nun die Klasse „Konto“, das Attribut „kontostand“ und die Methoden „einzahlen“ und „auszahlen“ ein. Die Bedienung sollte selbsterklärend sein. Bei Unklarheiten hilft auch die knapp gehaltene Anleitung „Kurzreferenz SEMI-OOS“ im Anhang.

Der Einfachheit halber wählen wir für das Attribut kontostand den Datentyp „GanzeZahl“ und führen demnach das Konto in ganzen Euro. Bei den Methoden sind keine Parameter anzugeben, da wir den jeweiligen Betrag bei der Methodenausführung vom Benutzer erfragen wollen. Ein Rückgabewert macht ebenso keinen Sinn, daher belassen wir die Einstellung „kein Rückgabewert“.

Die UML-Darstellung (im Arbeitsbereich rechts) der Klasse sieht nun folgendermaßen aus:



Nun sind noch die Algorithmen für die beiden Methoden (Befehlsschaltfläche „Methode“ der Steuerungskonsole) und eine Startanweisung (Befehlsschaltfläche „Startanweisung“ der Steuerungskonsole) zu definieren.

Die Festlegung der Algorithmen der Methoden erfolgt mit dem Struktogramm nach Nassi-Shneiderman, einer (programmiersprachenunabhängigen) Modellierungstechnik für Algorithmen.

Bei SEMI-OOS wird eine minimale aber effiziente Auswahl an algorithmischen Strukturen angeboten, welche auf der linken Seite angewählt (orange Markierung) und per rechtem Mausklick und dem Menüpunkt „Struktogrammelement einfügen“ ein- bzw. angefügt werden. Klickt man mit der linken Maustaste auf ein Element des Struktogramms, so öffnet sich für dieses Element der Editor im unteren Fensterbereich.

Beim Editieren der Anweisungen bzw. Bedingungen von Struktogrammelementen werden Textbausteine angeboten. Die Bausteine beinhalten auch eine Menge sehr nützlicher Basismethoden einer in SEMI-OOS integrierten Klasse BM (siehe Anhang).

Im Editor gelangt man mit der Cursor-Taste „nach-oben“ in das Auswahlfeld, die Eingabetaste fügt den markierten Baustein ein und mit der Esc-Taste kann das Auswahlfeld stets verlassen werden.

Wird mit dem Eintippen eines Bausteines begonnen, so erfüllt das Auswahlfeld die Aufgabe der Textvervollständigung, welche mit der Eingabetaste dann abgeschlossen werden kann.

Werden nur einstellige Bausteine eingegeben, z.B. bei der Eingabe eines Variablennamens und ist der Baustein eindeutig, so wird automatisch nach dem Tippen des Zeichens der Baustein übernommen.

Wurde ein korrekter und vollständiger Ausdruck zusammengesetzt, so wird die Befehlsschaltfläche „übernehmen“ aktiv, welche dann gedrückt werden kann. Ist „übernehmen“ aktiv, so wird sie beim Fortfahren z.B. mit der Bearbeitung eines anderen Struktogrammelementes automatisch gedrückt. Ist „übernehmen“ nicht aktiv, so werden aktuelle Änderungen beim Fortfahren verworfen.

Der Algorithmus für die Methode einzahlen() stellt sich wie folgt dar:

einzahlen()

DefiniereVariable betrag als GanzeZahl
BM.ausgabeTextZU("")
BM.ausgabeText("Einzahlungsbetrag in Euro: ")
Setze betrag := BM.eingabeGanzeZahl()
Setze kontostand := kontostand + betrag
BM.ausgabeTextZU("Aktueller Kontostand: " + BM.text_Aus_GanzeZahl(kontostand) + " Euro")
auszahlen()

Erläuterung:

Die erste Anweisung definiert eine lokale Variable betrag vom Datentyp GanzeZahl.

ausgabeTextZU(p_text) und ausgabeText(p_text) sind sogenannte Basismethoden einer in SEMI-OOS integrierten Klasse BM, welche auf der SEMI-OOS-Konsole mit oder ohne Zeilenumbruch den Text ausgeben, der als Parameter angegeben wird. Die Anweisung BM.ausgabeTextZU("") gibt also eine Leerzeile aus.

In Anweisung vier wird betrag ein Wert zugewiesen. Die Basismethode eingabeGanzeZahl() erfragt dabei den einzuzahlenden Betrag auf der Konsole.

Anweisung fünf setzt den Wert von kontostand auf kontostand (alt) plus den Wert von betrag.

In Anweisung sechs wird dann der neue Kontostand ausgegeben. Anzumerken ist hier, dass Texte mit einem „+“-Zeichen verkettet werden und der Kontostand zur Ausgabe mit der entsprechenden Basismethode zu einem Text verwandelt werden muss.

Nach jeder Einzahlung soll eine Auszahlung folgen, daher wird zum Abschluss die Methode auszahlen() aufgerufen.

Die Struktur des Algorithmus ist denkbar einfach, da nur aufeinander folgende Anweisungen ausgeführt werden.

Der Algorithmus für die Methode auszahlen() stellt sich sehr ähnlich dar:

auszahlen()

DefiniereVariable betrag als GanzeZahl
BM.ausgabeTextZU("")
BM.ausgabeText("Auszahlungsbetrag in Euro: ")
Setze betrag := BM.eingabeGanzeZahl()
Setze kontostand := kontostand - betrag
BM.ausgabeTextZU("Aktueller Kontostand: " + BM.text_Aus_GanzeZahl(kontostand) + " Euro")
einzahlen()

Als Startanweisung geben wir an: **erzeugeNeuesObjekt_Konto().einzahlen()**

Das bedeutet, dass beim Start unseres Programmes ein Objekt der Klasse Konto erzeugt (instantiiert) und die Methode einzahlen() ausgeführt wird. Der restliche Programmablauf ergibt sich dann aus den Anweisungen der beiden Methoden, die sich abwechselnd gegenseitig aufrufen.

Drücken wir nun die Befehlsschaltfläche „Implementierung“ in der Steuerungskonsole, können wir die entsprechenden Formulierungen unserer Eingaben in schülergemäßer SEMI-OOS-Sprache oder in Java anzeigen lassen.

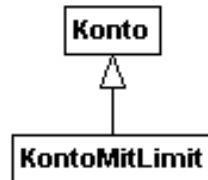
Zur Erläuterung der weiteren Möglichkeiten im Arbeitsbereich „Implementierung“ verweise ich an dieser Stelle auf die Kurzreferenz von SEMI-OOS.

5.2 KontoMitLimit

Projekt KontoMitLimit.semi

Wir erstellen eine Klasse KontoMitLimit, welche von der Klasse Konto erbt und beim Auszahlen ein Limit berücksichtigen soll.

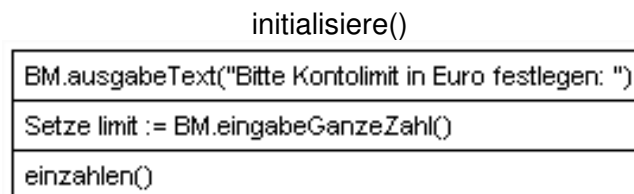
Dazu drücken wir nach der Definition der Klasse KontoMitLimit die Befehlsschaltfläche „Vererbungsbeziehung“ der Steuerungskonsole und erstellen die entsprechende Beziehung, welche dann in UML wie folgt dargestellt wird:



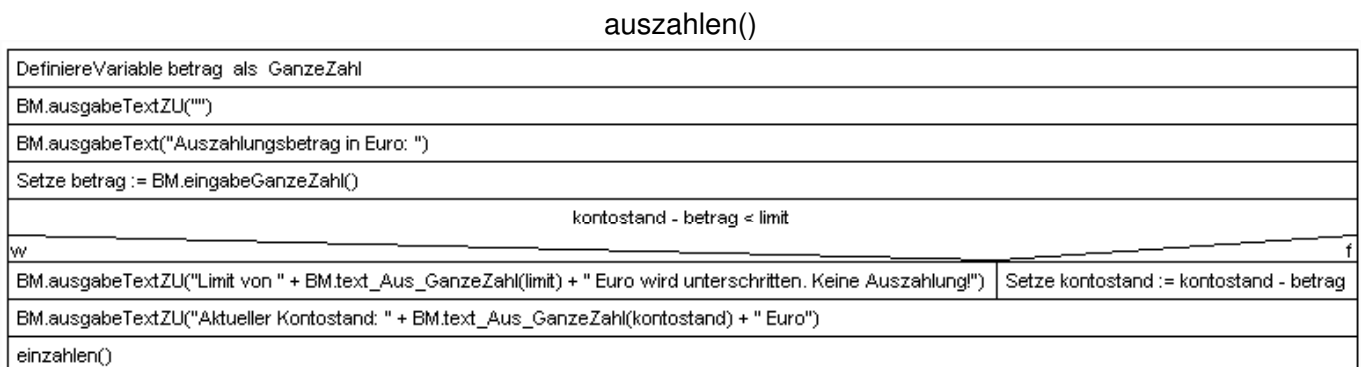
Für die Klasse KontoMitLimit ist lediglich ein Attribut limit zu definieren und die Methode auszahlen(), die nun das Limit berücksichtigen soll, zu überschreiben (Polymorphismus). Zum Festlegen des Limits und anschließendem Aufruf von einzahlen() wird zudem die Methode initialisiere() erstellt, welche in der Startanweisung ausgeführt werden soll.



Die Methode initialisiere() liest vom Benutzer das limit ein und ruft dann einzahlen() auf:



Die Methode auszahlen() benötigt nun eine bedingte Anweisung mit Alternative:



Als Startanweisung geben wir an: **erzeugeNeuesObjekt_KontoMitLimit().initialisiere()**

5.3 EinMalEinsTrainer

Projekt EinMalEinsTrainer.semi

Wir erstellen zunächst folgende Klasse:

EinMalEinsTrainer
aktuellerFaktor1 : GanzeZahl aktuellerFaktor2 : GanzeZahl aktuelleAntwort : GanzeZahl
stelleAufgabeUndLiesAntwort () bewerteAntwort ()

Die Methode `stelleAufgabeUndLiesAntwort()` ermittelt über die Basismethode `ganzeZufallszahl(p_minimum, p_maximum)` die beiden Faktoren:

`stelleAufgabeUndLiesAntwort()`

Setze aktuellerFaktor1 := BM.ganzeZufallszahl(1, 20)
Setze aktuellerFaktor2 := BM.ganzeZufallszahl(1, 20)
DefiniereVariable aufgabeText als Text
Setze aufgabeText := BM.text_Aus_GanzeZahl(aktuellerFaktor1) + " * " + BM.text_Aus_GanzeZahl(aktuellerFaktor2) + " = "
BM.ausgabeText(aufgabeText)
Setze aktuelleAntwort := BM.eingabeGanzeZahl()
bewerteAntwort()

Die Methode `bewerteAntwort()` prüft die Antwort und ruft dann wieder `stelleAufgabeUndLiesAntwort()` auf:

`bewerteAntwort()`

aktuelleAntwort = aktuellerFaktor1 * aktuellerFaktor2	
ww f	
BM.ausgabeTextZU("Korrekt !")	BM.ausgabeTextZU("Leider falsch !")
	BM.ausgabeText("Die korrekte Antwort lautet: ")
	BM.ausgabeTextZU(BM.text_Aus_GanzeZahl(aktuellerFaktor1 * aktuellerFaktor2))
stelleAufgabeUndLiesAntwort()	

Als Startanweisung geben wir an:

`erzeugeNeuesObjekt_EinMalEinsTrainer().stelleAufgabeUndLiesAntwort()`

5.4 EinMalEinsTrainerMitStatistik

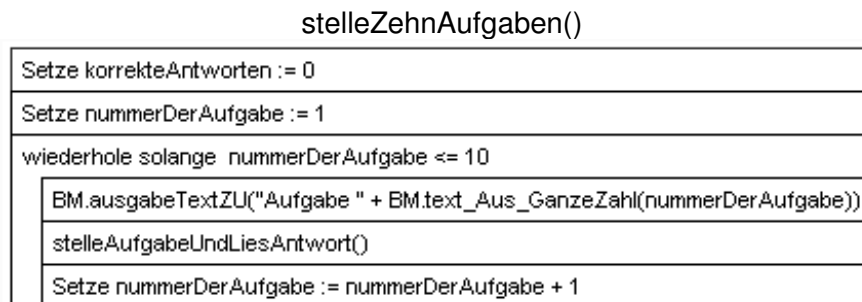
Projekt EinMalEinsTrainerMitStatistik.semi

Die Klasse EinMalEinsTrainerMitStatistik erbt von der Klasse EinMalEinsTrainer, wobei die Attribute nummerDerAufgabe und korrekteAntworten bei der Unterklasse hinzugefügt werden.

Da wir nun eine feste Anzahl an Aufgaben stellen wollen, wird die Methode stelleZehnAufgaben() definiert. bewerteAntwort() wird überschrieben, da wir beim Bewerten ja nun auch eine kleine Statistik ausgeben wollen.



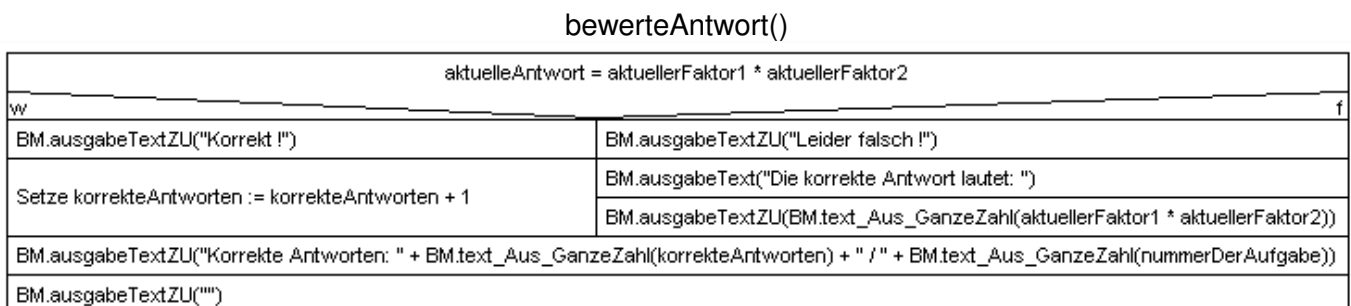
Bei der Methode stelleZehnAufgaben() verwenden wir dabei die algorithmische Struktur „Wiederholung mit Bedingungsprüfung am Anfang“.



Hinweis:

Einziger Unterschied bei den in SEMI-OOS angebotenen Wiederholungsvarianten ist, dass bei der Bedingungsprüfung am Ende zunächst der Wiederholungsblock durchgeführt und dann erst die Bedingung geprüft wird, d.h. der Wiederholungsblock wird in jedem Fall mindestens einmal durchgeführt. Bei der Wiederholung mit Bedingungsprüfung am Anfang kann es sein, dass der Wiederholungsblock gar nicht ausgeführt wird, wenn die Bedingung von Anfang an falsch ist.

Die Methode bewerteAntwort() ermittelt nun auch eine Statistik:



Als Startanweisung geben wir an:

erzeugeNeuesObjekt_EinMalEinsTrainerMitStatistik().stelleZehnAufgaben()

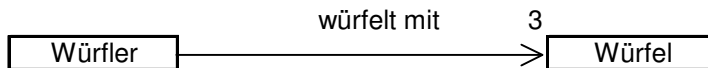
5.5 Würfler und Würfel

Projekt Wuerfel.semi

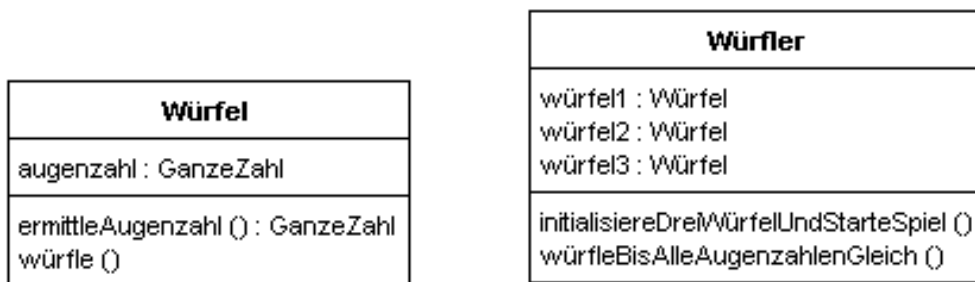
Wir erstellen bei diesem Beispiel zwei Klassen, wobei von der Klasse Würfler ein Objekt und von der Klasse Würfel drei Objekte instantiiert werden.

Der Würfler soll drei Würfel solange werfen, bis ihre Augenzahl identisch ist.

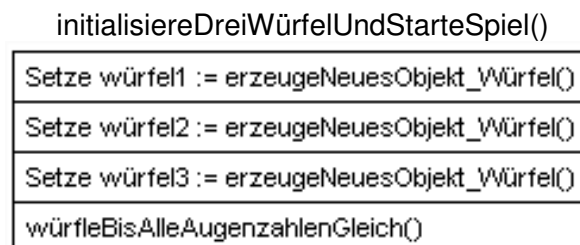
Da der Würfler Zugriff auf die Würfel haben soll, die Würfel aber nicht auf den Würfler (unidirektionale Assoziation), sieht das handschriftlich zu erstellende Klassendiagramm wie folgt aus:



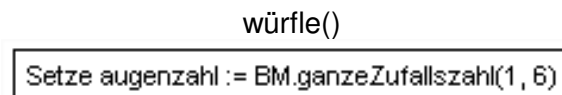
Zur Implementierung der Assoziation erhält die Klasse Würfler drei Attribute vom Typ Würfel, die als Referenzvariablen für die drei Würfel-Objekte dienen:



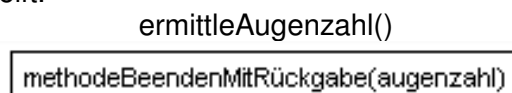
In der Methode initialisiereDreiWürfelUndStarteSpiel() werden nun die drei Würfelobjekte erzeugt und den Referenzattributen zugewiesen:



Die Methode würfle() der Klasse Würfel ermittelt lediglich einen Attributwert zum Attribut augenzahl:



ermittleAugenzahl() liefert die aktuelle Augenzahl, um zu vermeiden, dass die Klasse Würfler direkt auf das klassenfremde Attribut zugreift.



Hinweis:

Gemäß Kapselungsprinzip sollen auf die Attribute einer Klasse grundsätzlich immer nur klasseneigene Methoden zugreifen. Würde nämlich bei einer Klasse ein unzulässiger Attributwert gesetzt (z.B. bei einer Klasse Bruch das Attribut nenner auf den Wert 0), so ist der Fehler immer nur in der eigenen Klasse und nicht im ganzen Programm zu suchen.

Bei der Klasse Bruch müsste man also beispielsweise dafür sorgen, dass nur klasseneigene Methoden wie z.B. setzeNenner(p_nenner) auf das Attribut nenner zugreifen und dass diese Methoden dafür sorgen, dass der Attributwert auf keinen Fall 0 wird.

In der Methode würfleBisAlleAugenzahlenGleich() wird zunächst die lokale Variable wurf definiert und auf 0 gesetzt. Wir haben hier nun einen typischen Fall vorliegen, um eine Wiederholung mit Bedingungsprüfung am Ende einzusetzen, da auf jeden Fall zuerst einmal gewürfelt werden soll, bevor geprüft wird, ob die drei Würfel die gleiche Augenzahl haben. Besitzen sie die gleiche Augenzahl, so wird die Wiederholung und damit das Spiel beendet.

würfleBisAlleAugenzahlenGleich()

DefiniereVariable wurf als GanzeZahl									
Setze wurf := 0									
<table border="1"> <tr> <td>Setze wurf := wurf + 1</td></tr> <tr> <td>BM.ausgabeTextZU(BM.text_Aus_GanzeZahl(wurf) + ". Wurf")</td></tr> <tr> <td>würfel1.würfle()</td></tr> <tr> <td>würfel2.würfle()</td></tr> <tr> <td>würfel3.würfle()</td></tr> <tr> <td>BM.ausgabeTextZU("Würfel 1: " + BM.text_Aus_GanzeZahl(würfel1.ermittleAugenzahl()))</td></tr> <tr> <td>BM.ausgabeTextZU("Würfel 2: " + BM.text_Aus_GanzeZahl(würfel2.ermittleAugenzahl()))</td></tr> <tr> <td>BM.ausgabeTextZU("Würfel 3: " + BM.text_Aus_GanzeZahl(würfel3.ermittleAugenzahl()))</td></tr> <tr> <td>BM.ausgabeTextZU("")</td></tr> </table>	Setze wurf := wurf + 1	BM.ausgabeTextZU(BM.text_Aus_GanzeZahl(wurf) + ". Wurf")	würfel1.würfle()	würfel2.würfle()	würfel3.würfle()	BM.ausgabeTextZU("Würfel 1: " + BM.text_Aus_GanzeZahl(würfel1.ermittleAugenzahl()))	BM.ausgabeTextZU("Würfel 2: " + BM.text_Aus_GanzeZahl(würfel2.ermittleAugenzahl()))	BM.ausgabeTextZU("Würfel 3: " + BM.text_Aus_GanzeZahl(würfel3.ermittleAugenzahl()))	BM.ausgabeTextZU("")
Setze wurf := wurf + 1									
BM.ausgabeTextZU(BM.text_Aus_GanzeZahl(wurf) + ". Wurf")									
würfel1.würfle()									
würfel2.würfle()									
würfel3.würfle()									
BM.ausgabeTextZU("Würfel 1: " + BM.text_Aus_GanzeZahl(würfel1.ermittleAugenzahl()))									
BM.ausgabeTextZU("Würfel 2: " + BM.text_Aus_GanzeZahl(würfel2.ermittleAugenzahl()))									
BM.ausgabeTextZU("Würfel 3: " + BM.text_Aus_GanzeZahl(würfel3.ermittleAugenzahl()))									
BM.ausgabeTextZU("")									
wiederhole solange würfel1.ermittleAugenzahl() <> würfel2.ermittleAugenzahl() ODER würfel2.ermittleAugenzahl() <> würfel3.ermittleAugenzahl()									
BM.ausgabeTextZU("Spiel beendet!")									

Als Startanweisung geben wir an:

erzeugeNeuesObjekt_Würfler().initialisiereDreiWürfelUndStarteSpiel()

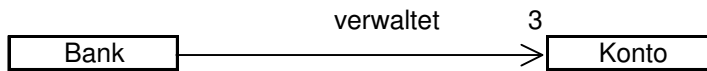
5.6 Bank und Konten

Projekt BankKonto.semi

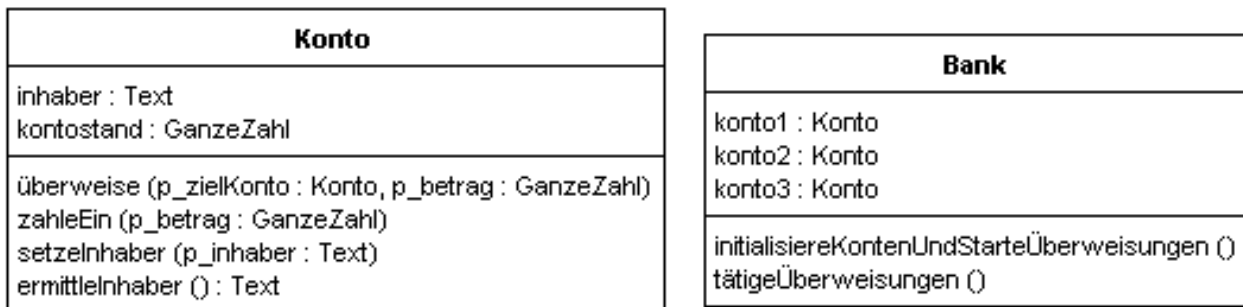
Ähnlich dem vorhergehenden Beispiel erstellen wir wieder zwei Klassen, wobei von der Klasse Bank erneut ein Objekt und von der Klasse Konto drei Objekte instantiiert werden.

Die Bank veranlasst bei den Konten untereinander Überweisungen.

Da die Bank Zugriff auf die Konten haben soll, sieht das Klassendiagramm wie folgt aus:

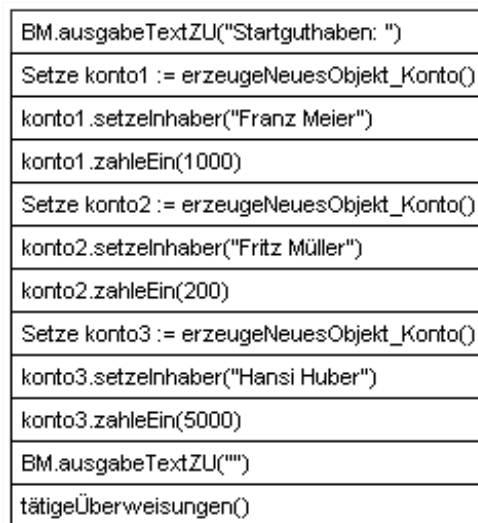


Die Klassen werden wie folgt definiert:



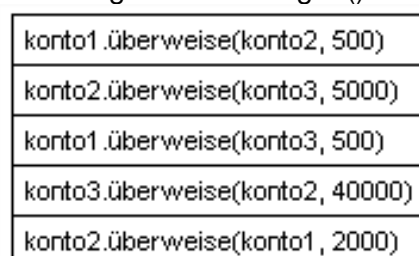
In der Methode `initialisiereKontenUndStarteÜberweisungen()` werden drei Kontoobjekte erzeugt und den Referenzattributen zugewiesen. Zugleich werden die Inhaber festgelegt und ein Startguthaben eingezahlt:

`initialisiereKontenUndStarteÜberweisungen()`



Die Methode `tätigeÜberweisungen()` ruft eine beliebig gewählte Folge der Methode `überweise(p_zielkonto, p_betrag)` bei den Konten auf:

`tätigeÜberweisungen()`



Die Methode `überweise(p_zielkonto, p_betrag)` gibt die Überweisungsdaten aus, zieht den Betrag vom eigenen Konto ab und zahlt es auf dem als Parameter gegebenen Zielkonto ein, indem es die Methode `zahleEin(p_betrag)` bei diesem Konto aufruft:

`überweise(p_zielkonto, p_betrag)`

<code>BM.ausgabeTextZU(inhaber + " überweist an " + p_zielkonto.ermittleInhaber() + " " + BM.text_Aus_GanzeZahl(p_betrag) + " Euro.")</code>
<code>Setze kontostand := kontostand - p_betrag</code>
<code>BM.ausgabeTextZU("Neuer Kontostand: " + inhaber + " " + BM.text_Aus_GanzeZahl(kontostand) + " Euro.")</code>
<code>p_zielkonto.zahleEin(p_betrag)</code>
<code>BM.ausgabeTextZU("")</code>

Die Methode `zahleEin(p_betrag)` erhöht den Kontostand entsprechend und gibt den neuen Kontostand aus:

`zahleEin(p_betrag)`

<code>Setze kontostand := kontostand + p_betrag</code>
<code>BM.ausgabeTextZU("Neuer Kontostand: " + inhaber + " " + BM.text_Aus_GanzeZahl(kontostand) + " Euro.")</code>

Die Methoden `setzeInhaber(p_inhaber)` und `ermittleInhaber()` wurden lediglich wieder dazu definiert, dass gemäß Kapselungsprinzip die Klasse `Bank` nicht direkt auf fremde Attribute der Klasse `Konto` zugreift.

`setzeInhaber(p_inhaber)`

<code>Setze inhaber := p_inhaber</code>

`ermittleInhaber()`

<code>methodeBeendenMitRückgabe(inhaber)</code>

Als Startanweisung geben wir an:

`erzeugeNeuesObjekt_Bank().initialisiereKontenUndStarteÜberweisungen()`

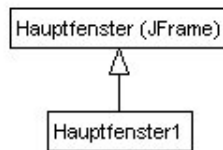
6. Ein- und Ausgaben mit SEMI-OOS über Oberflächen

6.1 Objektorientierte Struktur einer Oberfläche

Objektorientierte Oberflächen sind in der heutigen Softwareentwicklung nicht mehr wegzudenken. Aus diesem Grund ist bei SEMI-OOS ein entsprechender Editor integriert, der über die Befehlsschaltfläche „Oberflächen-Klasse“ der Steuerungskonsole aktiviert wird.

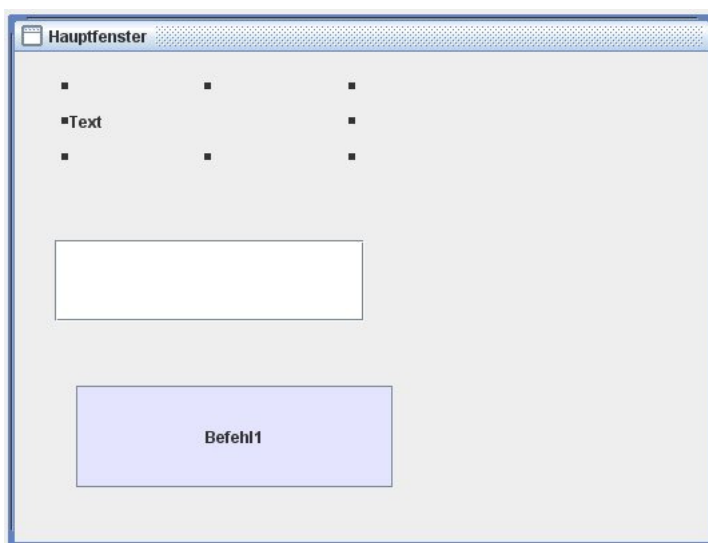
Für jede Oberfläche erstellt SEMI-OOS automatisch eine Klasse. Diese Klasse erbt von einer bereits existenten Klasse Hauptfenster, welche unter Java mit „JFrame“ bezeichnet ist. In der Klasse Hauptfenster (JFrame) sind alle nötigen Attribute und Methoden für ein solches Fenster bereits vorhanden. Durch die Vererbung übernehmen wir diese in unsere neue Fensterklasse.

Da die Unterklasse zunächst den Standardnamen „Hauptfenster1“ erhält, wird nach der Erstellung eines Fensters unter „Vererbungsbeziehung“ Folgendes angezeigt:



Zeichnen wir in unser Hauptfenster Oberflächenelemente wie beispielsweise Textfelder, Beschriftungsfelder oder Befehlsschaltflächen (dies sind Objekte von Klassen, die unter Java wie die Klasse JFrame bereits vorhanden sind), so wird automatisch für jedes gezeichnete Oberflächenobjekt ein Attribut, d.h. eine Referenzvariable für dieses Objekt, unserer Fensterklasse angelegt. Diese Attribute erhalten dabei abgekürzte Namen, die jederzeit noch geändert werden können. Textfelder werden beispielsweise tf1, tf2... , Befehlsschaltflächen bsf1, bsf2... und Beschriftungsfelder bf1, bf2... benannt. Für jede Befehlsschaltfläche wird zudem automatisch eine „Klick“-Methode erstellt, welche beim Programmablauf durch das Drücken der entsprechenden Befehlsschaltfläche ausgeführt wird. Ändert man nachträglich diese Namen ab, so ist zu beachten, dass identische Namen für verschiedene Elemente nicht erlaubt sind und SEMI-OOS daher die letzte Tastatureingabe, die zu einem solchen Fehler führen würde, verwirft.

Wird beispielsweise mit dem Editor folgendes Fenster erstellt, welches ein Beschriftungsfeld, ein Textfeld und eine Befehlsschaltfläche enthält, so entsteht dabei automatisch, verbunden mit der entsprechenden Vererbungsbeziehung, die aufgeführte Klasse:



Hauptfenster1
bf1 : Beschriftungsfeld tf1 : Textfeld bsf1 : Befehlsschaltfläche
bsf1Klick ()

In der Methode dieser Klasse können beispielsweise Anweisungen wie bf1.setzeBeschriftung(„hallo“) oder tf1.ermittleText() erfolgen, da jedes Oberflächenelement Methoden (siehe Anhang 7.2) besitzt, die entsprechend im Editor als Textbausteine angeboten werden.

6.2 Java Applikationen und Java Applets

Programme mit Oberflächen können unter „Implementierung“ als Applikation („Java-Archiv (*.JAR) erzeugen und ausführen“), d.h. als ein unabhängig von SEMI-OOS lauffähiges Programm, oder aber auch als Java-Applet (Java-Applet-Archiv (*.JAR) und HTML-Gerüst erzeugen) implementiert werden. Bei der Implementierung als Applet muss genau eine Oberflächenklasse (nicht mehrere) im Projekt vorhanden sein und die Startanweisung mit „erzeugeNeuesObjekt_NameDerFensterklasse“ beginnen. Um alle weiteren technischen Details kümmert sich SEMI-OOS.

Ein Applet läuft innerhalb einer HTML-Internetseite. Ein Gerüst für diese HTML-Seite, welche nur die Einbindung des Applets beinhaltet, wird von SEMI-OOS erzeugt und in dem Verzeichnis, in dem sich auch die Projektdatei befindet, abgelegt. Diese Seite kann beliebig mit einem entsprechenden HTML-Editor erweitert werden.

6.3 Beispiele

6.3.1 Multiplikator

Projekt: Multiplikator_Oberflaeche.semi

Unter „Oberflächen-Klasse“ wird dazu folgende Oberfläche erstellt und die Bezeichnungen der Oberflächenelemente auf tf_faktor1, tf_faktor2, bsf_produktBerechnen und bf_ergebnis umbenannt. Der Name der Klasse wird ebenfalls von Hauptfenster1 auf MultiplikatorOberfläche geändert:



MultiplikatorOberfläche
tf_faktor1 : Textfeld tf_faktor2 : Textfeld bsf_produktBerechnen : Befehlsschaltfläche bf_ergebnis : Beschriftungsfeld
bsf_produktBerechnenKlick ()

Der Algorithmus der Methode definiert zunächst die lokalen Variablen faktor1, faktor2 und ergebnis:

bsf_produktBerechnenKlick()
DefiniereVariable faktor1 als GanzeZahl
DefiniereVariable faktor2 als GanzeZahl
DefiniereVariable ergebnis als GanzeZahl
Setze faktor1 := BM.ganzeZahl_Aus_Text(tf_faktor1.ermittleText())
Setze faktor2 := BM.ganzeZahl_Aus_Text(tf_faktor2.ermittleText())
Setze ergebnis := faktor1 * faktor2
bf_ergebnis.setzeBeschriftung("Ergebnis: " + BM.text_Aus_GanzeZahl(ergebnis))

Anschließend wird faktor1 gleich dem Textinhalt des Textfeldes tf_faktor1 und faktor2 gleich dem Textinhalt von tf_faktor2 gesetzt. Der jeweilige Textinhalt wird mit der Methode ermittleText() ausgelesen und muss dann noch zur ganzen Zahl konvertiert werden.

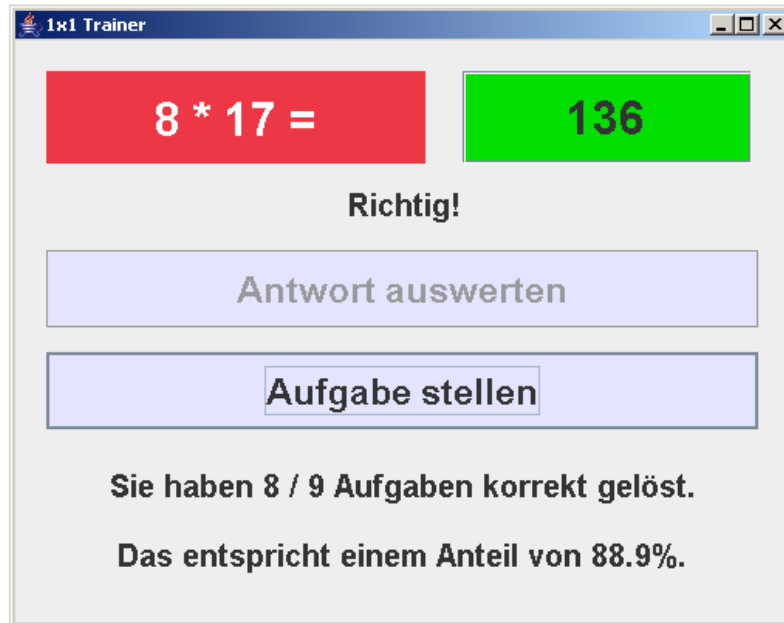
Die Methode setzeBeschriftung(...) erwartet als Parameter einen Text, welcher aus „Ergebnis: “ und dem zum Text gewandelten Ergebnis zusammengesetzt wird.

Die Startanweisung lautet: **erzeugeNeuesObjekt_MultiplikatorOberfläche()**

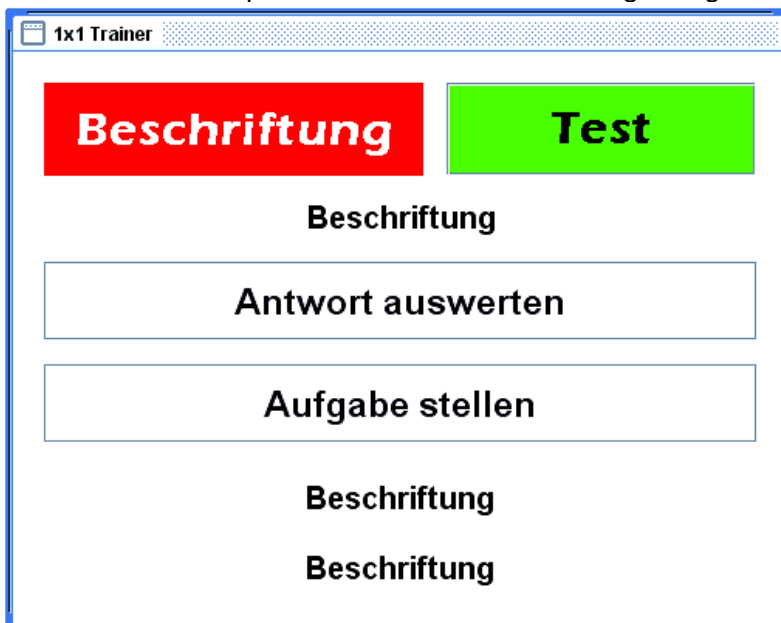
6.3.2 EinMalEinsTrainer mit Oberfläche

Projekt EinMalEinsTrainer_Oberflaeche.semi

Das Endprodukt soll folgendes Erscheinungsbild besitzen:



Dazu wird eine entsprechende Oberfläche mit zugehöriger Klasse erstellt:



EinMalEinsTrainerOberfläche
aktuellerFaktor1 : GanzeZahl aktuellerFaktor2 : GanzeZahl aktuelleAntwort : GanzeZahl korrekteAntworten : GanzeZahl anzahlAufgaben : GanzeZahl bf_aufgabe : Beschriftungsfeld bsf_antwortAuswerten : Befehlsschaltfläche bsf_aufgabeStellen : Befehlsschaltfläche bf_statistik1 : Beschriftungsfeld bf_statistik2 : Beschriftungsfeld bf_kommentar : Beschriftungsfeld tf_ergebnis : Textfeld
initialisiere () aufgabeStellen () statistikAusgeben () bsf_antwortAuswertenKlick () bsf_aufgabeStellenKlick ()

Die Attribute aktuellerFaktor1, aktuellerFaktor2, aktuelleAntwort, korrekteAntworten und anzahlAufgaben wurden dabei mit dem Klasseneditor (Befehlsschaltfläche „Klasse“ der Steuerungskonsole) ergänzt, die anderen Attribute entstanden durch das Zeichnen der Oberfläche im Editor.

Ebenso wurden die Methoden initialisiere(), aufgabeStellen() und statistikAusgeben() ergänzt und die Methoden bsf_antwortAuswertenKlick() und bsf_aufgabeStellenKlick() durch das Zeichnen erstellt. Die ergänzten Attribute und Methoden sind beim Klassendiagramm durch eine etwas größere Lücke von den durch die Oberflächenerstellung automatisch erzeugten abgesetzt.

Die Startanweisung **erzeugeNeuesObjekt_EinMalEinsTrainerOberfläche().initialisiere** erzeugt ein Objekt der Klasse EinMalEinsTrainerOberfläche und ruft initialisiere() auf.
Im Folgenden sind alle Struktogramme der definierten Methoden aufgelistet.

initialisiere()

bf_aufgabe.setzeBeschriftung("")
bf_statistik1.setzeBeschriftung("")
bf_statistik2.setzeBeschriftung("")
bf_kommentar.setzeBeschriftung("")
tf_ergebnis.setText("")
aufgabeStellen()

aufgabeStellen()

Setze aktuellerFaktor1 := BM.ganzeZufallszahl(1, 20)
Setze aktuellerFaktor2 := BM.ganzeZufallszahl(1, 20)
bf_aufgabe.setzeBeschriftung(BM.text_Aus_GanzeZahl(aktuellerFaktor1) + " * " + BM.text_Aus_GanzeZahl(aktuellerFaktor2) + " =")
bsf_aufgabeStellen.setzeAktiviert(falsch)
bsf_antwortAuswerten.setzeAktiviert(wahr)
setzeStandardBefehlsschaltfläche(bsf_antwortAuswerten)
tf_ergebnis.setText("")
bf_kommentar.setzeBeschriftung("")

bsf_aufgabeStellenKlick()

aufgabeStellen()

bsf_antwortAuswertenKlick()

Setze aktuelleAntwort := BM.ganzeZahl_Aus_Text(tf_ergebnis.ermittleText())	
aktuelleAntwort = aktuellerFaktor1 * aktuellerFaktor2	
vv	
Setze korrekteAntworten := korrekteAntworten + 1	bf_kommentar.setzeBeschriftung("Leider falsch! Das Ergebnis lautet: " + BM.text_Aus_GanzeZahl(aktuellerFaktor1 * aktuellerFaktor2))
bf_kommentar.setzeBeschriftung("Richtig!")	
Setze anzahlAufgaben := anzahlAufgaben + 1	
statistikAusgeben()	
bsf_aufgabeStellen.setzeAktiviert(wahr)	
bsf_antwortAuswerten.setzeAktiviert(falsch)	
setzeStandardBefehlsschaltfläche(bsf_aufgabeStellen)	

statistikAusgeben()

bf_statistik1.setzeBeschriftung("Sie haben " + BM.text_Aus_GanzeZahl(korrekteAntworten) + " / " + BM.text_Aus_GanzeZahl(anzahlAufgaben) + " Aufgaben korrekt gelöst.")
DefiniereVariable prozent als Kommazahl
DefiniereVariable prozentGerundet als Kommazahl
Setze prozent := 100 * BM.kommazahl_Aus_GanzeZahl(korrekteAntworten) / BM.kommazahl_Aus_GanzeZahl(anzahlAufgaben)
Setze prozentGerundet := BM.kommazahl_Aus_GanzeZahl(BM.ganzeZahl_Aus_Kommazahl(prozent * 10 + 0.5)) / 10
bf_statistik2.setzeBeschriftung("Das entspricht einem Anteil von " + BM.text_Aus_Kommazahl(prozentGerundet) + "%.")

6.3.3 Multiplikator mit separater Steuerungsklasse

Projekt: Multiplikator_Oberflaeche_SeparateSteuerungsklasse.semi

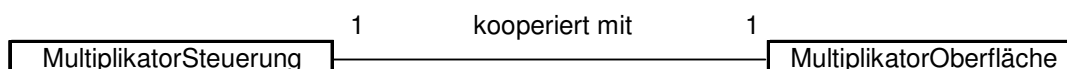
MVC (Model – View – Controller)

Bei der Erstellung von Software sollte man sich eigentlich stets an das Prinzip halten, Model (Datenmodell), View (Oberfläche) und Controller (Steuerung) in separate Programmteile, d.h. in separate Klassen, aufzuspalten.

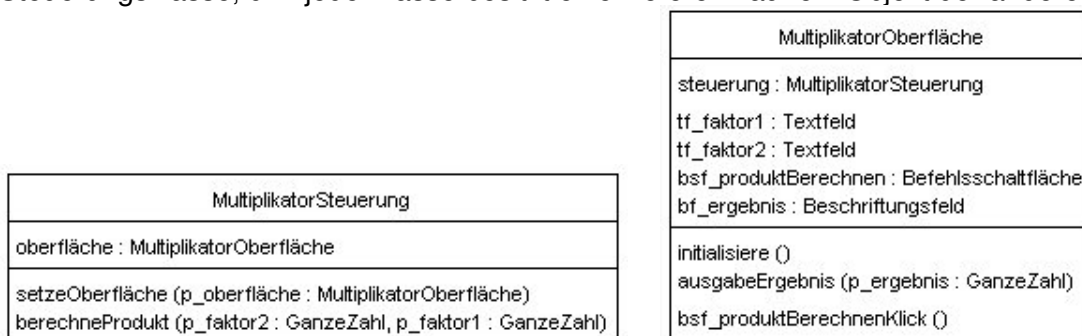
Die Berücksichtigung dieses Prinzips soll nun am einfachen Beispiel des Multiplikators aufgezeigt werden:

Da die Daten beim Multiplikator vernachlässigbar sind, verzichten wir auf die Dreiteilung. Eine Trennung in eine Steuerungsklasse (MultiplikatorSteuerung) und eine Oberflächenklasse (MultiplikatorOberfläche) erscheint jedoch sinnvoll.

Wir haben dann zwei Klassen und eine bidirektionale 1:1 Assoziation, da jedes Steuerungsobjekt genau auf ein Oberflächenobjekt zugreift und jedes Oberflächenobjekt genau ein Steuerungsobjekt mit eingegebenen Daten versorgen soll:



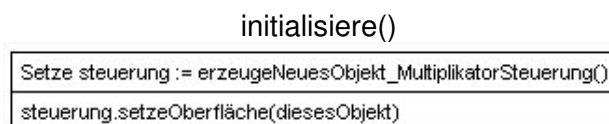
Implementiert wird diese Assoziation mit einem Attribut `steuerung` (Datentyp `MultiplikatorSteuerung`) der Oberflächenklasse und einem Attribut `oberfläche` (Datentyp `MultiplikatorOberfläche`) der Steuerungsklasse, d.h. jede Klasse besitzt eine Referenz auf ein Objekt der anderen Klasse.



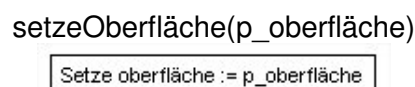
Man hat beim Programmstart dann nur noch dafür zu sorgen, dass beide Attribute tatsächlich auf ein Objekt der Gegenseite verweisen und nicht auf ein „nullObjekt“, d.h. ins Leere.

Dies geschieht folgendermaßen:

Die Startanweisung **erzeugeNeuesObjekt_MultiplikatorOberfläche().initialisiere()** erzeugt ein Objekt der Klasse `MultiplikatorOberfläche` und führt dessen nachfolgend aufgeführte Methode `initialisiere()` aus:



Der Algorithmus der Methode `setzeOberfläche(p_oberfläche : MultiplikatorOberfläche)` der Steuerungsklasse besteht nur aus einer Anweisung und setzt den erhaltenen Parameter gleich ihrem Attribut `oberfläche`:



Damit ist die Assoziation realisiert.

Die jeweiligen Methoden erledigen die Tätigkeiten, welche in der jeweiligen Klasse als Aufgabenbereich gesehen werden:

Die nachfolgend aufgeführte Methode `bsf_produkBerechnenKlick()` der Oberflächenklasse liest die Textfelder aus, die Berechnung ist die Aufgabe der Steuerung:

`bsf_produkBerechnenKlick()`

DefiniereVariable faktor1 als GanzeZahl
DefiniereVariable faktor2 als GanzeZahl
Setze faktor1 := BM.ganzeZahl_Aus_Text(tf_faktor1.ermittleText())
Setze faktor2 := BM.ganzeZahl_Aus_Text(tf_faktor2.ermittleText())
steuerung.berechneProdukt(faktor1, faktor2)

Die Methode `berechneProdukt()` der Steuerungsklasse berechnet das Ergebnis, wobei dessen Ausgabe wiederum Aufgabe der Oberfläche ist:

`berechneProdukt()`

DefiniereVariable ergebnis als GanzeZahl
Setze ergebnis := p_faktor1 * p_faktor2
oberfläche.ausgabeErgebnis(ergebnis)

Die Oberfläche gibt das Ergebnis mit der Methode `ausgabeErgebnis(p_ergebnis : GanzeZahl)` auf dem Beschriftungsfeld `bf_ergebnis` aus:

`ausgabeErgebnis(p_ergebnis)`

<code>bf_ergebnis.setzeBeschriftung("Ergebnis: " + BM.text_Aus_GanzeZahl(p_ergebnis))</code>
--

Die Aufteilung in zwei Klassen mag bei diesem Beispiel lächerlich erscheinen. Die Einhaltung dieses Prinzips bringt aber bei komplexeren Programmen wesentliche Vorteile da kleinere und sinngemäß getrennte Einheiten insbesondere im Rahmen der Vererbung Vorteile u.a. durch mehr Flexibilität bringen.

7. Anhang

7.1 Methoden der Klasse BM (Basismethoden) mit Beispiel

7.1.1 Eingaben - Konsole

eingabeText() : Text

Beispiel:

Definiere Variable eingegebenerText als Text

Setze eingegebenerText := BM.eingabeText()

Liest vom Benutzer über die Konsole eine Eingabe als Text ein.

eingabeGanzeZahl() : GanzeZahl

Beispiel:

Definiere Variable zahl als GanzeZahl

Setze zahl := BM.eingabeGanzeZahl()

Liest vom Benutzer über die Konsole eine Eingabe als GanzeZahl ein.

eingabeKommazahl() : Kommazahl

Beispiel:

Definiere Variable zahl als Kommazahl

Setze zahl := BM.eingabeKommazahl()

Liest vom Benutzer über die Konsole eine Eingabe als Kommazahl ein.

eingabeGanzeZahlGroß() : GanzeZahlGroß

Beispiel:

Definiere Variable zahl als GanzeZahlGroß

Setze zahl := BM.eingabeGanzeZahlGroß()

Liest vom Benutzer über die Konsole eine Eingabe als GanzeZahlGroß ein.

7.1.2 Ausgaben - Konsole

ausgabeText(p_text : Text)

Beispiel:

BM.ausgabeText(„Hallo Welt“)

Gibt auf der Konsole den angegebenen Text aus (ohne Zeilenumbruch).

ausgabeTextZU(p_text : Text)

Beispiel:

BM.ausgabeTextZU(„Hallo Welt“)

Gibt auf der Konsole den angegebenen Text aus (mit Zeilenumbruch).

7.1.3 Zufallszahlen

zufallszahl0Bis1() : Kommazahl

Beispiel:

Definiere Variable zahl als Kommazahl

Setze zahl := BM.zufallszahl0Bis1()

Gibt eine Zufallszahl aus dem Intervall [0 ; 1] zurück.

ganzeZufallszahl(p_minimum : GanzeZahl, p_maximum : GanzeZahl) : GanzeZahl

Beispiel:

Definiere Variable zahl als GanzeZahl

Setze zahl := BM.ganzeZufallszahl(1, 20)

Gibt eine ganze Zufallszahl aus dem Intervall [p_minimum ; p_maximum] zurück.

7.1.4 Dateiverwaltung

schreibZeileInFile(p_fileName : Text, p_text : Text)

Beispiel:

BM.schreibZeileInFile(„namen.txt“, „hallo“)

Hängt an die angegebene Textdatei eine Zeile mit dem entsprechenden Inhalt an.

liesZeileAusFile(p_fileName : Text, p_zeilenNummer : GanzeZahl) : Text

Beispiel:

Definiere Variable meinText als Text

Setze meinText := BM.liesZeileAusFile(„namen.txt“, 3)

Gibt den Textinhalt der angegebenen Zeile der Datei zurück.

initialisiereFile(p_fileName : Text)

Beispiel:

BM.initialisiereFile(„namen.txt“)

Initialisiert die angegebene Datei, d.h. der nächste Aufruf von `schreibZeileInFile` löscht das File und fügt als erste Zeile den als Parameter angegebenen Text an.

7.1.5 Texte

teilText(p_text : Text, p_start : GanzeZahl, p_ende : GanzeZahl) : Text

Beispiel:

Definiere Variable meinText als Text

Setze meinText := BM.teilText(„hallohallo“, 3,6)

Gibt den angegebenen Teiltext zurück, in diesem Fall „loha“, da sich gemäß Definition der erste Buchstabe auf Position 0 und nicht auf Position 1 befindet.

textLänge(p_text : Text) : GanzeZahl

Beispiel:

Definiere Variable länge als GanzeZahl

Setze länge := BM.textLänge(„hallohallo“)

Gibt die Zeichenlänge des Textes zurück, in diesem Fall 10.

textIdentisch(p_text1 : Text, p_text2 : Text) : Wahrheitswert

Beispiel:

Definiere Variable sindIdentisch als Wahrheitswert

Setze sindIdentisch := BM.textIdentisch(„hallo“, „allo“)

Überprüft die beiden Text auf Identität und gibt wahr oder falsch zurück, in diesem Fall falsch.

texteAlphabetischGeordnet(p_text1 : Text, p_text2 : Text) : Wahrheitswert

Beispiel:

Definiere Variable sindGeordnet als Wahrheitswert

Setze sindGeordnet := BM.texteAlphabetischGeordnet(„hallo“, „xyz“)

Überprüft die beiden Texte auf alphabetische Ordnung und gibt wahr oder falsch zurück, in diesem Fall wahr.

7.1.6 Mathematik

sin(p_winkelmaß : Kommazahl) : Kommazahl

Beispiel:

Definiere Variable zahl als Kommazahl

Setze zahl := BM.sin(3.1415)

Gibt den Sinuswert des angegebenen Winkelmaßes (Bogenmaß) zurück.

All weiteren mathematische Methoden agieren analog.

cos(p_winkelmaß : Kommazahl) : Kommazahl

tan(p_winkelmaß : Kommazahl) : Kommazahl

arcSin(p_sinWert : Kommazahl) : Kommazahl

arcCos(p_cosWert : Kommazahl) : Kommazahl
arcTan(p_tanWert : Kommazahl) : Kommazahl
natürlicherLog(p_wert : Kommazahl) : Kommazahl
log10(p_wert : Kommazahl) : Kommazahl
potenzwert(p_basis : Kommazahl, p_exponent : Kommazahl) : Kommazahl
eulerscheZahlHoch(p_exponent : Kommazahl) : Kommazahl

7.1.7 Typkonvertierungen

ganzeZahl_Aus_Text(p_text : Text) : GanzeZahl

Beispiel:

Definiere Variable eingegebenesAlter als Text

Definiere Variable alter als GanzeZahl

Setze eingegebenesAlter := tf_eingabeAlter.ermittleText()

Setze alter := BM.ganzeZahlAusText(eingegebenesAlter)

Bei dem Textfeld tf_eingabeAlter einer Oberfläche wird der Textinhalt ermittelt und gleich der Variable eingegebenesAlter gesetzt. Anschließend wird der zur ganzen Zahl umgewandelte Text der Variablen alter zugewiesen.

Alle weiteren Typkonvertierungen funktionieren analog.

text_Aus_GanzeZahl(p_ganzeZahl : GanzeZahl) : Text

ganzeZahlGroß_Aus_Text(p_text: Text) : GanzeZahlGroß

text_Aus_GanzeZahlGroß(p_ganzeZahlGroß : GanzeZahlGroß) : Text

kommazahl_Aus_Text(p_text: Text) : Kommazahl

text_Aus_Kommazahl(p_kommazahl : Kommazahl) : Text

ganzeZahlGroß_Aus_GanzeZahl(p_ganzeZahl : GanzeZahl) : GanzeZahlGroß

ganzeZahl_Aus_GanzeZahlGroß(p_ganzeZahlGroß : GanzeZahlGroß) : GanzeZahl

kommazahl_Aus_GanzeZahl(p_ganzeZahl : GanzeZahl) : Kommazahl

ganzeZahl_Aus_Kommazahl(p_kommazahl : Kommazahl) : GanzeZahl

kommazahl_Aus_GanzeZahlGroß(p_ganzeZahlGroß : GanzeZahlGroß) : Kommazahl

ganzeZahlGroß_Aus_Kommazahl(p_kommazahl : Kommazahl) : GanzeZahlGroß

7.1.8 Sonstiges

programmBeenden()

Beendet das laufende Programm.

warteTausendstelSekunden(p_tausendstelSekunden : GanzeZahl)

Beispiel:

BM.warteTausendstelSekunden(500)

Hält den Programmablauf für die angegebene Anzahl an Millisekunden an.

spieleWavDatei(p_fileName : Text)

Beispiel:

BM.spieleWavDatei(„hallo.wav“)

Spielt die angegebene WAV-Datei ab.

7.2 Methoden der Oberflächenelemente

Erläutert werden nur die Methoden, deren Handhabung nicht selbsterklärend erscheint.

7.2.1 Hauptfenster (JFrame)

setzeTitel(p_titel : Text)

setzeStandardBefehlsschaltfläche(p_befehlsschaltfläche : Befehlsschaltfläche)

Beispiel:

setzeStandardBefehlsschaltfläche(bsf_antwortAuswerten)

Die Standardbefehlsschaltfläche ist die Befehlsschaltfläche eines Fensters, welche durch die Eingabetaste gedrückt wird und etwas dicker umrandet ist.

setzeSichtbar(p_sichtbar : Wahrheitswert)

setzeAusdehnungMaximiert()

setzeAusdehnungNormal()

setzeFensterFarbe(p_farbe : Farbe)

Beispiel:

setzeFensterFarbe(erzeugeNeuesObjekt_Farbe(128,255,255))

Beim Aufruf wird ein Objekt der Klasse Farbe als Parameter übergeben. Die drei Werte (0-255) stehen für Rot, Grün und Blau, die RGB-Werte der additiven Farbmischung.

setzePosition(p_xPosition : GanzeZahl, p_yPosition : GanzeZahl)

setzeGröße(p_breite : GanzeZahl, p_hoehe : GanzeZahl)

ermittleTitel() : Text

7.2.2 Beschriftungsfeld (JLabel)

setzeSichtbar(p_sichtbar : Wahrheitswert)

setzeAktiviert(p_aktiv : Wahrheitswert)

setzeHintergrundFarbe(p_farbe : Farbe)

setzeVordergrundFarbe(p_farbe : Farbe)

setzeSchriftArt(p_schriftArt : SchriftArt)

setzeAusrichtung(p_ausrichtung : Konstante)

setzeBeschriftung(p_beschriftung : Text)

setzePosition(p_xPosition : GanzeZahl, p_yPosition : GanzeZahl)

setzeGröße(p_breite : GanzeZahl, p_hoehe : GanzeZahl)

ermittleBeschriftung() : Text

7.2.3 Textfeld (JTextField)

setzeSichtbar(p_sichtbar : Wahrheitswert)

setzeAktiviert(p_aktiv : Wahrheitswert)

setzeHintergrundFarbe(p_farbe : Farbe)

setzeVordergrundFarbe(p_farbe : Farbe)

setzeSchriftArt(p_schriftArt : SchriftArt)

setzeAusrichtung(p_ausrichtung : Konstante)

setzeText(p_text : Text)

setzePosition(p_xPosition : GanzeZahl, p_yPosition : GanzeZahl)

setzeGröße(p_breite : GanzeZahl, p_hoehe : GanzeZahl)

ermittleText() : Text

fordereFokusAn()

Durch den Aufruf dieser Methode erhält das Textfeld den Cursor.

7.2.4 Befehlsschaltfläche (JButton)

setzeSichtbar(p_sichtbar : Wahrheitswert)
setzeAktiviert(p_aktiv : Wahrheitswert)
setzeHintergrundFarbe(p_farbe : Farbe)
setzeVordergrundFarbe(p_farbe : Farbe)
setzeSchriftArt(p_schriftArt : SchriftArt)
setzeAusrichtung(p_ausrichtung : Konstante)
setzeBeschriftung(p_beschriftung : Text)
setzePosition(p_xPosition : GanzeZahl, p_yPosition : GanzeZahl)
setzeGröße(p_breite : GanzeZahl, p_hoehe : GanzeZahl)
ermittleBeschriftung() : Text
fordereFokusAn()

7.2.5 JaNeinAuswahl (JCheckBox)

setzeSichtbar(p_sichtbar : Wahrheitswert)
setzeAktiviert(p_aktiv : Wahrheitswert)
setzeHintergrundFarbe(p_farbe : Farbe)
setzeVordergrundFarbe(p_farbe : Farbe)
setzeSchriftArt(p_schriftArt : SchriftArt)
setzeAusrichtung(p_ausrichtung : Konstante)
setzeBeschriftung(p_beschriftung : Text)
setzeMarkiert(p_markiert : Wahrheitswert)
setzePosition(p_xPosition : GanzeZahl, p_yPosition : GanzeZahl)
setzeGröße(p_breite : GanzeZahl, p_hoehe : GanzeZahl)
ermittleBeschriftung() : Text
istMarkiert() : Wahrheitswert
fordereFokusAn()

7.2.6 JaNeinKnopf (JRadioButton)

setzeSichtbar(p_sichtbar : Wahrheitswert)
setzeAktiviert(p_aktiv : Wahrheitswert)
setzeHintergrundFarbe(p_farbe : Farbe)
setzeVordergrundFarbe(p_farbe : Farbe)
setzeSchriftArt(p_schriftArt : SchriftArt)
setzeAusrichtung(p_ausrichtung : Konstante)
setzeBeschriftung(p_beschriftung : Text)
setzeMarkiert(p_markiert : Wahrheitswert)
setzePosition(p_xPosition : GanzeZahl, p_yPosition : GanzeZahl)
setzeGröße(p_breite : GanzeZahl, p_hoehe : GanzeZahl)
ermittleBeschriftung() : Text
istMarkiert() : Wahrheitswert
fordereFokusAn()

7.2.7 Grafik (ImageIcon)

setzeSichtbar(p_sichtbar : Wahrheitswert)
setzeHintergrundFarbe(p_farbe : Farbe)
setzeGrafikDatei(p_grafikDatei : Text)
setzePosition(p_xPosition : GanzeZahl, p_yPosition : GanzeZahl)
setzeGröße(p_breite : GanzeZahl, p_hoehe : GanzeZahl)

7.3 Kurzreferenz SEMI-OOS

Schulgemäße Entwicklungsumgebung zur Modellierung und Implementierung objektorientierter Software

A. Allgemeines

Stellen Sie zunächst sicher, dass auf Ihrem Computer ein passender Java-Compiler mit Interpreter (Paket „JDK“, mindestens Version 5) installiert ist, den Sie kostenlos aus dem Internet beziehen können.

Das Programm SEMI-OOS hat kein Installationsprogramm, da es nur aus einer einzigen Datei, dem Java-Archiv „System_SEMIOOS.jar“, besteht. Diese Datei ist ähnlich wie eine „exe-Datei“ ausführbar.

Beim ersten Start wird aus dem Java-Archiv diese Kurzreferenz und der Lizenzvertrag extrahiert.

SEMI-OOS besitzt kein Hauptfenster-Menü. Alle Grundfunktionen sind über das Befehlsschaltflächen-Paket im oberen Bereich, welches im Folgenden Steuerungskonsole genannt wird, erreichbar. Im unteren Bereich wird die jeweilige Arbeitsoberfläche angezeigt. Kontextmenüs erreichen Sie über die rechte Maustaste.

Klicken Sie bitte nach dem ersten Start zunächst auf die Befehlsschaltfläche „Einstellungen / Preferences“ der Steuerungskonsole.

SEMI-OOS benötigt den Standort des Java-Compilers auf Ihrem System. Sie können diesen automatisch suchen lassen oder manuell die Datei „javac.exe“ wählen, die sich in der Regel im Installationsverzeichnis ... \ Programme \ Java \ jdk..... \ bin befindet.

Im unteren Bereich kann neben der Sprache und der System-Schriftart gewählt werden, ob die Oberfläche von SEMI-OOS für Einsteiger oder Fortgeschrittene dargestellt werden soll. Im Einsteigermodus können keine Klassenattribute und –methoden, keine Feldlisten und keine Zugriffsberechtigungen festgelegt werden.

Die Einstellungen werden in der Datei „semioos.pref“ fixiert.

B. Dokumentation der einzelnen Arbeitsbereiche

Auswahl über die Befehlsschaltflächen der Steuerungskonsole:

a) „Projekt neu“, „Projekt öffnen“, „Projekt speichern“ und „Projekt speichern unter“
Ein Projekt besteht aus nur einer einzigen Datei mit der Endung „.semi“, welche im üblichen Rahmen mit den genannten Befehlsschaltflächen verwaltet wird.

b) „Klasse“

Sie verwenden diesen Arbeitsbereich zur Definition von Klassen mit Attributen, Methoden und zugehörigen Parametern.

Im Textfeld wird jeweils die Bezeichnung eingegeben, gespeicherte Inhalte finden Sie im Auswahlfeld unter dem Befehlsschaltflächentrio „neu“, „speichern“ und „löschen“.

Die Speicherung erfolgt hierarchisch, d.h. falls Sie eine Klasse speichern, werden alle zugehörigen Attribute und Methoden, die im jeweiligen Auswahlfeld abgelegt sind, dieser Klasse zugeordnet.

Gleiches gilt beim Speichern einer Methode für die zugehörigen Parameter.

Die Oberfläche zur Definition der Algorithmen zu den Methoden erreichen Sie über die Befehlsschaltfläche „Methode“ der Steuerungskonsole.

c) „Oberflächen-Klasse“

Für jedes erstellte Fenster legt SEMI-OOS eine Klasse an, wobei für die im Fenster gezeichneten Objekte entsprechende Attribute erzeugt werden.

Zudem wird für jede Befehlsschaltfläche eine Methode, die beim Drücken der Befehlsschaltfläche ausgeführt wird, definiert.

Mit dem Klasseneditor (Befehlsschaltfläche „Klasse“, s.o.) können weitere Attribute und Methoden zu einer Oberflächen-Klasse hinzugefügt werden.

d) „Methode“

Die Festlegung der Algorithmen der Methoden erfolgt mit dem Struktogramm nach Nassi-Shneiderman, einer (programmiersprachenunabhängigen) Modellierungstechnik für Algorithmen.

Bei SEMI-OOS wird eine minimale aber effiziente Auswahl an algorithmischen Strukturen angeboten, welche auf der linken Seite angewählt (orange Markierung) und per rechtem Mausklick und dem Menüpunkt „Struktogrammelement einfügen“ ein- bzw. angefügt werden. Klickt man mit der linken Maustaste auf ein Element des Struktogramms, so öffnet sich für dieses Element der Editor im unteren Fensterbereich.

Beim Editieren der Anweisungen bzw. Bedingungen von Struktogrammelementen werden Textbausteine angeboten. Die Bausteine beinhalten auch eine Menge sehr nützlicher Basismethoden einer in SEMI-OOS integrierten Klasse BM.

Im Editor gelangt man mit der Cursor-Taste „nach-oben“ in das Auswahlfeld, die Eingabetaste fügt den markierten Baustein ein und mit der Esc-Taste kann das Auswahlfeld stets verlassen werden.

Wird mit dem Eintippen eines Bausteines begonnen, so erfüllt das Auswahlfeld die Aufgabe der Textvervollständigung, welche mit der Eingabetaste dann abgeschlossen werden kann.

Werden nur einstellige Bausteine eingegeben, z.B. bei der Eingabe eines Variablennamens und ist der Baustein eindeutig, so wird automatisch nach dem Tippen des Zeichens der Baustein übernommen.

Wurde ein korrekter und vollständiger Ausdruck zusammengesetzt, so wird die Befehlsschaltfläche „übernehmen“ aktiv, welche dann gedrückt werden kann. Ist „übernehmen“ aktiv, so wird sie beim Fortfahren z.B. mit der Bearbeitung eines anderen Struktogrammelementes automatisch gedrückt. Ist „übernehmen“ nicht aktiv, so werden aktuelle Änderungen beim Fortfahren verworfen.

e) „Vererbungsbeziehung“

Diese Arbeitsfläche dient zur Definition von Vererbungsbeziehungen zwischen Klassen.

Oberflächenklassen können unter SEMI-OOS nicht vererben, sie erben aber von einer vorgegebenen Klasse „Hauptfenster“. In Java entspricht diese der Klasse „JFrame“.

f) „Startanweisung“

Unterstützt durch die Vorgabe von Textbausteinen wird hier die Einstiegsanweisung in das Programm festgelegt.

g) „Implementierung“

Die Befehlsschaltflächen auf der linken Seite bieten folgende Optionen an:

„SEMIOOS Implementierung“

- Implementierung des Projektes in einer neu entwickelten, möglichst intuitiv verständlichen Sprache

„Java Implementierung“

- Implementierung des Projektes in der Sprache Java (ab Version 5.0)

„Java Implementierung kompilieren“

- Erzeugung einer Java-Quellcode-Datei, die den Namen der Projektdatei mit vorangestelltem „System_“ und der Endung „.java“ trägt
- Übersetzung des Quellcodes in Bytecode („class-Dateien“)

„Java Archiv (*.JAR) erzeugen und ausführen“

- Zusammenfassung und Komprimierung der „class-Dateien“ zu einem ausführbaren Java-Archiv („jar-Datei“) und anschließende Ausführung
- Aufräumen der „class-Dateien“

„Java Implementierung als Applet“

- Implementierung des Projektes in Java als Applet. Dazu muss genau eine Oberflächenklasse definiert sein und die Startanweisung mit „erzeugeNeuesObjekt_NameDerOberflächenklasse(“ beginnen.

„Java-Implementierung als Applet kompilieren“

- Erzeugung einer Java-Quellcode-Datei mit der Applet-Implementierung, die den Namen der Oberflächenklasse mit der Endung „.java“ trägt
- Übersetzung des Quellcodes in Bytecode („class-Dateien“)

„Java-Applet-Archiv (*.JAR) und HTML-Gerüst erzeugen“

- Zusammenfassung und Komprimierung der „class-Dateien“ zu einem Java-Applet-Archiv („jar-Datei“)
- Erzeugung einer extern mit einem Browser zu öffnenden HTML-Datei, die ebenfalls den Namen der Oberflächenklasse mit der Endung „.html“ trägt und ein Gerüst zur Einbindung des Applets enthält

h) „unabhängiges Struktogramm“

Sie können zu jeder Methode neben dem Struktogramm, welches in die Implementierung übernommen wird, ein frei formulierbares Struktogramm erstellen. Dieses Struktogramm dient nur der Veranschaulichung und hat für die Implementierung keine Bedeutung.

i) „Einstellungen / Preferences“

Erläuterungen dazu finden Sie unter „Allgemeines“.