Wozu JAVA?

Plädoyer für grafisches Programmieren

von Eckart Modrow, Jens Mönig und Kerstin Strecker

Dieser Beitrag ist ein Plädoyer dafür, das Aufkommen universeller grafischer Programmiersprachen zum Anlass zu nehmen, über die Rolle des Programmierens im Informatikunterricht neu nachzudenken. „JAVA“ ist dabei als Synonym für jegliches textbasierte Programmieren gemeint.

Zur aktuellen Situation

Machen wir uns nichts vor: Nach dreißig bis vierzig Jahren *Programmieren im Informatikunterricht* bewirkt diese Art von Unterricht, dass der größte Teil der so Unterrichteten, oft um die 80 Prozent, meist schon in der Sekundarstufe II befindlich, frustriert das Handtuch wirft. Und das bei einer Klientel, die überwiegend freiwillig und oft hoch motiviert ein neues Fach wählt. Wie würde es da erst bei einem Pflichtfach Informatik aussehen? Die Frage ist nicht rhetorisch gemeint – immerhin fordert die *Gesellschaft für Informatik* (GI) in Kenntnis dieser Ausgangslage ein Pflichtfach „Informatik für alle“ in der Sekundarstufe I, und sie beschreibt dies in ihren *Grundsätzen und Standards für die Informatik in der Schule* (AKBSI, 2008) mit einem gehäuften Teil Algorithmik einschließlich Implementation. Angaben dazu, wie das zu erreichen sein soll, bleibt sie leider schuldig.

Die Unterrichtenden und die – immer noch spärlich vertretene – Didaktik der Informatik an den Hochschulen haben auf diese Situation durchaus reagiert. „Informatikunterricht ist kein Programmierkurs“ – natürlich nicht, diese Debatte hat sich erledigt. Andere Themen wie Kryptologie, Datenbanken, Modellierungstechniken und Graphentheorie finden ihren berechtigten Platz sowohl im Unterrichtsgeschehen als auch in den Aufgaben des Zentralabiturs (vgl. Niedersächsisches Kultusministerium, 2008). Es wird erheblich weniger implementiert als früher. Wenn überhaupt Algorithmik betrieben wird, lässt man gegebene Algorithmen erforschen und nachvollziehen; Programmtexte, wenn überhaupt vorhanden, werden „dekonstruiert“.

Systeme werden mit völlig unterschiedlichen Methoden modelliert, und das bei Aufgaben, die auf das Zentralabitur vorbereiten, also in wenigen Minuten erfass- und bearbeitbar sein müssen. Solche Systeme können aber intuitiv verstanden werden, benötigen also in Wirklichkeit gar keine Modellierung (vgl. Strecker, 2009). Für den Unterricht ist das verhängnisvoll, denn die Lernenden erfahren so, dass von ihnen im Unterricht Unnötiges bzw. Unsinniges verlangt wird. Vor dem Zentralabitur hatten Modellierungstechniken im Projektunterricht ihren sicheren und sinnvollen Platz; ohne sie ging es nicht. Mit Zentralabitur und den damit verbundenen typischen Aufgaben erscheint diese Praxis in der Schule weitgehend als Bürokratie. Wir haben damit ein Beispiel, wie sich zwei gegenläufige, einzeln durchaus sinnvolle Entwicklungen konterkarieren.

Für den Praktiker sind die Vorgaben zum Zentralabitur und die dazu veröffentlichten Aufgabenbeispiele wesentliche Orientierungspunkte seines Unterrichts. Schließlich wird dessen Erfolg an der Bewältigung solcher Aufgaben gemessen. Nehmen wir als Beispiel eine Aufgabe des Informatik-Abiturs 2010 in Niedersachen (EA-Fach, d. h. Leistungskursniveau), dann konnte dort eine glatte Eins erreicht werden, ohne eine einzige Zeile Programmtext zu schreiben. Auch das ist eine Konsequenz der Probleme, die beim Programmieren im Unterricht auftreten. Natürlich sind nicht alle Aufgaben

Quelle: Zitty, Berlin, 15/85, S. 111 – LOG-IN-Archiv

Zeichnung: F. Wössner

derart extrem, aber die Tendenz, das Programmieren im Sinne des Implementierens in einer textbasierten Programmiersprache möglichst zu vermeiden, ist klar – und die Beispiele wirken. Normal interessierte und begabte Schülerinnen und Schüler müssen das Fach Informatik wählen und erfolgreich bewältigen können; eine Ausstiegsquote von 80 Prozent ist nicht akzeptabel. Wenn man aber einen wichtigen Bereich fortfallen lässt oder grundlegend ändert, ändert sich ein Fach qualitativ erheblich.

Die Frage stellt sich daher, ob die Vertreter des Schulfachs Informatik den Anspruch aufrechterhalten können, Algorithmik sei ein Werkzeug zur Entwicklung und Realisierung von Schülerideen und somit Informatik ein kreativitätsförderndes Fach, oder ob die Algorithmik auf die der Reproduktion zugänglichen Inhalte reduziert werden soll. Im zweiten Fall sollte das dann aber offen gesagt und zugegeben, im ersten Fall muss gezeigt werden, wie der Anspruch bei einer akzeptablen Erfolgsquote eingelöst werden kann. Darauf wird im Folgenden einzugehen sein.

Erfahrungen mit SCRATCH

Die Aussage, Informatikunterricht sei kein Programmierkurs, bedeutet wohl, dass Informatikunterricht kein „Kodierkurs“ sei, also keine Einführung in eine bestimmte Programmiersprache. Denn natürlich kann man den Begriff „Programmieren“ erheblich weiter fassen und meist tut man das auch. In einem ersten Schritt wollen wir daher untersuchen, ob im Informatikunterricht überhaupt eine Programmiersprache erforderlich ist. Wozu also JAVA (oder DELPHI, PYTHON usw.)?

Sehen wir uns die schon mehrfach zitierte 80-Prozent-Ausstiegsquote im Informatikunterricht an, so stellen wir fest, dass sie nur dann auftritt, wenn die Schülerinnen und Schüler „Programme schreiben“ sollen. Mit den anderen Themenbereichen des Informatikunterrichts gibt es kaum Schwierigkeiten. Keine Schwierigkeiten gibt es in der Regel auch mit dem Finden eigener Ideen zur Lösung der (möglichst selbst-) gestellten Probleme. Die treten erst „beim Schreiben“, also beim Verfassen des Programmtextes („Kodieren“) auf. Der Zyklus

Lösungsidee → Formulierung der Idee → Test → Änderung der Lösungsidee → …

wird i.d.R. durch Kodierungsprobleme unterbrochen. Die meisten Lernenden bleiben beim Kodieren stecken, nicht beim (hier weiter gefassten) Programmieren. Brauchen wir aber überhaupt Programmtext? Ist der Begriff „Programme schreiben“ im Sinne von „JAVA- (oder DELPHI-) Programmtext verfassen“ nicht eigentlich überholt?

In den letzten Jahren wurden mit grafischen Entwicklungssystemen überraschende Erfolge erzielt. Diese waren meist auf sehr enge Anwendungsgebiete beschränkt. Dennoch haben KARA, ALICE, ETOYS, LEGO MindStorms und in letzter Zeit besonders SCRATCH die Abbrecherquote an Schulen und auch Hochschulen drastisch reduziert. Dass es sich bei der grafischen Programmierung nicht durchweg um „Spiel-“ oder „Kindergarten“-Programmierung handelt, zeigen Werkzeuge wie *LabVIEW*, mit dem z.B. das südafrikanische Großteleskop SALT vollständig gesteuert wird.

Sehen wir uns SCRATCH etwas genauer an (vgl. auch Romeike, 2007). Konzipiert für Vor- und Grundschulen wird damit dennoch sehr erfolgreich in der gesamten Sekundarstufe I unterrichtet, teilweise auch darüber hinaus. Vorwürfe bezüglich der „kindlichen“ Oberfläche zielen deutlich daneben – schließlich handelt es sich bei der eigentlichen Zielgruppe um (recht kleine) Kinder. Wenn Unterrichtete und Unterrichtende höherer Jahrgänge trotzdem fröhlich damit arbeiten, dann zeigt das nur, dass die „richtigen“ Werkzeuge wohl nicht sehr geeignet gewesen sind. Beobachtet man Jugendliche bei der Arbeit mit SCRATCH, dann arbeiten eigentlich alle ohne große Einführung an eigenen Problemstellungen, die sie ständig umdefinieren, erweitern oder einschränken. Sie „entwanzen“ permanent, ohne sich darüber überhaupt Gedanken zu machen. Fehlersuche ist Teil des Entwicklungsprozesses; so sollte es sein, so ist es aber „mit JAVA“ meist nicht.

Betrachtet man die Komplexität der Problemlösungen, indem man die Schachtelungstiefe der algorithmischen Strukturen, die Zahl der Prozesse usw. analysiert, dann liegt die Komplexität von SCRATCH-Programmen der Mittelstufe meist deutlich über der von z.B. JAVA-Programmen der Oberstufen-Einführungskurse – und das wohlgemerkt bei selbstentwickelten Algorithmen, nicht bei „nachvollzogenen“. Die SCRATCH-Programmierer „modellieren“ zwar fast nie systematisch, sie legen einfach los. Allerdings besteht in einem JAVA-Einführungskurs wohl ebenfalls kaum Bedarf an Modellierung. Erst in elementarer Algorithmik Erfahrene können bei anspruchsvolleren Problemen Modellierungstechniken als hilfreich und sinnvoll erkennen.

Man kann zahlreiche Beispiele dafür anführen, dass sich bestimmte Arbeiten, z.B. die Eingabe und Verarbeitung mathematischer Terme, mit textbasierten Systemen wesentlich besser verrichten lassen als mit SCRATCH. Es lassen sich aber auch mindestens ebenso viele Beispiele finden, die viel eleganter mit SCRATCH lösbar sind. Trivialerweise sind unterschiedliche Werkzeuge für unterschiedliche Aufgaben geeignet. Und ebenso trivial ist es, dass für den Informatikunterricht fast immer nicht das gewählte Beispiel relevant ist, sondern die bei seiner Bearbeitung erworbenen Kompetenzen.

Beachten wir die Erfahrungen mit der Turtlegrafik oder den Robotern, dass es für die Lernenden viel einfacher ist, Fehler z.B. auf dem Bildschirm „zu sehen“ und dann nur noch den erkannten Fehler im Text zu korrigieren, als kryptische Programmtexte auf Fehler zu analysieren, dann scheinen grafische Anwendungen sehr viel geeigneter als mathematische Terme für den Anfangsunterricht zu sein – und motivierender sowieso.

Als Beispiel wollen wir mit der „Kindergarten-Version“ von SCRATCH ein Fadenpendel mit Erreger simu-

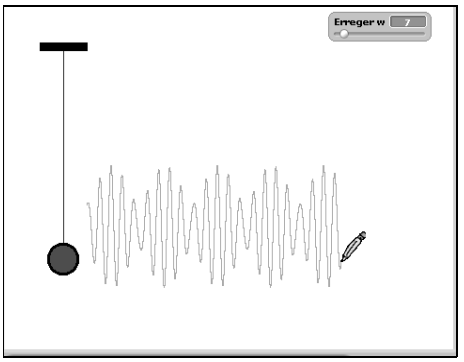


Bild 1: Aufzeichnung der Schwingung eines Fadenpendels.

lieren – ein ernsthaftes Projekt auch für die Oberstufe (siehe Bild 1)!

Das lässt sich mit ein paar kleinen Skripten (siehe Bilder 2 bis 4) leicht realisieren – vorausgesetzt, man hat die Physik verstanden. So lässt sich auch überprüfen, ob dies der Fall ist. Vor allem aber kann man z. B. Variablenbelegungen sofort am Bildschirm anzeigen und ändern und sich schrittweise dem gesuchten Ergebnis annähern. Der Erreger ändert periodisch seine Amplitude, und die Kugel reagiert darauf. Besonders einfach ist es, das Diagramm zeichnen zu lassen, indem ein Stift entsprechend mitgeführt wird.

BYOB als Beispiel grafischer Programmierung

Die Einschränkungen von SCRATCH sind offensichtlich. Weder sind in der Algorithmik geschachtelte Methoden und Rekursion noch bei den Datenstrukturen Listen von Listen möglich – und so fort. Jens Mönig hat dies zum Anlass genommen, SCRATCH um entsprechende Möglichkeiten zu erweitern. Die veränderte Version ist als BYOB 2.0 (= ***B**uild **y**our **o**wn **b**locks*) schon einigermaßen bekannt (vgl. Mönig, 2009). Für die vor Kurzem erschienene Version 3.0 hat er sich zusammen mit Brian Harvey (Universität Berkeley) noch weit anspruchsvollere Ziele gesetzt: Mit BYOB ist es möglich, die informatischen Konzepte des Lehrbuch-Klassikers *Struktur und Interpretation von Computerprogrammen* (Abelson/Sussman, [4]2001) komplett zu realisieren, also auf der Ebene von Berkeley-SCHEME zu arbeiten.

Mit BYOB 3.0 wird damit die konzeptionelle Ebene von Sprachen wie JAVA deutlich überschritten. Für uns ist aber Folgendes wichtig: wenn in Berkeley die Standard-Informatikvorlesung mit BYOB durchführbar ist, dann wird das System, was den geistigen Anspruch betrifft, auch für die deutsche Sekundarstufe II hinreichend anspruchsvoll sein.

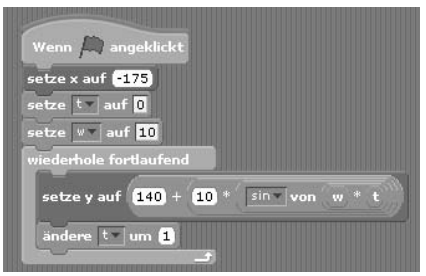
Ausgangspunkt des Projekts ist die Erkenntnis, dass sich zwischen dem Einstiegssystem SCRATCH und fortgeschritteneren Werkzeugen keine klare Grenze ziehen lässt. Egal, wo man ansetzt: Es findet sich immer ein weiter Überlappungsbereich. BYOB soll daher die Oberfläche von SCRATCH möglichst wenig verändern, sich gleichsam dahinter verstecken. In den Menüs finden sich aber Erweiterungen, mit denen die anspruchsvollen Konzepte von SCHEME realisiert werden – immer nach dem Motto: „So viel wie nötig und so wenig wie möglich."

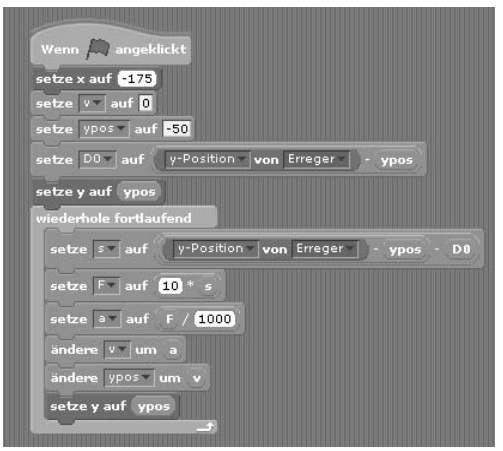
Beispiele für die Arbeit mit Blöcken

An dieser Stelle können wir keine systematische Einführung in BYOB geben; wir wollen auch nicht zeigen, dass und wie damit neue informatische Unterrichtskonzepte realisiert werden können. Stattdessen wollen wir demonstrieren, dass sich die tradierten Informatikinhalte der Oberstufe mit BYOB leicht und oft besser als mit textbasierten Systemen vermitteln lassen.

Beginnen wir mit den einfachen Erweiterungen. Mithilfe eines Blockeditors lassen sich Konzepte wie *strukturierte Zerlegung*, *Schachtelung von Operationen*, *lokale Variablen*, *Rekursion* usw. anschaulich realisieren. Bei einem *Block* kann es sich um einen neuen Befehl (command), eine Funktion (reporter) oder ein Prädikat handeln. Parameter können an beliebiger Stelle eingefügt und bei Bedarf typisiert werden. Mehrere Blockeditoren können gleichzeitig offen sein.

Wir wollen zuerst, weil es so einfach geht, Fibonacci-Zahlen berechnen. Dazu erzeugen wir einen Reporter mit dem Parameter n, den wir als Zahl deklarieren; lokale Variablen werden nicht benötigt. Danach erfolgt die übliche rekursive Definition (siehe Bilder 5a, b und Bild 6, nächste Seite).



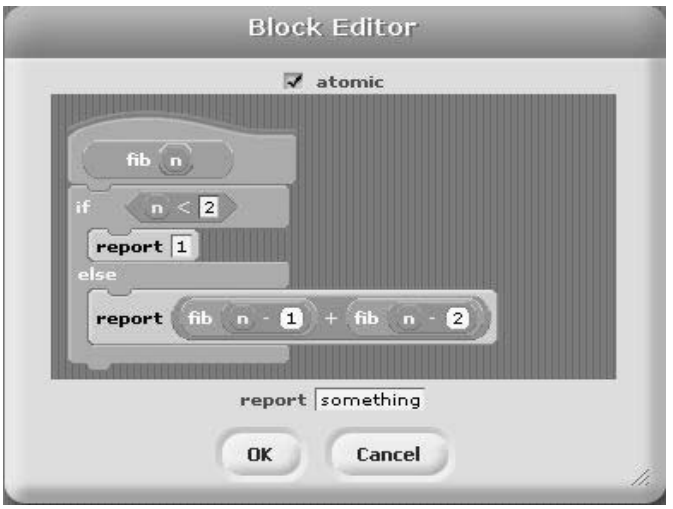


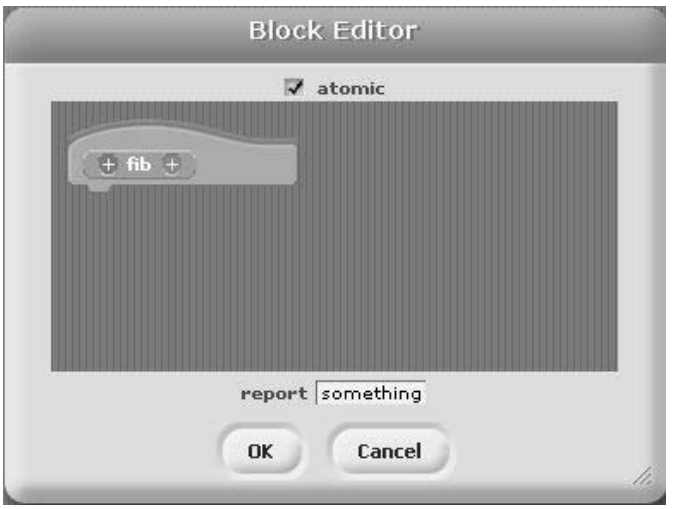
Bilder 2 bis 4: Skripte zur Erzeugung von Bild 1.

Zerlegen wir ein Problem in Teilprobleme, so können wir den Teilproblemen Operationen zuordnen, die als Blöcke realisiert

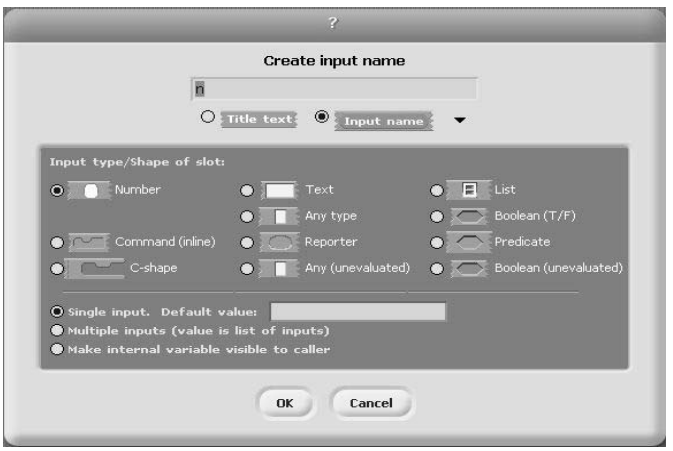
werden. Das soll im Zusammenhang mit Datenstrukturen demonstriert werden. BYOB kennt wie SCRATCH Zeichenketten und Listen. In BYOB-Listen können allerdings beliebige Elemente eingefügt werden, also auch weitere Listen. Damit lässt sich jede beliebige Datenstruktur erzeugen. Als Beispiel wollen wir die Datenstruktur „zweidimensionale Reihung (array)“ implementieren, und zwar ganz einfach ohne jede Zugriffskontrolle. Als Datenbehälter wählen wir eine einfache Liste; die Dimensionen n und m tragen wir ganz vorne ein. Zuerst wird mithilfe eines Reporter-Blocks namens

```
new array <name> [<breite>][<hoehe>]
```

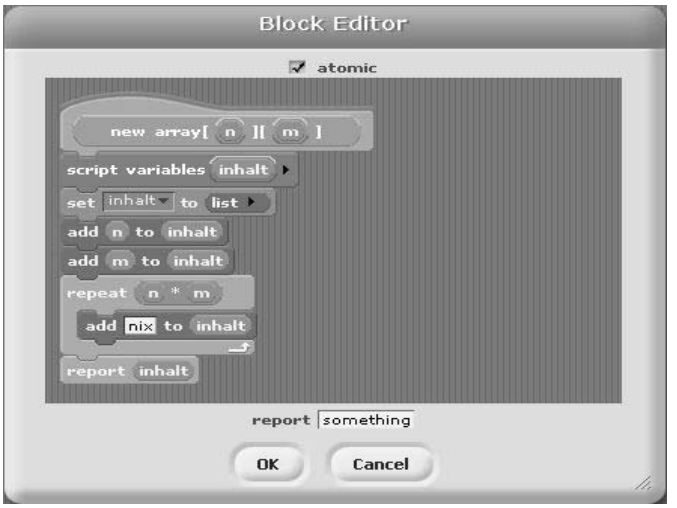


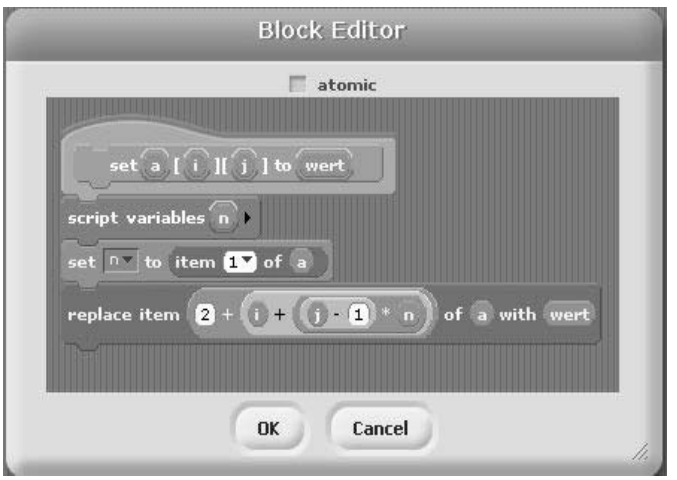


Bilder 5a und 5b (oben) und Bild 6 (unten):
Erzeugung von Fibonacci-Zahlen.



Bilder 7a, b, c:
Erzeugung einer zweidimensionalen Reihung.





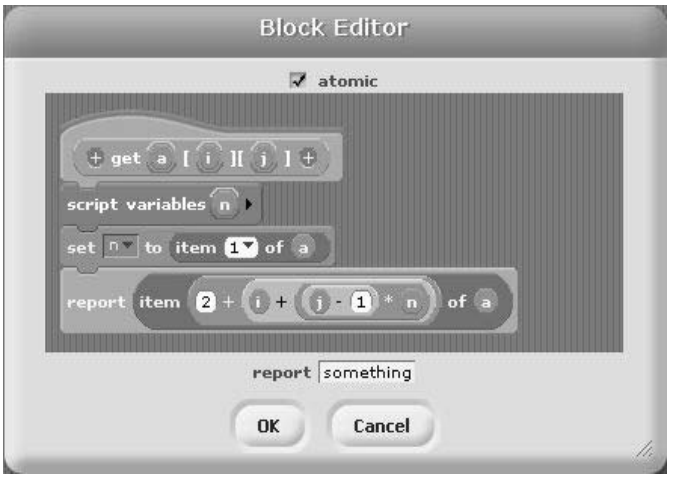


Bild 8: Anweisungen zur Bearbeitung der Reihung von Bild 7.

eine Reihung erzeugt. Dazu kreieren wir eine anfangs leere Liste namens *inhalt*, fügen die Dimensionen ein und danach n·m Elemente mit dem Wert *nix*. Dann wird diese Liste zurückgegeben (siehe Bilder 7a, b, c, vorige Seite). In diese Liste können an geeigneter Stelle Inhalte eingefügt werden. Dazu erzeugen wir in einer Methode

```
set <reihung>[<index1>][<index2>]
to <wert>
```

eine lokale Skript-Variable, um die Breite der Reihung aufzunehmen, und ersetzen den Wert an der richtigen Stelle. Die Bezeichnungen sind natürlich frei gewählt. Werte der Reihung erhält man zurück mit

```
get <reihung>[<index1>][<index2>]
```

Die Verwendung einer solchen Reihung zeigen die Anweisungen in Bild 8.

Bekanntlich gehört die Zählschleife als „algorithmischer Grundbaustein“ zur Reihung. Da BYOB über dergleichen nicht verfügt, jedenfalls nicht in der üblichen Form, wird das entsprechende Strukturelement nachgebildet (siehe Bilder 9 und 10); man beachte dabei den Abschnitt *run <action>*. Das Beispiel illustriert die weitergehenden Möglichkeiten des Systems, hier die „Lambdafizierung“ von BYOB-Strukturen, die es gestattet, diese wahlweise als Programmtext oder Daten zu interpretieren.

BYOB-Listen lassen sich trivialerweise als Schlangen oder Stapel nutzen. Geschachtelte Listen bilden Bäume, Wörterbücher, Graphen usw. Im Bereich der Datenstrukturen gib es hier keine Einschränkungen.

Objektorientierung mit BYOB

Ein wesentlicher Zug moderner Programmiersprachen ist die Möglichkeit, die Welt durch Objekte und Klassen zu beschreiben. Das sind für Lernende jedoch zwei wesentlich verschiedene Dinge. Beachtet man, dass der normale Weg der Erkenntnis vom Konkreten zum Abstrakten verläuft, dann ist es reichlich seltsam, mit dem abstrakten Konzept der Klasse zu beginnen, um erste einfache Objekte erzeugen zu können (wie dies z. B. in JAVA üblich ist).

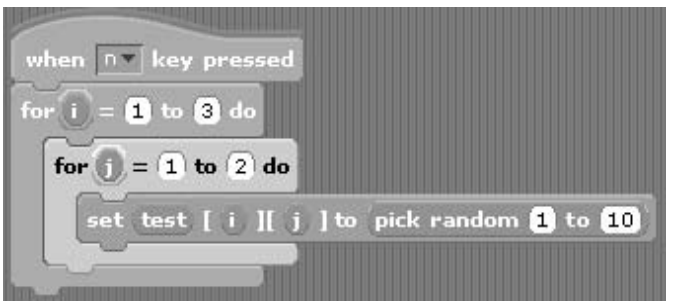
In BYOB haben wir beides: Erstens *Objekte* mit ihren Attributen und Methoden (Skripten), von SCRATCH geerbt, und zweitens natürlich *Klassen*. Für den Anfangsunterricht aller Jahrgangsstufen (einschließlich Universität) erscheint es uns am natürlichsten, zunächst Objekte mit den gewünschten Eigenschaften auszustatten und erst dann zu vervielfältigen und zur Klassenbildung überzugehen. Abstraktere Konzepte folgen erst dann, wenn sie benötigt werden, also in den Situationen, wo sie den Lernenden als sinnvoll erscheinen.

Als Beispiel wählen wir einen Schwarm von Fischen. Der besteht aus einzelnen, sehr ähnlichen Tieren, mindestens aber aus einem. Dieser Fisch schwimmt etwas ziellos in der Gegend herum und flieht eventuell vor einem Hai.

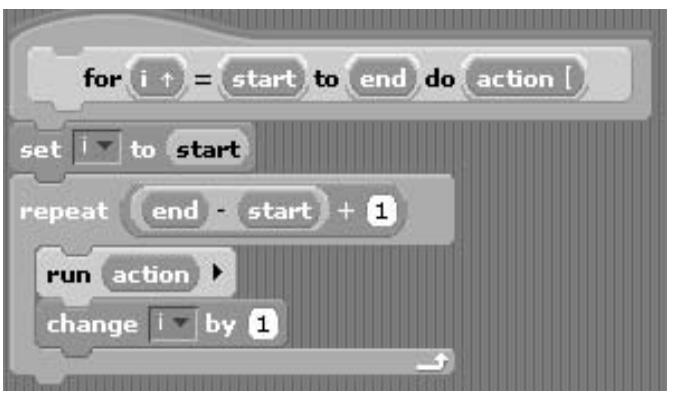
Ein einzelner Fisch bildet natürlich noch keinen Schwarm. Wir vervielfältigen daher unseren Fisch – mit der Folge, dass alle Fische unabhängig voneinander in der Gegend herumschwimmen, da sie nur die Methoden des Original-Objekts geerbt haben; das zur Schwarmbildung führende Verhalten müssen sie erst lernen. Beispielsweise können sie jeweils auf einen anderen, bestimmten Fisch zuschwimmen – allerdings nicht zu nahe! Oder sie bewegen sich auf alle anderen Fische jeweils ein Stückchen zu – das entspricht dann dem üblichen „Schwarmverhalten“. Oder sie beachten nur die nächsten Fische usw. Alle diese Verhaltensmuster führen zu unterschiedlichen Schwarmformen.

Ein Schwarm besteht natürlich aus einzelnen Fischen, also einzelnen Objekten. Dennoch handelt es sich bei ihm um etwas Neues, das Eigenschaften aufweist, die die Einzelobjekte nicht besitzen. Spätestens, wenn viele Fische ins Spiel kommen, können wir daher nicht mehr das Verhalten der einzelnen Fische untersuchen. Wir benötigen neue Strukturen, die den Schwarm als Ganzes beschreiben, und schreiten so „forschend“ im Lernprozess fort (siehe Bild 11, nächste Seite).

Da in BYOB Blöcke als Daten interpretiert werden können, impliziert das Verfahren eine elegante Form,



Bilder 9 und 10: Nachbildung einer Zählschleife.



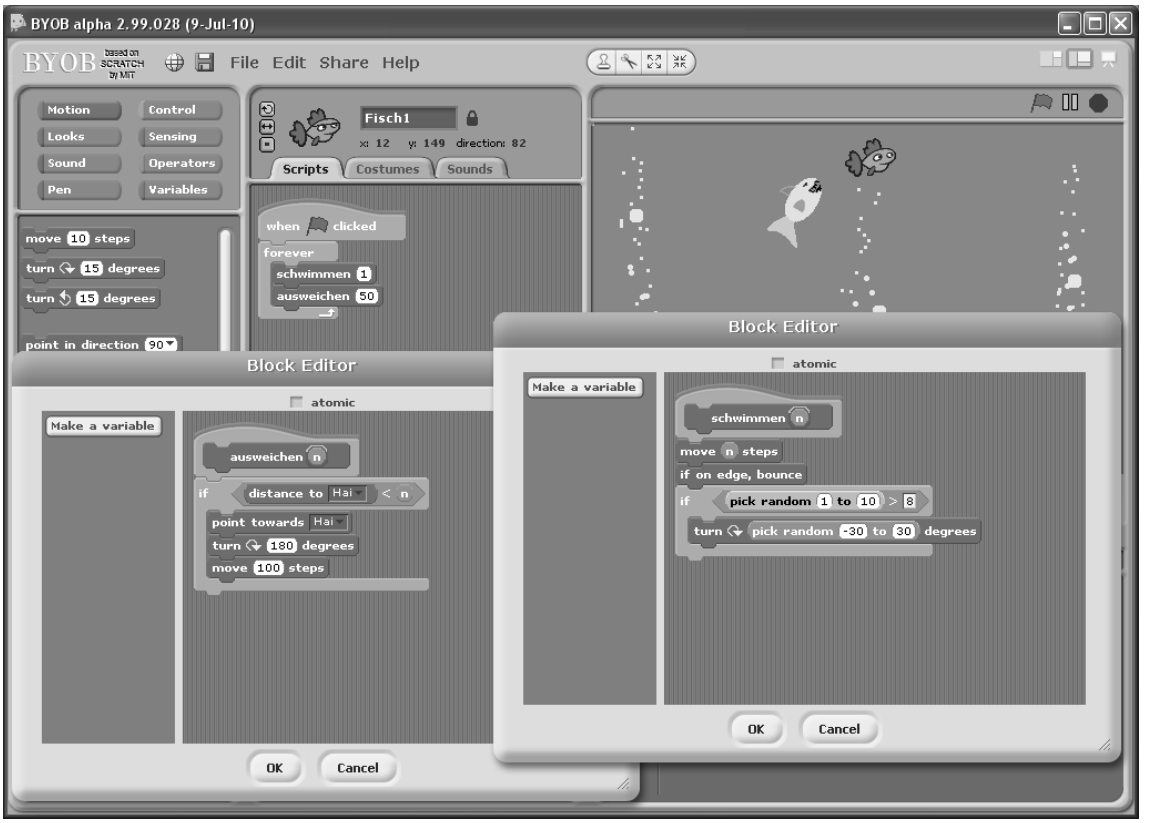


Bild 11: Modellierung eines Fisch-Schwarms.

Klassen zu implementieren: Methoden sind spezielle Attribute. Die Objekte einer Klasse unterscheiden sich in den „normalen“ Attributen wie Position, Größe, Farbe usw., besitzen aber den gleichen Satz von „Methoden-Attributen“, die man bei Bedarf über eine Nachricht aktiviert, indem sie, ähnlich wie bei der Zählschleife, über *call*- oder *run*-Anweisungen evaluiert werden. Leitet man von einer solchen Klasse eine Tochterklasse ab, dann „enthält“ diese ein Mutter-Objekt, dem sie nur solche Nachrichten weiterleitet, die nicht von der Tochterklasse selbst behandelt werden. Leider ist hier kein Platz, dies genauer auszuführen; Einzelheiten dazu findet man z. B. bei Harvey (2010).

BYOB und Abitur

Sichtet man die Zentralabitur-Aufgaben verschiedener Bundesländer aus den letzten Jahren im Bereich *Algorithmen und Datenstrukturen*, dann findet man typischerweise

- die Anwendung, Modifizierung und Implementation klassischer Datenstrukturen wie Zeichenketten, Reihungen, Listen verschiedener Art und Bäume,
- die Untersuchung von in unterschiedlicher Form gegebener Algorithmen hinsichtlich ihres Verhaltens, ihrer Komplexität usw. und
- die Bearbeitung (Beschreibung, Modifikation, Implementierung) eines durch seine Klassen, typischerweise durch Klassendiagramme, beschriebenen Modells.

Alle daraus abgeleiteten Aufgaben sind mit BYOB problemlos bearbeitbar – mit einer Ausnahme: Es ist ziemlich unpraktisch, BYOB-Programme auf dem Papier zu zeichnen. Da diese aber sowieso keine Syntaxfehler zulassen und Algorithmen abbilden, die ebenso durch Struktogramme oder UML-Diagramme beschrieben werden können, ist zu fragen, ob die Entwicklung von Algorithmen nicht sowieso besser in dieser letzteren Form erfolgt.

Konsequenzen und offene Fragen

Wenn wir ein Pflichtfach Informatik anstreben, dann muss dieses Fach für fast alle Schülerinnen und Schüler erfolgreich zu bewältigen sein, und das ist es auch schon bisher – bis auf den Bereich der Algorithmik. In der Argumentation für das Schulfach Informatik spielt der kreative, konstruktive, eben „technische“ Aspekt eine wesentliche Rolle. Zusammen mit dem zu vermittelnden Verständnis für die Auswirkungen der Informatiksysteme in unserer Gesellschaft ist das Gebiet *Algorithmen und Datenstrukturen* für das Fach konstituierend und kann nicht einfach stillschweigend gestrichen werden.

Mit grafischen Programmiersystemen, speziell BYOB, stehen uns heute Werkzeuge zur Verfügung, mit dem sich der Anspruch „Informatik für alle“ auch im Bereich der Algorithmik erfolgreich realisieren lässt. Inhaltlich sind uns durch die Verwendung solcher Systeme keine Grenzen gesetzt – im Gegenteil. Fassen wir den Begriff „Programmieren“ weit, beschreiben also damit den gesamten Prozess von der Modellbildung über die algorithmische Beschreibung bis zur Implementation, dann bilden zwar

die beiden ersten Schritte den anspruchsvollen, im eigentlichen Sinn „bildenden“ Teil des Prozesses. Die Motivation, diesen anspruchsvollen Weg auch zu gehen, folgt aber aus dem dritten: Schülerinnen und Schüler können nicht nur deskriptiv arbeiten, sondern sie setzen ihre eigenen Ideen in lauffähige Systeme um, sie erstellen vorzeigbare Produkte. Gerade darin liegt der Wert der Informatik in der Schule.

Systeme wie BYOB gestatten es nun, jenen letzten Schritt (die Implementation) einerseits für fast alle erfolgreich möglich zu machen, die dafür erforderliche Zeit aber im Vergleich zu textbasierten Sprachen zu minimieren. Der Gesamtprozess bleibt kreativ, schon weil sich die Phasen mischen; für die konzeptionelle Arbeit steht dann aber wesentlich mehr Zeit als früher zur Verfügung. Damit sind wir in einer ähnlichen Situation wie bei der Einführung der Softwarewerkzeuge zur Gestaltung grafischer Benutzungsoberflächen: der hierfür erforderliche Zeitbedarf wurde auch dort marginalisiert, die gewonnene Zeit steht nunmehr für inhaltlich anspruchsvollere Aufgaben zur Verfügung.

Damit stellen sich einige für die Informatik in der Schule interessante Fragen, die bisher nicht gestellt wurden, weil es schlicht keine Alternative zur textbasierten Programmierung gab.

1. Worin besteht der bildende Wert textbasierter Programmierung, also der Möglichkeit, Syntaxfehler machen zu können, wenn inhaltlich alle informatischen Konzepte, aber auch Standardanwendungsbereiche und -aufgaben mit grafischen Programmierwerkzeugen (wie z. B. BYOB) realisiert werden können? Ist die mit der Textbasierung verbundene Frustrationsrate gerechtfertigt, hat Syntax ihren eigenen Wert?
2. Ist der zu beobachtende Trend: „Weg von der Programmierung“ inhaltlich begründet oder ein Resultat der Probleme im Unterricht? Soll er beibehalten oder vielleicht angehalten, sogar wieder umgekehrt werden, wenn jetzt geeignetere Werkzeuge dafür zur Verfügung stehen? Um nicht missverstanden zu werden: Programmieren wird hier immer noch als Synonym für „selbstständiges produktorientiertes Problemlösen“ verwendet, nicht für „Kodieren“!
3. Auf welcher Beschreibungsebene ist zu arbeiten? Algorithmen können natürlich als BYOB-Blöcke vorgegeben werden; bearbeitbar sind sie in dieser Form allerdings nur am Computer. Soll und kann auf „papiergeeignete“ Notationsformen umgestiegen werden, wenn Syntaxeigenheiten keine Rolle mehr spielen, korrekte Syntax in diesem Zusammenhang keinen eigenen Wert mehr hat?
4. Ist Informatikunterricht nur realitätsnah, wenn echte Produktionssysteme wie JAVA, PYTHON usw. im Unterricht verwendet werden? Benötigt der Informatikunterricht Ausbildungswerkzeuge ähnlich wie die Naturwissenschaften, die auch fast ausschließlich Geräte einsetzen, die man außerhalb der Schule kaum findet?

Diese Fragen stellen sich erst jetzt aufgrund der neu zur Verfügung stehenden Techniken; sie können nunmehr präziser gestellt werden, weil es Alternativen zu den herkömmlichen Möglichkeiten gibt. Die Antworten werden unsere Sicht auf einen wichtigen Bereich der Informatik in der Schule und damit unsere Argumentation schärfen; wir werden sie nicht am Schreibtisch, sondern durch Erfahrungen in der Schulpraxis finden. Die Unterrichtserfahrungen sind aber wichtig, weil erst auf dieser Basis ein sachlicher Diskurs geführt werden kann. Also: Probieren Sie es aus!

Prof. Dr. Eckart Modrow
Max-Planck-Gymnasium
Theaterplatz 10
37073 Göttingen

E-Mail: emodrow@informatik.uni-goettingen.de

Jens Mönig
Lange Straße 23/2
71126 Gäufelden

E-Mail: jens.moenig@miosoft.com

Dr. Kerstin Strecker
Max-Planck-Gymnasium
Theaterplatz 10
37073 Göttingen

E-Mail: kerstin.strecker@gmx.de

Literatur und Internetquellen

Abelson, H.; Sussman, G.: Struktur und Interpretation von Computerprogrammen – Eine Informatik-Einführung. Berlin u. a.: Springer, ^{4}2001.

AKBSI – Arbeitskreis „Bildungsstandards“ der Gesellschaft für Informatik (Hrsg.): Grundsätze und Standards für die Informatik in der Schule – Bildungsstandards Informatik für die Sekundarstufe I. Empfehlungen der Gesellschaft für Informatik e. V. vom 24. Januar 2008. In: LOG IN, 28. Jg. (2008), Heft 150/151, Beilage.

AP Computer Science: Principles – Big Ideas, Key Concepts, and Supporting Concepts (2010).
http://www.csprinciples.org/docs/APCSPrinciplesBigIdeas20110204.pdf

Harvey, B.: BYOB Reference Manual Version 3.0 (2010).
http://byob.berkeley.edu/BYOBManual.pdf

Mönig, J.: BYOB 2.0 (August 2009).
http://chirp.scratchr.org/dl/BYOB%202.0.pdf

Mönig, J.; Harvey, B.: BYOB 3.0 – Build Your Own Blocks (August 2010).
http://byob.berkeley.edu/

Niedersächsisches Kultusministerium: VZIN – Vorgaben Zentralabitur Informatik Niedersachsen 2011 – Thematische Schwerpunkte Informatik, Juni 2008.
http://www.nibis.de/nli1/gohrgs/zentralabitur/zentralabitur_2011/18informatik2011.pdf

Romeike, R.: Animationen und Spiele gestalten – Ein kreativer Einstieg in die Programmierung. In: LOG IN, 27. Jg. (2007), Heft 146/147, S. 36–44.

Strecker, K.: Informatik für Alle – Wie viel Programmierung braucht der Mensch? Göttingen: Georg-August-Universität (Dissertation), 2009.
http://webdoc.sub.gwdg.de/diss/2009/strecker/strecker.pdf

Alle Internetquellen wurden zuletzt am 31. Mai 2011 geprüft.