universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

„Vermittlung von Grundkonzepten der Programmierung im Kontext der Schule mithilfe von Mikrowelten.

Eine Anleitung für LehrerInnen, die den beispielhaften Aufbau eines Storytelling-Szenarios mit Java-Hamster und BYOB gegenüberstellt.“

Verfasser

Klemens Winkler

angestrebter akademischer Grad

Magister der Naturwissenschaft (Mag.rer.nat.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 190 412 884
Studienrichtung lt. Studienblatt:	Lehramtsstudium UF Physik und UF Informatik und Informatikmanagement
Betreuer:	Univ.-Prof. Dr. Wilfried Grossmann

Inhaltsverzeichnis

A Einleitung

Wenn ich mich an meine Anfänge im Programmieren zurück erinnere, dann handelt es sich dabei leider nicht um allzu positive Erinnerungen. Ich war damals nämlich absoluter Neuling auf diesem Gebiet und hatte die Aufgabe in zwei Semestern Grundkenntnisse in der Programmiersprache Java zu erlangen. Schon bei meinem ersten „hello world“-Programm bot sich mir ein unübersichtliches Gewirr, angefangen bei `public static void main`, dar, ohne auch nur ansatzweise Verständnis dafür zu empfinden. Nach einer gewissen Zeit konnte ich die genannten Begriffe zwar schon unterscheiden und definieren, jedoch waren die Einsatzmöglichkeiten in meiner Programmierpraxis nur äußerst beschränkt. Zwei erfolglose Prüfungs-Antritte später, widmete ich mich zwei Semester lang ausführlicher Zusatzlektüre. Gepaart mit zahlreichen Stunden vor dem PC, konnte ich danach schlussendlich von mir behaupten, dass ich eigenständig programmieren kann.

Legt man nun dieses Szenario auf die Schule um, so wird man schnell erkennen, dass es nicht der Realität entspricht, Schülern[1] Programmieren anhand von zusätzlichem Textstudium beizubringen. Deswegen haben Entwickler schon früh erkannt, dass das Erlernen der Grundkonzepte der Programmierung leichter fällt, wenn eventuell verwirrende, theoretische Hintergründe schlicht und einfach ausgeblendet werden. Das ist nun der Punkt an denen die, im Titel erwähnten, Mikrowelten zum Einsatz kommen. Die hier vorgestellten Mikrowelten sind Lernumgebungen, in denen theoretische Grundlagen des Programmierens, sowie Syntax und Fehlermeldungen auf ein Minimum reduziert werden. So kann man sich mit vollem Eifer den eigentlich spannenden Dingen, wie Problemlösung und der Gestaltung eigener Programme, widmen. Die Zahl dieser Mikrowelten ist jetzt schon kaum mehr überschaubar, jedoch zeigt sich bei genauerer Betrachtung so mancher Qualitätsunterschied.

Diese Arbeit beschäftigt sich deshalb mit zwei ausgewählten Mikrowelten, die jeweils einen unterschiedlichen Zugang zum Erlernen von Programmieren haben. Einerseits der Java-Hamster-Simulator, der den Ansatz verfolgt textuelles Programmieren zu vereinfachen, indem Befehle und Methoden in Umgangssprache formuliert werden. Andererseits die Lernumgebung BYOB, die komplett auf Programmcode verzichtet und diesen stattdessen mit vorgefertigten Kommandoblöcken zur Verfügung stellt. Beide haben jedoch gemein, dass die Resultate der Programmier-Tätigkeit mithilfe einer visuellen Simulation veranschaulicht werden können.

1 Aus Gründen der Lesbarkeit wird in dieser Arbeit stets die maskuline Form verwendet.

Ziel der folgenden Ausarbeitung ist es ein, im Unterricht brauchbares, Konzept zu erstellen, das Lehrern als Anleitung für den schulischen Einstieg ins Programmieren dienen soll. Im Zuge dessen sollen Fragen wie „Mit welchen Programmier-Konzepten haben Anfänger gewöhnlich die größten Schwierigkeiten?“, „Wie kann man es erreichen, dass die Programmierbeispiele das Interesse der Schüler wecken und der Lernerfolg dabei möglichst hoch ist?“ oder „Wo unterscheiden sich die beiden Mikrowelten bei der Lösung eines bestimmten Problems?“ beantwortet werden.

Um dies zu erreichen bietet diese Arbeit zu Beginn eine theoretische Einführung darüber was überhaupt unter *Programmieren* zu verstehen ist und wie dessen Einsatz in der Schule gerechtfertigt werden kann. Anschließend wird das Problemlösen mithilfe von Algorithmen vorgestellt und die Zielgruppe ermittelt. Es folgt ein Überblick über wesentliche Konzepte des Programmierens, wobei zuerst auf rein imperative oder *klassische* Programmierkonzepte eingegangen wird und danach auf die erweiterten, objektorientierten Konzepte. Im Anschluss wird gezielt darauf eingegangen welche Konzepte bei Programmieranfängern, laut gängiger Studien, die größten Schwierigkeiten verursachen. Das Ganze geht über in eine bildungstheoretischen Einführung zum *Konstruktivismus*, der als Grundlage für die Entwicklung von Mikrowelten dient. Hier wird besonderer Wert auf die Unterform des *Konstruktionismus* gelegt, der spezifisch das Lernen anhand vom Nutzer erstellten Programmen erklärt. Darauf aufbauend folgt ein thematischer Ausflug in die Mikrowelt. Hier werden historische Fakten und grundlegende Eigenschaften dargestellt.

In Bezugnahme auf die Zielgruppe werden passende Auszüge aus dem Lehrplan ausgewählt, die die Relevanz dieser Arbeit für die Schulinformatik bestätigen. Auf Grundlage der bisher erwähnten Theorie-Konzepte werden anschließend realistische Lehr- und Lernziele formuliert. Der Theorieteil wird abgeschlossen durch einen Einblick in die Praxis des *Storytelling*, also des Geschichten-Erzählens. Der Fokus liegt hierbei auf emotionalem Lernen und dem daraus entwickelten *goal-based scenario*, welches ein Modell zur Gestaltung von Lernumgebungen darstellt und in dieser Arbeit zum Einsatz kommt.

Der zweite Teil der Arbeit befasst sich mit der wissenschaftlichen Vorgehensweise. Wobei der Leser einen Einblick erhält wie der Praxisteil aufgebaut ist und wie er vonstatten geht. Außerdem wird das für diese Arbeit umgesetzte goal-based scenario vorgestellt, welches die Beispielaufgaben in einen erzählerischen Rahmen verpackt.

Auf Grundlage des vorherigen Abschnitts befasst sich der Praxisteil mit der Analyse der

Mikrowelten. Bei diesem Punkt werden die beiden Mikrowelten anhand, für den Einsatz in der Schule, relevanter Punkte einander gegenübergestellt. Dazu zählen die Installation, die Funktionsweise, das Design und die Handhabung der Software. Dabei wird besonderer Wert auf Hilfestellungen und wichtige Informationen im Umgang mit den Lernumgebungen gelegt, welche der Lehrperson den Einstieg in die Materie erleichtern sollen. Der Großteil dieses Vergleichs beschäftigt sich jedoch mit der exemplarischen Ausarbeitung der Beispielaufgaben und den zugehörigen Infotexten, die die Schüler bei der eigenen Bearbeitung unterstützen sollen. Dazu sei gesagt, dass die exemplarische Ausarbeitung dem Lehrer auch hier nur als Orientierung und nicht als Vorschrift dienen soll. Zuguterletzt wird untersucht ob sich bei der Bearbeitung dieser Aufgaben für den Schüler Erweiterungs- oder Verbesserungsmöglichkeiten finden lassen.

B Theorie

1 Programmieren

Unter Programmieren versteht man die „Entwicklung von Computerprogrammen“ (Boles 2008, S. 5). Zum Aufgabenbereich des Programmierens zählt die Lösung von Problemen mithilfe von Computern, wozu auch die Methoden und Denkweisen, die an dem Lösungsprozess beteiligt sind, gehören. Das soll bedeuten, dass nicht nur das Schreiben eines Programms unter Programmieren verstanden wird, sondern auch die Denk- und Handlungsschritte, die dazu führen. Dies zeigt bereits die Vielschichtigkeit jenes Bereiches, da sowohl Entwurfstechniken, Lösungsstrategien als auch kreative Prozesse zum Programmieren zählen. Das Programm kommt hierbei einer Art Anleitung gleich, die dem Computer bei der Lösung des Problems dient (vgl. Boles 2008, S. 5). Um Programme zu gestalten werden oft allgemeine Verfahren zur Lösung von Problemen eingesetzt, welche in der Fachsprache unter dem Namen *Algorithmus* bekannt sind (vgl. Schubert et al. 2011, S. 4).

Als Werkzeug zur Formulierung von Algorithmen dient dem Programmierer eine einfache und eindeutige Sprache, die es ihm ermöglicht mit dem Computer zu kommunizieren. Zwischen dem Programm und dem Computer agiert der sogenannte *Compiler*. Er ist eine Art *Übersetzer*, der das Programm in eine, für den Computer verständliche, Form bringt (vgl. Ullenboom 2011, Abschnitt 1.5.2). Die Rückmeldungen in diesem Interaktionsprozess beschränken sich jedoch höchstens auf Fehlermeldungen, welche bei klassischen Programmiersprachen vom Programmierer oftmals hohe Sprachkenntnisse erfordern.

2 Motivation für Programmierunterricht

Wenn es um die Sinnhaftigkeit dessen geht, ob man Programmieren in der Schule überhaupt in die Jahresplanung aufnehmen sollte, so muss man die aktuelle Stellung der Informatik in der Gesellschaft betrachten. Oftmals wird kritisiert, dass der Informatikunterricht sich ausschließlich damit beschäftigt die Anwender-Ebene abzudecken und dabei nicht hinter die Software-Fassade blickt (vgl. Reichert et al. 2005, S. 1). Studien aus Deutschland (vgl. Hanselmann 2009, S. 2) und der Schweiz (Umbach-Daniel et al. 2008) bekräftigen dieses Bild. Besonders schwerwiegend ist die daraus folgende öffentliche Meinung, dass die Informatik es zum Ziel habe, Software-Experten auszubilden, die alle Tricks und Kniffe eines jeden aktuellen Programms kennen (vgl. Hromkovic 2012, S. 6). Dass diese Meinung unangebracht ist, erkennt

man schnell, wenn man bedenkt, wie zahlreich Software am Markt vertreten ist, wie häufig diese vom Hersteller abgeändert wird und wie selten Befehle der einen Anwendung auf eine Andere übertragbar sind (vgl. Reichert et al. 2005, S. 5).

Diesem Bild gilt es mit aller Kraft entgegen zu arbeiten, da es erstens eine falsche Kurzlebigkeit der Informatik vermittelt und zweitens bei weitem nicht die enorme Vielfalt der Informatik widerspiegelt. Informatik bedeutet nämlich nicht, dass man an die Fesseln einer vorgegebenen Programmstruktur gelegt bleiben muss, sondern, dass man beispielsweise selbst kreativ tätig sein und eigene Ideen für Anwendungen realisieren kann. Das ist, laut Kölling (vgl. 2010, S. 34), auch der Punkt wo die intrinsische Motivation eines jeden Menschen Dinge zu kreieren ins Spiel kommt. Um dies umsetzen zu können, bedarf es jedoch geeigneter Fertigkeiten um mit dem Computer situationsspezifisch in Interaktion zu treten und da ist das Erlernen von Programmierkonzepten eine Grundbedingung dafür.

Außerdem bietet der Programmierunterricht den Schülern ideale Gelegenheiten, bei der Erstellung komplexer Programme, modulare Entwurfstechniken kennenzulernen. Die Schüler sind damit konfrontiert selbstständig erste einfache Konzepte zu entwickeln und diese dann, mit steigender Komplexität, auf weitere Problemstellungen umzusetzen. Diese modulare Arbeitsweise ist nicht nur in vielen Ingenieursberufen von großer Bedeutung (vgl. Hromkovic 2012, S. 7) sondern fördert auch die Problemlösefähigkeit im Allgemeinen (siehe Punkt 3).

3 Problemlösen mithilfe von Algorithmen

3.1 Begriffsdefinition: Problem

Es wurde bereits erwähnt, dass Programmieren immer Hand in Hand mit dem Lösen von Problemen geht. Nun gilt es erst einmal zu klären was überhaupt unter einem *Problem* verstanden wird. Eine allgemeine Definition liefert der Psychologe Karl Duncker: „Ein Problem entsteht z.B. dann, wenn ein Lebewesen ein Ziel hat und nicht ‚weiß', wie es dieses Ziel erreichen soll." (Duncker 1935, S. 1). Probleme stellen uns also vor Herausforderungen, die wir nicht ohne weiteres lösen können.

Anschließend fährt Duncker fort: „Wo immer der gegebene Zustand sich nicht durch bloßes Handeln (Ausführen selbstverständlicher Operationen) in den erstrebten Zustand überführen lässt, wird das Denken auf den Plan gerufen." (s. o.). In ähnlicher Art und Weise unterscheidet Dietrich Dörner prinzipiell zwischen einem Problem und einer Aufgabe. So beschreibt eine

Aufgabe eine geistige Anforderung, die mithilfe einer bereits bekannten Methode gelöst werden kann. Beispielsweise die Division zweier Zahlen (vorausgesetzt das Individuum weiß wie man zwei Zahlen dividiert). Beim Problemlösen jedoch muss immer etwas Neues geschaffen werden. Der Unterschied was eine Aufgabe und was ein Problem darstellt, liegt also im Vorwissen des Individuums (vgl. Dörner 1987, S. 10f.).

3.2 Problemlösen und Problemlösestrategien

Außerdem kommt bei Dunckers letztem Zitat die Methodik bzw. das menschliche Handeln ins Spiel, welches zur Lösung des Problems beitragen soll. Abhängig vom Problem gibt es Verfahren und Strategien, die man dabei als Unterstützung einsetzen kann. Ich habe bereits erwähnt, dass dies beim Programmieren der Algorithmus ist. Im folgenden werden zwei Modelle vorgestellt, die den algorithmischen Denkprozess veranschaulichen sollen:

„Beim Problemlösen besteht das Ziel darin, das Problem einer Lösung und einer Transformation zuzführen“ (Preiwisch-Seibold 1999, S. 81). Eine Möglichkeit diesen Transformations-Prozess zu gestalten, ist folgendes sechs-Stufen-Schema nach Syslo et al. Es ist speziell auf das Problemlösen mithilfe von Algorithmen ausgelegt (vgl. Syslo et al. 2008):

- Problemsituation: Erfassen und Verstehen des Problems.
- Spezifikation: Anforderung an die Daten, Verhältnis zwischen Input und Output.
- Design: Wahl einer algorithmischen Technik und Datenstruktur.
- Programmierung: Erstellen einer Lösung für den Computer (eigentliches Programm).
- Testen: Feststellung ob gewünschtes Ergebnis eintritt und Effizienz beurteilen.
- Präsentation: Präsentieren der Lösung und Diskussion der Anwendung.

Speziell der Punkt *Design* verdeutlicht dabei aber, dass es in diesem Modell zur Lösung eines Problems unter Umständen eines Vorwissens bzw. an Erfahrung im Bereich des Programmierens bedarf. Natürlich kommt es vor, dass Programmieranfänger durchaus in der Lage sind bereits nach kurzer Zeit eigenständige Lösungsalgorithmen zu entwickeln, in der Regel ist dies jedoch nicht der Fall (vgl. Syslo et al. 2013, S. 43ff.; Preiwisch-Seibold 1999, S. 69ff.; Hornecker 1998, S. 43ff.). Viel mehr wird bei genauerer Betrachtung klar, dass das Problemlösen mithilfe von Computern für eine große Anzahl von Menschen keinen intuitiven

Prozess darstellt, sondern häufig an Erfahrungen mit ähnlichen algorithmischen Lösungen und deren Entwurfsmustern geknüpft ist (dazu mehr unter Punkt 6).

Ein weiterer Ansatz, welcher etwas weniger auf bereits bekannte Programmier-Muster zurückgreift, wäre das *analytische Verfahren zur Problemlösung* (vgl. Preiwisch-Seibold 1999, S. 136), welches ursprünglich von der Lehrerfortbildung der Landeshauptstadt München stammt:

1. Wie lautet das Problem?
   - Suchen einer präzisen und kurzen Problembenennung;
   - Gibt es ein vergleichbares Problem?
   - Kann man das Problem verallgemeinern?
   - Läßt sich das Problem in leichter bearbeitbare, übersichtlichere Teilprobleme zerlegen?
2. Was ist gesucht?
   - Aufzählung sämtlicher Ausgangsobjekte.
3. Was ist bekannt?
   - Aufzählung sämtlicher Eingangsobjekte.
4. Spezifizierung sämtlicher Eingangs- und Ausgangsgrößen
   genaue Festlegung der:
   - gegebenen Eigenschaften aller Eingangsobjekte,
   - gewünschten Eigenschaften aller Ausgangsobjekte,
   - Einführung der notwendigen Hilfobjekte (Zwischen objekte) und der
   - logischen Beziehungen zwischen den Eingangs- und Ausgangsobjekten
5. Entwurf, Überprüfung und Verbesserung des Problemlösungsplans

Die Punkte 1 und 5 dieses Modells könnte man als Richtlinie zum Lösen von Problemen im Allgemeinen bezeichnen. Die Punkte 2-4 hingegen sind speziell auf das Lösen von Problemen mithilfe von Programmen ausgelegt. Der Hauptgrund warum ich dieses Modell überhaupt gewählt habe, liegt darin, dass dabei versucht wird das Problem selbst zu hinterfragen. Es behandelt also zusätzlich zum WAS? auch schon gezielter das WIE? Der Problemlösung.

3.3 Weiterführende Gedanken

Ziel dieser Ausarbeitung wird es, auf Grundlage der oben genannten Punkte, sein eine Art *hybride* Fragestellung zu wählen, die sowohl die Umsetzung von neuen als auch von bekannten Strategien voraussetzt. So wird etwa der Einsatz von Konstrukten wie Schleifen oder bedingten Anweisungen für die meisten Schüler etwas Neues darstellen, jedoch eine Bewegung wie das Stufensteigen durch bekannte Denkmuster abstrahierbar sein. Das heißt, dass ein entscheidender Schritt bei der Lösungsfindung insofern darin besteht eine Verbindung zwischen Alltagserfahrungen und einer simulierten, virtuellen Welt herzustellen. Hier bieten die visuellen Darstellungen von Programmen innerhalb der beiden Lernumgebungen die ideale Möglichkeit die Probleme im wahrsten Sinne *vor Augen zu führen*. Welche entscheidende Rolle dabei außerdem Geschichten spielen können, wird später noch vorgestellt.

Darüber hinaus ist für diese Arbeit insbesondere die Zerlegung des Problems in mehrere Teilprobleme relevant. Bei der Umsetzung wird auf diesen Aspekt noch gesondert eingegangen. Abschließend sei erwähnt, dass die beiden beschriebenen Modelle zur Problemlösung etwas vereinfacht zu einem eigenen Modell zusammengefügt und auch den Schülern bei der Ausarbeitung als eine Art *Rezept* zur Lösung der Aufgaben bzw. Probleme mitgegeben werden. Wie genau dieser Schritt realisiert wird, wird im späteren Verlauf der Arbeit noch gezeigt.

4 Zielgruppe

Die später folgenden Beispielaufgaben sollen Programmieranfängern dabei helfen erste Konzepte der Programmierung kennen zu lernen. Hauptaugenmerk wird hierbei auf die Ausbildung in höheren Schulen gelegt, was, in Übereinstimmung mit dem Curriculum (Lehrplan Informatik, S. 2; Lehrplan Wahlfach Informatik, S. 1), Schüler ab der 5. Klasse AHS als konkrete Zielgruppe ergibt. Dazu sei gesagt, dass speziell BYOB für deutlich jüngeres Publikum geeignet ist und daher, mit entsprechender Adaption der Beispielaufgaben, sicherlich schon Acht- bis Zehnjährige von der Ausarbeitung profitieren könnten.

5 Grundkonzepte der Programmierung

Da die Programmiersprache Java als Grundlage für den Java-Hamster-Simulator dient, wird der Theorieteil ausschließlich Definitionen aus dementsprechender Literatur enthalten. Jedoch wird nur auf die Konzepte eingegangen, die bei der Erarbeitung der Beispiele mit den

Mikrowelten relevant sind. Es handelt sich daher nicht um eine vollständige Einführung in die Programmiersprache Java. Aus Gründen der Simplizität, werden hier nämlich einige Konzepte, wie die der statischen Methoden oder der exakten Funktionsweise der Bildschirmausgabe in Java, weggelassen.

Die folgenden theoretischen Grundlagen sind in zwei große Unterkapitel gegliedert und zwar in die *imperative* und die *objektorientierte* Programmierung. Die Grundlagen sollen dazu dienen den Lehrer bzw. den Leser auf die Materie einzustimmen. Die Schüler werden in dieser technischen Form nie mit Programmierkonzepten konfrontiert, sondern erhalten die Grundlagen in weiterer Folge mithilfe eines alternativen Ansatzes. In Abschnitt D unter dem Punkt Exemplarische Ausarbeitung der Beispiele folgt die praktische Umsetzung der sogleich vorgestellten Inhalte. Es sei jedoch jetzt schon erwähnt, dass nicht jedes dieser Konzepte schlussendlich in der Ausarbeitung zur Anwendung kommt. Viel mehr soll der theoretische Teil dazu dienen, Konzepte nahezulegen mit denen die Lösung der Beispiele aus dem praktischen Teil *möglich* wäre.

5.1 Imperative Programmierung

Der imperative Programmierstil ist der älteste und damit sozusagen der „klassische“ Programmierstil. Hierbei wird Code fabriziert, indem man Anweisung an Anweisung reiht, bis schlussendlich das gewünschte Ergebnis vorliegt. Es können auch Methoden oder Prozeduren deklariert werden. Das Programm befindet sich jedoch in nur einer einzigen Klasse, besteht also aus nur einem File, was gerade bei hohem Programmieraufwand sehr schnell unübersichtlich wird (vgl. Müller et al. 2005, S. 208).

5.1.1 Klassen und Methoden

Da diese beiden Begriffe in den folgenden Abschnitten oft genannt werden, ihre genaue Definition aber erst in den Abschnitten 5.1.10 und 5.2.1 folgt, soll hier eine kurze Einführung in die Begrifflichkeiten erfolgen. So gibt eine Klasse einen bestimmten Typ vor und beschreibt wie einzelne Objekte dieses Typs auszusehen haben. Beispielweise könnte die Klasse *Hamster* Objekte namens *Tapsi*, *Hanni*, *Gusti*, etc. liefern. Das was die Objekte der Klasse *können* sollen, beschreiben die Methoden. Im Falle der Hamster-Klasse also *laufen*, *Körner-hamstern*, *Fell-pflegen*, etc. (vgl. Ullenboom 2011, Abschnitt 3.2).

5.1.2 Variablen

Grundsätzlich können Variablen in der Informatik, ähnlich wie wir es aus der Mathematik kennen, verschiedene Werte annehmen. Da Variablen in Java Felder sind, können sie zudem diese Werte speichern, um sie im späteren Verlauf wieder zu verwenden (vgl. Zakhour et al. 2007, S. 68). Bevor man auf eine Variable zugreifen kann, muss die Variable deklariert werden. Eine Deklaration besteht immer aus dem Typ, den die Variable annehmen soll und dem Variablennamen.

Beispiel:

```
int anzahlKoerner;
```

Die Bezeichnung int (=Integer) steht hier für die Werte der ganzen Zahlen (was beim Zählen von Körnern ja Sinn machen würde). Variablen beginnen prinzipiell mit einem Buchstaben, dürfen aber nicht den selben Namen wie Schlüsselwörter, beispielsweise `class` oder `if`, haben.

Um Variablen einen Wert zuzuweisen, wird der Zuweisungsoperator „=“ verwendet (Details zum Operator folgen):

```
anzahlKoerner = 0;
```

Die Deklaration und die Wertzuweisung kann auch in einem Schritt erfolgen:

```
int anzahlKoerner = 0;
```

(vgl. Müller et al. 2005, S. 47ff.).

Grob unterscheidet man zwischen Instanzvariablen und lokalen Variablen. Lokale Variablen dienen häufig zur Speicherung von Werten innerhalb einer Methode. Wie der Name schon sagt, kann man nur lokal in der Methode selbst darauf zugreifen und beispielsweise ihren Wert verändern (vgl. Zakhour et al. 2007, S. 68). Instanzvariablen gehören immer zu einer Klasse und sind, im Gegensatz zu den lokalen Variablen, global in der ganzen Klasse verfügbar. Auf Instanzvariablen bzw. Attribute wird später im Zuge der objektorientierten Programmierung genauer eingegangen.

Eine weitere Form der Variablen sind Parameter, die unter dem Punkt 5.1.10 *Methoden und Prozeduren* genauer behandelt werden.

5.1.3 Operatoren und Ausdrücke

Operatoren verknüpfen zwei oder mehrere Operanden zu einem Ausdruck, wobei ein Ausdruck „nichts anderes als eine Verarbeitungsvorschrift zum Ermitteln eines Wertes“ ist (Balzert 1999, S. 208) und ein Operand gleichzusetzen ist mit den zu verarbeitenden Daten. Diese können Variablen oder Literale sein, welche in dieser Arbeit auf Ganz- und Gleitkommazahlen beschränkt sein sollen (vgl. Ullenboom 2011, Abschnitt 2.1.4).

5.1.4 Der Zuweisungsoperator

Wie bereits erwähnt dient in Java das Gleichheitszeichen „=“ als Zuweisungsoperator. Eine Zuweisung besteht aus einer Variable und dem Wert, den diese Variable erhalten soll:

```
anzahlKoerner = 0;
```

Zuweisungen sehen dadurch zwar wie mathematische Gleichungen aus, gehorchen jedoch eigenen Gesetzen. So wäre mathematisch die Gleichung a = a +1 eine falsche Aussage, da kein a die Gleichung erfüllen kann. Programmiertechnisch geht das aber durchaus in Ordnung, weil zuerst der rechte Ausdruck ausgewertet wird und dann in die Variable kopiert wird.

Die Zuweisung

```
a = a + 1;
```

würde den Wert der Variable `a` also um 1 erhöhen (vgl. Ullenboom, Abschnitt 2.4.1 2011).

5.1.5 Die Rechenoperatoren

Hierzu zählen, die aus der Mathematik gewohnten, Operatoren der Addition `+`, Subtraktion `-`, Multiplikation `*` und Division `/`. Außerdem gibt es in Java auch noch die Modulo-Division `%`, die nur den Rest einer Division ermittelt.

Die Ausdrücke sind hierbei nichts anderes als Formeln zur Berechnung von Werten:

```
i+1, 3*(2+number).
```

Die Berechnung erfolgt nach der mathematischen Regel: „Punkt- vor Strichrechnung“. Die Ergebnisse können dann auch in Variablen gespeichert werden:

```
anzahlKoerner = (3 + 9) + 4 * (5 + 3);
```

(vgl. Müller et al. 2005, S. 57f.).

5.1.6 Die Vergleichsoperatoren

Vergleicht man zwei Werte miteinander so bezeichnet man diese Form von Vergleichen als „Boolesche Ausdrücke“ (Müller et al. 2005, S. 56). Boolesche Ausdrücke ergeben immer entweder den Wert `true` oder `false`. Folgende Operatoren zählen zu den Vergleichsoperatoren: größer `>`, größer oder gleich `>=`, kleiner `<`, kleiner oder gleich `<=`, gleich `==` und ungleich `!=`.

Beispiele:

`3 < 7`, liefert `true`

`3 >= 7`, liefert `false`

`3 == 7`, liefert `false`

`3 != 7`, liefert `true`

Auch Variablen können mit Operatoren verglichen werden. Erstellt man beispielsweise eine Variable `anzahlKoerner` und weist ihr den Wert 5 zu, so würde die Abfrage ob die Variable größer als 3 ist das Ergebnis `true` liefern:

```
int anzahlKoerner = 5;
```

`anzahlKoerner > 3`, liefert `true`

5.1.7 Die logischen Operatoren

Will man in Java beispielsweise die Abarbeitung eines Programmteils an mehrere Bedingungen knüpfen, so kann man diese mithilfe der logischen Operatoren zusammensetzen. Die wichtigsten Operatoren sind hierbei das *Und* (Konjunktion), das *Oder* (Disjunktion) und das *Nicht* (Negation).

- Und: `a && b`
- Oder: `a || b`
- Nicht: `!a`

Wobei `a` und `b` ausschließlich boolesche Ausdrücke sein dürfen. Andernfalls würde der Compiler einen Fehler melden. Folgende Tabelle gibt genauen Aufschluss über die möglichen Werte verknüpfter, boolescher Ausdrücke (vgl. Ullenboom, Abschnitt 2.4.7 2011):

a	**b**	**a && b**	**a || b**	**!a**
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Beispiele:

```
3 > 7 && 5 <= 8
```

→ `3 > 7` liefert `false`.

→ `5 <= 8` liefert `true`.

→ Daher liefert der gesamte Ausdruck `false`.

```
2 != 4 || 5 < 5
```

→ `2 != 4` liefert `true`.

→ `5 < 5` liefert `false`.

→ Daher liefert der gesamte Ausdruck `true`.

```
!(anzahlKoerner == 4)
```

→ `anzahlKoerner == 4` liefert `true`, wenn `anzahlKoerner` den Wert 4 hat, ansonsten `false`.

→ Daher liefert der gesamte Ausdruck `false`, wenn `anzahlKoerner` den Wert 4 hat, ansonsten `true`.

5.1.8 Anweisungen und Blöcke

Wie bereits erwähnt, kann man mithilfe von Operatoren Ausdrücke kreieren. Nun bilden in weiterer Folge Ausdrücke den Kern von Anweisungen und diese wiederum lassen sich zu (Anweisungs-) Blöcken zusammenfassen (vgl. Zakhour et al. 2007, S. 91).

Eine Anweisung wäre zum Beispiel das vorhin erwähnte:

```
anzahlKoerner = (3 + 9) + 4 * (5 + 3);
```

Anweisungen können vom Programm vollständig ausgeführt werden und enden immer mit einem Semikolon (`;`). Vergleichsweise werden Anweisungen als „Gegenstück zu den Sätzen in den natürlichen Sprachen“ (Zakhour et al. 2007, S. 93) angesehen.

Blöcke sind Anweisungen, die durch geschweifte Klammern zu Gruppen zusammengefasst werden.

```
{
      if (anzahlKoerner < i)
            anzahlKoerner = anzahlKoerner + 1;
      i = i – 1;
}
```

Hier wurden die zwei Anweisungen „`if`...“ und „`i =` ...“ zu einem Block zusammengefasst. Dazu sei gesagt, dass auch eine einzelne Anweisung einen Block ausmachen kann. In diesem Beispiel ist `anzahlKoerner = anzahlKoerner + 1;` eigentlich auch ein Block der `if`-Anweisung und man könnte die Anweisung daher auch in geschweifte Klammern setzen.

5.1.9 Bedingte Anweisung und Schleifen

Die bedingte Anweisung, auch `if`-Anweisung oder Verzweigung genannt, wird verwendet um einen Anweisungs-Block nur unter einer bestimmten Bedingung auszuführen. Die einfachste

bedingte Anweisung hätte die Form:

```
if(Bedingung)
      Anweisung(en);
```

wie bereits weiter oben angedeutet ist dies dasselbe wie:

```
if(Bedingung) {
      Anweisung(en);
}
```

Wobei die zweite Schreibweise vor allem bei Anweisungsblöcken gebräuchlich ist. Prinzipiell erfolgt zuerst die Überprüfung der Bedingung. Ergibt diese den Wert `true`, dann wird die darunter stehende Anweisung ausgeführt. Entspricht der Wert der Bedingung `false`, dann wird das Programm bei der ersten Anweisung nach dem Block der `if`-Anweisung fortgesetzt.

Die bedingte Anweisung ist zudem erweiterbar. Man kann der `if`-Anweisung nämlich auch eine Handlungsalternative vorgeben. Dazu verwendet man in Java die `if-else`-Anweisung:

```
if(Bedingung) {
      Anweisung 1;
} else {
      Anweisung 2;
}
```

Ergibt nun die Überprüfung der Bedingung den Wert `false`, so wird die `if`-Anweisung nicht verlassen, sondern die Anweisung(en) im `else`-Zweig abgearbeitet (vgl. Müller et al. 2005, S. 65f.).

Um Anweisungen mehrfach auszuführen, werden in Java Schleifen zur Verfügung gestellt. Es gibt mehrere Arten von Schleifen, ich möchte mich aber auf die `while-` und auf die `for`-Schleife im Einsatz als Zählschleife beschränken.

Die `while`-Schleife ist vom Grundprinzip der einfachen `if`-Anweisung sehr ähnlich.

Nur, dass der Anweisungsblock *solange* wiederholt wird, *bis* die Bedingung nicht mehr erfüllt (`false`) ist.

```
while(Bedingung) {
      Anweisung(en);
}
```

Auch die `while`-Schleife beginnt mit der Überprüfung der Bedingung. Ist diese `true`, dann wird die Anweisung bzw. der Anweisungsblock ausgeführt und an den Beginn der `while`-Schleife zurückgekehrt. Es folgt eine weitere Überprüfung der Bedingung. Erst wenn das Ergebnis der Überprüfung `false` ist, wird die Schleife verlassen.

Dazu ein Beispiel aus der Hamster-Welt:

```
while(vornFrei() == true) {
            vor();
}
```

Solange die Methode `vornFrei()` also `true` zurückliefert (näheres zu den Methoden folgt im Abschnitt 5.1.10), bewegt sich der Hamster ein Feld nach vorne. Wobei man die Methode `vornFrei()` hier so auffassen muss, dass sie in Blickrichtung des Hamsters, von seiner aktuellen Position, quasi ein Feld vorausschaut und überprüft, ob das Feld begehbar ist oder nicht. Erst wenn ein Weiterkommen nicht mehr möglich ist, liefert `vornFrei() false` zurück und die Schleife wird folglich verlassen.

Es sei erwähnt, dass der Code hier noch etwas optimiert werden kann. Da `vornFrei()` eine boolesche Methode ist, also nur `false` oder `true` zurückliefert, kann man auch

```
while(vornFrei()) {
            vor();
}
```

schreiben, da es in Java reicht, wenn in der Bedingung `true` oder `false` steht.

Eine weitere Variante, um Anweisungen wiederholt durchzulaufen ist die `for`-Schleife, die auch als Zählschleife verwendet werden kann.

Die Standard-Form der `for`-Schleife ist die folgende:

```
for(Initialisierung; Bedingung; Iteration [Inkrement]) {
      Anweisung(en);
}
```

Die Initialisierung wird zu Beginn der Schleife einmal ausgeführt und dient hier der Festlegung einer Zählvariable. Eine Zählvariable ist so etwas wie eine Hilfsvariable und speichert einen veränderbaren Wert, der für die Überprüfung der Bedingung herangezogen wird. Wie auch bei der `while`-Schleife, so endet die `for`-Schleife, wenn der Wert der Bedingung `false` ist. Am Ende jedes Schleifendurchlaufs wird der Ausdruck Iteration (Inkrement) aufgerufen.

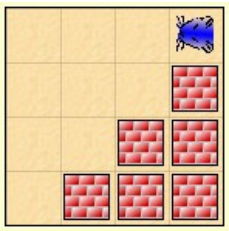
Das folgende Beispiel zeigt die Verwendung der `for`-Schleife als Zählschleife. Für die kommenden Beispielprogramme ist diese Verwendung ausreichend.

```
for(int i = 0; i < 10; i++) {
      System.out.println(„Der Zähler ist: “ + i);
}
```

Hier wird an erster Stelle die Zählvariable `i` initialisiert. Es folgt die Überprüfung der Bedingung, also ob `i` kleiner als zehn ist. Anschließend wird der Ausdruck `System.out.println(„Der Zähler ist: “ + i)` ausgeführt, welcher „Der Zähler ist: “ und den Wert von `i` am Bildschirm ausgibt. Zum Schluss wird die Zählvariable um 1 erhöht und die Schleife startet den nächsten Durchlauf. Am Ende sollte die Schleife zehn mal „Der Zähler ist: “ und daneben jeweils den aktuellen Wert der Zählvariable ausgegeben haben.

Ein weiteres Beispiel dazu soll den möglichen Einsatz der `for`-Schleife im Java-Hamster-Simulator veranschaulichen:

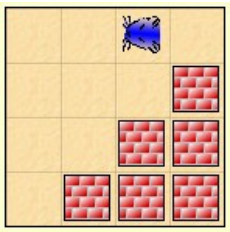
```
for(int stufe = 0; stufe < 3; stufe++) {
      vor();
      linksUm();
      vor();
      rechtsUm();
}
```

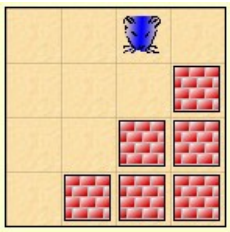


Die vorgestellte Schleife könnte zum Beispiel für das Herabsteigen von drei Stufen, wobei der Hamster nach links blicken müsste, verwendet werden. Eine mögliche Variante wie diese Ausgangssituation aussehen könnte, zeigt die Abbildung auf der linken Seite.

Der Ablauf der Schleife

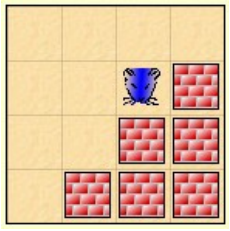
Dabei wird zuerst die Variable `stufe` initialisiert und auf den Wert 0 gesetzt. Da die darauffolgende Überprüfung der Bedingung `false` ergibt – 0 ist kleiner als 3 – wird im Anweisungsblock fortgefahren. Der Hamster wird nun mit der Methode `vor()` ein Feld nach vorne bewegt (siehe Abbildung rechts).

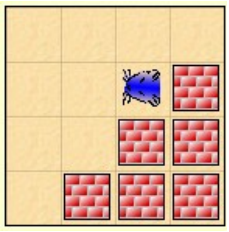




Der Block wird nacheinander abgearbeitet und daher folgt die Ausführung der Methode `linksUm()`. Der Hamster wird also dadurch um 90 Grad nach links gedreht (siehe Abbildung links).

Anschließend folgt ein weiterer Schritt nach vorne, sodass der Hamster auf der zweiten Stufe ankommt.





Schließlich wird der Hamster mithilfe der Methode `rechtsUm()` um 90 Grad nach rechts gedreht. Es sei dazu erwähnt, dass im Java-Hamster-Simulator eigentlich keine Drehungen nach rechts möglich sind und die Methode daher erst selbst implementiert werden muss. Mit drei Drehungen nach links ist dies aber leicht erreicht.

Die Schleife ist damit natürlich noch nicht beendet. Es folgt die Erhöhung der Variable `stufe` auf den Wert 1 und der zweite Durchlauf, mit der Überprüfung ob `stufe` kleiner als 3 ist, startet. Da die Bedingung erst nicht mehr erfüllt ist, wenn die Variable `stufe` den Wert 3 hat, wird der Anweisungsblock der Schleife für die Werte 0, 1 und 2, also drei mal, ausgeführt. Das heißt, dass der Hamster am Ende alle drei Stufen überwunden hat und am Fußpunkt der Stiege landet.

Ist die Schleifen-Bedingung noch erfüllt, will man die Ausführung der Schleife aber dennoch vorzeitig beenden, so kann man auf die `break`-Anweisung zurückgreifen.

```
while(vornFrei()) {
      vor();
      linksUm();
      if(vornFrei()) {
            vor();
      } else {
            break;
      }
}
```

5.1.10 Methoden und Prozeduren

In der Programmiersprache Java stellen Methoden und Prozeduren „Dienstleistungen zur Verfügung, die durch Botschaften in Anspruch genommen werden können“ (Balzert 1999, S. 222). Anders ausgedrückt, könnte man sie als Operationen bezeichnen, die zur Manipulation von Daten dienen.

Um Botschaften zu übermitteln, bedarf es eines Informationsaustauschs. Deswegen können Methoden sogenannte Parameter übergeben werden, die während des Methodenablaufs verwendet werden. Methoden sind Anweisungsfolgen, die in Java immer zu einer Klasse

gehören (mehr dazu in Abschnitt 5.2.1). Eine spezielle Form der Methode sind die sogenannten *Prozeduren*. Prozeduren können zwar genauso wie Methoden den Zustand eines Objekts bzw. von Objekt-Daten ändern, geben dabei aber nichts zurück. Die übrigen Methoden hingegen liefern immer genau einen Rückgabewert (vgl. Balzert 1999, S. 222ff.).

Methoden bestehen aus einem Rückgabetyp, dem Methodennamen, gegebenenfalls aus einem oder mehreren Parametern und dem Methodenrumpf, der einen Anweisungsblock beinhaltet.

```
Rückgabetyp Methodenname(Parameter) {
      Anweisung(en);
      Rückgabewert;
}
```

Methoden ohne Rückgabewert wird das Schlüsselwort `void` vorangestellt. Hat die Methode einen Rückgabewert, so muss dieser im Methodenrumpf mit dem `return`-Befehl angezeigt werden (vgl. Müller et al. 2005, S. 117f.).

Ein simples Beispiel einer eigenen Methode aus dem Java Hamster-Simulator, wäre die Prozedur `rechtsUm()`:

```
void rechtsUm() {
      linksUm();
      linksUm();
      linksUm();
}
```

Dabei zeigt das Schlüsselwort `void` an, dass es sich um eine Prozedur, also eine Methode ohne Rückgabewert, handelt. Der Grund dafür, dass die Methode überhaupt implementiert werden muss, liegt in der Eigenart des Hamsters sich ausschließlich um 90 Grad nach links drehen zu können. Anstatt jedes mal, wenn eine Drehung um 90 Grad nach rechts erwünscht ist, drei mal die Prozedur `linksUm()` in den Code zu schreiben, genügt es nun die Prozedur `rechtsUm()` aufzurufen.

Eine Methode mit Rückgabewert wäre beispielsweise `koernerZaehlen()`, welche den Hamster, wie der Name schon sagt, die Körner auf dem aktuellen Feld zählen lässt:

```
int koernerZaehlen() {
      int anzahlKoerner = 0;
      while (kornDa()) {
            nimm();
            anzahlKoerner++;
      }
      return anzahlKoerner;
}
```

Der Rückgabetyp wird durch das `int` signalisiert, was einem ganzzahligen Wert entspricht. Innerhalb der Methode kommt es nun zur Deklaration einer lokalen, ganzzahligen Variable `anzahlKoerner`, deren Wert anfangs auf 0 gesetzt wird. Es folgt eine `while`-Schleife, deren Bedingung die Methode `kornDa()` ist. Diese Methode liefert `true`, wenn sich ein Korn im aktuellen Feld befindet und `false` wenn dies nicht der Fall ist. Es folgt der Anweisungsblock der `while`-Schleife, welcher den Hamster mit `nimm()` ein Korn aufnehmen lässt und die Variable `anzahlKoerner` um den Wert 1 erhöht (mithilfe von „`++`“). Wird die `while`-Schleife verlassen, so gibt die Methode den ganzzahligen Wert von `anzahlKoerner` zurück.

Wichtig: *Rückgabe* ist nicht gleichbedeutend mit *Ausgabe*! Das soll heißen, dass, wenn eine Methode fertig ausgeführt ist, das Ergebnis für den Benutzer eigentlich noch nicht ersichtlich ist. Was mit dem Ergebnis passiert, bleibt schlussendlich dem Benutzer überlassen. Es kann beispielsweise auf dem Bildschirm ausgegeben oder zur Überprüfung einer Bedingung verwendet werden (siehe `while(kornDa())`).

Parameter wurden schon bei den Variablen kurz erwähnt. Sie werden Methoden oder Prozeduren gleich bei der Deklaration mitgegeben (im Beispiel unten: `int anzahl`). Während die Methode abgearbeitet wird, können die Parameter dann wie Variablen verwendet werden. Parameter dienen in erster Linie als Schnittstelle mit der Außenwelt der Methode.

```
void nimm(int anzahl) {
      for(int i = 0; i < anzahl; i++) {
            nimm();
      }
}
```

Diese Prozedur lässt den Hamster eine vorgegebene Anzahl an Körnern aufnehmen. Der Parameter `anzahl` kommt hierbei in der Bedingung der `for`-Schleife zum Einsatz. Hat die Zählvariable `i` also den selben Wert wie `anzahl` erreicht, so wird die Schleife und damit auch die ganze Prozedur beendet.
Zu beachten gilt es hier die Tatsache, dass der Compiler bei der Übersetzung des Programms die Prozeduren `nimm(int anzahl)` und `nimm()` sehr wohl unterscheiden kann. In der Fachsprache, spricht man davon, dass die beiden nämlich eine unterschiedliche *Signatur* besitzen (vgl Ullenboom 2011, Abschnitt 2.7.1). Einfach formuliert bedeutet dies, dass, wenn sich nach dem Rückgabewert die ersten Zeilen von Methoden oder Prozeduren unterscheiden (egal ob durch Parameter oder Methodennamen), dann können sie vom Computer als individuelle Methoden bzw. Prozeduren erkannt werden.

Eingesetzt werden Methoden speziell deswegen weil sie es einerseits verhindern, dass wiederkehrende Programmteile im Programmcode mehrfach vorkommen (siehe `rechtsUm()`). Daraus ergibt sich ein weiterer Vorteil, nämlich, dass Änderungen im Code nur an einer Stelle durchgeführt werden müssen um die Funktionalität des gesamten Programms zu beeinflussen. Andererseits lassen sich so komplexe Programme in kleinere Teilprogramme zerlegen, was die Übersichtlichkeit des Programms erhöht (vgl Ullenboom 2011, Abschnitt 2.7).

Anmerkung: In Java wird grundlegend nicht zwischen Methoden und Funktionen unterschieden. Der Begriff Methode steht in dieser Arbeit daher immer als synonym für Funktion.

5.1.11 Einschub: Rekursion

Der Begriff Rekursion stammt vom lateinischen recurrere (=„zurücklaufen“) und bedeutet so viel wie „Selbstbezug“ (Hattenhauer 2010, S. 217). Die Rekursion ist ein Hilfsmittel der Informatik, das auf den Mathematiker John McCarthy zurückgeht. Sie beschreibt allgemein die

Abhängigkeit eines zu beschreibenden Objekts von einem Parameter derselben Art (vgl. Schöning 2006, S. 6). Auf das Programmieren angewendet, bedeutet das nichts anderes als, dass eine Methode sich in ihrem Rumpf bzw. im Zuge ihrer Definition noch einmal selbst aufruft.

Rekursive Beispiele aus dem Alltag wären ein Familienstammbaum (jeder Mensch hat Vorfahren und jeder Vorfahre hat Vorfahren) und die russischen Matroschka-Puppen (in einer Puppe steckt eine weitere Puppe) (vgl. Weigend 2007, S. 99).

Innerhalb dieser Arbeit beschränke ich mich auf die Form der linearen Rekursion. Diese besagt, dass beim Aufruf der rekursiven Methode es zu höchstens einem weiteren Aufruf der Methode kommt. Weitere Formen der Rekursion wären die verzweigte, die verschachtelte oder die offene Rekursion, auf die im Rahmen dieser Arbeit jedoch nicht genauer eingegangen wird (vgl. Logofatu 2008, S. 106).

Hierzu folgendes Beispiel aus dem Java-Hamster-Simulator:

```
void allesnehmen() {
      if (kornDa()) {
            nimm();
            allesnehmen();
      }
}
```

Die Prozedur `allesnehmen()` soll dazu dienen den Hamster alle, auf einem Feld befindlichen, Körner aufnehmen zu lassen. Dazu erfolgt zuerst die Überprüfung der Bedingung, nämlich, ob sich in dem Feld, auf dem sich der Hamster aktuell aufhält, ein Korn befindet. Ist dies der Fall, so nimmt der Hamster mithilfe der Prozedur `nimm()` ein Korn auf und `allesnehmen()` wird ein weiteres mal ausgeführt. Ist dies nicht der Fall, so endet die Prozedur. Die Prozedur wird also so lange ausgeführt bis sich kein Korn mehr auf dem Feld befindet.
Wichtig hierbei ist die Abbruchbedingung, die dafür sorgt, dass die Methode ab einem gewissen Punkt nicht mehr aufgerufen wird. Ist diese Bedingung nicht gegeben, dann kann es sein, dass die Aufrufe endlos stattfinden und die Prozedur nicht mehr verlassen wird.

Das obige Beispiel einer rekursiven Prozedur ist ein weiterer Spezialfall der Rekursion, nämlich die „rekursive Selbstaufforderung“ (Weigend 2007, S. 100). Sie bezieht sich auf die sogenannten „endrekursiven Prozeduren“, welche, wie der Name schon sagt, einerseits nichts zurückgegeben und andererseits am Ende einen rekursiven Aufruf haben. In diesem Sonderfall kann die Rekursion als Schleife interpretiert werden. Es handelt sich hierbei nämlich nur um eine Wiederholung der Ausführung, bei der die Prozedur auch mit anderen Parametern aufgerufen werden kann(vgl. Weigend 2007, S. 100).

5.2 Objektorientierte Programmierung

Die objektorientierte Programmierung ist eine Erweiterung der imperativen Programmierung. Dieser Programmierstil kommt speziell bei größeren Projekten zum Einsatz, da er es erlaubt, das zu lösende Problem in Teilprobleme aufzuteilen. Hierzu dienen miteinander interagierende Objekte, die mithilfe unterschiedlicher Klassen deklariert werden können.

5.2.1 Klassen und Objekte

Zakhour et al. (vgl. 2007, S. 58) erklären in ihrem *Java Tutorial* das Konzept des Softwareobjekts anhand von realen Objekten. So weisen alle realen Objekte einen Zustand und ein Verhalten auf. Beispielsweise kann ein Hund u.A. die Zustände Name, Farbe, Rasse oder hungrig haben und dabei das Verhalten bellend, Stöckchen holend oder Schwanz wedelnd aufweisen.
Softwareobjekte können dies ebenfalls und zwar mithilfe von Feldern (bei Müller et al. „Attribute“ genannt) und Methoden. Wobei Felder den Zustand eines Objekts beinhalten und Methoden das Verhalten dieses Objekts.

Damit die Komplexität für die Schüler nicht zusätzlich erhöht wird, bin ich bisher nicht genauer auf die Datentypen eingegangen. Ich möchte mich in diesem Abschnitt darauf beschränken, dass Datentypen „eine Menge von Daten gleicher Art“ (Müller et al. 2005, S. 109) sind. Klassen sind ebenso Datentypen, die sich jedoch aus Datenkomponenten, auch Attributen genannt, zusammensetzen und Methoden zur Manipulation dieser Komponenten bereitstellen. Innerhalb einer Klasse können zudem Konstruktoren deklariert werden, die zur Erzeugung von Objekten dienen (vgl. Müller et al. 2005, S. 113).

Die allgemeine Form der Klassendeklaration in Java lautet:

```
class MeineKlasse {
      Attributdeklaration(en);
      Konstruktordeklaration(en);
      Methodendeklaration(en);
}
```

Wobei der Klassenrumpf (in den geschwungenen Klammern befindlich) nicht zwingend alle der drei Deklarationen beinhalten muss, er könnte sogar leer bleiben. Grundlegend beginnt die Klassendeklaration immer mit dem Schlüsselwort `class`, worauf der Klassenname folgt (vgl. Müller et al. 2005, S. 116). Die Namen von Klassen beginnen in Java gewöhnlich mit einem Großbuchstaben (vgl. Zakhour et al. 2007, S. 112).

Müller et al. beschreiben die Klassendeklaration als „eine Art Schablone oder Maske für das Aussehen der Objekte der Klasse“ (Müller et al. 2005, S. 121). Zakhour et al. (vgl. 2007, S. 124) wiederum formulieren diesen Zusammenhang so, dass eine Klasse einen Entwurf für Objekte bereitstellt. Egal welche der Definitionen man bevorzugt, der Kern der beiden Aussagen ist doch, dass Klassen vorgeben, welche Attribute die Objekte einer Klasse haben.

5.2.2 Attributdeklaration

Die Attribute einer Klasse werden wie Variablen deklariert.

```
typ name;
```

Wie bereits erwähnt, sind sie die Datenkomponenten aus denen die Objekte bestehen (vgl. Müller et al. 2005, S. 117).

5.2.3 Methodendeklaration

Die Deklaration von Methoden wurde bereits im Abschnitt 5.1.10 vorgestellt.

5.2.4 Konstruktordeklaration

Die Deklaration des Konstruktors ist mit der Deklaration von Methoden zu vergleichen. Am Anfang steht immer der Name des Konstruktors, der immer identisch mit dem Namen der Klasse sein muss. Genauso wie bei den Methoden, können dem Konstruktor Parameter in geschwungenen Klammern übergeben werden. Der Rumpf besteht aus einem Anweisungsblock.

```
Konstruktorname(Parameter) {
      Anweisung(en);
}
```

Wird dem Konstruktor bei der Deklaration kein Parameter übergeben, so handelt es sich um einen „default“-Konstruktor, welcher ausschließlich ein Objekt anlegt (vgl. Müller et al. 2005, S. 119).

Zusammenfassend kann man sagen, dass, wenn man beispielsweise eine Klasse `Schueler` implementieren würde, die Objekte dieser Klasse die einzelnen Schüler wären. Diese könnten dann zum Beispiel die Attribute `name`, `geburtsjahr` und `schuelernummer` haben, wobei sie anhand letzterem eindeutig zu identifizieren wären.

6 Häufige Probleme von Programmieranfängern

Seitdem Programmieren als unterrichtenswert aufgefasst wurde, existieren Probleme, Missverständnisse und Fehlvorstellungen bei den Lernenden, die sich bis zum heutigen Tage wiederholen. In den folgenden beiden Abschnitten möchte ich klären welche Probleme das sind und wie diese Probleme vermieden werden können.

6.1 Studien zu häufigen Problemen

Nachdem nun geklärt ist wie an eine Problemlösung herangegangen werden kann und welche Inhalte bei der Lösung der Aufgaben von Bedeutung sind, gilt es zu untersuchen welche Konzepte Programmieranfängern, für gewöhnlich, die meisten Probleme bereiten. Ginat et al. (2013) und Lobnig (2008) liefern eine gute Übersicht über die Ergebnisse bedeutender Studien:

> „[..] difficulties were observed in programs that were inefficient, or cumbersome, due to insufficient recognition of task characteristics [...]. Sometimes the task characteristics were recognized but novices still erred in the selection and utilization of patterns.“ (Ginat et al., S. 57)

Häufige Probleme ergeben sich demnach aus mangelhaftem Erkennen der Anforderungen zur Lösung der Aufgabenstellung. Die daraus entstehenden Programme sind dann oft ineffizient und umständlich. Darüber hinaus stellen Ginat et al. fest, dass, selbst wenn die Aufgabenstellung richtig erkannt wird, Programmieranfänger falsche Entwurfsmuster einsetzen oder diese fehlerhaft anwenden. Ebensolche Fehler werden in der Literatur oft der Kategorie der *Plankomposition* zugeordnet (vgl. Ebrahimi 1994; Soloway et al. 1989). *Pläne* sind dabei mit den oben-genannten *Patterns* zu vergleichen. Sie sind Muster bzw. Vorlagen typischer Programmkomponenten wie Schleifen oder bedingter Anweisungen. Die angeführten Studien zeigen, dass Anfänger große Schwierigkeiten in der Plankomposition, also der Zusammensetzung und Verschachtelung der Pläne, haben. Dies erweist sich als eine der Kern-Schwierigkeiten des klassischen Programmierens, da das Problemlösen meist Vorwissen bzw. Erfahrungen voraussetzt.

Außerdem streichen Ginat et al. (in Bezug auf Spohrer et al.) heraus, dass sich Schwierigkeiten mit grundlegenden Programmkomponenten, wie `for`-Schleifen und `if`-Anweisungen, zumeist durch fehlende Codestücke, falsche Formulierungen, falschen Einsatz oder einfaches Übernehmen (aus anderen Programmteilen) äußern (vgl. Ginat et al., S. 57f.). Dies lässt sich häufig in Fehlvorstellungen zum Programmablauf – weil oft in Verbindung mit falschen Vorstellungen von bedingten Anweisungen – begründen (vgl. Lobnig 2008, S. 6). Zwar ist vielen Anfängern klar, dass Anweisungen sequentiell, also in der Reihenfolge in der sie im Programm vorkommen, abgearbeitet werden. Jedoch glauben sie dabei gerne, dass das Programm in speziellen Fällen selbstständig entscheidet welche Anweisungen auszuführen sind und welche nicht. Missverständnisse im Programmablauf entstehen in erster Linie durch die mangelnde Kompetenz der Lernenden Code zu lesen und zu verfolgen. Um dies zu erreichen bedarf es nämlich einer gedanklichen Simulation des Programms oder einer schriftlichen Rekonstruktion des Ablaufs, was jedoch die Fähigkeiten der meisten Anfänger übersteigt. Das zeigt sich insbesondere in der mangelnden Fähigkeit den Wert von Variablen über den gesamten Programmverlauf nachzuverfolgen, was häufig dazu führt, dass der Output von Programmen nicht nachvollzogen werden kann (vgl. Lobnig 2008, S. 10).
Zudem wird bei Zählschleifen (laut du Boulay und Putnam et al.) nicht verstanden, dass beispielsweise die Zählvariable bei jedem Durchlauf erhöht bzw. erniedrigt wird und danach einen anderen Wert besitzt. Ebenso Probleme bereitet die `while`-Schleife, bei der oft angenommen wird, dass sie sofort terminiert, da nicht verstanden wird, dass die Überprüfung

der Bedingung am Anfang der Schleife geschieht (vgl. Lobnig 2008, S. 6).

Ein Großteil der Schwierigkeiten im Verständnis objektorientierter Konzepte ergibt sich, laut Lobnig (in Bezug auf Holland et al.), aufgrund des Wechsels vom imperativen zum objektorientierten Programmierstil. Ein Problem, das dabei, im Zusammenhang mit dieser Arbeit, von besonderer Wichtigkeit ist, ist die Verschmelzung von Objekt und Variable. So entwickeln Anfänger häufig die Vorstellung, dass ein Objekt nur eine Art *Hülle* für Variablen darstellt. Daraus ergibt sich wiederum eine weitere Fehlvorstellung, nämlich, dass Objekte einfach Datensätze sind. Hier kann es passieren, dass die Lernenden es nicht verstehen, dass Objektverhalten abhängig von deren Zustand ist (vgl. Lobnig 2008, S. 7f.).

6.2 Methoden zur Verbesserung der Situation

Auf Grundlage der Ergebnisse aller oben genannten Arbeiten und Studien, lassen sich folgende Richtlinien, die dem Lehrer im Unterricht als Handlungsorientierung dienen sollen, ausmachen:

- Ausgewählte Beispielprogramme sollten vor den Augen der Schüler gestaltet und/oder in ihre einzelnen Komponenten *zerlegt* werden. Die Schüler sollen so ein Gefühl für die einzelnen Pläne und die Möglichkeiten ihrer Zusammensetzung erhalten.
- Um Missverständnissen beim Programmablauf vorzubeugen, empfiehlt es sich Programme in Einzelschritten durchlaufen zu lassen. Dafür soll auch eine händische Variante auf der Tafel verwendet werden, da die Lernenden so ein Gefühl für die manuelle Repräsentation eines Programms entwickeln können.
- Um die Fähigkeit zur Verfolgung von Variablenwerten zu verbessern, hilft es den Schülern außerdem oft, wenn man verschiedene *Rollen* von Variablen vorstellt (Zählvariablen, Zwischenspeicher, etc.).
- Den Schülern sollten sowohl bessere als auch schlechtere Varianten bereits bekannter Problemlösungen vorgeführt werden. Einerseits kann so die Gestaltungsfreiheit beim Porgrammieren vor Augen geführt werden, andererseits erhalten die Schüler einen Einblick darin, dass Funktionalität nicht das einzige Programmierungsziel sein sollte.
- Außerdem sollte den Schülern klar sein, was es bedeutet, wenn eine Programmkomponente falsch gestaltet, falsch eingesetzt oder schlicht und einfach fehlend ist. So können die Schüler ein Bewusstsein für mögliche Gestaltungsfehler

entwickeln (vgl. Ginat et al. 2013, S. 66).

Des weiteren geht Preiwisch-Seibold in ihrer Arbeit auf ein Modell von Perkins et al. (1988) ein, das die Umsetzung neuer Programmbefehle für Anfänger erleichtern soll. Perkins et al. stellten nämlich in einer ihrer Studien fest, dass Schüler den Schritt von der Theorie zur Anwendung eines neuen Befehls eher nachvollziehen können, wenn die Anwendung eines Befehls sofort mit einem Beispiel illustriert wird. Sie geben dabei folgendes drei-Punkte-Modell zur Aufbereitung von Befehlen vor (vgl. Preiwisch-Seibold 1999, S. 85f.):

- Sinn und Zweck
- Syntax
- Anwendung

Auf Grundlage dieses Modells werden in weiterer Folge bei den Infotexten für die Schüler (siehe Punkt D5) neue Befehle präsentiert und veranschaulicht.

7 Konstruktivismus

Als Grundlage für die Konzipierung von Mikrowelten dient die konstruktivistische Lerntheorie. Bei der konstruktivistischen Lerntheorie wird davon ausgegangen, dass das Individuum selbst - mit seiner Wahrnehmung und all seinem Vorwissen - dafür verantwortlich ist, was es in welcher Form aufnimmt. So besitzt jeder Mensch eine eigene Vorstellung der Wirklichkeit, die er sich selbst konstruiert und daher kann es auch nicht zur „Übertragung“ dieser Vorstellungen bzw. des Wissens darüber kommen. Lernen geht dadurch weit über die reine Wissens-Reproduktion hinaus.

In der konstruktivistischen Theorie entsteht Wissen in erster Linie aus Problemlösesituationen, was den Lernenden schlussendlich zu einem handlungsfähigen Experten machen soll. Auf der anderen Seite nimmt der Wissensvermittler die Rolle des Gestalters effektiver Lernumgebungen, mithilfe derer die Lernenden zum Expertenstatus herangeführt werden sollen, ein (vgl. Esslinger-Hinz et al. 2011, S. 88; Süßenbacher 1997, S. 53).

7.1 Konstruktionismus

Der Mathematiker Seymour Papert entwickelte seine eigene Variante des Konstruktivismus, nämlich den *Konstruktionismus*. Im Zentrum seiner Theorie stehen von den Lernenden geschaffene, für andere sichtbare Artefakte (*public entities*). Diese spiegeln symbolisch die Lösung einer gegebenen Problemstellung wider, die anschließend mit anderen Personen diskutiert werden kann, was einen besonders intensiven Lernprozess bewirken soll. Papert geht dabei so weit, dass ihm das reine Handeln während des Lernprozesses nicht ausreicht, sondern er viel mehr für ein stetes Erschaffen eintritt. So wird aus Deweys Theorie des *learning by doing* bei Papert schnell ein *learning by making* (vgl. Wolf 2003, S. 52f.).

Früh erkennt Papert, dass diese Form des Lernens im traditionellen Schulsystem kaum möglich ist und fordert daher schon Anfang der 80er-Jahre die Ausstattung eines jeden Schülers mit einem Computer (vgl. Papert 1985, S. 25). Deshalb steht im konstruktionistisch-geprägten Unterricht der Schüler im Mittelpunkt und nicht der Lehrer oder der Lehrplan. Schüler sollen nicht darauf gedrillt werden Wissen anzuhäufen und zu reproduzieren, sondern sollen durch die Erschaffung von persönlich-bedeutsamen Dingen lernen. Daraus resultiert, laut Papert, ebenso eine emotionale Bindung zu den Artefakten. Der Lehrer schlüpft im Zuge dieses Lernprozesses in die Rolle eines *Beraters, Kollegen* bzw. *Mitlernenden*, der Ratschläge sowohl gibt als auch verweigert (vgl. Wolf 2003, S. 54).

8 Mikrowelten

Auch bei der Entwicklung von Mikrowelten spielt der Südafrikaner Seymour Papert eine sehr bedeutende Rolle. Er entwickelte nämlich, zusammen mit zahlreichen Kollegen, im Artificial Intelligence Laboratory am MIT ab dem Jahre 1967 die Programmiersprache LOGO (Papert 1985, S. 217f.). LOGO ist eine konstruktivistisch-geprägte Lernumgebung, die Papert in seinen späteren Werken als *Mikrowelt* bezeichnen sollte.

Papert selbst sieht die Mikrowelt als „Brutkasten“ (ebenda, S. 131). Sei es für theoretische Konzepte, abstrakte Denkmodelle oder Ideen jeglicher Art. Mikrowelten sind, laut Papert, Welten, die sich der Lernende selbst gestalten kann und sich im Zuge dieses Prozesses eigenständig neues Wissen aneignet. Ein Grund dafür warum Papert die Entwicklung von Mikrowelten für notwendig hält, ist, dass es seiner Meinung nach, speziell in mathematisch-geprägten Gebieten, oftmals eines Vorwissens, zur Erlangung neuer Erkenntnisse, bedarf. Dies wiederum steht eng im Zusammenhang mit Piagets Theorie der Assimilation, bei der der

Lernende das Neue in das Alte integriert (vgl. ebenda, S. 126ff.).
Um diesen Lernprozess zu erleichtern sollten Mikrowelten, nach Papert, einfach aufgebaut und anhand von Beispielen anwendbar sein, sowie an den Erfahrungen des Nutzers anknüpfen (vgl. ebenda, S. 132).

Der Sprach- und Literaturwissenschaftler Rolf Schulmeister definiert Mikrowelten als „geschlossene artifizielle Umgebungen mit eigenen Regeln“ (Schulmeister 2007, S. 45). Er beschreibt sie als „Kunstwelten, abstrakte Welten, in denen freie Kombinationen möglicher und unmöglicher Konzepte erlaubt sind“ (s. o.). Man könnte außerdem behaupten Schulmeister vergleicht das Lernen mithilfe von Mikrowelten mit einer Art Schatzsuche. Er spricht nämlich davon, dass das zu lernende Wissen in Mikrowelten „versteckt“ sei und es am Lernenden läge „es wieder auszugraben“ (vgl. s. o.).

9 Lehrplanbezug

Der allgemeine Lehrplan der österreichischen AHS für das Fach Informatik ist in die Bereiche „Bildungs- und Lehraufgabe“ und „Lehrstoff“ unterteilt. Im Abschnitt Bildungs- und Lehraufgabe bezieht sich diese Arbeit auf folgende Planpunkte:

> „Es ist eine wesentliche Aufgabe des Informatikunterrichts, Schülerinnen und Schülern informatische und informationstechnische Grundkenntnisse zu vermitteln, um sie zu befähigen, diese zur Lösung einer Problemstellung sicher und kritisch einzusetzen.“ (Lehrplan Informatik, S. 1)

> „Bei der kritischen Auseinandersetzung mit den dabei ablaufenden Prozessen und deren Ergebnissen sollen die Schülerinnen und Schüler ihr kognitives, emotionales und kreatives Potenzial nützen.“ (Lehrplan Informatik, S. 1)

Ein weiterer wichtiger Punkt, in Bezug auf die Mikrowelten, findet sich unter der Überschrift „Kreativität und Gestaltung“:

> „Der Umgang mit Informationstechnologie gibt den Schülerinnen und Schülern Gelegenheit, selbst Gestaltungserfahrungen zu machen. Sinnliche Wahrnehmungen ermöglichen Zugänge zu kognitiven Erkenntnissen.“ (Lehrplan Informatik, S. 1)

Zusammenfassend lässt sich also feststellen, dass in dieser Arbeit, in Übereinstimmung mit dem Lehrplan, ein Hauptaugenmerk auf Problemlösungen und den damit verbundenen

kreativen Prozessen liegt. Weiters wird hierbei insbesondere hoher Wert auf eine kritische Auseinandersetzung mit der Thematik gelegt.

Im Abschnitt Lehrstoff sieht der Lehrplan der 5. Klassen folgende Inhalte vor, welche in weiterer Folge diese Arbeit rechtfertigen:

> „Einblicke in wesentliche Begriffe und Methoden der Informatik, ihre typischen Denk-und Arbeitsweisen, ihre historische Entwicklung sowie ihre technischen und theoretischen Grundlagen gewinnen und Grundprinzipien von Automaten, Algorithmen und Programmen kennen lernen." (Lehrplan Informatik, S. 2)

Da dieses Unterrichtskonzept, wie bereits angesprochen, auch für höhere Jahrgänge geeignet ist, lassen sich ebenso Übereinstimmungen im Lehrplan des Wahlfachs Informatik feststellen, in welchem für den Stoff der 6. bis 8. Klasse der Punkt „Konzepte von Programmiersprachen" (Lehrplan Wahlfach Informatik, S. 1) angeführt ist.

10 Lehr- und Lernziele

Wenn man bedenkt, dass der Lehrer in einer konstruktivistischen Lernumgebung all seine Macht über den Lernprozess, seine mögliche Einflussnahme und seine Rolle als Wissensvermittler beinahe komplett aufgibt, so ist es müßig darüber zu diskutieren welche Lehrziele er damit transportieren will. Konstruktivisten lehnen konkrete Lehrziele sogar ab, da sie als *einengend* und *manipulativ* empfunden werden (vgl. Tiberius 2011, S. 167). Einzig die sogenannte *Differenzwahrnehmung* (nach Siebert) wird als vertretbares Lehrziel angesehen, da sie sich selbst wieder auf den Konstruktivismus bezieht. Den Schülern soll bewusst sein, dass sich jeder Lernende seine eigene Umwelt, seine eigene Perspektive und sein eigenes Wissen kreiert und daher auch Perspektivenwechsel nötig sind, um damit umgehen zu können (vgl. ebenda).

Oberstes Lernziel bei der Arbeit mit den Mikrowelten ist es erste eigene Programme bzw., um es einfach auszudrücken, Handlungsvorschriften für die Hauptfigur der virtuellen Geschichte zu gestalten. Die Schüler sollen grundlegende Konzepte wie Variablen, Methoden und Objekte unterscheiden und diese eigenständig einsetzen können.
Darüber hinaus sollten die Schüler lernen, dass es oft nicht nur einen richtigen Weg gibt um eine Problemstellung zu lösen, sondern, dass unterschiedliche Herangehensweisen zum selben

Ziel führen können. Gelingt die Problemlösung nicht auf Anhieb, so sollen die Lernenden in der Lage sein, mithilfe von bereitgestellten Unterlagen, selbst an der Behebung eines potentiellen Fehlers zu arbeiten.

11 Storytelling

Wie ich es bereits in der Einleitung vorweggenommen habe, ist es ein Ziel dieser Arbeit ein Konzept zu entwickeln, dass möglichst nah an den Interessen bzw. am Alltag der Schüler anknüpft. Diese Arbeit soll daher nicht dazu dienen klassische Programmieraufgaben wie die Visualisierung von Zahlenreihen oder Sortieralgorithmen zum gefühlten hundertsten mal aufzuwärmen. Ich möchte hier viel lieber einen alternativen Ansatz vorstellen, nämlich das *Storytelling*.

Storytelling, also das Erzählen von Geschichten, ist allgegenwärtig. Sei es im Kindergarten, in der Volksschule oder zu Hause als abendliches Einschlafritual – mit Geschichten ist vermutlich ein jeder schon konfrontiert worden. Liest man sich einige dieser kindgerechten Geschichten durch, dann merkt man schnell, dass sie eines gemeinsam haben, nämlich eine Botschaft. Sei es der Umgang mit seinen Mitmenschen, das Verhalten an der Fußgängerampel oder der Ablauf eines Einkaufs im Supermarkt. Anhand von Geschichten sollen Kinder lernen wie die Dinge in der großen weiten Welt funktionieren.

Jetzt stellt sich natürlich die Frage: Warum sollen wir plötzlich aufhören mithilfe von Geschichten zu lernen wie die Dinge funktionieren? Warum kann man dieses Prinzip nicht auch auf das Erlernen von Programmierkonzepten umlegen?

11.1 Geschichten und Emotionen

Geschichten stehen immer in einem engen Zusammenhang mit Emotionen. Gute Geschichten sollen den Hörer dazu bringen sich in die Figuren der Geschichte hinein zu versetzen und deren Gefühle nach zu empfinden (vgl. Bredella 2005, S. 229).

Neurologische Studien zeigen zudem, dass die Verknüpfung von Lerninhalten mit Emotionen einen positiven Einfluss auf die Lernfähigkeit und die Kreativität des menschlichen Hirns haben (vgl. Giessen 2009, S. 10f.).

Domagk und Niegemann (vgl. 2009, S. 40) beschäftigten sich mit den wichtigsten Studien zu diesem Thema und nennen folgende drei Hauptgründe, warum Emotionen im Kontext des

Lernens berücksichtigt werden sollten: Sie haben erstens einen Einfluss auf die Qualität der Lernprozesse und deren Ergebnisse, zweitens lässt sich ein direkter Zusammenhang zum subjektiven Wohlbefinden feststellen und drittens beeinflussen sie die Qualität der Kommunikation zwischen Lehrer und Schüler.

11.2 Goal-based scenario

Ein Modell, das diese Theorie des emotionalen Lernens in die Tat umsetzt, ist das *goal-based scenario* (GBS) von Roger C. Schank. Das GBS dient als Anleitung für die Gestaltung von virtuellen Lernumgebungen im Bereich der Schul- und Hochschulbildung. Ziel des GBS ist es Fertigkeiten zu verbessern und nicht nur reines Faktenwissen zu kreieren. Das gelernte Wissen sollte im Zusammenhang damit stehen wie es in der Praxis umgesetzt werden kann.

Wobei das Szenario sieben wesentliche Komponenten umfasst (vgl. Schank et al. 1999, S. 163f.; Domagk et al. 2009, S. 56f.):

1. Lernziele: Es bedarf einer klaren Vorstellung davon welche Fertigkeiten und welches Wissen sich der Lernende während der Arbeit mit dem Szenario aneignen soll.
2. Arbeitsauftrag: Der Auftrag sollte motivierend, halbwegs realistisch und immer an ein Ziel gekoppelt sein.
3. Rahmenhandlung: Bettet den Auftrag in eine Handlung ein und bietet dem Lernenden Gelegenheiten Fertigkeiten zu üben und sich Informationen zu beschaffen.
4. Rolle: Die Rolle, die der Lernende in der Rahmenhandlung einnimmt, soll so gewählt werden, dass die erlernten Fertigkeiten und das gesammelte Wissen zur Ausübung der Rolle benötigt werden.
5. Szenario-Handlungen: Die Handlungen sollten in engem Zusammenhang mit dem Arbeitsauftrag stehen und diesen schlussendlich zielgerecht erfüllen. Der Fortschritt des Handlungsverlaufs sollte für den Lernenden erkennbar sein. Genauso sollten dem Lernenden positive wie negative Konsequenzen vor Augen geführt werden.
6. Ressourcen: Sie beinhalten alle Informationen, die nötig sind um den Arbeitsauftrag zu erfüllen. Idealerweise werden diese Informationen anhand von Geschichten bereitgestellt.
7. Feedback: Rückmeldungen sollten *just in time* erfolgen, also im Moment des Fehlers bzw. des Erfolgs.

11.2.1 Lehrerrolle im goal-based scenario

Roger C. Schank schreibt dazu in einem seiner Bücher in dem Kapitel „Learning by Doing“:

> „In other words, the role of the teacher in a goal-based scenario is to open up interesting problems and provide tools for solving them when asked by the student to do so. The accomplishment of the goal should be its own reward. The curriculum must be oriented towards, and satisfied with, the idea that a student will learn what they need to in order to accomplish goals.“ (Schank 1999, S. 189)

Die Aufgabe des Lehrers in einem GBS besteht also darin die Schüler mit Problemstellungen zu konfrontieren und ihnen gegebenenfalls Werkzeuge zur Lösung der Probleme bereitzustellen. Außerdem soll der Lehrplan nach den Szenario-Zielen ausgelegt sein und dem Schüler ermöglichen so zu lernen, dass die Ziele erreicht werden können. Das macht deutlich, wie wichtig es in diesem Szenario ist, adäquate Lernziele zu formulieren. Weiters beinhaltet diese Aussage auch, genau wie in der konstruktionistischen Theorie von Papert, die vermehrt passive Rolle des Lehrers innerhalb des Lernprozesses. Der Lehrer rückt nur dann in eine aktive Rolle wenn dies ausdrücklich von den Lernenden gewünscht wird. Selbst dann sollte aber nur eine Art Anleitung zur Selbsthilfe erfolgen, da die Problemlösung immer in der Hand des Schülers liegen sollte.

11.2.2 Umsetzung des goal-based scenario

Wie bereits erwähnt, dient das GBS als Anleitung zur Gestaltung von interaktiven Lernumgebungen. Ich persönlich werde im Zuge dieser Arbeit zwar keine eigene Lernumgebung kreieren, jedoch anhand der Richtlinien des GBS (im speziellen Punkt 5-7) vorhandene Umgebungen adäquat anwenden.

Des weiteren werde ich das GBS aber vor allem als Leitfaden für die Aufbereitung der Programmierbeispiele in einem erzählerischen Rahmen verwenden. Wie diese Umsetzung konkret aussieht wird später im praktischen Teil der Arbeit erläutert.

11.3 Eigenschaften einer guten Geschichte

Wie bereits oben erwähnt, bedarf es bei Schanks Modell einer Rahmenhandlung und mehrerer, mit den Lernschritten verknüpften, Szenario-Handlungen. Es stellt sich nun die Frage wie dieser Handlungsverlauf aufgebaut werden soll. Hierzu ein Kommentar des Pädagogen Ingo Reinhardt:

> „Der Plot geht [...] über die bloße Chronik der Ereignisse hinaus. Er ist eine Kette von Zusammenhängen aus Ursachen und Wirkungen, die zwischen dem Verhalten der Figuren und der Handlung laufend ein Muster erzeugen.“ (Reinhardt 2003, S. 55)

Reinhardt, der sich hierbei auf Lämmert (1972, S. 25) bezieht, spricht in weiterer Folge davon, dass nicht nur das „Was“, sondern auch das „Warum“ von höher Bedeutung ist, wenn es um das Erwecken der Neugier bei den Lesern geht. So sei es ein großer Unterschied ob man beispielsweise davon spricht, dass ein König und daraufhin auch seine Gattin starb oder ob man zusätzlich erwähnt, dass die Königin aus Kummer wegen des Todes ihres Ehemannes starb (vgl. Reinhardt 2003, S. 54f.).

C Vorgehensweise

Der Praxisteil soll in erster Linie nicht dazu dienen jegliche Potentiale, Stärken oder Schwächen der Mikrowelten auszumachen. Viel mehr soll der konkrete Einsatz beider Lernumgebungen in einer Unterrichtssituation vor- bzw. nebeneinandergestellt werden. Dadurch, dass diese Arbeit, wie bereits angesprochen, als Anleitung oder Leitfaden dienen soll, liegt mehr Augenmerk darauf, den Lehrer selbst entscheiden zu lassen welche Mikrowelt welche Programmierkonzepte besser oder schlechter veranschaulicht.

Daher wird die, im nächsten Kapitel folgende, Ausarbeitung besonders auf die Schritte eingehen, die notwendig sind, um einen möglichst reibungsfreien Gebrauch in der Schule zu gewährleisten. Diesbezüglich werden im speziellen interessante Fakten, Tipps und Tricks zum Java-Hamster-Modell und BYOB herausgearbeitet, welche dem Lehrer einen besonders schnellen Einstieg in die Materie ermöglichen sollen. Im Detail beginnt dies mit einer Darstellung der Homepages und deren Inhalt, führt weiter zur Installation der Software und gelangt über Design, Funktionalität und Bedienbarkeit schlussendlich zu Hilfestellungen und Dokumentationen.

Weiters beinhaltet der praktische Teil die technische Realisierung des Hamsterbaus, die Ausformulierung der einzelnen Arbeitsaufträge zu den Szenario-Handlungen und Infotexte zu beiden Lernumgebungen, die den Schülern als Hilfestellungen dienen sollen.

1 Umsetzung des goal-based scenarios

Im Theorieteil wurde bereits erläutert worum es sich beim goal-based scenario handelt. Es folgt nun die Umsetzung für die Gestaltung eines möglichen Szenarios für den Informatikunterricht.

1.1 Lernziele

Die Lernziele stimmen in diesem Fall mit den allgemeinen Lernzielen dieser Arbeit überein (siehe Punkt B10). Grob zusammengefasst wären das also eigenständiges Arbeiten und der Einsatz von Grundkonzepten der Programmierung. Zusätzlich wird jedoch bei jeder der Szenario-Handlungen ein gesondertes Lernziel ausgegeben.

1.2 Arbeitsauftrag

Der Auftrag besteht darin einen jungen Hamster, der noch nicht einmal einfache Dinge wie

gehen oder fressen beherrscht, auf das Leben in der Wildbahn vorzubereiten. Dies geschieht indem man dem Jungtier Handlungsvorschriften in Form von Programmen erteilt, anhand derer er für seine Zukunft lernen soll.

1.3 Rahmenhandlung

Die Rahmenhandlung erzählt davon, dass der Schüler auf einem Feld einen kleinen, hilflosen Hamster findet, den er anschließend mit zu sich nach Hause nimmt. Um den Hamster ideal auf ein Leben in der Natur einzustellen, hat der Schüler dem Hamster einen eigenen Bau errichtet, in dem der Hamster nun seinen Alltag zu meistern hat.

1.4 Rolle

In dem Szenario nimmt der Schüler die Rolle eines Tierzüchters ein, der sich speziell auf Nagetiere konzentriert. Hauptsächlich gehört es zu seiner Arbeit verletzte oder verstoßene Tiere aufzunehmen und diese auf eine Auswilderung vorzubereiten. Es handelt sich dabei um eine verantwortungsbewusste Person, die darauf bedacht ist aus jedem seiner Tiere ein überlebensfähiges Wildtier zu machen.

1.5 Szenario-Handlungen

1.5.1 Die ersten Gehversuche

Die Premieren-Handlung besteht darin den Hamster seine ersten Schritte vollführen zu lassen. Belohnt wird die Erfüllung der Aufgabe durch ein Korn für das kleine Nagetier.

Im Java-Hamster-Simulator lernen die Schüler nun zum ersten mal die Funktionen `vor()` und `nimm()` kennen. In BYOB wiederum wird der Umgang mit den Bewegungs-Blöcken und die Interaktion von Objekten geübt.

1.5.2 Auf zu höheren Gefilden

Nun soll der Hamster das erste mal die Stufen zur Vorratskammer in seinem Bau erklimmen.

Hier haben die Schüler erstmalig die Gelegenheit größere Anweisungsblöcke zu gestalten. Da sich dabei Ereignisse wiederholen, können die Schüler versuchen diese mithilfe von Schleifen ablaufen zu lassen.

1.5.3 Mahlzeit

Nach den Strapazen des Stufensteigens hat der Hamster großen Hunger und deshalb soll der Schüler ihn die ganze Vorratskammer leer fressen lassen.

Bei dieser Aufgabe sollte im Java-Hamster-Simulator auf jeden Fall eine Schleife zum Einsatz kommen. In BYOB lässt sich das Beispiel auch ohne Schleife lösen.

1.5.4 Hamsterliche Bettruhe

Die nächste Handlung besteht darin, den Hamster mit den bisher erlernten Fähigkeiten wieder sicher die Stufen hinunter zu geleiten und ihn anschließend in seinem Hamsterbett rasten zu lassen.

Hierbei soll besonders darauf geachtet werden, dass Code bzw. Kommandoblöcke nicht einfach dupliziert werden, sondern eine eigene Funktion für das Hinuntersteigen von Stufen kreiert werden soll. Diese sollte im Idealfall in abgewandelter Form dazu verwendet werden, den Hamster die fünf Stufen aus dem Hamsterbau gehen zu lassen.

1.5.5 Ein ganz normaler Hamstertag

Abschließend soll der Hamster einen ganzen Tag in seinem neuen zu Hause verbringen. Dazu soll der Schüler ein Programm schreiben, das den Tagesablauf des Hamsters simuliert.

Dafür sollen die programmtechnischen Fertigkeiten aus den bisherigen Szenario-Handlungen kombiniert werden.

1.5.6 Tapetenwechsel

Nachdem der Hamster sich schon wie zu Hause fühlt, ein guter Tierzüchter seine Schützlinge

jedoch ständig vor Herausforderungen stellt, soll der Schüler nun einen neuen Bau für den Hamster eines Kollegen errichten. Anschließend sollen die Schüler gegenseitig überprüfen, ob sich ihr Hamster-Programm auch auf die Umgebung des Mitschülers anwenden lässt. Es sei dazu erwähnt, dass die Anzahl der Hindernisse bzw. Objekte auf vier beschränkt bleiben soll.

Beim Java-Hamster-Simulator stehen ihm dabei die üblichen Werkzeuge zur Gestaltung des Territoriums zur Verfügung. In BYOB sind zu diesem Zweck in einem Ordner vorgegebene Objekte in beliebiger Weise auf der Bühne zu verteilen. Diese Objekte müssten vom Schüler selbst aber zuerst angelegt und dann wie gewünscht platziert werden.

Die Aufgabe erfordert in zweierlei Hinsicht die Kreativität der Schüler. Einerseits bei der Gestaltung des Hamsterbaus und andererseits beim Durschreiten des Baus mithilfe eines Programms. Hauptziel dabei ist es zu testen, wie *portabel* diese Programme – insbesondere die Methoden und Prozeduren – schlussendlich sind. Damit sei jetzt nicht gemeint, ob sie auf anderen Betriebsystemen läuffähig sind, sondern viel mehr, ob sie bei Veränderung der Ausgangssituation trotzdem noch zufriedenstellende Ergebnisse liefern. Die Schüler sollen so einen Anstoß erhalten Methoden möglichst allgemein zu formulieren und nicht nur auf ein spezifisches Problem zuzuschneiden.

1.5.7 Zeit zu „hamstern“

Aufgrund der guten Lernfortschritte des Hamsters, entschließt sich der Tierzüchter nun dazu den Hamster gelegentlich auf kleinen abgesteckten Feldern für seinen zukünftigen Alltag im Freien üben zu lassen. Einer dieser Testläufe führt den Hamster zu einem kleinen Feld, auf dem es Futter im Überfluss zu geben scheint. Da der Winter vor der Tür steht, ist dies die ideale Gelegenheit für den Hamster sich einen Vorrat zuzulegen. Der Hamster soll nun also das gesamte Feld abgehen und dabei alle Körner aufnehmen, die sich auf dem Feld befinden.

Es sollen hier weitere *Automatismen*, in Form von eigenständig ablaufenden Methoden und Prozeduren, kreiert werden, die die Hauptfigur schließlich an ihr Ziel führen. Das Schreiben von eigenen Methoden und der Umgang mit bedingten Anweisungen soll hier speziell intensiviert werden. Für die Lösung mit dem Java Hamster-Modell kommt hierbei erstmals der klassische Java Hamster-Simulator zum Einsatz.

1.5.8 Der Eindringling

Um den Hamster auf seinen Umgang mit Artgenossen zu testen, beschließt der Tierzüchter ihn mit einem weiteren Hamster zu konfrontieren.

Im Java-Hamster-Simulator ist es damit notwendig erstmalig ein eigenes Hamster-Objekt anzulegen und ihm passende Attribute zuzuweisen. Dieser Schritt ist in BYOB bereits in der vorherigen Szenario-Handlung geschehen. Um die Aufgabe nicht zu simpel zu gestalten, sollte daher der neue Hamster eine andere Fellfarbe verpasst bekommen.

1.5.9 Hamsterliebe

Da der neue Hamster sehr scheu ist, der Auszuwildernde aber überaus neugierig reagiert, entsteht sofort eine wilde Verfolgungsjagd, bei der der Hamster des Tierzüchters vor lauter Aufregung doppelt so schnell wie der neue läuft. Die Verfolgung soll erst beendet werden, wenn die Hauptfigur den Eindringling eingeholt hat. Der Hamster merkt anschließend, dass es sich bei dem Eindringling um eine Hamster-Dame handelt und verliebt sich auf der Stelle in sie.

Zum ersten mal wird im Java-Hamster-Simulator nun gelernt wie man Operationen auf verschiedene Objekte anwenden kann. In BYOB wiederum können die Schüler nun die Möglichkeit erproben Skripte für unterschiedliche Objekte zusammenzustellen und dann parallel auszuführen.

1.6 Ressourcen

Jeder Schüler erhält die Arbeitsaufträge in Form von Szenario-Handlungen in schriftlicher Form. Außerdem werden den Lernenden zu jedem der Beispiele Infotexte zur Verfügung gestellt, die hilfreiche Programmierkonzepte kurz erläutern und im Einsatz zeigen sollen.

1.7 Feedback

Rückmeldung erhalten die Schüler in diesem Szenario in erster Linie von der Software, aber natürlich auch über die Lehrperson. Speziell BYOB erlaubt es hier, etwa mithilfe einer Sprechblase oder dem Verschwinden von Objekten, direkt mit dem Benutzer zu interagieren bzw. Konsequenzen aufzuzeigen. Zusätzlich muss man jedoch auch bedenken vorgegebenes

Feedback nicht im Übermaß einzusetzen, da man nie wissen kann welcher Weg zum angestrebten Ziel geführt hat. Dabei spielt ein eigentlich erfreulicher Aspekt der Programmierung eine wesentliche Rolle, nämlich, dass es immer mehrere Lösungen für ein und dasselbe Problem gibt. Da der Kreativität der Schüler in diesem Szenario bewusst keine Grenzen gesetzt sind, ist es auch schwer auszumachen welche möglichen Lösungen schlussendlich erreicht werden. Im großen Stil vorgegebenes Feedback zu verwenden, ist daher sicherlich nicht der richtige Ansatz. Außerdem ermöglichen es die Visualisierungen in den Lernumgebungen ohnehin, dass man direkt nachvollziehen kann ob die Aufgabe erfüllt ist oder nicht.

D Umsetzung

1 Java Hamster

Das Hamster-Modell wurde in den 80er Jahren von der Gesellschaft für Mathematik und Datenverarbeitung (GMD) entwickelt. Der deutsche Informatiker Dietrich Boles hat dieses Modell übernommen, an die Programmiersprache Java angepasst und für parallele und objektorientierte Programmierung tauglich gemacht. Prinzipiell wurde das Modell entworfen um speziell Programmieranfängern die Flut an technischen und methodischen Neuerungen des klassischen Programmierens zu ersparen. Die Problemlösung sollte im Vordergrund stehen und nicht die Entschlüsselung von Compiler-Botschaften (vgl. Boles 2008, S. V).

Inzwischen ist das Modell zu einem mächtigen Werkzeug herangewachsen mit dem nicht nur einfache Anweisungen veranschaulicht werden können, sondern bereits komplexere Sachverhalte wie Datenstrukturen oder objektorientierte Softwareentwicklung. Neben Java unterstützt der Simulator zusätzlich die Sprachen *Scheme, Prolog, Python, Ruby* und *Scratch*, sowie die Einbindung endlicher Automaten und Struktogrammen (Stand: Mai 2013).

1.1 Überblick über die Mikrowelt

1.1.1 Daten und Fakten

Die Homepage (http://www.java-hamster-modell.de/) bietet auf der Startseite eine kurze Aufklärung darüber was das Java-Hamster-Modell ist und was auf der Website zu erwarten ist. Weiters finden sich Informationen zu den Büchern über das Java-Hamster-Modell, ein Überblick über die Funktionsweisen des Simulators, ein Link zum Forum, weiterführende Materialien, Links über Java und zu alternativen Programmierumgebungen und das Impressum auf der Homepage (Stand: 07.03.2013). Der Simulator ist aktuell in der Version 2.9.1 (Stand: 06.03.2013) verfügbar und als nur 31 MB-großes ZIP-File herunterzuladen. Mit ein paar Mausklicks ist er am Computer entpackt und bereit entdeckt zu werden. Praktisch dabei ist, dass das Handbuch sich ebenfalls im File befindet und man so direkt ins Medium einsteigen kann.

Zusätzlich zum mächtigen Java-Hamster-Simulator kann man auch eine *light-Version* des Simulators auf der Website herunterladen. Diese ist speziell an Programmieranfänger gerichtet und bietet daher eine ideale Basis für die ersten Aufgaben der später folgenden Ausarbeitung.

1.1.2 Voraussetzungen

Um den Hamster-Simulator zu installieren bedarf es, zumindest unter Windows, Linux und Solaris, keiner außergewöhnlichen Extras. Es reicht wenn auf dem verwendeten Rechner entweder das „Java SE Development Kit“ (JDK) oder eine „Java SE Runtime Environment“ (JRE) installiert ist (vgl. Hamster-Handbuch, S. 17). Bei der light-Version reicht jedoch eine JRE nicht aus, hier muss ein JDK der Version 6 oder höher auf der Plattform installiert sein (vgl. Hamster-Handbuch light).

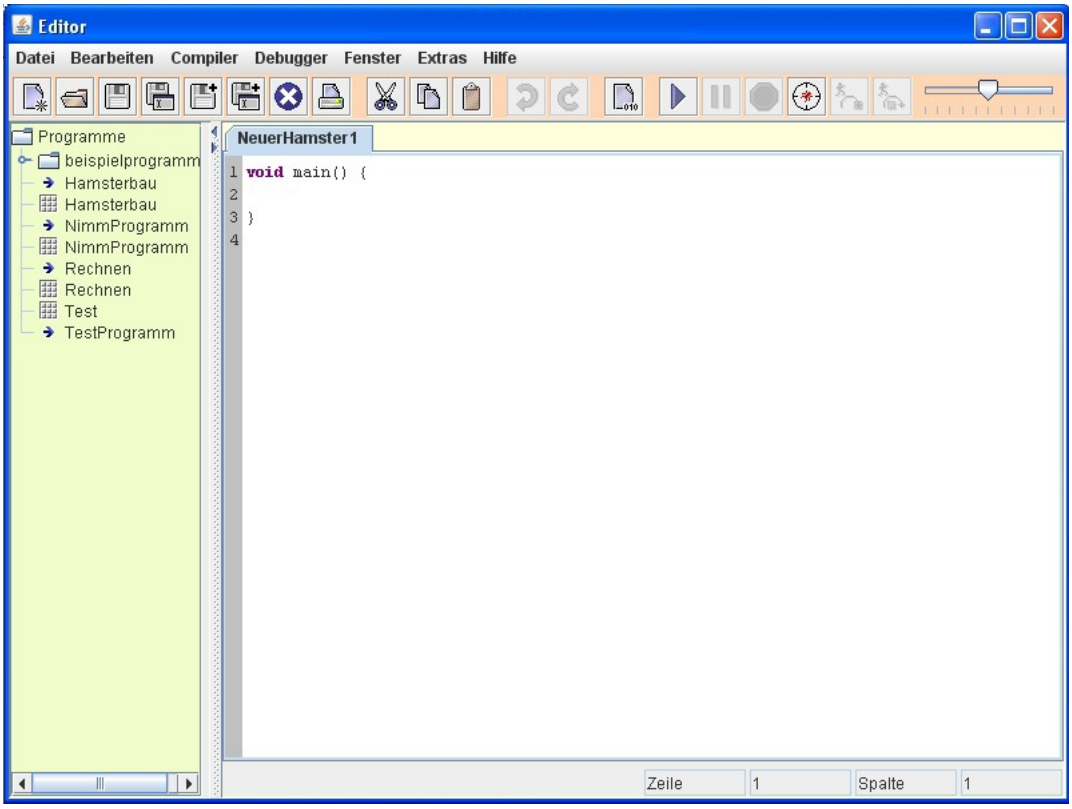
[Der Einfachheit halber und um Missverständnissen vorzubeugen, werde ich in Zukunft nur noch auf die Funktionsweise und Befehlsformen in Windows eingehen.]

1.1.3 Installation

Unter Windows vollzieht sich die Installation des Simulators sehr schnell und einfach. Sind alle, oben genannten, Voraussetzungen erfüllt, dann reicht das entpacken der Daten aus dem ZIP-File und nach einem Klick auf die Datei „hamstersimulator.jar“ bzw. „simulator.jar“ bei der light-Version kann das Hamstern auch schon beginnen. Es kann jedoch sein, dass der light-Simulator nach dem Starten nach der Angabe des Ordners verlangt in dem sich das JDK befindet.

1.1.4 Design und Funktionalität

Der klassische Hamster-Simulator ist in ein Editor-Fenster und in ein Simulations-Fenster aufgeteilt.



Im Editor-Fenster werden die Hamster-Programme geschrieben und können dann anschließend kompiliert und ausgeführt werden. Die Buttons unterhalb der Menüzeile bieten eine große Auswahl aus den darüberliegenden Punkten. Wobei sich die ersten acht Buttons auf den Menüpunkt „Datei“ beziehen, die darauffolgenden fünf auf die Kategorie „Bearbeiten“, Button Nummer 14 kompiliert das File und die übrigen Buttons dienen zum Starten und Anhalten der Ausführung und der Kontrolle des Debug-Modus. Der Schieberegler am rechten Ende dient zur Variation der Geschwindigkeit der Ausführung.

Auf der linken Seite unterhalb der Menüleiste befindet sich der Datei-Baum des Ordners in den der Java-Hamster-Simulator entpackt wurde. Hier können Programme (durch Pfeil angezeigt) und Territorien (durch Feld angezeigt) mit einem Klick direkt in den Simulator geladen werden. Der Editor unterstützt Syntaxhervorhebung, wobei ausschließlich Schlüsselwörter und Kommentare farblich unterschiedlich dargestellt werden. Die Zeilennummer wird am linken, grauen Rand des Code-Eingabe-Fensters mitgeführt.

Über den Menüpunkt „Hilfe“ sind zahlreiche Quellen, wie eine eigene Hamster-API, das Handbuch und Links zu den von Dietrich Boles verfassten Büchern zu finden. Wobei dazu gesagt sein muss, dass nur ein Link zu einer eingeschränkten Leseprobe führt und die anderen beiden lediglich eine Inhaltsangabe des jeweiligen Buches liefern.

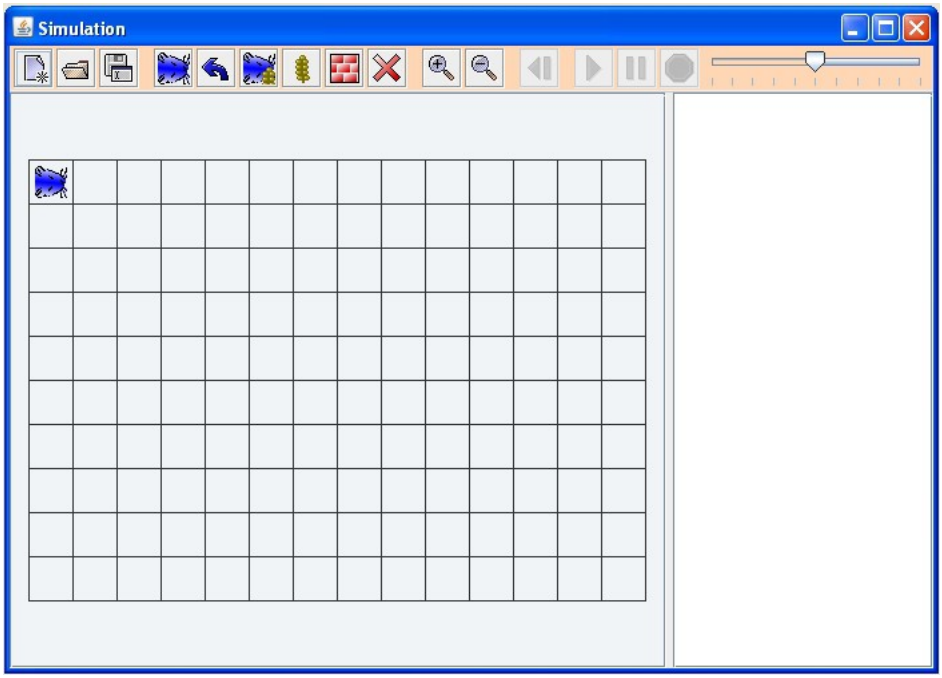
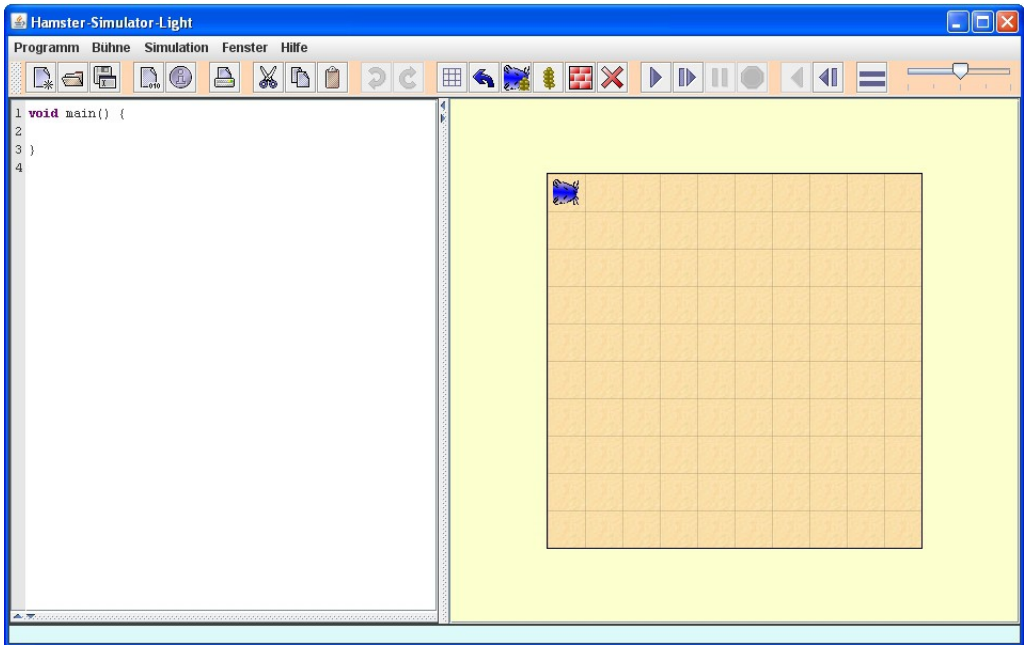


Abbildung 7: Screenshot des Simulations-Fensters vom Java Hamster-Simulator

Das Simulations-Fenster wird ausschließlich über Buttons angesprochen und gesteuert. Dabei dienen die ersten drei Buttons ebenso der Dateimanipulation, mit den folgenden sechs kann man das Territorium verändern, indem man beispielsweise den Hamster umplatziert oder Körner und Mauern auf dem Feld verteilt, die nächsten zwei dienen zur Vergrößerung und Verkleinerung der Ansicht und die letzten Buttons beziehen sich abermals auf die Ausführung des Programms.

Ein interessantes Detail befindet sich rechts neben dem Hamster-Feld. In dieser Textfläche werden nämlich, während der Ausführung eines Programms, sämtliche Schritte (aufgerufene Funktionen, Anweisungen, etc.) mitprotokolliert. Eine alternative Darstellung bietet die 3D-Simulation, die unter dem Menüpunkt „Fenster“ im Editor geöffnet werden kann.

Die light-Variante des Simulators fasst die beiden Fenster der Normalversion in einem zusammen.



Die Funktionsweise der Buttons ist komplett gleich wie in der Standardversion. Der einzige Unterschied ist, dass sich zusätzlich vier weitere Buttons im Menü befinden. Von besonderer Bedeutung sind hierbei der Button mit dem Feld-Symbol (Button 12), mit dem man die Feldgröße ändern kann, der Button rechts neben dem Play-Symbol, der es ermöglicht einen Einzelschritt des Programms ausführen zu lassen und der letzte Button, der die Ablaufverfolgung aktiviert bzw. deaktiviert.

Das Design erhält alleine durch den erdigen Ton des Hamster-Territoriums etwas mehr Abwechslung und wirkt dadurch nicht mehr so steril wie in der Normalversion.

1.2 Die Mikrowelt im Einsatz

1.2.1 Bedienbarkeit

Die Buttons sind intuitiv gestaltet und reichen bei der Bedienung des Simulators und des Editors absolut aus. Einzig die 3D-Darstellung, die Veränderung der Schriftgröße und die Hilfe kann in der Normalversion nicht direkt über die Buttons aufgerufen werden. Sowohl die Normal- aus auch die light-Version bieten außerdem die Möglichkeit, falls die bildliche Darstellung nicht ausreichen sollte, mithilfe von *Mouseover-Events* nähere Informationen zu den

einzelnen Buttons zu erhalten.

Angenehm bei der Handhabung des Java Hamster-Simulators ist außerdem, dass der Editor die klassischen Tastenkombinationen zum Speichern des Dokuments, sowie zum Einfügen und Kopieren von Text zulässt.

In der light-Variante stellte sich bei der Bearbeitung größerer Programme heraus, dass der Simulator gelegentlich dazu neigt *hängen zu bleiben* und keine Befehle mehr entgegen zu nehmen. Manchmal ließ sich der Simulator zwar mit einem Klick auf den Kompilier-Button wieder aus seinem Tiefschlaf holen, in manchen Fällen blieb aber nichts anderes übrig als den Simulator brutal zu beenden und ihn neu zu starten.

1.2.2 Dokumentation und Hilfestellungen

Das mit dem Standard-Simulator mitgelieferte Handbuch bietet zwar zahlreiche Hilfestellungen, um sich bis ins kleinste Detail mit der Funktionsweise des Editors und des Simulators vertraut zu machen, jedoch erfährt man über die gesamte Lektüre hinweg nur wenig bis gar nichts über die Programmierbefehle und deren Einsatz. So haben mich Sätze wie: „Der Hamster-Simulator bietet jedoch noch weitere Möglichkeiten. Diese können Sie nun durch einfaches Ausprobieren selbst erkunden oder im nächsten Abschnitt nachlesen.“ (Hamster Handbuch, S. 27) während des Lesens doch sehr stutzig gemacht, da ich mich in diesem konkreten Fall nach ca. 20 Seiten über die Veränderungen gegenüber der Vorgängerversionen, die Installation des Simulators und die Funktionsweise der wichtigsten Buttons ganz und gar nicht in der Lage fühlte *darauf los zu erkunden*. Das Handbuch bietet insgesamt einen sehr ausführlichen Überblick über die Möglichkeiten (inklusive zusätzlichem Einsatz von Python, Scratch, endlichen Automaten, etc.), die man mit dem Java Hamster-Simulator hat, jedoch ist davon auszugehen, dass man als Neuling nichts darüber erfährt wie man eigentlich mit dem Simulator programmiert.

Besserung bringt dabei erst das Handbuch zum Hamster-Simulator-light, welches schon viel eher das liefert, was für Anfänger interessant ist, nämlich eine Einführung in die Gestaltung eigener Programme. Ein gehöriger Nachteil dabei ist aber, dass das Handbuch nicht auf der Homepage verlinkt ist, sondern man ausschließlich von seiner Existenz erfährt, wenn man die

light-Version herunterlädt.

Als äußerst hilfreich und qualitativ hochwertige Lernbegleitung erweisen sich jedoch die Bücher, die ergänzend zum Java Hamster-Modell veröffentlicht wurden. Von spezieller Bedeutung für diese Arbeit sind die Bücher *Programmieren spielend gelernt* und *Objektorientierte Programmierung spielend gelernt*, welche auf der Homepage des Hamster-Modells teilweise als Leseproben verfügbar sind. Die Bücher enthalten, im Gegensatz zu so manch anderer Programmier-Lektüre, nur selten Fachsprache und sind dabei so aufgebaut, dass der Schwierigkeitsgrad der Inhalte sukzessive gesteigert wird. Zusätzlich bieten sie zahlreiche Beispielprogamme und Übungsaufgaben, um das Gelernte selbstständig umzusetzen. Zusammen mit dem light-Handbuch bilden sie für die Schüler eine ideale Ergänzung als Sekundärliteratur.

Außerdem existiert, wie bereits erwähnt, ein eigenes Java Hamster-Forum, welches auf der Homepage verlinkt ist. Hier finden sich kleine Diskussionen zu den Aufgaben aus den einzelnen Büchern, zu potentiellen Problemen bei der Installation oder allgemeine Anregungen zum Modell. Die Betonung liegt dabei aber auf *klein*, da die Aktivität insgesamt doch sehr überschaubar ist und das Forum dadurch nicht gerade sehr umfangreich erscheint. Ein Vorteil des Forums liegt jedoch immerhin darin, dass Dietrich Boles, der Entwickler des Java Hamster-Simulators, selbst noch im Forum tätig ist und zahlreiche Fragen persönlich beantwortet (Stand: Mai 2013).

2 BYOB

Die Lernumgebung BYOB baut auf dem Programmcode der Mikrowelt *Scratch* auf und wurde von Jens Mönig mit Unterstützung von Brian Harvey entwickelt. In Scratch steuert man einen Akteur – die Scratch-Katze – indem man ihn vorgegebene Programmblöcke ausführen lässt. Diese können per *drag & drop* zusammengefügt und anschließend gemeinsam ausgeführt werden. Die darauffolgenden Aktionen können direkt auf einem kleinen Bildschirm mitverfolgt werden. Hintergrund dafür ist, dass dazu einerseits keine eigene Programmiersprache inklusive Syntax gelernt werden muss und andererseits Programmierergebnisse sofort nachvollziehbar sind. Dies ermöglicht im Unterricht einen schnellen Einstieg, ohne große theoretische Vorarbeit leisten zu müssen (vgl. Scratch-Website).

BYOB unterscheidet sich von Scratch insofern, dass, wie die Langfassung des Namens *build your own blocks* vermuten lässt, auch eigene Blöcke erstellt und eingesetzt werden können. Für diese Arbeit wesentlich ist, dass man so eigene Funktionen kreieren und objektorientierte Konzepte veranschaulichen kann. Aktuell arbeiten Mönig und Harvey zusammen mit Studenten der Berkeley University of California an einer eigenen, vom Programmcode von Scratch unabhängigen, Version von BYOB, die unter dem Namen *Snap!* veröffentlicht werden soll (vgl. Snap!-Website).

2.1 Überblick über die Mikrowelt

2.1.1 Daten und Fakten

Wie es der URL (http://snap.berkeley.edu/) der BYOB-Homepage zu entnehmen ist, beschäftigt sich diese bereits vermehrt mit dem offiziellen Nachfolger von BYOB. Zu BYOB selbst findet man eine kurze Beschreibung, ein Handbuch, Installationsanleitungen für gängige Plattformen, zusätzliche Sprachpakete, Tutorials, Beispielprogramme und Beiträge von Usern.

2.1.2 Voraussetzungen

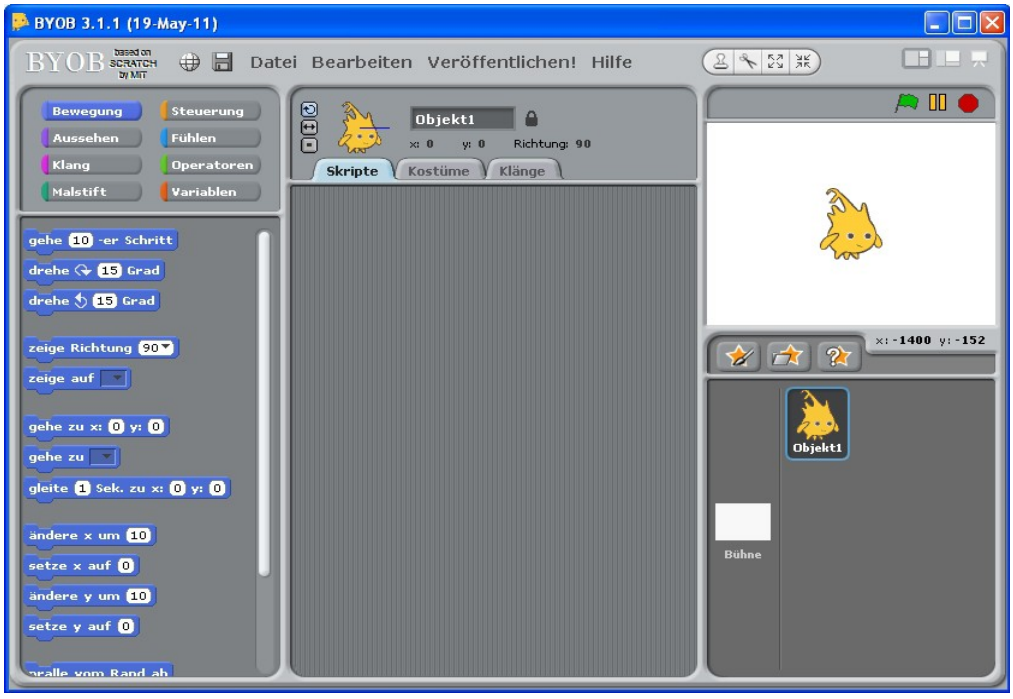
Auf der Homepage werden keine besonderen Voraussetzungen für die Installation erwähnt. Einzig für Unix-Varianten, die nicht Linux entsprechen, bedarf es der Installation von *Squeak* - einer Implementation von Smalltalk.

2.1.3 Installation

Die Installation unter Windows ist entweder als .exe-Datei oder auch als ZIP-File auf der Homepage verfügbar. Auch hier lassen sich bei der Installation unter Windows keine Probleme feststellen. Sowohl die .exe-Version als auch das entpackte ZIP-File funktionieren anschließend einwandfrei.

2.1.4 Design und Funktionalität

Die BYOB-Programmierumgebung ist grundlegend in drei Teile gegliedert. In der linken Spalte befinden sich die Kommandoblöcke, die in die Kategorien *Bewegung*, *Aussehen*, *Klang*, *Malstift*, *Steuerung*, *Fühlen*, *Operatoren* und *Variablen* unterteilt sind. Diese können in der mittleren Spalte wie Puzzleteile zusammengesetzt und den Akteuren bzw. Objekten zugeordnet werden. Hier können neue Programmblöcke kreiert, das Aussehen verändert und Klänge hinzugefügt werden. Die rechte Spalte ist aufgeteilt in einen Bildschirm, der in BYOB *Bühne* genannt wird und eine Auflistung aller, in der aktuellen Datei befindlichen, Akteure.



In der Menüleiste links befinden sich zwei Buttons, die einerseits zum Ändern der Sprache und andererseits zum Speichern des aktuellen Projekts dienen. Darüber hinaus gliedern sich die Punkte des Menüs in Datei-Operationen (Speichern, Öffnen, etc.), Bearbeitungs-Maßnahmen, Methoden der Veröffentlichung (Umwandeln in .exe-Datei und Netzwerksitzungen) und

Hilfestellungen (Handbuch, Website und Online-Hilfe-Seiten). Als am umfangreichsten erweist sich hierbei der Punkt *Bearbeiten*, wo man etwa Schritte rückgängig machen, einen Turbomodus aktivieren oder alle unbenutzten Blöcke entfernen kann. Des weiteren findet sich hier auch die Möglichkeit Bilder und Klänge zu komprimieren, was den Einsatz von zusätzlicher Software obsolet macht. Oberhalb der Bühne befinden sich vier weitere Buttons, die zum Duplizieren, Ausschneiden, Vergrößern und Verkleinern von Objekten dienen. Weiters lässt sich mit einem Klick auf eines der drei Symbole in der rechten oberen Ecke die Bühnengröße variieren. Darunter finden sich Möglichkeiten zum Starten, Anhalten und Stoppen der Simulation. Zuguterletzt lassen sich mit den drei Buttons unterhalb der Bühne neue Objekte malen, aus einer Datei laden, sowie ein Zufallsobjekt erstellen. Wie es vielleicht schon zu entnehmen war, zeigen sich hier, im Gegensatz zum Java-Hamster-Simulator, jedoch keine Redundanzen zwischen dem Menü und den Buttons.

2.2 Die Mikrowelt im Einsatz

2.2.1 Bedienbarkeit

Die Bedienbarkeit ist die große Stärke von BYOB. Wie bereits weiter oben erwähnt, werden die Programmblöcke hier nämlich per drag & drop zusammengefügt, wobei dem Programmierer visuell angezeigt wird, was zusammengehört und was nicht. Teile, die über- oder untereinander geschachtelt werden können, symbolisieren dies indem sie einem Puzzleteil nachempfunden sind. Abgerundete Blöcke wiederum gehören in abgerundete Felder und sechseckige Blöcke in sechseckige Felder. Zudem unterscheiden sich die einzelnen Kategorien farblich voneinander. Somit ist immer auf einem Blick ersichtlich was sich wie im Programm einsetzen lässt.

Als sehr nützlich in der Handhabung der Mikrowelt erweisen sich auch hier die *Mouseover-Events*, die es dem Nutzer, mithilfe der entsprechenden Positionierung des Mauszeigers, ermöglichen Kurzbeschreibungen zu den wichtigsten Menüfunktionen zu erhalten.

Ein kleiner Schwachpunkt in der Bedienung von BYOB zeigt sich bei der Ausführung von Skripten per Tastendruck. Diese Funktion lässt sich leider nämlich nicht zu jedem beliebigen Zeitpunkt nutzen. Speziell, wenn gleichzeitig andere Skripte ausgeführt werden, kommt es gelegentlich zu Komplikationen. Hier hilft ein Klick auf den orangen Pause-Button, da somit alle Skripte angehalten werden und die gegenseitige Beeinflussung verschiedener Skripten

unterbunden wird.

2.2.2 Dokumentation und Hilfestellungen

Um den Einstieg in die Mikrowelt zu vereinfachen, bieten sich idealerweise die zahlreichen Tutorials auf der Homepage zur Lernumgebung Scratch an, auf der BYOB, wie bereits erwähnt, basiert. Unter http://info.scratch.mit.edu/Languages (Stand: Mai 2013) findet man beispielsweise mehrsprachige Files über erste Schritte mit Scratch, grundlegende Funktionen der Lernumgebung, Hilfe-Seiten oder die vermittelbaren Programmierkonzepte.

Weiters gibt es unter http://scratch-dach.info/index.php?title=Hauptseite (Stand: Mai 2013) ein sehr gutes *Wiki* online, in dem Artikel, Links, Tutorials und andere essentielle Informationen zu Scratch und BYOB gesammelt und veröffentlicht werden. Sehr interessant ist dabei auch, dass es eine eigene Kategorie zum Thema *Bildung* gibt, in der speziell auf den Einsatz der beiden Lernumgebungen in der Schule eingegangen wird.

Eine eigene Website, die sich wiederum gesondert mit dem Einsatz von Scratch und BYOB in der Schule befasst ist http://scratched.media.mit.edu/ (Stand: Mai 2013). Hier finden sich abermals zahlreiche Links zu Projekten, Veröffentlichungen und Tutorials.

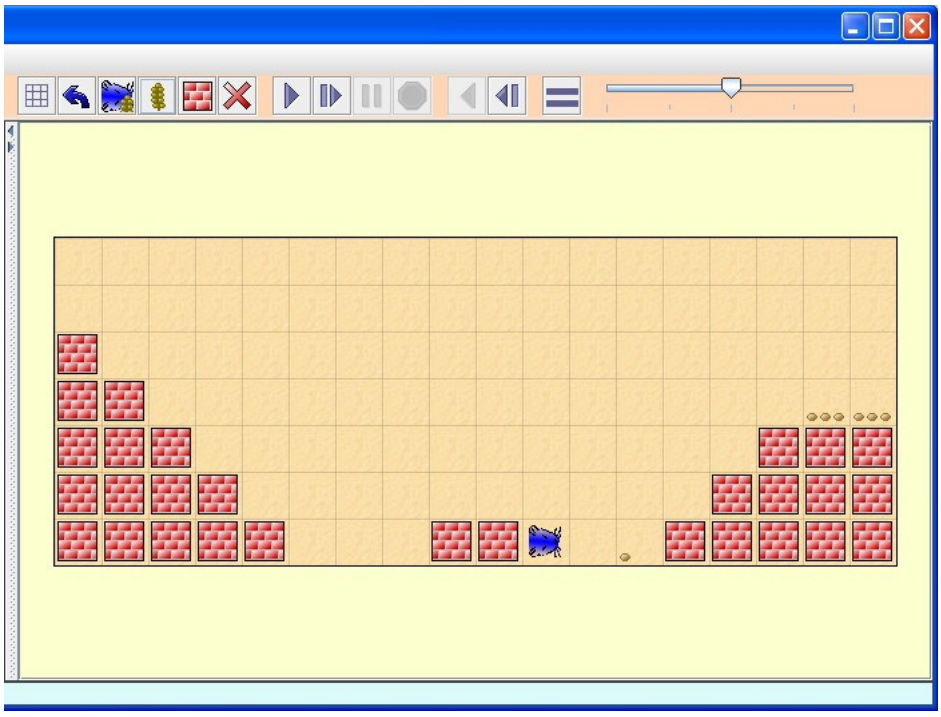
Das Handbuch zu BYOB, welches direkt über den Menüpunkt *Hilfe* der Lernumgebung aufgerufen werden kann, liefert einen guten und detaillierten Überblick über, für diese Arbeit, mehr als genügende Funktionen der Mikrowelt. Da es sich dabei jedoch um ein Skript in englischer Sprache handelt, findet sich darüber hinaus unter dem Link http://touchit.fh-joanneum.at/fileadmin/PDF/BYOB_Anleitung.pdf (Stand: Mai 2013) eine Kurzanleitung in deutscher Sprache, die speziell auf die Kreation eigener Blöcke in BYOB ausgerichtet ist.

Bereits dieser kleine Auszug an online-Resourcen lässt erahnen welches Repertoire an Material zu BYOB alleine im Internet verfügbar ist. Diese Vielfalt ist, im Gegensatz zum Java Hamster-Modell, sicherlich mitunter dadurch zu begründen, dass die Lernumgebung in zahlreichen Sprachen veröffentlicht wurde und so eine deutlich größere Community entstehen konnte. Auch hier ergäbe sich natürlich wieder eine weitere interessante Komponente für den Einsatz in der Schule, nämlich die Möglichkeit des *polylingualen* Unterrichts.

3 Der Hamsterbau

Wie bereits beim Punkt Rahmenhandlung im *goal-based scenario* angedeutet, erhalten die Schüler einen vorgegebenen Hamsterbau, der als Umgebung für die Erarbeitung erster eigener Programme dienen soll. Es folgt ein Einblick wie dieser Bau in der jeweiligen Mikrowelt technisch zu realisieren ist. Prinzipiell besteht der Bau aus fünf Stufen, die sich am linken Rand befinden, einem kleinen Bett in der Mitte und drei weiteren Stufen auf der rechten Seite, die zu einer Vorratskammer, die mit Körnern oder Getreide gefüllt ist, führen.

3.1 Java-Hamster-Simulator



Beim Java-Hamster-Simulator beschränkt sich der Gestaltungsaufwand für den Hamsterbau der ersten fünf Beispiele in erster Linie auf die Festlegung der Territoriumsgröße von 7 Reihen und 18 Spalten. Anschließend können, mithilfe der zugehörigen Buttons, die Mauerblöcke, die Körner und der Hamster im Feld positioniert werden.

Das Territorium der letzten drei Aufgaben besteht aus 8 Reihen und 14 Spalten, wobei sich außen ein Rechteck aus Mauern befindet. Auf den freien Feldern soll jeweils, wie oben beschrieben, eine beliebige Anzahl an Körnern platziert werden.

3.2 BYOB

Als etwas aufwändiger erweist sich jedoch die Gestaltung des Hamsterbaus in BYOB. Hierzu habe ich mir mit dem frei-verfügbaren Bildbearbeitungsprogramm *GIMP* beholfen. Der Abschnitt, der die überirdische Welt repräsentieren soll, ist ein vorgegebener Hintergrund von BYOB, der sich im Ordner *Backgrounds/Outdoors* unter dem Namen *hay_field* befindet. Den unterirdischen Bau habe ich gestaltet indem ich die frei-bleibende Fläche mit dem, in GIMP zur Verfügung stehenden, Muster *Walnut* gefüllt und mit transparenter, schwarzer Farbe verdunkelt habe. Die Stufen bzw. die Hindernisse habe ich aus einzelnen, mit dem Muster *Burlwood* gefüllten, Quadraten wie Bauklötze in GIMP zusammengesetzt.

Die Abbildungen des Hamsters und der Weizenähre stammen aus der *Open Clipart-Library* (*http://openclipart.org/*), wobei ich die Cliparts noch ein wenig verändert habe. Beim Weizen[2] beschränken sich die Änderungen auf die Farbe, beim Hamster[3] war der Aufwand höher, da das Clipart eigentlich ein dunkelbraunes Eichhörnchen zeigt.

Damit nun die Szenario-Handlungen in BYOB möglichst gut umgesetzt werden können, bedarf

2 http://openclipart.org/detail/144535/wheat-black-and-white-by-serioustux, Zugriff: 22.05.13 – 8:18
3 http://openclipart.org/detail/103189/squirrel-by-pesasa, Zugriff: 22.05.13 – 8:18

es noch der einen oder anderen Adaptierung mithilfe von Skripten. So sollte beispielsweise, sobald der Hamster eine der Weizenähren berührt, diese verschwinden und der Hamster ein gesättigtes „Lecker!“ von sich geben, um dem Schüler zu signalisieren, dass die Essensaufnahme erfolgreich war. Ich habe dies mit einem Hamster-Skript und vier gleichen Skripten für die Weizenähren realisiert:

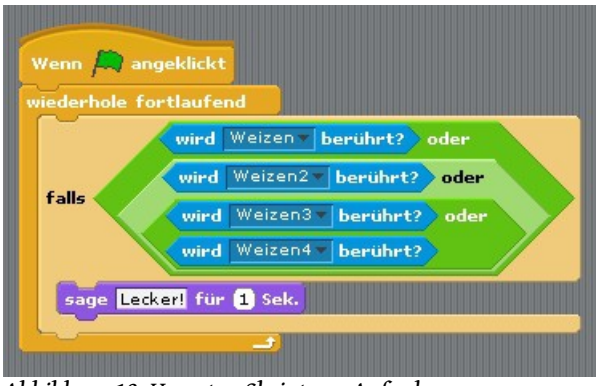


Abbildung 12: Hamster-Skript zur Aufnahme

Abbildung 13: Weizen-Skript zur Aufnahme

Unbedingt zu berücksichtigen ist dabei der *warte 0.1 Sekunden*-Block im Weizen-Skript. Dieser verhindert, dass der Hamster nicht vergisst „Lecker!“ zu sagen. In den Probeläufen stellte sich auf langsamen Rechnern nämlich heraus, dass der Weizen manchmal so schnell verschwindet, dass der Hamster gar nicht mehr *merkt*, dass er ihn gerade berührt hat. Das liegt wohl daran, dass das Hamster-Skript bei einem Schleifen-Durchlauf bis zu vier Bedingungen überprüfen muss, das Weizen-Skript hingegen nur eine einzige und daher schneller einen Durchlauf beendet.

Um den Ausgangszustand immer wieder herstellen zu können, ohne dabei die Datei ständig neu zu laden, habe ich mir ebenso mit Skripten beholfen:

Abbildung 14: Hamster-Skript zur Herstellung der Ausgangssituation

Abbildung 15: Weizen-Skript zur Herstellung der Ausgangssituation

Drückt der Schüler also die Leertaste, so gehen alle Hamster- und Weizen-Objekte in ihre Ausgangslage zurück. Dazu sei erwähnt, dass Abbildung 15 nur ein Beispiel für eine Weizenähre

ist und sich daher die Koordinaten der übrigen Objekte selbstverständlich unterscheiden.

Wichtiger Hinweis: Die Wiederherstellung der Ausgangslage funktioniert nicht immer einwandfrei. Manchmal ist es notwendig zuerst den Pause- oder den Stopp-Button zu betätigen.

Auch für den Hintergrund der letzten drei Aufgaben verwende ich ein Clipart aus der Open-Clipart-Library[4], das ich selber wieder mit den bereits bekannten Methoden etwas umgestaltet habe.

Abbildung 16: Hintergrund für die letzten drei Aufgaben in BYOB

4 Programmieraufgaben

In diesem Abschnitt finden sich die Angaben zu den Programmierbeispielen und zwar in der Form, in der sie den Schülern zum Ausarbeiten zur Verfügung gestellt werden.

4 http://openclipart.org/detail/86617/guadua-by-rastrojo, Zugriff: 22.05.13 – 8:18

4.1 Einführung in das Szenario

Zuerst wird der Schüler auf die Situation eingestimmt indem er ein kurzes Briefing darüber erhält was in dem Szenario bisher geschehen ist und was nun von ihm verlangt wird:

> Du bist ein Tierzüchter, der sich besonders auf Nagetiere spezialisiert hat. Bei einem deiner Spaziergänge, entdeckst du auf einem Feld plötzlich einen hilflosen kleinen Hamster. Verantwortungsbewusst wie du bist, nimmst du ihn, zum Schutz vor gefräßigen Raubtieren, mit zu dir nach Hause. Dort angekommen, machst du dich eifrig ans Werk und baust dem kleinen Tier seinen eigenen Hamsterbau...
>
> Öffne nun die Datei Hamsterbau.ter bzw. Hamsterbau.ypr!

4.2 Die ersten Gehversuche

> Der Bau ist also fertig. Nun liegt es an dir den Hamster auf ein eigenständiges Leben in der freien Wildbahn vorzubereiten!
>
> Du stellst fest, dass der junge Hamster erst kürzlich zur Welt gekommen ist und daher noch nicht einmal sehen kann. Tollpatschig stolpert er vor sich hin, wenn er probiert eine Pfote vor die andere zu setzen. Noch mag es ihm nicht so recht gelingen sich vorwärts zu bewegen. Kannst du ihm mit einem Programm helfen? Lass den Hamster dabei bis vor die Stufen auf der rechten Seite gehen! Damit er weiß, dass er etwas richtig gemacht hat, soll der Hamster dort zur Belohnung ein Korn erhalten!
>
> Sei dir bei all deinen künftigen Aufgaben bewusst, dass der Hamster seine Umwelt zwar nicht sehen aber sehr wohl fühlen kann. Du musst ihn also immer bis zu einem Punkt führen, an dem er merkt, dass er nicht mehr weiter gehen kann.

4.3 Auf zu höheren Gefilden

> Der kleine Hamster hat zwar gerade erst gehen gelernt, aber er fühlt sich schon für größere Aufgaben berufen. Er möchte jetzt nämlich die drei Stufen zur Vorratskammer erklimmen. Leider hat er noch gar keine Ahnung wie er das anstellen soll. Hilf ihm, indem du ein Programm schreibst, welches die nötigen Schritte vorgibt! Am Ende sollte sich dein Hamster auf der obersten Stufe mit Blick nach rechts befinden.

4.4 Mahlzeit

> Jetzt hat der kleine Hamster einen Riesen-Hunger. Er nimmt sich vor, solange zu essen, bis die ganze Vorratskammer leer ist. Schreibe ein Programm, das den Hamster solange essen lässt, bis keine Vorräte mehr vorhanden sind!

4.5 Hamsterliche Bettruhe

> Vollgefressen sehnt sich Hamster nach einer kleinen Ruhepause. Die Vorratskammer scheint ihm dafür aber unangemessen. Viel lieber würde er sein Bett unten nach den Stufen nutzen. Dass das Hinuntergehen im Prinzip genauso funktioniert wie das Hinaufgehen weiß

der kleine Hamster schon. Doch er möchte in Zukunft auch die Stiege mit den fünf Stufen verwenden, die aus seinem Bau herausführt, ohne dafür wieder etwas Neues lernen zu müssen. Hilf ihm dabei!

4.6 Ein ganz normaler Hamstertag

Schreibe abschließend ein Programm, welches den Tagesablauf des kleinen Hamsters simuliert! Er steht vom Bett auf, geht in die Vorratskammer zwei Körner (in BYOB: eine Ähre) essen, geht anschließend aus dem Bau, kehrt in den Bau zurück, isst noch einmal drei Körner (in BYOB: zwei Ähren) und geht schließlich schlafen. Du kannst dafür selbstverständlich den Code aus den vorangegangenen Aufgaben verwenden. Achte dabei darauf, dass du für sich-wiederholende Schritte den Code nicht doppelt verwendest, sondern ihn in Methoden verpackst!

4.7 Tapetenwechsel

Dein Hamster weiß jetzt dank dir schon wie der Alltag in seinem Bau auszusehen hat. Da du deine Nagetier-Kinder aber ständig vor neue Herausforderungen stellst, hältst du es für das Beste zu testen, wie sich dein Hamster in einem anderen Bau zurechtfindet. Tue dich dafür mit einem Kollegen oder einer Kollegin zusammen und baut euch dann gegenseitig neue Territorien! Testet dann, ob euer Ablauf-Programm auch für die Konstruktion eurer Kollegen funktioniert! Wenn nicht, passt es so lange an, bis es funktioniert!

4.8 Zeit zu „hamstern“

In der erdigen Unterwelt der Felder ist dein Hamster nun schon zu einem wahren Meister der Aufgabenbewältigung geworden. Das es aber auch zu deinem Ziel gehört den Hamster auf das Leben auf den Feldern vorzubereiten, ist es nun an der Zeit einen ersten Testlauf unter freiem Himmel durchzuführen. Damit dein Hamster nicht weglaufen kann, hast du dafür einen kleinen Feldabschnitt eingezäunt. Da du auch darauf geachtet hast, dass dein Hamster dort reichlich zu futtern hat, ist dies die ideale Gelegenheit für deinen Schützling einen Vorrat für den bevorstehenden Winter zu hamstern. Dein Hamster soll nun das gesamte Feld abgrasen und alles Essbare aufnehmen was ihm vor das Maul kommt!

4.9 Der Eindringling

Nachdem dein Hamster seinem Namen alle Ehre gemacht und zum ersten mal reichlich gehamstert hat, entschließt du dich ihn gleich vor eine weitere Herausforderung zu stellen. Er soll nun nämlich den Erstkontakt mit einem Artgenossen proben. Setze dazu irgendwo an den Rand des Feldes einen weiteren Hamster!

4.10 Hamsterliebe

Als dein Hamster den Eindringling entdeckt, will er sofort wissen mit wem er es zu tun hat und beginnt neugierig auf ihn zuzulaufen. Etwas ängstlich und scheu eilt der neue Hamster sofort in die selbe Richtung davon, worauf eine wilde Verfolgungsjagd entsteht. Leider können die beiden Hamster dabei nur am Rande des Feldes, entlang des Zaunes, laufen, da der Boden in der Mitte des Feldes voller leerer Getreidehülsen ist. Neugierig wie dein Schützling ist, ist er bei der Verfolgung aber sogar mit der doppelten Geschwindigkeit wie

der gejagte Hamster unterwegs.

5 Infotexte

Hier werden die, zu den einzelnen Aufgaben gehörigen, Infotexte vorgestellt. Zuerst erhalten die Schüler jedoch eine Einführung zur Lösung von Problemstellungen mithilfe von Programmen und zur jeweiligen Programmierumgebung, anhand von Texten, welche sich jeweils am Anfang der Punkte 5.1, 5.2 und 5.3 befinden.

5.1 Lösung von Problemstellungen mithilfe von Programmen

Um dir den Einstieg in die Pröblemlösung mit Computer-Programmen zu erleichtern, soll dir folgende Anleitung behilflich sein. Gehe die einzelnen Fragen zu jedem der Punkte bei der Lösung der Aufgaben durch und mache dir bei gegebenenfalls Notizen!

- Problemsituation: Erfassen und Verstehen des Problems.
  - Suchen einer präzisen und kurzen Problembenennung.
  - Gibt es ein vergleichbares Problem?
  - Kann man das Problem verallgemeinern?
  - Läßt sich das Problem in leichter bearbeitbare, übersichtlichere *Teilprobleme* zerlegen?
- Spezifikation: Anforderung an die Daten, Verhältnis zwischen Input und Output.
  - Was soll verändert werden?
  - Gegenüberstellung des Ausgangszustands zum Endzustand.
- Design: Wahl einer algorithmischen Technik.
  - Wie kommt man vom Ausgangszustand zum Endzustand?
  - Mit welchen Programmier-Konzepten lassen sich die Teilprobleme umsetzen?
  - Wo müssen Entscheidungen getroffen werden?
  - Wo wiederholen sich Ereignisse?
- Programmierung: Erstellen einer Lösung für den Computer (eigentliches Programm).
- Testen: Feststellung ob gewünschtes Ergebnis eintritt und Effizienz beurteilen.
  - Starten der Simulation in der Mikrowelt.
  - Wenn Fehler eintreten, wo treten diese ein?

- Verfolgung des Programms in Einzelschritten.

- Präsentation: Präsentieren der Lösung und Diskussion der Anwendung.

5.2 Java-Hamster-Simulator

Das Fenster des Java-Hamster-Simulators ist unterteilt in eine Menüleiste (oben), in einen Editor-Bereich (links) und einen Territoriums-Bereich (rechts). Im Editor-Fenster findet das eigentliche Programmieren statt. Hier schreibst du deine Programme, die du danach im Territorium veranschaulichen kannst. Erstellst du ein neues Programm (), dann wird automatisch die `main()`-Methode geöffnet. In diese Methode schreibst du dann die Anweisungen hinein, die du deinem Hamster erteilen möchtest. Dein Hamster kann die folgenden Aktions-Befehle ausführen: `vor()`, `linksUm()`, `nimm()` und `gib()`. Probiere einfach selbst aus was passiert, wenn du einen der ersten beiden Befehle in dein Programm schreibst und dieses anschließend ausführst! Außerdem ist er in der Lage mit Befehlen bestimmte Dinge zu überprüfen. Dies tut er mit den Methoden `vornFrei()`, `kornDa()` und `maulLeer()`.

Beispiel:

```
void main() {
      vor();
}
```

Wichtig: Jede Anweisung muss mit einem Strichpunkt (`;`) beendet werden!

Beachte auch, dass jedes deiner Programme vor dem Ausführen *kompiliert* werden muss. Das tust du indem du auf den dementsprechenden Button klickst (). Dieser Schritt ist notwendig um etwaige Fehler im Programm auszumachen und falls alles richtig ist, verwandelt der Computer das Programm darauffolgend in eine für ihn verständliche Sprache. Um nach der Ausführung eines Programms zum Ausgangszustand zurück zu kehren, verwende den Button  .

5.2.1 Die ersten Gehversuche

Das obige Code-Beispiel soll dir veranschaulichen wie du Befehle in deine `main()`-Methode schreibst. Tue dies nun genauso mit den Befehlen, die nötig sind um diese Aufgabe zu lösen.

Vergiss nicht das Programm zu speichern und zu kompilieren!

5.2.2 Auf zu höheren Gefilden

Überlege dir welche Schritte bzw. welche Drehungen dein Hamster vollführen muss um eine Stufe zu überwinden! Leider kann sich dein Hamster nicht einfach nach rechts drehen. Weißt du vielleicht eine Möglichkeit eine rechts-Drehung durch links-Drehungen zu ersetzen?

Wenn du eine Lösung gefunden hast, kannst du diese in eine eigene *Methode* schreiben. Methoden dienen dazu Daten und Zustände von Objekten (wie dein Hamster) zu ändern. Was du nun gestalten sollst, ist wiederum eine spezielle Form der Methode, nämlich eine *Prozedur*. Diese liefert, im Gegensatz zur Methode, kein Ergebnis zurück, sondern dient ausschließlich zur Datenmanipulation.

Allgemein besteht eine Methode aus einem Rückgabetyp, dem Methodennamen, gegebenenfalls aus einem oder mehreren Parametern und dem Methodenrumpf, der einen Anweisungsblock beinhaltet.

```
Rückgabetyp Methodenname(Parameter) {
      Anweisung(en);
      Rückgabewert;
}
```

Prozeduren wird das Schlüsselwort `void` vorangestellt. Hat die Methode einen Rückgabewert, so muss dieser im Methodenrumpf mit dem `return`-Befehl angezeigt werden.

Ein Beispiel für eine Prozedur wäre:

```
void zweiMalVor() {
      vor();
      vor();
}
```

Wie der Name schon sagt, lässt diese Prozedur den Hamster zwei Schritte nach vorne gehen. Das Schlüsselwort `void` zeigt dabei an, dass es sich um eine Prozedur handelt. Würde anstelle

von `void` beispielsweise `int` (für *Integer* = Ganzzahlen) stehen, dann würde es sich um eine Methode mit einem ganzzahligen Rückgabewert handeln.
Achte darauf, dass du alle eigenen Methoden und Prozeduren unterhalb der `main()`-Methode schreibst und nicht die geschweiften Klammern am Anfang und am Ende der Methode vergisst! Außerdem sollten Methoden immer mit einem Kleinbuchstaben beginnen. Wenn du die Methode aufrufen willst, dann schreibe den Methodennamen in die `main()`-Methode und setzte ans Ende einen Stichpunkt!

```
void main() {
      zweiMalVor();
}
```

5.2.3 Mahlzeit

Es sollte hier nun nicht das Ziel sein sechs mal den `nimm()`-Befehl in dein Programm zu schreiben. Programmiersprachen wie Java bieten nämlich die Möglichkeit mithilfe von sogenannten *Schleifen* Anweisungen mehrfach auszuführen. Eine Form der Schleife ist die `while`-Schleife. Diese führt eine Anweisung so lange aus, bis eine Bedingung nicht mehr erfüllt ist.

```
while(Bedingung erfüllt) {
      Anweisung;
}
```

Dazu ein Beispiel aus der Hamsterwelt:

```
while(vornFrei()) {
      vor();
}
```

Mit dieser Schleife geht der Hamster solange einen Schritt nach vorne, bis er keine freie Bahn mehr hat. Probiere nun selbst eine ähnliche Schleife für das Aufnehmen von Körnern zu schreiben! Erinnere dich, dass der Hamster mit folgenden drei Methoden Dinge überprüfen kann: `vornFrei()`, `kornDa()` und `maulLeer()`. Die Schleife kannst du nun entweder in die `main()`-Methode schreiben oder in eine eigene Prozedur verpacken.

5.2.4 Hamsterliche Bettruhe

Versuche nun das Herabschreiten der Stufen ebenfalls mithilfe einer Schleife zu formulieren! Hilfreich könnte dir dabei eine *bedingte Anweisung* sein. Bedingte Anweisungen oder auch `if`-Anweisungen genannt, führen ein Ereignis nur unter einer bestimmten Bedingung aus.

```
if(Bedingung) {
      Anweisung(en);
}
```

Prinzipiell erfolgt zuerst die Überprüfung der Bedingung. Ergibt diese den Wert `true`, dann wird die darunter stehende Anweisung ausgeführt. Entspricht der Wert der Bedingung `false`, dann wird das Programm bei der ersten Anweisung nach dem Block der `if`-Anweisung fortgesetzt.

Ein Beispiel dafür wäre:

```
if(vornFrei()) {
      vor();
}
```

Ist der Weg für den Hamster frei, so bewegt er sich ein Feld nach vorne. Außerdem gibt es die Möglichkeit mit einem zusätzlichen `else`-Zweig eine Handlungsalternative vorzuschlagen:

```
if(Bedingung) {
      Anweisung 1;
} else {
      Anweisung 2;
}
```

In der Hamsterwelt könnte dies beispielsweise so aussehen:

```
if(vornFrei()) {
      vor();
} else {
      linksUm();
}
```

Ist der Weg frei, geht der Hamster nach vorne, ist der Weg nicht frei, dann dreht sich der Hamster nach links. Wichtig ist, dass du die `if`-Anweisung nicht mit einer Schleife verwechselst! Das Funktionsprinzip ist zwar gleich, jedoch wird die bedingte Anweisung im Gegensatz zur Schleife nur ein einziges mal ausgeführt.

Die Bedingung, egal ob bei einer Schleife oder einer bedingten Anweisung, wird immer auf ihren Wahrheitswert geprüft. Wahrheitswerte sind in Java `true` und `false` und werden auch als *boolesche Werte* bezeichnet. Eigentlich liefern die Methoden `vornFrei()`, `kornDa()` und `maulLeer()` jeweils `true` oder `false` zurück und können nur deswegen in die Bedinung geschrieben werden.

Deshalb könnte dir bei der Ausarbeitung der Verneinungs-Operator „`!`“ sehr hilfreich sein. Dieser verneint einen booleschen Wert, indem er seinen Wahrheitswert umkehrt. Aus `true` wird also `false` und umgekehrt.

Hierzu folgendes Beispiel:

```
if(!kornDa()) {
      vor();
}
```

In deutsche Sprache übersetzen, lässt sich dies mit: „Wenn *kein* Korn da ist, dann gehe einen Schritt nach vorne.“

5.2.5 Ein ganz normaler Hamstertag

Setzte nun das bisher gelernte in einem großen eigenen Programm um! Verpacke dabei alle Einzelhandlungen (das Erklimmen der Vorratskammer, das Aufnehmen der Körner, etc.) in eigene Prozeduren! Es kann sein, dass das Programm schlussendlich aus sehr vielen Zeilen besteht, solange du aber darauf achtest, dass die Prozeduren aussagekräftige Namen erhalten, sollte die Funktionalität aus dem Hauptprogramm in der `main()`-Methode ersichtlich sein.

5.2.6 Tapetenwechsel

Klicke dafür zu aller erst auf den Button  und wähle anschließend eine geeignete Territoriumsgröße!
Mit einem Klick auf den Mauer-Button  kannst du einzelne Mauerstücke in deinem Territorium verteilen und mithilfe des Korn-Buttons  einzelne Körner.

Achte bei dieser Aufgabe darauf, dass du nicht mehr als vier Hindernisse einbaust, da es sonst zu komplex wird! Ziel sollte es sein den Hamster im neuen Hamsterbau einmal von einer Seite zur anderen laufen zu lassen und dabei alle Körner aufzunehmen.

5.2.7 Zeit zu „hamstern“

Starte nun die Standard-Version des Java Hamster-Simulators und öffne das Territorium ZeitZuHamstern.ter! In der Standard-Version hast du die Möglichkeit eine 3D-Darstellung anzeigen zu lassen. Du kannst sie aktivieren, indem du im Editor-Fenster in der Menüleiste auf *Fenster* klickst.
Selbstverständlich bleibt es dir selbst überlassen wie du deinen Hamster das Feld abgrasen lässt. Einer der wohl leichtesten Varianten ist es aber den Hamster eine Reihe hamstern zu schicken, ihn wieder zurücklaufen und in die nächste Reihe wechseln zu lassen.

Bevor du deinen eigenen Lösungsweg formulierst, sollst du deinem Hamster bei dieser Aufgabe einen Namen geben. Dazu musst du zu aller erst einmal eine Variable mit dem Namen deines Hamsters erstellen. Variablen dienen allgemein, ähnlich wie in der Mathematik, zur Speicherung von Werten.

Variablen werden in Java wie folgt erstellt:

```
variablentyp name;
```

Bei den Variablentypen beschränken wir uns in der Hamsterwelt auf die Typen *Integer* (`int`) für Ganzzahlen und den Hamstertyp `Hamster`.

Um einen Hamster später im ganzen Programm ansprechen zu können, musst du außerhalb

der `main()`-Methode zum Beispiel folgendes schreiben:

```
Hamster tapsi;
```

Diese Anweisung erzeugt eine Variable vom Typ `Hamster` mit dem Namen `tapsi`. Du kannst dir natürlich auch einen eigenen Namen für deinen Hamster überlegen. Der Hamster `tapsi` ist zwar nun kreiert, jedoch hat er noch gar keine Eigenschaften. Mit einer Zuweisung kannst du dies ändern. Dabei musst du nur schauen, dass die linke Seite der Zuweisung den selben Typ wie die rechte Seite hat. Einfach gesagt dürfen Hamster-Variablen also nur Hamster enthalten und Integer-Variablen nur Ganzzahlen.

Mit folgender Anweisung wird anschließend der Variable der Standard-Hamster zugewiesen:

```
tapsi = Hamster.getStandardHamster();
```

Das heißt, dass künftig der Hamster im Territorium mit seinem Namen angesprochen werden muss. Für deine weiteren Programme bedeutet das, dass du jedem Befehl den Namen deines Hamsters und einen Punkt voranstellen musst. Aus `vor();` wird also `tapsi.vor();` und aus `vornFrei();` wird `tapsi.vornFrei();`. Bei eigenen Methoden und Prozeduren musst du den Namen übrigens nicht voranstellen.

Du kannst die Schritte aus dem obigen Beispiel übrigens auch in einem machen:

```
Hamster tapsi = Hamster.getStandardHamster();
```

Versuche bei der Lösung des Beispiels zudem eine eigene *Überprüfungs-Methode* zu schreiben! Wie schon einmal angemerkt, musst du bei der Definition einer Methode einen Rückgabetyp angeben. Eine Methode, die `true` oder `false` zurückliefern soll, hat immer den Rückgabetyp `boolean`. In der Methode selbst, wird der Punkt an dem ein Wert zurückgegeben werden soll mit dem `return`-Befehl angezeigt.

Hier ein Beispiel:

```
boolean linksFrei() {
```

```
        tapsi.linksUm();
        return tapsi.vornFrei();
    }
```

Die Methode lässt den Hamster also einmal nach links drehen und überprüft dann, ob der Weg nach vorne frei ist. Das Ergebnis dieser Überprüfung ist der Rückgabewert der Methode `linksFrei()`.

5.2.8 Der Eindringling

Im Ordner des Java Hamster-Simulators befindet sich ein weiterer Ordner namens „pseudo-hamsterklassen“. Darin beinhaltet ist die Datei Hamster.java, die du nun mit einem Editor öffnen sollst. In der Datei siehst du unter anderem sogenannte *Konstruktoren*, die zur Erzeugung neuer Hamster dienen.

Eine Möglichkeit einen Konstruktor aufzurufen, wäre beispielsweise:

```
Hamster gusti = new Hamster();
```

Dies erstellt einen Hamster, der aber weder einen Ort, noch eine Blickrichtung oder sonstiges besitzt. In der Datei Hamster.java ist in diesem Fall von einem „nicht initialisierten Hamster“ die Rede.
Finde nun den Konstruktor mit dem man die Position, die Blickrichtung, die Anzahl der Körner im Maul und die Farbe angeben kann! Nachdem du ihn gefunden hast, erzeuge einen neuen Hamster, der die Angaben im Beispieltext erfüllt!

5.2.9 Hamsterliebe

Ganz nach dem revolutionären Motto „neue Hamster – neue Methoden“ stehen dir ab jetzt neue Befehle zur Verfügung. Nämlich `getReihe()`, `getSpalte()`, `getAnzahlKoerner()` und `getBlickrichtung()`. Diese Methoden liefern allesamt eine Ganzzahl zurück, wobei `getReihe()` und `getSpalte()` die Koordinaten des aktuellen Feldes, `getAnzahlKoerner()` die Anzahl der Körner im Maul des Hamsters und `getBlickrichtung()` die Blickrichtung des Hamsters zurückgibt. Dabei stehen die Ganzzahlen der Blickrichtung jeweils für 0 = Norden, 1 = Osten, 2 = Süden und 3 = Westen.

Natürlich gibt es auch bei dieser Aufgabe wieder Programmierkonzepte, die dir bei der Ausarbeitung hilfreich sein könnten. Beispielsweise kann man mit den sogenannten Vergleichsoperatoren zwei Werte miteinander vergleichen. Das Ergebnis eines solchen Vergleichs kann entweder wahr oder falsch sein, also wieder `true` oder `false`. Deswegen nennt man diese Form von Vergleichen auch *boolesche Ausdrücke.* Folgende Operatoren zählen zu den Vergleichsoperatoren: größer `>`, größer oder gleich `>=`, kleiner `<`, kleiner oder gleich `<=`, gleich `==` und ungleich `!=`.

Beispiele:

- `3 < 7`, liefert `true`
- `3 >= 7`, liefert `false`
- `3 == 7`, liefert `false`
- `3 != 7`, liefert `true`

Für den Hamster aus den vorherigen Beispielen, könnte ein beispielhaftes Programmstück so aussehen:

```
if(tapsi.getAnzahlKoerner() == 4) {
      tapsi.vor();
}
```

Befinden sich also 4 Körner im Maul des Hamsters `tapsi`, dann begibt er sich ein Feld nach vorne.

Mithilfe von logischen Operatoren können mehrere Vergleiche oder allgemein boolesche Ausdrücke miteinander verknüpft werden. Damit kann man beispielsweise die Abarbeitung eines Programmteils an mehrere Bedingungen knüpfen. Die wichtigsten Operatoren sind hierbei das *Und* `&&`, das *Oder* `||` und das bereits bekannte *Nicht* `!`:

- Und: `a && b`, liefert `true`, wenn `a` und `b true` sind.
- Oder: `a || b`, liefert `true`, wenn `a` oder `b true` sind.
- Nicht: `!a`, liefert `true`, wenn `a false` ist.

Wenn nun zwei Hamster namens `tapsi` und `gusti` existieren, könnte man daher folgende

Bedingung formulieren:

```
if(tapsi.getBlickrichtung() == gusti.getBlickrichtung() &&
tapsi.getAnzahlKoerner() ==  tapsi.getAnzahlKoerner()) {
      tapsi.vor()
      gusti.vor()
}
```

Besitzen also beide Hamster sowohl die gleiche Blickrichtung als auch die gleiche Anzahl an Körnern im Maul, dann sollen sich beide um ein Feld nach vorne bewegen.

5.3 BYOB

In BYOB bastelst du deine Programme indem du einzelne Programmblöcke wie Puzzleteile zusammensteckst. Wie du auf einem Blick erkennen kannst, sind die Programmblöcke dabei in die Kategorien *Bewegung*, *Aussehen*, *Klang*, *Malstift*, *Steuerung*, *Fühlen*, *Operatoren* und *Variablen* aufgeteilt. Die einzelnen Blöcke aus den jeweiligen Kategorien kannst du in das mittlere Feld in den Abschnitt *Skripte* ziehen. Ausführen kannst du einen Block, indem du anschließend einmal auf ihn klickst. Am besten du probierst selbst einmal aus was passiert wenn du den einen oder anderen Block in dein Skript-Feld ziehst und ihn dann ausführst!

All deine Programmblöcke, die du für deinen Hamster zusammenstellst, solltest du jedoch immer mit einer *grünen Fahne* versehen. Diesen Block findest du unter der Kategorie *Steuerung*. Wenn du dann im rechten oberen Eck auf die grüne Fahne klickst, startet dein Programm. Anhalten kannst du es mit dem orangen *Pause-Button* und stoppen mit dem roten *Stopp-Button*. Wenn du den Ausgangszustand im Hamsterbau herstellen willst, dann betätige die *Leertaste*.

5.3.1 Die ersten Gehversuche

Mit welchem Block du deinen Hamster durch den Bau gehen lassen kannst, wirst du wahrscheinlich schnell herausgefunden haben. Es stellt sich nun natürlich die Frage wieviele Schritte dein Hamster bis zu den Stufen zu gehen hat. Indem du mit dem Mauszeiger über den Hamsterbau fährst, kannst du dir mithilfe der x-Koordinaten rechts unterhalb des Baus selbst ermitteln wieviele Schritte der Hamster zu gehen hat.
Versuche das Ergebnis aber noch etwas zu verfeinern! Finde dafür einen Weg, bei dem du nicht

vorgeben musst wieviele Schritte zu gehen sind. Kleiner Tipp: Dein Hamster ist in der Lage zu *fühlen*! ;)

5.3.2 Auf zu höheren Gefilden

Solltest du feststellen, dass der Block 

deinen Hamster zu schnell durch den Bau flitzen lässt, dann sieh dir mal den Block 

an! Hier kannst du einstellen wie lange der Hamster brauchen soll um zu einem bestimmten Punkt zu gelangen. Um nicht ständig den passenden Punkt eingeben zu müssen, kannst du dir mit den Blöcken 

und 

helfen. Diese speichern die aktuellen Koordinaten deines Hamsters. Möchtest du nun, dass dein Hamster 10 Schritte nach rechts macht, dann verbinde den Block x-Position mit dem Plus-Operator und trage die Zahl 10 in das andere Feld ein. Schon gleitet dein Hamster langsam durch seine Wohnstätte.

Finde nun wieder mithilfe der Koordinatenanzeige heraus wie hoch und wie breit eine Stufe ist! Wenn du das weißt, kannst du deinem Hamster auch mitteilen welche Schritte er zum Überwinden einer Stufe zu gehen hat. Da der Hamster zur Vorratskammer drei Stufen zu überwinden hat, muss er eigentlich drei mal dasselbe machen. Finde einen Block mit dem du sich-wiederholende Anweisungen *steuern* kannst!

5.3.3 Mahlzeit

Wie du bereits im ersten Beispiel festgestellt haben solltest, läuft das Hamstern der Weizenähren automatisch beim Berühren des Weizens ab. Versuche bei diesem Beispiel aber nun nicht wieder mit den Koordinaten herauszufinden wie weit der Hamster bis zur letzten Ähre zu gehen hat, sondern wirf einen Blick in die Kategorie *Fühlen*! Dort befindet sich nämlich ein Block namens wenn berührt? Jetzt muss es dir nur noch gelingen ein Skript zu basteln, dass deinen Hamster so lange in kleinen Schritten zu Weizen 4 gleiten lässt, bis er ihn berührt.

5.3.4 Hamsterliche Bettruhe

Tja, das Bett des Hamsters liegt offensichtlich links von seiner Position auf der Vorratskammer. Um nach links zu gehen, sollte er also erst mal seine Blickrichtung ändern. Jetzt bietet BYOB dir zwar die Möglichkeit Objekte um eine gewisse Anzahl an Grad zu drehen, wenn du dies mit 180 Grad ausprobierst, wirst du jedoch feststellen, dass dein Hamster danach auf dem Kopf

steht. Deswegen bleibt einem in diesem Fall nichts anderes übrig, als das Kostüm des Hamsters zu ändern. In der Kategorie *Aussehen* findest du den passenden Befehl dazu. Wähle nun das passende Kostüm aus und füge den Block deinem Skript hinzu!

Der erste Schritt ist getan, doch jetzt folgt der eigentliche Auftrag für deinen kleinen Schützling, nämlich das Herabschreiten der Stufen. In der letzten Aufgabe solltest du deinem Hamster beigebracht haben Objekte zu erfühlen bzw. sich Objekten anzunähern bis er sie berührt. Das kann er nun auch für die Stufen sehr gut benötigen. Wenn man sich es genau überlegt, muss man nämlich so lange eine Stufe entlang gehen, bis man sie nicht mehr berührt. Danach wechselt man hinunter zur nächsten Stufe und der Prozess beginnt wieder von vorne.

Damit deine Skripte nicht zu lange werden, solltest du außerdem künftig einzelne Handlungen wie das Herabschreiten von Stufen oder das Gleiten zu einem Objekt in eigene Blöcke zusammenfassen. Dies kannst du tun, indem du in der Kategorie *Variablen* unten auf Neuer Block klickst. Es öffnet sich ein Fenster, welches dich auffordert die Art und den Namen des Blocks zu bestimmen. Da es sich in deinem Fall immer um Bewegungen des Hamsters handelt, kannst du eigentlich immer *Bewegung* als Kategorie auswählen. Den Namen kannst du selbstständig wählen, es empfiehlt sich jedoch einen Namen zu wählen, der die Funktion des Blocks widerspiegelt. Also beispielsweise *Vorratskammer hinunter* für den Block zum Herabschreiten der Vorratskammer. Wenn du im Fenster auf OK gedrückt hast, sollte sich ein Skript-Fenster öffnen in das du dann die Blöcke hineinziehen kannst aus denen dein eigener Block bestehen soll.

5.3.5 Ein ganz normaler Hamstertag

Um das Tagesablauf-Skript für deinen Hamster nicht ewig lang werden zu lassen, solltest du wieder darauf achten möglichst viele eigene Blöcke zu kreieren. Um manche deiner Blöcke vielfältiger einzusetzen, kannst du nun auch probieren Parameter in deine Blöcke aufzunehmen. Ein Parameter kann eine Zahl, ein Objekt oder allgemein eine Variable sein, die einem Block mitgegeben und in der Ausführung des Blocks verwendet wird. Hinzufügen kannst du Parameter indem du mit der rechten Maustaste auf einen deiner eigenen Blöcke und anschließend auf b*earbeiten* klickst. Fährst du nun mit der Maus über die Kopfzeile des Blocks, so erkennst du kleine orange Plus-Zeichen. Klickst du an einer Stelle deiner Wahl auf ein solches Zeichen, so kannst du entweder eine weitere Beschriftung oder eine Eingabe in die

Kopfzeile des Blocks aufnehmen. Wählst du eine Eingabe, was einem Parameter entspricht, kannst du mit einem Klick auf den kleinen schwarzen Pfeil auswählen um was für eine Art von Parameter es sich handeln soll. Bevor du auf OK klickst, vergiss nicht dem Parameter einen Namen zu geben, sonst bleibt dein Block unverändert! Der neue Parameter sollte nun orange hinterlegt in der Kopfzeile erscheinen. Willst du, dass der Parameter während der Ausführung des Blocks verwendet wird, dann kannst du ihn aus der Kopfzeile an eine Stelle deiner Wahl ziehen. Achte dabei darauf, dass der Typ des Parameters an dieser auch Stelle verwendet werden darf! Ein Zahl-Parameter kann beispielsweise nicht dort eingefügt werden, wo nach einem Objekt verlangt wird.

5.3.6 Tapetenwechsel

Damit deine eigenen Blöcke beim Wechsel auf ein neues Projekt nicht verloren gehen, klicke in der Objekt-Liste deinen Hamster an, anschließend in der Menü-Zeile auf *Datei* und schlussendlich auf *Objekt exportieren...*! Wähle nun Speicherort und -name und klicke auf OK. Jetzt kannst du in deinem BYOB-Fenster die Datei HamsterbauLeer.ypr öffnen. Um Hindernisse in das neue Hamster-Territorium einzufügen, musst du diese als neue Objekte anlegen. Das machst du, indem du unterhalb der Bühne auf *Objekt aus Datei laden* klickst und anschließend das gewünschte Objekt auswählst! Achte bei dieser Aufgabe darauf, dass du nicht mehr als vier Hindernisse einbaust, da es sonst zu komplex wird!

5.3.7 Zeit zu „hamstern“

Öffne die Datei Hamstern.ypr!

Zu aller erst solltest du dir überlegen wie dein Hamster das Feld abgrasen soll (Schlangenlinien, Spiral-förmig, etc.). Ist das geklärt, beginn damit die Einzelschritte in eigene Blöcke zu fassen! Versuche während der Ausarbeitung vor allem auch zu bedenken wie weit dein Hamster gehen soll und wann er sich dazu entscheiden soll stehen zu bleiben.

5.3.8 Der Eindringling

Wie du ein neues Objekt anlegst, sollte dir ja bereits klar sein. Versuche nun selbstständig herauszufinden wie du die Fellfarbe deines neuen Hamsters ändern kannst! Lege außerdem ein zweites Kostüm mit entgegengesetzter Blickrichtung an und bastle einen Block, der den neuen

Hamster beim Drücken der Leertaste auf seine Ausgangsposition zurückkehren lässt!

5.3.9 Hamsterliebe

Damit sich die beiden Hamster gleichzeitig bewegen, sollte jeder deiner Hamster ein eigenes Skript erhalten. Solltest du dazu neue Blöcke erstellen müssen, dann achte darauf, dass du beim Erstellen *Nur für dieses Objekt* auswählst!

6 Exemplarische Ausarbeitung der Beispiele

In diesem Abschnitt werden zu den einzelnen Programmieraufgaben beispielhafte Lösungen vorgestellt. Dazu sei gesagt, dass diese aber mehr zur Handlungsorientierung als als Musterlösung dienen sollen. Der Lehrer soll dadurch einen besseren Einblick in die praktische Anwendung der jeweiligen Lernumgebung erhalten und seine eigene Praxis dadurch vorantreiben.

6.1 Java Hamster Simulator

6.1.1 Die ersten Gehversuche

Der Hamster soll bei dieser Aufgabe bis vor die Stufen zu seiner Rechten gehen und dort ein Korn aufnehmen.

Ausgangssituation:

Exemplarische Lösung:

```
void main() {
      vor();
      vor();
      nimm();
}
```

Dies ist sicherlich die einfachste Lösung der Problemstellung. Zwischen dem Hamster und der Mauer befinden sich zwei freie Felder, die mit den zwei `vor()`-Befehlen überwunden werden können. Der `nimm()`-Befehl lässt den Hamster das, auf dem zweiten freien Feld befindliche, Korn aufnehmen.

6.1.2 Auf zu höheren Gefilden

Jetzt soll der Hamster die Stufen zur Vorratskammer solange erklimmen bis er an der obersten Stufe mit Blick nach rechts angekommen ist.

Ausgangssituation:

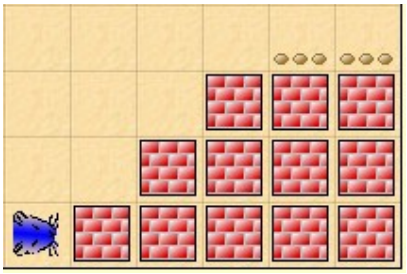


Abbildung 18: Ausgangssituation des Gefilde-Beispiels im Java Hamster-Simulator

Exemplarische Lösung:

```
void main() {
      linksUm();
      vor();
      rechtsUm();
      vor();
      linksUm();
      vor();
      rechtsUm();
      vor();
      linksUm();
      vor();
      rechtsUm();
      vor();
}

void rechtsUm() {
      linksUm();
      linksUm();
      linksUm();
}
```

Es handelt sich hierbei zwar nicht um die schönste Lösung, jedoch liegt das Augenmerk bei dieser Aufgabenstellung ohnehin viel mehr darin erste größere Anweisungsblöcke und eine erste eigene Methode zu gestalten. Man erkennt nämlich auf einen Blick, dass sich in der `main()`-Methode der selbe Anweisungsblock drei mal wiederholt. Dieses `linksUm(); vor();`

`rechtsUm(); vor();` ist es also was den Hamster genau eine Stufe überwinden lässt.

6.1.3 Mahlzeit

Auf der Vorratskammer angekommen, soll der Hamster nun alle Körner, die sich auf den beiden Feldern der Kammer befinden, fressen.

Ausgangssituation:

Abbildung 19: Ausgangssituation des Mahlzeit-Beispiels im Java Hamster-Simulator

Exemplarische Lösung:

```
void main() {
      vor();
      allesNehmen();
      vor();
      allesNehmen();
}

void allesNehmen() {
      while(kornDa()) {
            nimm();
      }
}
```

Auch diese Lösung ist noch nicht perfekt, da sich in der `main()`-Methode abermals die Anweisungen wiederholen. Jedoch liegt der Schwerpunkt auch hier eher dabei erstmals eine Schleife in einer eigens geschriebenen Prozedur einzusetzen. Die effizientere Gestaltung der `main()`-Methode wäre eine Zusatzaufgabe für Fortgeschrittene. Wie dies aussehen könnte wird im Abschnitt 7 Verbesserungs- und Erweiterungsmöglichkeiten gezeigt.

6.1.4 Hamsterliche Bettruhe

Der kleine Hamster soll nun die Stiegen hinab auf sein Bett in der Mitte des Baus klettern und dort eine Ruhepause einlegen. Das Herabsteigen sollte in eine eigene Prozedur verpackt und eine weitere Prozedur für das Hinaufsteigen von Stufen sollte kreiert werden. (Anmerkung: Die Prozedur für das Hinaufsteigen wird erst in der Lösung der nächsten Aufgabe bereitgestellt.)

Ausgangssituation:

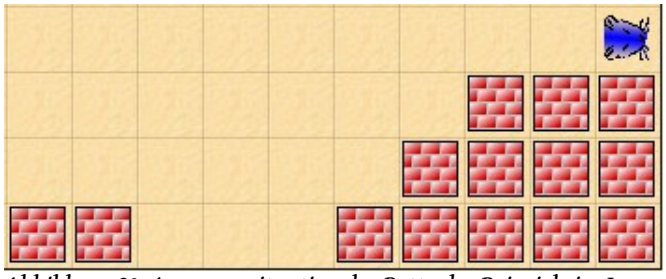


Abbildung 20: Ausgangssituation des Bettruhe-Beispiels im Java Hamster-Simulator

Exemplarische Lösung:

```
void main() {
      umdrehen();
      stufenRunter();
      bettRauf();
}

void bettRauf() {
      rechtsUm();
      vor();
      linksUm();
      vor();
}

void rechtsUm() {
      linksUm();
      linksUm();
      linksUm();
}

void stufenRunter() {
```

```
        while(vornFrei()) {
                linksUm();
                if(vornFrei()) {
                        vor();
                        rechtsUm();
                } else {
                        rechtsUm();
                        vor();
                }
        }
}

void umdrehen() {
        linksUm();
        linksUm();
}
```

Da die Prozeduren `bettRauf()`, `rechtsUm()` und `umdrehen()` nur eine Folge von Befehlen beinhalten und die Funktionalität so auf einen Blick ersichtlich sein sollte, erspare ich mir weitere Erläuterungen dazu. Interessant zu untersuchen ist jedoch sicherlich die Prozedur `stufenRunter()`.

Diese besteht prinzipiell aus einer `while`-Schleife, in der sich eine `if-else`-Anweisung befindet. Die `while`-Schleife wird solange ausgeführt, bis das Feld vor dem Hamster nicht mehr frei ist. Ist das Feld frei, so dreht sich der Hamster zuerst einmal nach links. Da der Hamster nach Ausführung der Prozedur `umdrehen()` nach links blickt, wird er nach dem ersten Befehl des Schleifenrumpfes hinunter auf die Mauern der Stufen blicken. Dort testet er mithilfe der bedingten Anweisung (`if(vornFrei())`), ob er nun weitergehen kann oder nicht. Kann er nicht weitergehen (`else`), so wird er sich noch auf der Stiege befinden und sich daher wieder nach rechts drehen (`rechtsUm()`) und einen Schritt vor gehen (`vor()`). Kann er erstmals weitergehen (`if(vornFrei())`), so wird er die Stiege soeben verlassen haben und wird deswegen zuerst einen Schritt nach vorne (`vor()`) machen und sich dann nach rechts drehen (`rechtsUm()`). Diesen Test, ob er Boden unter den Pfoten hat oder nicht, macht der Hamster eben solange bis er, bei Blickrichtung nach links, eine Mauer vor sich hat. Dies ist erst der Fall, wenn er vor seinem Bett angekommen ist.

6.1.5 Ein ganz normaler Hamstertag

Bei dieser Herausforderung soll der Hamster vom Bett auf die Vorratskammer zwei Körner fressen gehen, anschließend die fünf Stufen auf der linken Seite erklimmen, wieder zurück in die Vorratskammer drei Körner fressen und schlussendlich zurück ins Bett gehen.

Ausgangssituation:

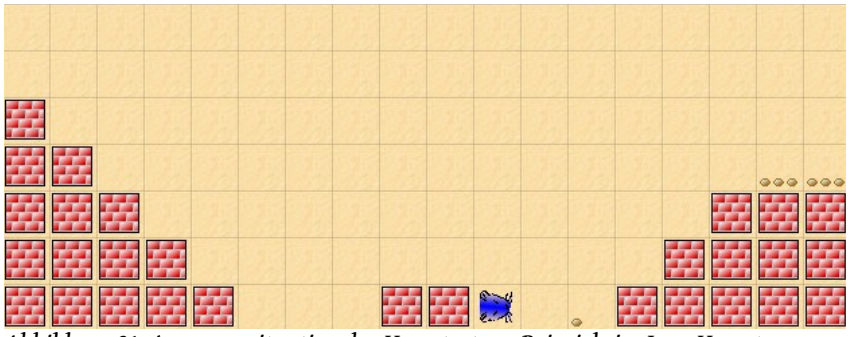


Abbildung 21: Ausgangssituation des Hamstertags-Beispiels im Java Hamster-Simulator

Exemplarische Lösung:

```
void main() {
      vorwaertsGehen();
      stufenRaufRechts();
      vor();
      nimm2();
      umdrehen();
      stufenRunterLinks();
      aufBettLinks();
      stufenRunterLinks();
      stufenRaufLinks();
      stufenRunterRechts();
      aufBettRechts();
      stufenRunterRechts();
      stufenRaufRechts();
      vor();
      nimm();
      vor();
      nimm2();
      umdrehen();
      stufenRunterLinks();
```

```
	aufBettLinks();
}

void aufBettLinks() {
	rechtsUm();
	vor();
	linksUm();
	vor();
}

void aufBettRechts() {
	linksUm();
	vor();
	rechtsUm();
	vor();
}

void nimm2() {
	nimm();
	nimm();
}

void rechtsUm() {
	linksUm();
	linksUm();
	linksUm();
}

void stufenRaufLinks() {
	while(!vornFrei()) {
		rechtsUm();
		vor();
		linksUm();
		if(vornFrei()) {
			vor();
		} else {
			linksUm();
			vor();
			linksUm();
			break;
		}
```

```
	}
}

void stufenRaufRechts() {
	while(!vornFrei()) {
		linksUm();
		vor();
		rechtsUm();
		vor();
	}
}

void stufenRunterLinks() {
	while(vornFrei()) {
		linksUm();
		if(vornFrei()) {
			vor();
			rechtsUm();
		} else {
			rechtsUm();
			vor();
		}
	}
}

void stufenRunterRechts() {
	while(vornFrei()) {
		rechtsUm();
		if(vornFrei()) {
			vor();
			linksUm();
		} else {
			linksUm();
			vor();
		}
	}
}

void umdrehen() {
	linksUm();
	linksUm();
```

```
}

void vorwaertsGehen() {
      while(vornFrei()) {
            vor();
      }
}
```

Die Lösung mag auf den ersten Blick sehr lange aussehen, schließlich handelt es sich dabei auch um über 100 Codezeilen, jedoch beziehe ich mich hierbei ausschließlich auf die, in den vorherigen Aufgaben gestalteten und eingesetzten, Methoden. Das soll heißen, dass sich die Entwicklung und Produktion von neuem Code eigentlich in Grenzen hält und es viel mehr gefragt ist eine Adaption bzw. Anwendung des bisher Gelernten zu erreichen. Eine Besonderheit dabei ist sicherlich, dass ich mich dazu entschieden habe für die Blickrichtungen links und rechts jeweils eigene Prozeduren zu schreiben. Dies verdoppelt zwar wiederum die Anzahl an Codezeilen, jedoch halte ich es für eine gute Übung, um ein Gefühl für den Einsatz von Methoden zu bekommen. Voraussetzung dafür wäre, dass dem Schüler klar wird, dass sich die Prozeduren `stufenRunterLinks()` und `stufenRunterRechts()` nur dadurch unterscheiden, dass die Drehungen des Hamsters in die Gegenrichtung stattfinden.

Zusätzliche Beachtung gilt es darüber hinaus der Prozedur `stufenRaufLinks()` zu schenken, da sie als einzige eine `break`-Anweisung im `else`-Zweig beinhaltet. Aufgrund dessen, dass ich die beiden `stufenRauf`-Prozeduren nämlich so konzipiert habe, dass sie so lange ausgeführt werden bis der Hamster kein Hindernis mehr vor sich hat (`while(!vornFrei())`), würde sie im Falle der Stufen auf der linken Seite ohne `break`-Anweisung eine Fehlermeldung liefern. Das liegt daran, dass sich nach der obersten Stufe, nicht wie auf der rechten Stiege, ein freies Feld, sondern die Territoriumsgrenze befindet. Diese interpretiert der Hamster als weitere Stufe und daher würde die `while`-Schleife erst terminieren wenn der Hamster über die Territoriumsgrenzen hinaus schreiten will. Die `else`-Verzweigung verhindert dies indem der Hamster wieder auf die letzte Stufe zurückkehrt und dort mittels `break` die Schleife und damit die gesamte Prozedur beendet wird.

6.1.6 Tapetenwechsel

Da die Schüler bei dieser Aufgabe angehalten sind eigene Territorien für den Hamster zu kreieren, werde ich dazu selbstverständlich keine beispielhafte Lösung vorstellen.

6.1.7 Zeit zu „hamstern“

Bei seinem ersten Ausflug in die freie Natur hat der Hamster nun die Aufgabe sämtliche Körner auf einem eingegrenzten Feldabschnitt aufzunehmen. Wie bereits in Punkt C1.5.7 erwähnt, hat bei diesem Beispiel die Standard-Version des Java Hamster-Simulators seine Premiere. Dieser Umstand führt dazu, dass erstmals die, optisch attraktivere, 3D-Simulation gestartet werden kann.

Ausgangssituation:

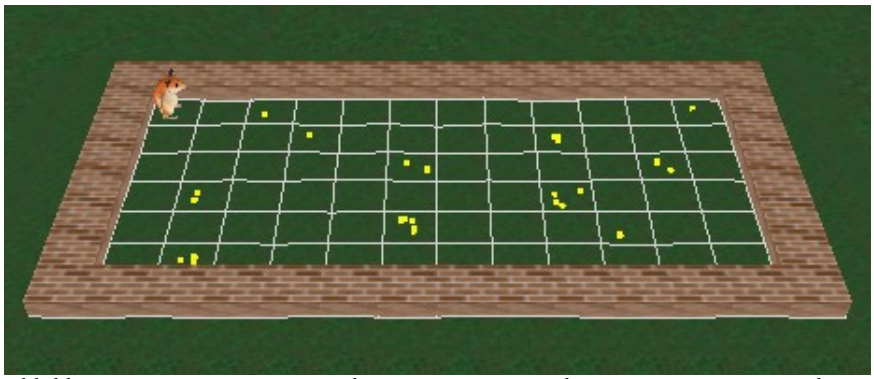


Abbildung 22: Ausgangssituation des Hamstern-Beispiels im Java Hamster-Simulator

Exemplarische Lösung:

```
Hamster tapsi = Hamster.getStandardHamster();

void main() {
	reiheAufnehmenUndZurueck();
	while(nochEineReiheVorhanden()) {
		geheInDieNaechsteReihe();
		reiheAufnehmenUndZurueck();
	}
}

void allesNehmen() {
	while(tapsi.kornDa()) {
		tapsi.nimm();
	}
}
```

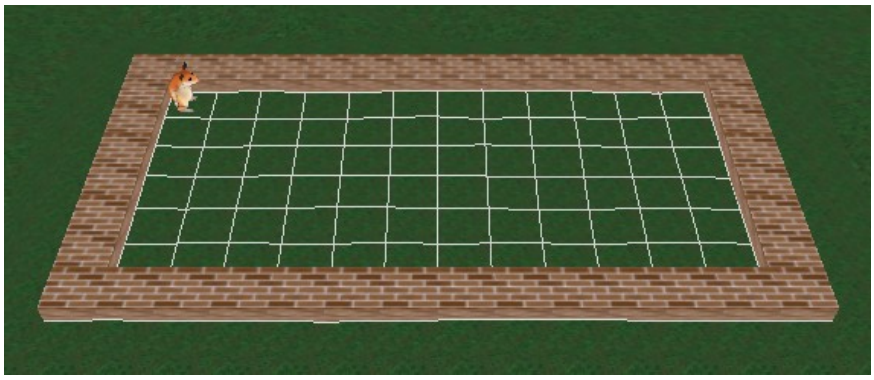
```
void geheInDieNaechsteReihe() {
	tapsi.vor();
	tapsi.linksUm();
}

boolean nochEineReiheVorhanden() {
	tapsi.linksUm();
	return tapsi.vornFrei();
}

void rechtsUm() {
	tapsi.linksUm();
	tapsi.linksUm();
	tapsi.linksUm();
}

void reiheAufnehmenUndZurueck() {
	vorwaertsUndNehmen();
	umdrehen();
	zurueckGehen();
}

void umdrehen() {
	tapsi.linksUm();
	tapsi.linksUm();
}

void vorwaertsUndNehmen() {
	allesNehmen();
	while(tapsi.vornFrei()) {
		tapsi.vor();
		allesNehmen();
	}
}

void zurueckGehen() {
	while(tapsi.vornFrei()) {
		tapsi.vor();
	}
}
```

Zu aller erst wird bei dieser Lösung eine globale Variable vom Typ `Hamster` mit dem Namen `tapsi` angelegt und ihr danach der Standard-Hamster zugewiesen. So kann im weiteren Verlauf des Programms der Standard-Hamster mit seinem Variablennamen `tapsi` angesprochen werden. Die Prozeduren in der main-Methode sollten sich anhand ihrer Namen eigentlich von selbst erklären. So nimmt der Hamster mit der Prozedur `reiheAufnehmenUndZurueck()` alle Körner in einer Reihe auf und kehrt dann zum ersten Feld der Reihe zurück. Die zweite Prozedur `geheInDieNaechsteReihe()` veranlasst den Hamster sich in die nächste Reihe zu begeben und sich in Blickrichtung nach rechts zu orientieren. Wirklich Neues gibt es erst bei der Methode `nochEineReiheVorhanden()`, welche einen `boolean`-Wert zurückliefert. Während des Ablaufs der Methode dreht sich der Hamster in Blickrichtung nach unten (`linksUm()`) und kontrolliert dann, ob der Weg nach unten frei ist (`tapsi.vornFrei()`). Das Ergebnis dieser Abfrage wird mit dem `return`-Befehl zurückgeliefert. Also `true`, wenn der Weg frei ist und `false`, wenn dem nicht der Fall ist. Das dieses Ergebnis nun als Bedingung für den Durchlauf der `while`-Schleife dient, ist ebenso eine Neuerung dieses Beispiels. Wenn man bedenkt, dass eine `boolean`-Methode nur `true` oder `false` zurückliefert und eine Bedingung diese Information ausreicht, um als überprüft zu gelten, dann sollte auch dieser Umstand keine allzu großen Verständnisschwierigkeiten bedeuten.

6.1.8 Der Eindringling

Hier soll ein weiterer Hamster erstellt und an den Rand des Feldes gesetzt werden.

Ausgangssituation:



Exemplarische Lösung:

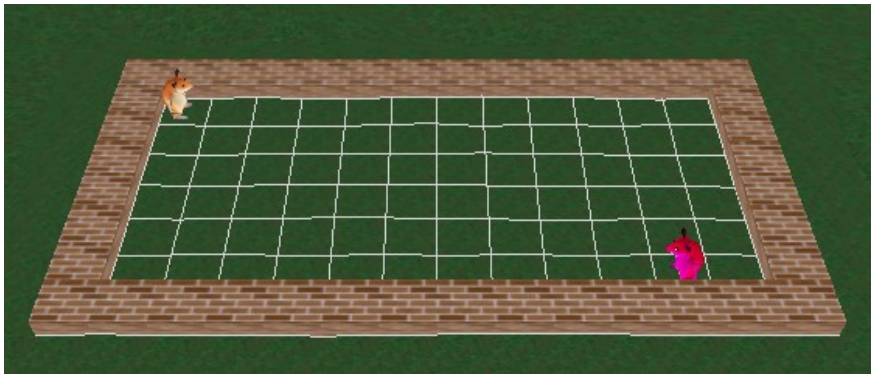
```
Hamster hanni = new Hamster(6, 11, Hamster.WEST, 0, 5);
```

Auch wenn sich die Code-Produktion diesmal in Grenzen hält, so steckt doch so manch bedeutende Erkenntnis in dieser Zeile. Abermals wird in diesem Beispiel ein Hamster-Variable erzeugt, diesmal wird ihr aber nicht der Standard-Hamster, sondern ein neuer Hamster zugewiesen. Dieser Hamster wird mit dem fünf Parameter umfassenden Konstruktor `Hamster()` erzeugt. Die ersten zwei Parameter stehen hierbei für die Position des Hamsters, wobei der erste die Reihennummer und der zweite die Spaltennummer angibt. Der dritte Parameter steht für die Blickrichtung des Hamsters, der vierte für die Anzahl der Körner im Maul und der letzte für die Farbe des Hamsters.

6.1.9 Hamsterliebe

Die beiden Hamster sollen sich bei dieser Aufgabe eine wilde Verfolgungsjagd liefern. Der Standard-Hamster soll dabei aber doppelt so schnell sein wie der neue Hamster.

Ausgangssituation:



Exemplarische Lösung:

```
Hamster tapsi = Hamster.getStandardHamster();
Hamster hanni = new Hamster(6, 11, Hamster.WEST, 0, 5);

void main() {
      while(hamsterNichtAmSelbenFeld()) {
            hanniVor();
            tapsiVor();
      }
}

boolean hamsterNichtAmSelbenFeld() {
      if(tapsi.getReihe() == hanni.getReihe() &&
      tapsi.getSpalte() == hanni.getSpalte()) {
            return false;
      } else {
            return true;
      }
}

void hanniRechtsUm() {
```

```
        hanni.linksUm();
        hanni.linksUm();
        hanni.linksUm();
    }

    void hanniVor() {
        if(hanni.vornFrei()) {
            hanni.vor();
        } else {
            hanniRechtsUm();
            hanni.vor();
        }
    }

    void tapsiRechtsUm() {
        tapsi.linksUm();
        tapsi.linksUm();
        tapsi.linksUm();
    }

    void tapsiVor() {
        for(int i = 0; i < 2; i++) {
            if(tapsi.vornFrei()) {
                tapsi.vor();
            } else {
                tapsiRechtsUm();
                tapsi.vor();
            }
        }
    }
```

Die `while`-Schleife in der `main()`-Methode wird so lange ausgeführt bis sich die Hamster auf dem selben Feld befinden. Während der Ausführung bewegen sich die beiden Hamster mit den Prozeduren `hanniVor()` und `tapsiVor()` abwechselnd vorwärts. Die Hamster befinden sich auf dem selben Feld wenn sowohl die Reihen- als auch die Spaltennummer ihres Aufenthaltsorts übereinstimmt, wobei diese Überprüfung in der Methode `hamsterNichtAmSelbenFeld()` geschieht. Die Methode liefert `false`, wenn sich die Hamster am selben Feld aufhalten und `true`, wenn nicht.

Schließlich noch zur Erklärung der beiden `Vor()`-Prozeduren. Die Hamster-Dame `hanni` bewegt

sich nur unter der Voraussetzung, dass ihr Weg frei ist, nach vorne. Anderenfalls dreht sie sich nach rechts und geht dann einen Schritt vor. Unser Schützling `tapsi` muss diese Überprüfung, mithilfe der `for`-Schleife, zwei mal durchführen, da er sich ja doppelt so schnell wie sein weibliches Pendant fortbewegt.

6.2 BYOB

6.2.1 Die ersten Gehversuche

Der Hamster soll seine ersten Schritte zur Stiege auf der rechten Seite machen.

Ausgangssituation:

Abbildung 25: Ausgangssituation des Gehversuche-Beispiels in BYOB

Exemplarische Lösung:

Abbildung 26: Skript für das Gleiten zur Vorratskammer

Eine sehr simple Methode diese Aufgabe zu lösen, ist es zu erruieren auf welcher x-Koordinate sich der Hamster und die Weizenähre vor der Stiege befinden. Die Differenz der beiden Koordinaten hat der Hamster anschließend zu überwinden.

Da sich der Hamster dabei aber so schnell bewegt, dass es beinahe schon an Teleportation grenzt, gibt es eine weitere Variante, mit der man die Bewegung etwas entschleunigen kann.

6.2.2 Auf zu höheren Gefilden

Auch in BYOB soll der Hamster nun die Stiege zur Vorratskammer erklimmen.

Ausgangssituation:

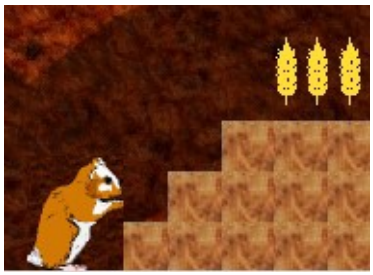


Abbildung 28: Ausgangssituation des Gefilde-Beispiels in BYOB

Exemplarische Lösung:

Abbildung 29: Skript für den Ablauf des Gefilde-Beispiels

Dabei werden die zwei blauen Bewegungsblöcke mithilfe einer simplen Wiederholungsschleife drei mal ausgeführt. In den Bewegungsblöcken kommen zusätzlich Variablen und Operatoren zum Einsatz. Die Variablen *x-Position* und *y-Position* speichern, wie der Name bereits vermuten lässt, die aktuellen Koordinaten des Hamsters und aktualisieren den Wert laufend. Ermittelt man nun die Höhe bzw. die Länge einer Stufe in Koordinateneinheiten (hier: 26), so kann man es durch eine einfache Addition erreichen, dass der Hamster eine Stufe erklimmt. Der erste

Block ist dabei für das Hochsteigen, der zweite für das Entlanggehen einer Stufe verantwortlich.

Wie ich später zeigen werde, können die x- und y-Position des Hamsters auch mit den Blöcken *ändere x um...* und *ändere y um...* um einen konstanten Wert verändert werden. Jedoch bietet die oben vorgestellte Variante den Vorteil die Schüler bereits in einem frühen Lernstadium mit den Konzepten der Variablen und Operatoren zu konfrontieren.

6.2.3 Mahlzeit

Oben auf der Stiege angekommen, soll der Hamster nun die gesamten Vorräte verputzen.

Ausgangssituation:

Abbildung 30: Ausgangssituation des Mahlzeit-Beispiels in BYOB

Exemplarische Lösung:

Abbildung 31: Skript zur Lösung des Mahlzeit-Beispiels

Hier kommt nun erstmals eine bedingte Anweisung ins Spiel und zwar um genau zu sein, eine bedingte Wiederholungsanweisung. Dies sollte eigentlich nichts Unbekanntes mehr darstellen, denn in Java entspricht dies beispielsweise der `while`-Schleife. Es gibt bei dieser Schleife jedoch auch Unterschiede zur klassischen `while`-Schleife in Java. So findet die Überprüfung der Bedingung nicht am Beginn eins Schleifendurchlaufs, sondern erst am Ende eines solchen statt. Dies würde in java eigentlich der `do-while`-Schleife entsprechen. Hinzu kommt, dass, um einen

weiteren Durchlauf zu starten, die Auswertung der Bedingung nicht `true` ergeben muss, sondern `false`. Dies entspräche einer `do-until`-Schleife, welche in Java jedoch nicht existiert. Sobald also nun in unserem Beispiel der Hamster die letzte Weizenähre berührt, hört er damit auf in x-Richtung zu *gleiten*.

6.2.4 Hamsterliche Bettruhe

Der Hamster soll nun die Stiegen hinunter steigen und danach auf seinem Bett eine kleine Rast einlegen.

Ausgangssituation:

Abbildung 32: Ausgangssituation des Bettruhe-Beispiels in BYOB

Exemplarische Lösung:

Abbildung 33: Skript für den Ablauf des Bettruhe-Beispiels

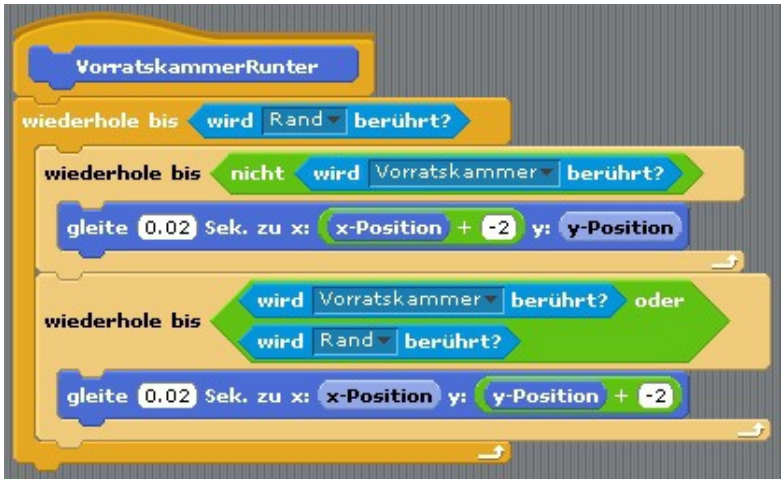


Abbildung 35: Block zum Hinaufsteigens des Betts in negativer x-Richtung

Da es in BYOB nicht möglich ist, ein Objekt mit einem Befehl umdrehen zu lassen (ohne, dass es danach auf dem Kopf steht), muss man sich eines kleinen Tricks behelfen. Man erstellt ein zweites Kostüm, das so gespiegelt wird, dass es in die entgegengesetzte Richtung blickt. Will man nun, dass sich der Akteur auf der Bühne umdreht, so zieht man ihm einfach das alternative Kostüm an (*ziehe Kostüm hamsterLinks an*). Um nun das Herabschreiten der Stufen möglichst allgemein ablaufen zu lassen, sollte der Hamster Kontakt zur Vorratskammer haben. Dazu dient der Block mit der Wiederholungsanweisung (*wiederhole bis wird Vorratskammer berührt?*).

Ist dies erreicht, startet die Ausführung der Prozedur *VorratskammerRunter*. Hat der Hamster nämlich Kontakt zur Vorratskammer, so kann man ihm auftragen so lange nach links zu gehen (*x-Position - 2*) bis er keinen Kontakt mehr zur Kammer hat (*wiederhole bis nicht wird Vorratskammer berührt?*). Dann befindet er sich genau über der Stufe darunter und kann anschließend in y-Richtung (*y-Position – 2*) hinunter zu dieser Stufe gleiten (*wiederhole bis wird Vorratskammer berührt?*). Dies wiederholt er solange bis er schlussendlich den Rand der Bühne

berührt (*wiederhole bis wird Rand berührt?*) und damit am Fußpunkt der Vorratskammer angelangt ist.

Es folgt die Ausführung der Prozedur `BettRauf`. Dabei gleitet der Hamster solange bis er das Bett berührt (*wiederhole bis wird Bett berührt?*), begibt sich auf die Höhe des Bettes (*wiederhole bis nicht wird Bett berührt?*) und gleitet schließlich auf das Bett (*gleite 1 Sek. zu x: -8 y: -125*).

6.2.5 Ein ganz normaler Hamstertag

Auch in BYOB sollen die Schüler nun den Tag des Hamsters mit einem Programm simulieren. Der Hamster soll dabei die Stufen zur Vorratskammer erklimmen und eine Weizenähre fressen. Danach soll er die fünf Stufen auf der linken Seite hochsteigen, abermals auf die Vorratskammer, um die beiden übrigen Ähren zu fressen und abschließend auf sein Bett steigen.

Ausgangssituation:

Abbildung 36: Ausgangssituation des Hamstertag-Beispiels in BYOB

Exemplarische Lösung:

Auch in BYOB ergibt sich bei dieser Aufgabe eine sehr lange Abfolge von Befehlen. Wobei es sich zu einem Großteil um eigens erstellte Prozeduren handelt, die ich in weiterer Folge etwas genauer untersuchen möchte.

Abbildung 38: Block zum Hinaufsteigen der Vorratskammer in positiver x-Richtung

Ich habe vorhin bereits eine Möglichkeit vorgestellt wie der Hamster die Stufen zur Vorratskammer überwinden kann. Voraussetzung dafür war jedoch, dass man die Höhe und die Länge der Stufen wissen musste. Dies ist hier nun nicht mehr der Fall. Der Hamster *erfühlt* nämlich selbst wie lang und wie hoch eine Stufe ist. Diese Variante würde daher auch funktionieren, wenn sich die Länge und Höhe der einzelnen Stufen unterscheiden würde.

Abbildung 39: Block für das Gleiten zu einem beliebigem Objekt in positiver x-Richtung

Zum ersten mal kommt hierbei in BYOB eine Prozedur mit Parameter zum Einsatz. Bei dem Parameter handelt es sich um einen Objekt-Parameter, der der Schleife als Abbruchbedingung dient. Sobald das Objekt vom Akteur berührt wird, terminiert die Schleife. Bis dahin nähert sich der Akteur in positiver x-Richtung an das Objekt an.

Abbildung 40: Block zum Gleiten in negativer y-Richtung bis beliebiges Objekt berührt wird

Im letzten Beispiel wurde schon erwähnt, dass der automatische Ablauf mancher Prozeduren leichter fällt, wenn der Akteur Kontakt zu den Objekten hat. Dies ermöglicht diese Prozedur.

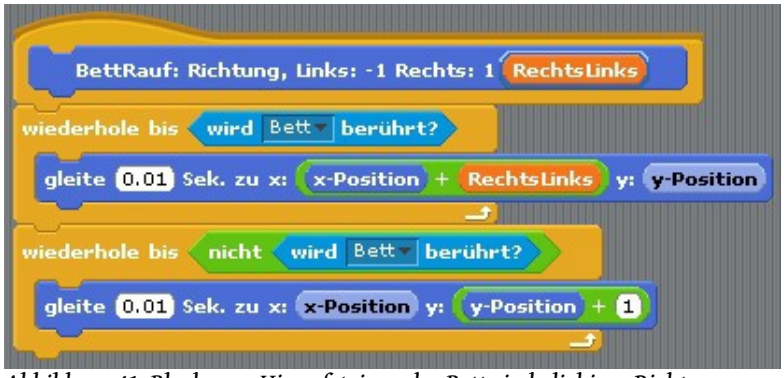


Abbildung 41: Block zum Hinaufsteigen des Betts in beliebiger Richtung

Wie es im Kapitel Fehler: Referenz nicht gefunden ersichtlich ist, muss man im Java-Hamster-Simulator beim Hinauf- oder Herabsteigen von Hindernissen für jede Blickrichtung eine eigene Methode bzw. Prozedur schreiben. Dies ist in BYOB nicht mehr von Nöten. Die *BettRauf*-Prozedur aus der Lösung des vorherigen Beispiels kann nämlich mithilfe eines Parameters so adaptiert werden, dass der Parameter die Ausführungsrichtung beeinflusst. Übergibt man der Prozedur die Zahl 1, so bewegt sich der Akteur nach rechts. Übergibt man die Zahl -1, so gleitet er nach links.

Abbildung 42: Block zum Herabsteigen des Betts in beliebiger Richtung

Auch in dieser Prozedur kommt der Parameter mit dem selben Zweck zum Einsatz. Der einzige Unterschied zwischen den beiden Prozeduren liegt darin, dass die Abbruchbedingungen der Schleifen vertauscht sind. Sobald der Hamster das Bett also nicht mehr *erfühlen* kann, hört er auf sich in x-Richtung zu bewegen und gleitet darauffolgend vom Bett hinab bis zum Rand der Bühne.

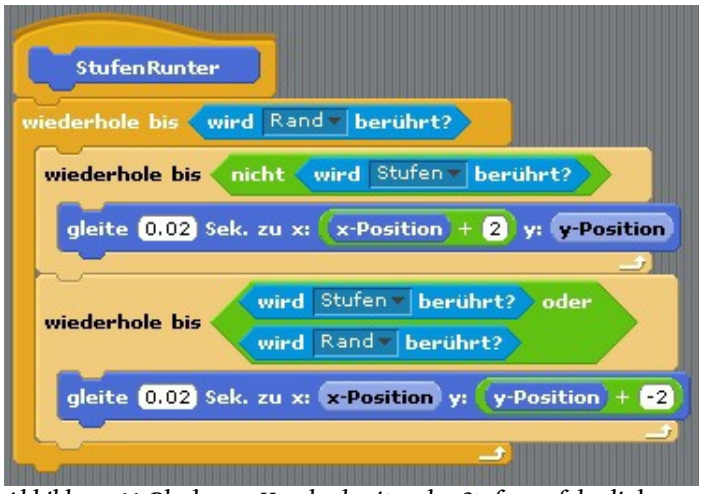


Abbildung 44: Block zum Herabschreiten der Stufen auf der linken Seite des Hamsterbaus

Das Prinzip der *StufenRauf*- bzw. der *StufenRunter*-Prozedur ist exakt das selbe wie bei den *VorratskammerRauf*- bzw. *VorratskammerRunter*-Prozeduren. Die einzigen Unterschiede sind, dass sich das Objekt der Abbruchbedingung verändert und die Anzahl der Stufen, was sich in der *StufenRauf*-Prozedur unter *wiederhole 5 mal* zeigt.

Abbildung 45: Block für das Gleiten zum Rand in negativer x-Richtung

Zuguterletzt die Prozedur, die den Hamster, oben auf der Stiege auf der linken Seite angekommen, zum Rand gleiten lässt. Wie im Ablauf-Programm ersichtlich, befindet sich danach der Block *pralle vom Rand ab*. Dieser ist notwendig, da die folgende *StufenRunter*-

Prozedur als Abbruchbedingung *wird Rand berührt?* enthält. Würde man also den Hamster bis zum Rand gleiten lassen und ihm dann die Prozedur *StufenRunter* befehlen, so würde er nichts tun, da die Bedingung *wird Rand berührt?* bereits erfüllt wäre.

6.2.6 Tapetenwechsel

Auch hier wird keine beispielhafte Lösung vorgestellt.

6.2.7 Zeit zu „hamstern“

Bei dieser Aufgabe soll der Hamster auf einem eingezäunten Feld sämtliche Vorräte plündern.

Ausgangssituation:

Abbildung 46: Ausgangssituation des Hamstern-Beispiels in BYOB

Exemplarische Lösung:

Der Ablauf des Hauptprogramms ist dabei sehr ähnlich zu dem des Java-Hamster-Simulators. Die Ausführung wird beispielsweise erst unterbrochen, wenn der untere Rand der Bühne berührt wird (*wird Rand berührt?*) bzw. keine weitere Reihe mehr vorhanden ist. Bis dies so weit ist, gleitet der Hamster eine Reihe entlang und wechselt, am Rand angekommen, in die nächste Reihe. Als Reihe ist dabei immer ein Feldabschnitt, dessen Höhe der Größe des Hamsters in Koordinaten entspricht, anzusehen. Zu beachten gilt es hierbei vor allem den Befehl den Hamster am rechten Rand um 180 Grad zu drehen. Dies ist nötig, da sich infolge des *pralle vom Rand ab*-Blocks der Hamster am rechten Rand (und zwar ausschließlich am rechten Rand) immer auf den Kopf dreht.

Abbildung 48: Block für das Gleiten zum Rand in beliebiger Richtung mit anschließendem Abprallen vom Rand

Dieser Prozedur kann man, genau wie im vorherigen Beispiel, wieder die Ausführungsrichtung als Parameter mitgeben. Übergibt man dabei den Wert -1, so bewegt sich der Hamster nach links, übergibt man 1, nach rechts.

In dieser Prozedur kommt zum ersten mal eine Zählvariable (hier: *i*) zum Einsatz. Die Ausführung der Schleife findet dabei entweder so lange statt bis die Variable *i* den Wert 14 hat oder der untere Rand der Bühne berührt wird. Bei genauerer Betrachtung sieht man, dass der Schleifenrumpf dabei maximal 15 mal (für die Werte von 0 bis 14) ausgeführt wird. Diese Ausführungszahl rührt daher, dass die Größe des Hamsters ca. 60 Koordinateneinheiten entspricht. Bei Schritten der Größe 4 überwindet der Hamster dabei also genau seine eigene Größe.

6.2.8 Der Eindringling

Ein neuer Hamster soll auf der Bühne platziert werden. Damit man die beiden Hamster dann auch optisch unterscheiden kann, soll der neue Hamster eine andere Fellfarbe erhalten.

Ausgangssituation:

Abbildung 50: Ausgangssituation des Eindringling-Beispiels in BYOB

Exemplarische Lösung:

Die einfachste Möglichkeit einen neuen Hamster zu erstellen, ist sicherlich den vorhandenen Hamster zu kopieren. Dies geschieht indem man unterhalb der Bühne den Hamster per Rechtsklick auswählt und dann auf *Duplizieren* klickt. Es erscheint ein neuer Hamster, der exakt wie der bisherige aussieht. Um dies zu ändern, klickt man den neuen Hamster unterhalb der Bühne an und wechselt dann auf die Karteikarte *Kostüme*. Klickt man nun bei einem Kostüm auf *Bearbeiten*, so erscheint ein kleines Bildbearbeitungsfenster, welches es erlaubt die Kostümfarben zu ändern.

6.2.9 Hamsterliebe

Die Aufgabe besteht darin die beiden Hamster auf eine Verfolgungsjagd entlang des Bühnenrandes zu schicken. Dabei soll der neue Hamster sich nur halb so schnell wie der alte bewegen.

Ausgangssituation:

Abbildung 51: Ausgangssituation des Verfolgungs-Beispiels in BYOB

Exemplarische Lösung:

Dies sind die beiden Hauptprogramme, wie sie dem alten (links) und dem neuen Hamster (rechts) mitgegeben werden. Erstmals rechtfertigt sich hierbei entscheidend der Einsatz der grünen Fahne als Startsignal, da die beiden Programme so parallel ablaufen können. Im Gegensatz zum Java-Hamster-Simulator muss man den beiden Hamstern also nicht nacheinander sequentielle Befehle geben, sondern kann dies in BYOB gleichzeitig tun.

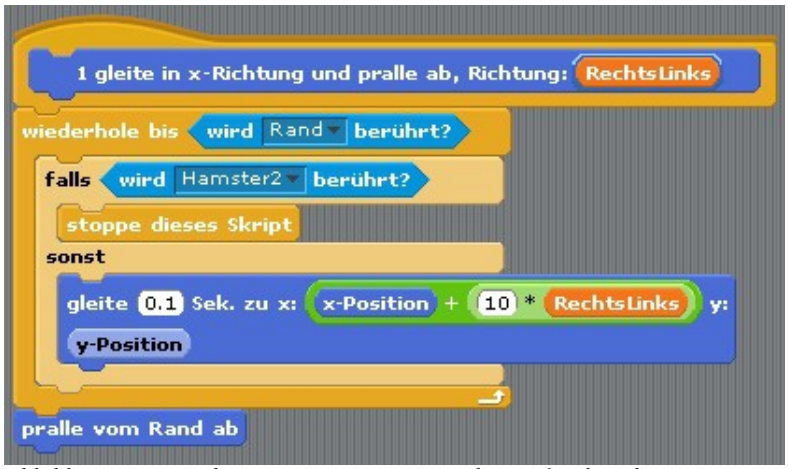
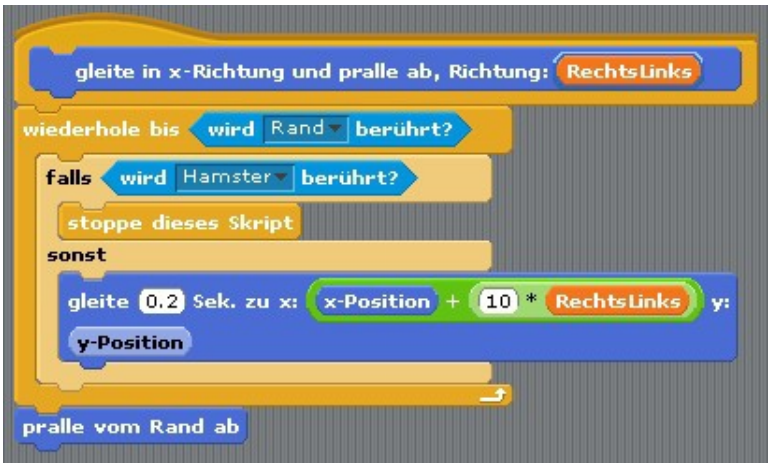


Abbildung 54: Prozedur zur Bewegung in x-Richtung für den alten Hamster



Die beiden Prozeduren unterscheiden sich auf den ersten Blick nur minimal. Einerseits gibt es, je nach Hamster, Unterschiede in der Bedingung der if-else- bzw. falls-sonst-Anweisung, andererseits ist die Ausführungszeit des gleit-Blockes beim alten Hamster, aufgrund seiner höheren Geschwindigkeit, kürzer als beim Neuen. Auf die Funktionsweise bezogen, sollten diese beiden Prozeduren jedoch keine sonderlichen Neuerungen darstellen. Einzig der Block *stoppe dieses Skript* ist bisher noch nicht in Erscheinung getreten. Dieser ist mit der `break`-Anweisung in Java zu vergleichen. Jedoch stoppt der Block nicht nur die Ausführung der Schleife, sondern auch das gesamte Skript, in der der Block eingebunden ist. Er ist hier notwendig um ein Berühren der beiden Hamster überhaupt erst zu ermöglichen. Die Einbindung der Abfrage *wird Hamster berührt?* mit einer oder-Verknüpfung in die Schleifen-Bedingung würde das Programm nämlich nur in manchen Fällen richtig terminieren lassen. Das liegt daran, dass die Überprüfung der Bedingung immer erst am Ende der Schleife passiert. Wird nun also eine Prozedur, aufgrund der Erfüllung der Bedingung, verlassen, so wird im Hauptskript gegebenenfalls in die nächste Prozedur gewechselt, die wiederum zuerst eine Bewegung ausführen und dann die Bedingung überprüfen würde. Dadurch kann es sein, dass sich die Hamster wieder voneinander entfernen und eines oder beide Skripte weiter ausgeführt werden.

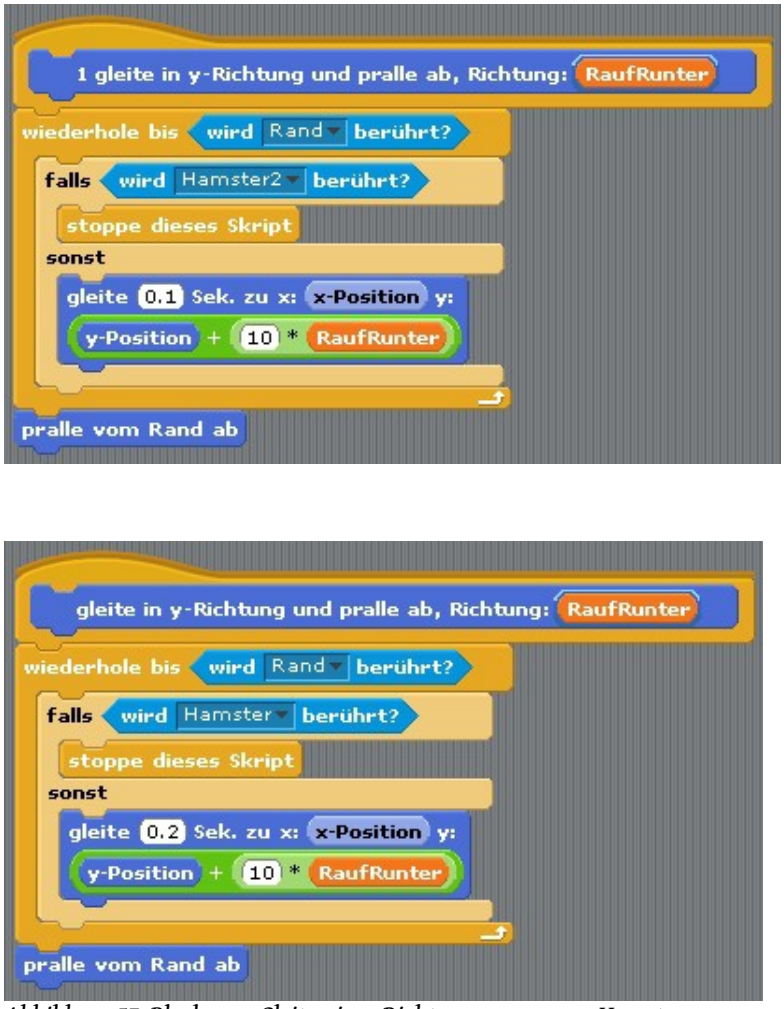


Abbildung 57: Block zum Gleiten in y-Richtung vom neuen Hamster

Diese zwei Prozeduren sollten funktional eigentlich keine Fragen mehr offen lassen. Die beiden Hamster bewegen sich, solange der jeweils andere Hamster oder der Rand der Bühne nicht berührt wird, zehn Schritte in y-Richtung.

7 Verbesserungs- und Erweiterungsmöglichkeiten

In diesem Abschnitt werden nun noch alternative Varianten der soeben vorgestellten Lösungen präsentiert. Dies soll speziell dazu dienen die Vielfältigkeit von Problemlösungen beim Programmieren exemplarisch aufzuzeigen und kann im Unterricht unterstützend für Schüler mit vergleichsweise hohem Lernfortschritt eingesetzt werden.

7.1 Java-Hamster-Simulator

7.1.1 Die ersten Gehversuche

Um die Lösung dieser Aufgabe etwas allgemeiner zu formulieren, bietet es sich an anstelle einer vorgegebenen Anzahl an `vor()`-Befehlen, diese unter der Bedingung einer while-Schleife ausführen zu lassen.

```
void main() {
      vorwaertsGehen();
      nimm();
}

void vorwaertsGehen() {
      while(vornFrei()) {
            vor();
      }
}
```

7.1.2 Auf zu höheren Gefilden

Auch bei diesem Beispiel gibt es eine bessere, effizientere und vor allem allgemeinere Variante, die in einer späteren Lösung zum Einsatz gekommen ist. Die im Punkt 6.1.2 vorgestellte Befehlssequenz in der `main()`-Methode, kann nämlich durch die `stufenRaufRechts()`-Prozedur des Tages-Ablaufs-Programms ersetzt werden.

```
void stufenRaufRechts() {
      while(!vornFrei()) {
            linksUm();
            vor();
            rechtsUm();
            vor();
      }
}
```

Die Sequenz aus der exemplarischen Lösung wird nun nicht mehr drei mal in Folge niedergeschrieben, sondern mithilfe einer `while`-Schleife wiederholt ausgeführt. Dies führt speziell dazu, dass die Anzahl der überwindbaren Stufen mit der Prozedur jetzt nicht nur auf

drei beschränkt ist, sondern sich damit beliebig viele Stufen meistern lassen.

7.1.3 Mahlzeit

Eine weitere Möglichkeit dieses Beispiel zu lösen wäre mithilfe der, im Punkt 6.1.7 Zeit zu „hamstern“ vorgestellten, Prozedur `vorwaertsUndNehmen()`.

```
void vorwaertsUndNehmen() {
      allesNehmen();
      while(tapsi.vornFrei()) {
            tapsi.vor();
            allesNehmen();
      }
}
```

Diese ist zwar etwas komplexer als die exemplarische Lösung, jedoch funktioniert sie allgemein für jeden Fall, in dem beliebig viele Körner auf einer geraden Strecke aufzunehmen sind.

7.1.4 Hamsterliche Bettruhe

Aufgrund dessen, dass sich die Verbesserungsmöglichkeiten des in Punkt 6.1.4 Programms auf das Zusammenfassen von mehreren Methoden zu einer einzigen beschränken, werde ich hier nicht genauer auf diese Aufgabe eingehen. Natürlich ist es in vielen der Beispiele möglich Befehlssequenzen in eigene Methoden zu verpacken, jedoch sehe ich die Sinnhaftigkeit davon in Frage gestellt, wenn dabei weder eine bessere Nachvollziehbarkeit für den Schüler noch eine effizientere oder gar kürze Programm-Version entsteht.

7.1.5 Ein ganz normaler Hamstertag

Besonders das Aufnehmen einer bestimmten Anzahl von Körnern lässt sich beim, in Punkt 6.1.5 angeführten, Programm sicherlich noch verbessern. So kann man eine Methode schreiben, der man per Parameter die Anzahl der aufzunehmenden Körner mitgeben kann. Eine mögliche Realisierung könnte so aussehen:

```
void nimm(int anzahl) {
      for(int i = 0; i < anzahl; i++) {
```

```
            nimm();
        }
    }
```

Außerdem kann man die Prozeduren `bettRaufRechts()` und `bettRaufLinks()` durch die Prozeduren `stufenRaufRechts()` bzw. `stufenRaufLinks()` ersetzen. Da diese nämlich terminieren wenn der Weg nach dem Erklimmen einer Stufe frei ist, kann man sie auch zum Aufstieg ins Bett, welches als einzelne Stufe anzusehen ist, verwenden. Dadurch ergibt sich in der `main()`-Methode wiederum eine sich-wiederholende Folge an `rauf-` und `runter-`Prozeduren, welche deswegen in eigenen Prozeduren in Schleifen verpackt werden können:

```
    void runterRaufLinks(int wiederholungen) {
        for(int i = 0; i < wiederholungen; i++) {
            stufenRunterLinks();
            stufenRaufLinks();
        }
    }

    void runterRaufRechts(int wiederholungen) {
        for(int i = 0; i < wiederholungen; i++) {
            stufenRunterRechts();
            stufenRaufRechts();
        }
    }
```

Die abgewandelte `main()`-Methode sieht nun folgendermaßen aus:

```
    void main() {
        vorwaertsGehen();
        stufenRaufRechts();
        vor();
        nimm(2);
        umdrehen();
        runterRaufLinks(2);
        runterRaufRechts(2);
        vor();
        nimm();
        vor();
```

```
        nimm(2);
        umdrehen();
        runterRaufLinks(1);
}
```

7.1.6 Tapetenwechsel

Da es zu diesem Punkt keine exemplarische Lösung gibt, werde ich selbstverständlich auch keine Verbesserungsmöglichkeiten vorstellen.

7.1.7 Zeit zu „hamstern“

Anstatt den Hamster jedes mal wieder an den Anfang der Reihe zurücklaufen zu lassen, kann man auch eine elegantere Variante wählen bei der der Hamster quasi in Schlangenlinien über das Feld pflügt. Kommt er also am Ende einer Reihe an, so soll er sich nach unten drehen, testen ob noch eine Reihe existiert und wenn ja, dann soll er in die nächste Reihe gehen und sich so drehen, dass er die nächste Reihe abgrasen kann.

```
Hamster tapsi = Hamster.getStandardHamster();

void main() {
        while(tapsi.vornFrei()) {
                vorwaertsUndNehmen();
                dreheNachUnten();
        }
}

void allesNehmen() {
        while(tapsi.kornDa()) {
                tapsi.nimm();
        }
}

void dreheNachUnten() {
        int blickrichtung = tapsi.getBlickrichtung();
        if(blickrichtung == 1) {
                rechtsUm();
                geheInNaechsteReiheUndDrehe(blickrichtung);
        } else if(blickrichtung == 3) {
```

```
            tapsi.linksUm();
            geheInNaechsteReiheUndDrehe(blickrichtung);
      }
}

void geheInNaechsteReiheUndDrehe(int blickrichtung) {
      if(tapsi.vornFrei() && blickrichtung == 1) {
            tapsi.vor();
            rechtsUm();
      } else if(tapsi.vornFrei() && blickrichtung == 3) {
            tapsi.vor();
            tapsi.linksUm();
      }
}

void rechtsUm() {
      tapsi.linksUm();
      tapsi.linksUm();
      tapsi.linksUm();
}

void vorwaertsUndNehmen() {
      allesNehmen();
      while(tapsi.vornFrei()) {
            tapsi.vor();
            allesNehmen();
      }
}
```

Die beiden Prozeduren, die hier neu dazu gekommen sind, sind `dreheNachUnten()` und `geheInNaechsteReiheUndDrehe(int blickrichtung)`. Eine besondere Herausforderung, um die Läuffähigkeit des Programms zu gewährleisten, war es die Blickrichtung des Hamsters vor der Drehung nach unten zu speichern. Dies habe ich schließlich erreicht, indem ich die Blickrichtung in einer Variable in der erst-genannten Prozedur abgespeichert habe und diese dann der zweit-genannten Prozedur als Parameter übergeben habe. Bildlich gesprochen kann man behaupten, dass ich dem Hamster so ein Kurzzeit-Gedächtnis verpasst habe.

Der Ablauf eines Durchlaufs sieht dann folgendermaßen aus:

- `vorwaertsUndNehmen()`: Der Hamster geht eine Reihe ab und nimmt dabei alle Körner auf.

- `dreheNachUnten()`: Der Hamster ist am Ende der Reihe angekommen und fragt sich nun in welche Richtung er blickt:
  - `if(blickrichtung == 1)`: Blickt er nach rechts, so muss er sich nach rechts drehen um darauffolgend auf die nächste Reihe zu blicken.
  - `if(blickrichtung == 3)`: Blickt er nach links, so muss er sich nach links drehen.
- `geheInNaechsteReiheUndDrehe(int blickrichtung)`: Der Hamster erinnert sich, in welche Richtung er vor dem nach-unten-Drehen geblickt hat und entscheidet nun anhand dessen wie er sich zu verhalten hat.
  - `if(tapsi.vornFrei() && blickrichtung == 1)`: Ist der Weg nach vorne frei und hat er vorhin nach rechts geblickt, so geht er einen Schritt vor und dreht sich anschließend wieder nach rechts.
  - `if(tapsi.vornFrei() && blickrichtung == 3)`: Ist der Weg nach vorne frei, hat der Hamster aber vorhin nach links geblickt, so geht er ebenso einen Schritt vor und dreht sich nach links.

7.1.8 Der Eindringling

Sollte die Erstellung und Initialisierung in einem einzigen Schritt für die Schüler zu schnell gehen (siehe Punkt 6.1.8), so könnte man die einzelnen Schritte etwas aufteilen:

```
Hamster hanni;

void main() {
      hanni = new Hamster();
      hanni.init(6, 11, Hamster.WEST, 0, 5);
}
```

Mit `Hamster hanni;` wird nun zuerst eine Hamster-Variable mit dem Namen `hanni` deklariert. Dieser Variable wird im nächsten Schritt ein neu-erzeugter, aber noch nicht initialisierter, Hamster zugeordnet (`hanni = new Hamster();`). Erst der letzte Schritt (`hanni.init(...)`) initialisiert den Hamster.

7.1.9 Hamsterliebe

Da sich weitere, von mir getestete, Lösungsvarianten als umständlicher oder ineffizienter als

die oben gezeigte Lösung erwiesen, werde ich bei bei dieser Aufgabenstellung auf die Vorstellung alternativer Lösungswege verzichten.

7.2 BYOB

7.2.1 Die ersten Gehversuche

Um nicht extra ausmessen zu müssen wieviele Koordinaten der Hamster zu überwinden hat, würde es sich natürlich auch schon bei der ersten Aufgabe empfehlen den *wird ... berührt?*-Block der Kategorie *Fühlen* einzusetzen.

Wie ich es bereits angekündigt habe, kann die Änderung der Koordinaten auch alternativ mit dem *ändere x um ...*-Block erfolgen.

7.2.2 Auf zu höheren Gefilden

Um die Lösung dieser Aufgabenstellung auch für ähnliche Probleme verwenden zu können, bietet es sich an eine Prozedur zu gestalten, die auf sämtliche Stufen-Objekte anwendbar ist. Hier eine Möglichkeit dies zu bewerkstelligen:

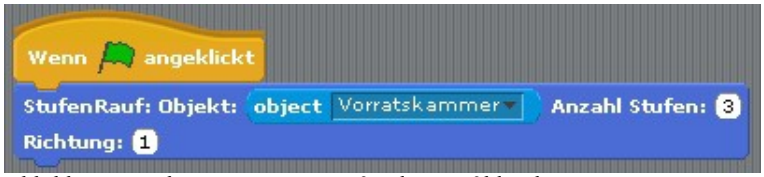


Abbildung 59: Alternativer Ansatz für das "Gefilde-Skript"

Wie im obigen Hauptskript erkennbar, kann man der Prozedur das zu überwindende Objekt, die Stufenanzahl und die Richtung, in der die Stufen überquert werden sollen, als Parameter übergeben. Da bei diesem Beispiel die Stufen zur Vorratskammer überwunden werden sollen, wird als Objekt die Vorratskammer, als Stufenanzahl drei und als Richtung eins (was wieder für rechts steht) übermittelt.

Die Prozedur selbst enthält keine algorithmische Weiterentwicklung, außer, dass die Koordinaten abermals mit der in Punkt 7.2.1 vorgestellten Variante verändert werden.

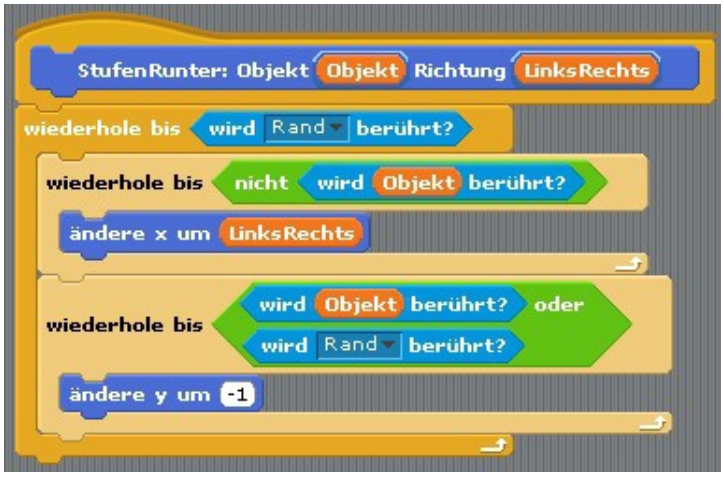
7.2.3 Mahlzeit

In diesem Fall beschränken sich die Variationsmöglichkeiten ausschließlich auf die alternative Änderung der Koordinaten.

7.2.4 Hamsterliche Bettruhe

Abbildung 61: Alternativer Ansatz für das "Bettruhe-Skript"

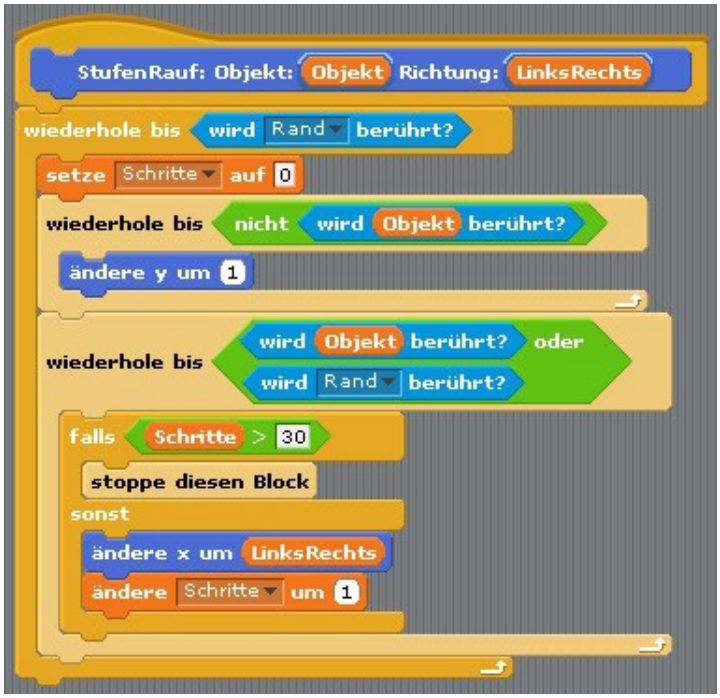
Auch hier gibt es die Möglichkeit die Lösung etwas allgemeiner zu formulieren. Das Haupt-Skript wird dadurch zwar sogar ein bisschen länger, jedoch mithilfe der Möglichkeit jeder Prozedur ein Objekt und eine Richtung mitzugeben, auch flexibler einsetzbar.



Die grundlegende Funktionsweise der Prozedur *StufenRunter* wurde in einer ähnlichen Form bereits in Punkt 6.2.5 vorgestellt. Da hat es sich aber noch um eine parameterfreie Prozedur gehandelt. Mit dieser Prozedur können nun sowohl das zu überwindende Objekt als auch die Ausführungsrichtung mitgegeben werden.

Abbildung 63: Block zum Gleiten zu einem beliebigen Objekt in beliebiger Richtung

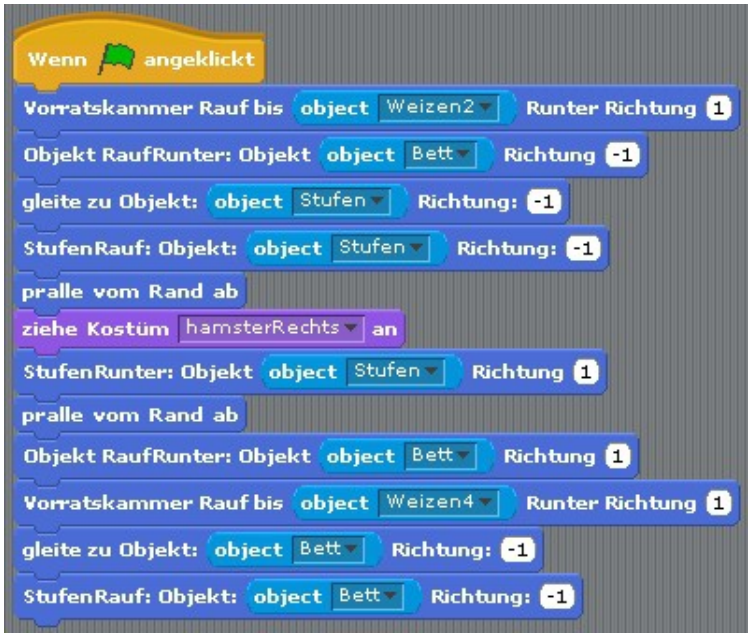
Auch diese Prozedur sollte bereits bekannt sein. Zusätzlich zum Objekt kann hierbei nun auch die x-Richtung der Ausführung mitgegeben werden.



Bei der Lösung von Aufgabe 6.2.4 war das Gleiten zur Endposition noch kein automatisierter Prozess, sondern mit der Angabe der Koordinaten dieser Position verbunden. Dass dies auch eleganter zu lösen ist, zeigt die obere Prozedur. Dazu habe ich die Variable *Schritte* deklariert. Diese wird zu Beginn der Prozedur auf 0 gesetzt und kommt schließlich in der Schleife, die die Bewegung in x-Richtung realisiert, als Bedingung zum Einsatz. So wird vor jedem Schritt in x-Richtung überprüft, ob die Variable einen Wert größer als 30 hat. Ist dies nicht der Fall, so wird die Variable um 1 erhöht. Dadurch soll verhindert werden, dass, wenn die oberste Stufe einer Stiege erreicht ist, der Hamster sich so lange in x-Richtung bewegt bis er den Rand der Bühne berührt. Nach jeder erfolgreich überwundenen Stufe erhält die Variable *Schritte* wieder den Wert 0 und der Prozess beginnt von vorne.

7.2.5 Ein ganz normaler Hamstertag

Mit den bisher entwickelten Prozeduren lässt sich die Lösung von Punkt 6.2.5 noch etwas umgestalten.



Prinzipiell besteht dieses Skript nur noch aus den drei Bewegungsprozeduren *StufenRauf*, *StufenRunter* und *gleite zu Objekt*. Um sich wiederholende Ereignisse nicht mehrfach anzuschreiben, habe ich zusätzlich die Prozeduren *Vorratskammer Rauf bis Objekt Runter* und *Objekt RaufRunter* erstellt.

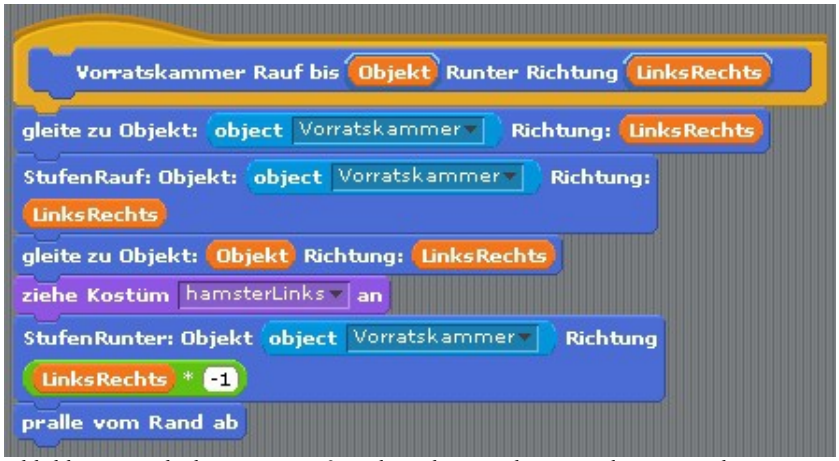


Abbildung 66: Block zum Herauf- und wieder Herabsteigen der Vorratskammer bis zu einem beliebigen Objekt in beliebiger Richtung

Dem Objekt-Parameter soll dabei die Weizenähre mitgegeben werden, die der Hamster sich, auf der Vorratskammer angekommen, einverleiben soll.

Abbildung 67: Block zum Überwinden eines beliebigen Objekts in beliebiger Richtung

Diese Prozedur ist speziell für die Fälle gemacht, wenn ein Objekt dem Hamster den Weg versperrt. Mit der Prozedur *Objekt RaufRunter* kann also ein quadratisches oder rechteckiges Hindernis beliebiger Größe überwunden werden.

7.2.6 Tapetenwechsel

Erneut wird zu dieser Problemstellung keine beispielhafte Lösung präsentiert.

7.2.7 Zeit zu „hamstern“

Ein Schönheitsfehler der Lösung aus Punkt 6.2.7 ist sicherlich der, dass der Schleifenrumpf eine Bewegung zum gegenüberliegenden Bühnenrand, in die nächste Reihe und wieder zum gegenüberliegenden Bühnenrand beinhält. Besser wäre es, wenn die Bedingung der Schleife nach jeder Bewegung in x-Richtung überprüft wird. Dies ermöglicht der Einsatz einer weiteren Variable.

Abbildung 68: Alternativer Ansatz für das "Hamstern-Skript“

Zu Beginn des Skripts wird die Variable *Richtung* auf 1 gesetzt, was einer Bewegung nach rechts

entspricht. Anschließend beginnt die Ausführung des Schleifenrumpfes mit der Bewegung des Hamsters zum gegenüberliegenden Bühnenrand. Ist der Hamster am Rand angekommen, so wird mit den beiden bedingten Anweisungen überprüft in welche Richtung der Hamster unterwegs war. War er nach rechts unterwegs, so soll er darauffolgend das Kostüm *hamsterLinks* anziehen (und sich wieder um 180 Grad drehen), war er jedoch nach links unterwegs, so das Kostüm *hamsterRechts*. Ist der Kostümwechsel vollzogen, gleitet der Hamster in die nächste Reihe. Schlussendlich wird die Variable *Richtung* mit -1 multipliziert und das Ergebnis als neuer Wert zugewiesen, was beim nächsten Schleifendurchlauf einem Richtungswechsel entspricht.

7.2.8 Der Eindringling

Die in Punkt 6.2.8 vorgestellte Variante ist sicherlich die einfachste und effizienteste diese Aufgabe zu lösen.

7.2.9 Hamsterliebe

Eine sehr kurze und effiziente Möglichkeit diese Aufgabe zu lösen, ist es die beiden Hamster nach jeder Rand-Berührung um 90 Grad zu drehen und anschließend wieder bis zur nächsten Rand-Berührung laufen zu lassen.

Abbildung 69: Alternativer Ansatz für das "Verfolgungs-Skript" von Hamster 1

Abbildung 70: Alternativer Ansatz für das "Verfolgungs-Skript" von Hamster 2

Es sei jedoch erwähnt, dass diese Lösung rein auf Funktionalität und Effizienz ausgelegt ist. In

diesem Fall muss man darüber hinweg sehen, dass sich die beiden Hamster phasenweise auf dem Kopf stehend fortbewegen. Da es dabei aber nicht einmal der Erstellung eigener Prozeduren bedarf, lässt sich bereits vermuten wie effizient das Skript ist.

Ein interessantes Detail in diesem Skript ist der *gehe -20* bzw. *20-er Schritt.* Wie man erkennen kann, ist die Richtung dieses 20-er Schritts jeweils entgegen der Laufrichtung in der Schleife darüber. Es handelt sich also nur um das Zurückprallen vom Rand, um ein Weiterkommen zu ermöglichen. Wählt man nämlich nur den klassischen *pralle vom Rand ab*-Block, so berühren die Hamster nach der Drehung um 90 Grad abermals den Rand und die Hamster würden immer wieder eine Reihe hin- und zurücklaufen.

E Zusammenfassung

Wie in der Arbeit gezeigt, geht es beim Programmieren nicht nur um die Erstellung von Computer-Programmen, sondern auch um Denk- und Handlungsprozesse, die an der Erstellung beteiligt sind. Programmieren bedeutet vor allem aber auch *kreativ sein*. So bieten die, in der Arbeit vorgestellten, Lösungen nur einen Auszug von zahlreichen Lösungs-Varianten der Aufgaben. Diese wiederum decken beispielhaft lediglich einen Bruchteil der Möglichkeiten im Einsatz der Mikrowelten ab, was verdeutlicht wie umfassend gearbeitet werden kann und welches Kreativitäts-Potential sich durch diese Bandbreite ausleben lässt. Nicht umsonst meint der Informatiker Michael Kölling daher: „In computer science, writing programs ist the most direct way of creating things. We can make new things out of nothing." (Kölling 2010, S. 34).

Genau hier muss, meiner Meinung nach, auch beim modernen Informatik-Unterricht angesetzt werden, da dieser wegführen soll von einer Beschränkung auf die Anwender-Ebene, hin zu einer offenen Form, die auch Freiheiten zur persönlichen Gestaltung lässt. Anstatt Schüler mit den Worten: „Heute habe ich gelernt wie ich etwas anwende, was jemand anderer gestaltet hat" aus dem Unterricht gehen zu lassen, sollte der Fokus darauf gelegt werden, dass die Schüler selbst gestalterische Rollen einnehmen. Sie sollen nach einer Unterrichtseinheit viel mehr sagen: „Heute habe *ich* etwas gestaltet, was jemand anderer anwenden kann". Nur so kann eine Generation moderner *PC-Heimwerker* geschaffen werden, die auch interessiert daran ist neue Dinge zu kreieren.

Alles was man als Informatik-Lehrer dafür tun muss, ist es den Schülern mit den richtigen Methoden die nötigen Werkzeuge näher zu bringen und gleichzeitig damit ihr Interesse für das Programmieren zu wecken. So habe ich in dieser Arbeit den Ansatz verfolgt die Schüler mit Problemstellungen, also Herausforderungen, die nicht ohne Denkprozesse zu meistern sind, zu konfrontieren, um kreative Prozesse anzustoßen. Bevor aber eifrig ans Problemlösen gegangen werden kann, werden den Schülern dabei Strategien zum Lösen von Problemen vorgestellt. Wie gezeigt, beinhalten die Strategien beispielsweise das Erfassen des Problems, das Zerlegen des Problems in Teilprobleme, die Spezifizierung des Ist- und des Soll-Zustands, das darauf-basierende Design einer Lösung und das Programmieren, das Testen und gegebenenfalls das Präsentieren dieser Lösung.

Speziell beim Punkt *Design* bedarf es dabei aber wieder zusätzlicher Informationen, schließlich brauchen die Schüler, um Probleme eigenständig lösen zu können, eine Ahnung davon, welche grundlegenden Konzepte von Programmiersprachen überhaupt unterstützt werden. Für den Lehrer ist dabei von imminenter Bedeutung welche Konzepte erfahrungsgemäß die größten

Schwierigkeiten für Programmier-Anfänger darstellen. Die Ergebnisse, die die Arbeit im Hinblick auf diese Problematik liefert, stehen häufig im Zusammenhang mit fehlerhaften Vorstellungen von Funktionsweisen verschiedener Grundkonzepte, wie bedingten Anweisungen und Schleifen. Darüber hinaus birgt das objektorientierte Modell ebenso Potential für Unklarheiten, da hier vor allem der Begriff des Objekts missverstanden wird. Um genau für solche Situationen gewappnet zu sein bzw. um sie gar nicht erst entstehen zu lassen, soll der Lehrer Gegenmaßnahmen, wie das Zerlegen von Programmen in Einzelkomponenten oder die Ausführung von Programmen in Einzelschritten, im Unterricht einleiten. Außerdem werden neue Befehle und Konzepte mit dem Modell *Sinn und Zweck – Syntax – Anwendung* vorgestellt, was das Verständnis zusätzlich stärken soll.
Um den Lernprozess weiter anzuregen, den Erkenntnisgewinn zu erhöhen und vor allem das Interesse der Schüler zu wecken, bietet diese Arbeit den Ansatz Programmieren mithilfe von *Storytelling* zu unterstützen. Die Festlegung auf das sogenannte *goal-based scenario* bietet den idealen Rahmen eine Geschichte mit Lernerfolgen zu versehen und zu gestalten. Um die Identifikation mit der Geschichte zu erhöhen, bekommen die Schüler zu aller erst eine Rolle und einen Auftrag zugeteilt. Die Lernenden werden so zu einem Teil der Geschichte und sind in weiterer Folge entscheidend am Fortlauf derselben beteiligt.

Entscheidende Erkenntnis liefert natürlich auch die Gegenüberstellung der beiden Lernumgebungen. Dabei zeigt sich der Java Hamster-Simulator als deutlich umfangreicher in der Möglichkeit seines Einsatzes, da er zusätzlich noch zahlreiche andere Programmiersprachen wie Python, Ruby oder Scratch unterstützt. BYOB hingegen ist spezialisierter und daher beschränkt auf den Einsatz der gegeben Programmstrukturen, die jedoch auch durch eigene Strukturen erweitert werden können. Auch unterscheiden sich die beiden Mikrowelten in ihrer prinzipiellen Auslegung. BYOB setzt voll und ganz auf visuelle Programmierung. Der Java Hamster-Simulator hingegen auf die Vereinfachung klassischer textueller Programmierung. Hier obliegt es dem Lehrer selbst zu entscheiden welchen Ansatz er für besser hält, um erste Konzepte des Programmierens kennenzulernen. Sicherlich besitzen beide Umgebungen ihre Vor- und Nachteile. Beispielsweise ist es als positiv anzusehen, dass die Blöcke in BYOB in Form einer kleinen Datenbank verfügbar und außerdem in deutscher Sprache formuliert sind. So liegt es, meiner Meinung nach, immerhin nahe, dass ihr Einsatz für die Schüler als intuitiver wahrgenommen wird, weil die Bezeichnungen die Funktion vorwegnehmen. Der Java Hamster-Simulator bietet wiederum einen sofortigen Einstieg in die

textuelle Programmierung und das vor allem indem die Befehlswelt auf ein Minimum reduziert wird. Was in weiterer Folge den Fokus mehr auf Steuerungselemente wie Variablen, bedingte Anweisungen und Schleifen legt.
So kann man zusammenfassend sagen, dass es mit BYOB von Beginn an möglich ist frei und eigenständig zu programmieren, wohingegen Anfänger beim Arbeiten mit dem Java Hamster-Simulator ab der ersten Minute ein gewisse Form der Führung und Anleitung brauchen, was schlussendlich auch aus den Infotexten dieser Arbeit herausgeht. Letzten Endes hängt der Einsatz der Mikrowelt aber auch stark von dem Vorwissen der Schüler und des Lehrers ab.

Auf Grundlage der Arbeit stellt sich nun die Frage wie die Umsetzung des vorgestellten Szenarios schlussendlich im Unterricht stattfinden soll. Dem Lehrer obliegt es hier wieder selbst, ob die Aufgaben den Schülern beispielsweise in Form eines Projekts oder offenen Lernens zur Verfügung gestellt werden sollen. Darüber hinaus kann diskutiert werden welche Beispiele vielleicht teilweise oder vollständig mit dem Lehrer ausgearbeitet werden sollen. Dabei lässt sich zudem natürlich darüber diskutieren, ob bei der Ausarbeitung noch mehr oder weniger die beim Punkt *Grundkonzepte der Programmierung* eingegangenen Konzepte berücksichtigt werden sollen. Auch bezüglich der Lernform sind noch Fragen offen. Legt man die Ausarbeitung vielleicht sogar prinzipiell als Gruppenarbeit aus, nur in Einzelfällen oder gar nicht? Oder wie genau soll die Vermittlung der Infotexte stattfinden? Sollen sie in .pdf-Format gepackt werden oder in eine E-Learning-Plattform eingebunden werden? Fragen über Fragen, die schließlich nur die Praxis im Unterricht selbst beantworten kann und dem Lehrer die Möglichkeit bieten sollen im Rahmen dieser Anleitung einen eigenen Weg der Vermittlung von Grundkonzepten der Progammierung zu finden.

Egal wie die Umsetzung schlussendlich aussieht, ein Ziel sollte jeder Programmier-Unterricht überall auf diesem Planeten haben: „Let kids program in the classroom, let them be creative, let them have fun!“ (Kölling 2010, S. 34).

Abbildungsverzeichnis

F Literaturverzeichnis

Balzert, Helmut: Lehrbuch Grundlagen der Informatik, Konzepte und Notationen in UML, Java und C++, Algorithmik und Software-Technik, Anwendungen. Heidelberg; Berlin 1999.

Boles, Dietrich: Programmieren spielend gelernt mit dem Java-Hamster-Modell. Wiesbaden 2008.

Bredella, Lothar: Die Rolle von Emotionen bei der Rezeption von Geschichten. in: Duxa, Susanne / Hu, Adelheid / Schmenk Barbara (Hrsg.): Grenzen überschreiten, Menschen, Sprachen, Kulturen. Tübingen 2005, S. 225-234

Domagk, Steffi / Niegemann, Helmut M.: Emotion, Narration und Lernen mit interaktiven Lernangeboten. in: Giessen, Hans (Hrsg.): Emotionale Intelligenz in der Schule, Unterrichten mit Geschichten. Weinheim und Basel 2009, S. 39-62.

Dörner, Dietrich: Problemlösen als Informationsverarbeitung. Stuttgart 1987.

Duncker, Karl: Zur Psychologie des produktiven Denkens. Berlin, 1935.

Ebrahimi, Alireza: Novice programmer errors: language constructs and plan composition. In: International Journal of Human-Computer Studies 41, September 1994. S. 457-480.

Esslinger-Hinz, Ilona / Sliwka, Anne: Schulpädagogik. Weinheim und Basel 2011.

Giessen, Hans: Emotion und Narration – neue Medien, neue Formen des Lernens. in: Giessen, Hans (Hrsg.): Emotionale Intelligenz in der Schule, Unterrichten mit Geschichten. Weinheim und Basel 2009, S. 7-18.

Ginat, David / Menashe, Eti / Taya, Amal: Novice Difficulties with Interleaved Pattern Composition. In: Diethelm, Ira / Mittermeir, Roland T. (Hrsg.): Informatics in Schools. Sustainable Informatics Education for Pupils of all Ages, Oldenburg 2013, S. 57-66.

Hamster-Handbuch – http://www-is.informatik.uni-oldenburg.de/~dibo/hamster/download/v28/handbuch.pdf, Zugriff: 08.10.12 – 15:20.

Hamster-Handbuch light – http://www-is.informatik.uni-oldenburg.de/~dibo/hamster/download/light/v20/handbuch.htm#_Toc270423040, Zugriff: 06.0313 – 11:31.

Hamster-Simulator – http://www-is.informatik.uni-oldenburg.de/~dibo/hamster/simulator.html, Zugriff: 08.10.12 – 14:50.

Hanselmann, Ulla: Ein bisschen Word, in: Die Zeit, 19.02.2009, Nr. 9; Online Ausgabe: http://www.zeit.de/2009/09/C-Informatik, Zugriff: 05.03.13 – 13:20.

Hattenhauer, Rainer: Informatik für Schule und Ausbildung. München 2010.

Hornecker, Eva: Programmieren als Handwerkszeug im ersten Semester, In: Claus, Volker (Hrsg): Informatik und Ausbildung, GI-Fachtagung '98, 30.3.-1.4.1998, Stuttgart.

Informatik Aktuell, Berlin Heidelberg 1998, S. 43-51.

Hromkovic, Juraj: Einführung in die Programmierung mit LOGO. Lehrbuch für Unterricht und Selbststudium. Wiesbaden 2012.

Kölling, Michael: The Secret of Programming Education, Creating Motivation. In: Brandhofer, Gerhard / Futschek, Gerald / Micheuz, Peter / Reiter, Anton / Schoder, Karl (Hrsg.): 25 Jahre Schulinformatik, Zukunft mit Herkunft. Wien 2010, S. 33-34.

Lehrplan Informatik – http://www.bmukk.gv.at/medienpool/11866/lp_neu_ahs_14.pdf, Zugriff: 04.02.13 – 09:12.

Lehrplan Wahlfach Informatik – http://bmukk.gv.at/medienpool/11876/lp_neu_ahs_21.pdf, Zugriff: 11.02.13 – 10:19.

Lobnig, Tanja: Probleme von Programmieranfänger/inne/n. http://www.ddi.edu.tum.de/fileadmin/tueds10/www/material/Lehre/Seminare/BestOf/08-KL-Lobnig-Programmieranfaenger.pdf, Version von 2008, Zugriff: 06.04.2013 – 12:30.

Logofatu, Doina: Grundlegende Algorithmen mit Java. Wiesbaden 2008.

Müller Heinrich / Weichert Frank: Vorkurs Informatik, Der Einstieg ins Informatikstudium. Wiesbaden 2005.

Preiwisch-Seibold, Elvira: Das Erlernen einer Programmiersprache im Wirtschaftsinformatikunterricht unter besonderer Berücksichtigung wissenspsychologischer Theorien, dargestellt am Beispiel der prozeduralen Sprache COBOL. München 1999.

Reichert, Raimond / Nievergelt, Jürg / Hartmann, Werner: Programmieren mit Kara: Ein spielerischer Zugang zur Informatik. Berlin Heidelberg 2005.

Reinhardt, Ingo: Storytelling in der Pädagogik, Eine Einführung in die Arbeit mit Geschichten. Stuttgart 2003.

Schank, Roger C.: Dynamic Memory Revisited. Cambridge 1999.

Schank, Roger C. / Berman, Tamara. R. / Macpherson, Kimberli A.. Learning by doing. In: Charles M. Reigeluth (Hrsg.), *Instructional-design - theories and models: Vol. 2, A new paradigm of instructional theory*. Mahwah, NJ 1999, S. 161-181.

Schöning, Uwe: Ideen der Informatik. Grundlegende Modelle und Konzepte, München 2006.

Schubert, Sigrid / Schwill, Andreas: Didaktik der Informatik. Heidelberg 2011.

Scratch-Website: http://scratch.mit.edu/, Zugriff: 09.03.2013 – 11:13.

Snap!-Website: http://snap.berkeley.edu/, Zugriff: 09.03.2013 – 11:04.

Soloway, Elliot I. / Spohrer, James C.: Novice Mistakes: Are the Folk Wisdoms correct? In: Soloway, Elliot I. / Spohrer, James C. (Hrsg): Studying the Novice Programmer, Hillsdale, NJ 1989, S. 401-416.

Süßenbacher, Winfried: Software-Bildung. Innsbruck 1997.

Syslo, Maciej M. / Kwiatkowska, Anna Beata: The Challenging Face of Informatics Education in Poland. In: Mittermeir Roland T. / Syslo, Maciej M. (Hrsg.): Informatics Education – Supporting Computational Thinking, Heidelberg 2008, S. 1-18.

Syslo, Maciej M. / Kwiatkowska, Anna Beata: Informatics for All High School Students. In: Ira Diethelm, Ira / Mittermeir, Roland T.: Informatics in Schools. Sustainable Informatics Education for Pupils of all Ages, 6th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2013, Oldenburg, Germany, February 26–March 2, 2013. Proceedings. Berlin Heidelberg 2013, S. 43-56.

Tiberius, Victor: Hochschuldidaktik der Zukunftsforschung. Wiesbaden 2011.

Ullenboom, Christian: Java ist auch eine Insel, Das umfassende Handbuch. http://openbook.galileocomputing.de/javainsel/, Version von 2011, Zugriff: 03.04.13 – 11:05.

Umbach-Daniel, Anja / Wegmann, Armida: Das Image der Informatik in der Schweiz. http://www.ruetter.ch/cs/images/fit_image_informatik_schweiz_zusammenfassung.pdf, Zürich 2008. Zugriff: 05.03.13 – 13:17.

Weigend, Michael: Intuitive Modelle der Informatik. Potsdam 2007.

Zakhour, Sharon / Hommel, Scott / Royal, Jacob / Rabinovitch, Isaac / Risser, Tom / Hoeber, Marc: Das Java Tutorial, Eine Einführung in die Grundlagen, aktuell zu Version 6. München 2007.

G Lebenslauf

Persönliche Daten

Name:	Klemens Winkler

Ausbildung

2011 – heute:	Lehramtsstudium UF Informatik und Informatikmanagement und UF Psychologie und Philosophie an der Universität Wien.
2006 – heute:	Lehramtsstudium UF Physik und UF Informatik und Informatikmanagement an der Universität Wien.
1997 – 2005:	BG/BRG Purkersdorf (NÖ), Absolvierung des Gymnasial-Zweiges mit den Fremdsprachen Französisch und Italienisch.
1993 – 1997:	Volksschule Purkersdorf (NÖ)