Eckart Modrow
Jens Mönig
Kerstin Strecker

Wozu Java?

Abstract:
Der folgende Artikel ist ein Plädoyer, das Aufkommen universell geeigneter grafischer Programmiersprachen wie BYOB zum Anlass zu nehmen, neu über die Rolle des Programmierens im Informatikunterricht nachzudenken. „Java“ ist dabei nur als Synonym für textbasierte Programmierung zu sehen.

1. Zur Situation

Machen wir uns nichts vor: nach 30 bis 40 Jahren – je nach Zählung – Programmieren im Informatikunterricht bewirkt dieser Teil des Unterrichts immer noch, dass der größte Teil der Unterrichteten, oft um die 80%, 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? Sie halten die Frage für rhetorisch? Nun denn: immerhin fordert die Gesellschaft für Informatik (GI) in Kenntnis dieser Ausgangslage ein Pflichtfach „Informatik für alle“ in der Sekundarstufe I, und sie beschreibt dieses in ihrem Entwurf für Standards (AKBSI, 2008) mit einem gehäuften Teil Algorithmik incl. Implementation. Angaben dazu, wie sie das erreichen will, bleibt sie leider schuldig. Mutig sind die Vertreter der GI! Oder nur waghalsig?

Die Unterrichtenden und die immer noch spärlich vertretene Didaktik an den Hochschulen haben auf diese Situation durchaus reagiert. Informatik ist kein Programmierkurs, natürlich nicht, diese Debatte brauchen wir nicht mehr zu führen. Andere Themen wie Kryptologie, Datenbanken, Modellierungstechniken und Graphentheorie finden ihren berechtigten Platz im Unterrichtsgeschehen wie in den Aufgaben des Zentralabiturs (VZIN, 2008). Es wird weniger implementiert, viel weniger. Wenn überhaupt Algorithmik betrieben wird, dann werden gegebene Algorithmen erforscht und nachvollzogen, der Code, wenn überhaupt vorhanden, dekonstruiert. Systeme werden mit den unterschiedlichsten Methoden modelliert, und das bei Aufgaben, die auf das Zentralabitur vorbereiten, also in wenigen Minuten erfass- und bearbeitbar sein müssen – das ist eine ziemlich präzise Beschreibung für Systeme, die intuitiv verstanden werden können, also gar keine Modellierung erfordern (STR, 2009). Für den Unterricht ist das schlimm, denn die Lernenden erfahren so die Unsinnigkeit dessen, was von ihnen 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 erscheinen sie in der Schule weitgehend als Bürokratie. Wir haben damit ein schönes Beispiel, wie sich zwei gegenläufige, einzeln durchaus sinnvolle Entwicklungen konterkarieren.

Für den Praktiker sind die Vorgaben zum Zentralabitur und die dazu veröffentlichen Aufgabenbeispiele wesentliche Orientierungspunkte seines Unterrichts. Schließlich wird dessen Erfolg an solchen Aufgaben gemessen werden. Nehmen wir als Beispiel eine Aufgabe des aktuellen Informatikabiturs 2010 in Niedersachen (EA-Fach, d.h. Leistungskursniveau), dann konnten dort eine glatte Eins erreicht werden, ohne eine einzige Zeile Code zu schreiben. Auch das ist eine Konsequenz aus den Problemen mit dem Programmieren in der Schule. Natürlich sind nicht alle Aufgaben derart extrem, aber die Tendenz ist klar – und die Beispiele wirken. Natürlich müssen normal interessierte und begabte Schülerinnen und Schüler das Fach Informatik wählen können, und natürlich ist eine Ausstiegsquote von 80% nicht akzeptabel. Aber ebenso natürlich ändert sich ein Fach qualitativ, wenn es einen wichtigen Bereich weg lässt oder grundlegend ändert.

Die Frage ist deshalb, ob die Vertreterinnen und Vertreter des Schulfachs Informatik den Anspruch aufrecht erhalten können, Algorithmik als Werkzeug zur Entwicklung und Realisierung von Schülerideen zu betrachten, Informatik also als kreatives Fach weiterentwickeln wollen, oder ob dieser Bereich auf die der Reproduktion zugängliche Inhalte reduziert werden soll. Im zweiten Fall muss das dann auch offen gesagt werden, im ersten Fall muss gezeigt werden, wie es bei einer akzeptablen Erfolgsquote geschehen kann. Hiervon handelt dieser Artikel.

2. Erfahrungen mit Scratch

Informatikunterricht ist kein *Programmierkurs* (s.o.). In dieser Deutlichkeit bedeutet die Aussage wohl, dass Informatikunterricht kein *Kodierkurs* ist, also keine Einführung in eine bestimmte Programmiersprache. Denn natürlich kann man „Programmieren" erheblich weiter fassen, meist tut man das auch. In einem ersten Schritt wollen wir deshalb untersuchen, ob denn überhaupt eine Programmiersprache in der Schule erforderlich ist. Wozu also Java?

Sehen wir uns die schon mehrfach zitierte 80%-Ausstiegsquote im Informatikunterricht an, dann tritt die nur auf, wenn die Schülerinnen und Schüler „Programme schreiben" müssen. Mit den anderen Themenbereichen des Informatikunterrichts gibt es meist keine 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 Kodieren auf. Der Zyklus „Lösungsidee → Formulierung der Idee →Test → Änderung der Lösungsidee → …" wird durch Kodierungsprobleme unterbrochen. Die meisten Schülerinnen und Schüler bleiben beim Kodieren stecken, nicht beim (hier weiter gefassten) Programmieren. Brauchen wir aber überhaupt Code? Ist der Begriff „Programme schreiben" nicht eigentlich überholt?

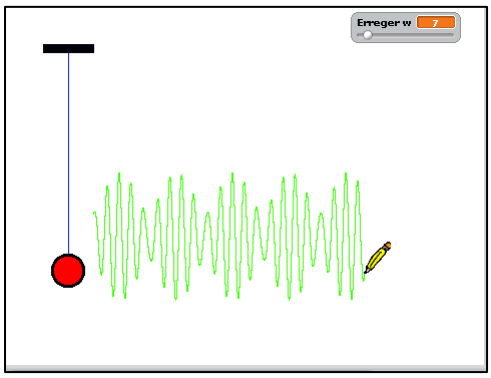
In den letzten Jahren wurden überraschende Erfolge mit grafischen Entwicklungssystemen gemacht. Diese waren meist auf sehr enge Anwendungsgebiete beschränkt. Trotzdem haben *Kara*, *Alice*, *eToys, Lego-Mindstorms* und in letzter Zeit besonders *Scratch* die Abbrecherquote an Schulen und auch Hochschulen drastisch reduziert. Dass es sich bei der grafischen Programmierung nicht durchweg um „Spielzeug-" oder „Kindergartenprogrammierung" handelt, zeigen Werkzeuge wie *LabView*, mit dem z. B. das südafrikanische Großteleskop SALT vollständig gesteuert wird – und bei dem handelt es sich gewiss nicht um ein Spielzeug.

Sehen wir uns *Scratch* etwas genauer an: konzipiert für Vor- und Grundschulen wird damit trotzdem sehr erfolgreich in der gesamten Sekundarstufe I unterrichtet, teilweise auch darüber hinaus. Vorwürfe bzgl. der „kindischen" Oberfläche gehen 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 sind. Beobachtet man Jugendliche bei der Arbeit mit Scratch, dann arbeiten eigentlich alle ohne große Einführung an eigenen Problemstellungen, die sie ständig umdefinieren, erweitern, einschränken, … Sie „debuggen" permanent, ohne sich darüber überhaupt Gedanken zu machen. Debuggen ist Teil des Entwicklungsprozesses – na klar. So sollte und soll es sein, so ist es aber „mit *Java*" meist nicht. Betrachtet man die Komplexität der Problemlösungen, indem man die Schachtelungstiefe der Kontrollstrukturen, die Zahl der Prozesse usw. analysiert, dann liegt die Komplexität von *Scratch*-Programmen der Mittelstufe meist sehr deutlich über der von z. B. *Java*-Programmen der Oberstufen-Einführungskurse – und das wohlgemerkt bei selbst entwickelten 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 Schülerinnen

und Schüler können bei anspruchsvolleren Problemen Modellierungstechniken als hilfreich und sinnvoll erkennen – und dann: siehe oben.

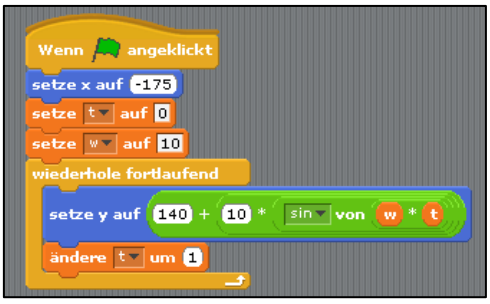
Man kann jetzt zahlreiche Beispiele dafür anführen, dass sich bestimmte Arbeiten, z. B. die Eingabe und Verarbeitung von mathematischen Termen, mit textbasierten Systemen wie *Java*, *Python*, … sehr viel besser verrichten lassen als mit *Scratch*. Man kann natürlich mindestens ebenso viele Beispiele finden, die sich viel eleganter mit *Scratch* lösen lassen. Trivialerweise sind unterschiedliche Werkzeuge für unterschiedliche Aufgaben optimiert. Und ebenso trivial ist es, dass für den Informatikunterricht fast immer nicht das gewählte Beispiel relevant ist, sondern die bei seiner Bearbeitung erworbenen Kompetenzen. Beispiele sind austauschbar. 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 anfangs 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 Federpendel mit Erreger simulieren – ein richtig ernsthaftes Projekt auch für die Oberstufe!

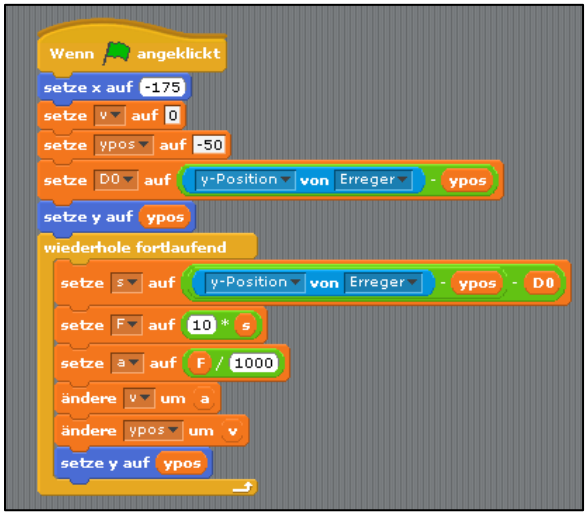


Das lässt sich mit ein paar kleinen Skripten leicht realisieren – wenn man die Physik verstanden hat. Man kann so überprüfen, *ob* man die Physik verstanden hat. Vor allem aber kann man z. B. Variablenbelegungen sofort am Bildschirm anzeigen und ändern und sich schrittweise dem gesuchten Ergebnis nähern.

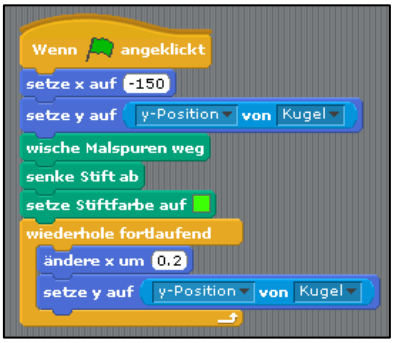
Der Erreger ändert periodisch seine y-Position:



Und die Kugel reagiert darauf:



Besonders einfach ist es, das Diagramm zu zeichnen. Ein Stift wird entsprechend bewegt:



Nur zur Erinnerung: wir arbeiten immer noch mit der Grundschulversion!

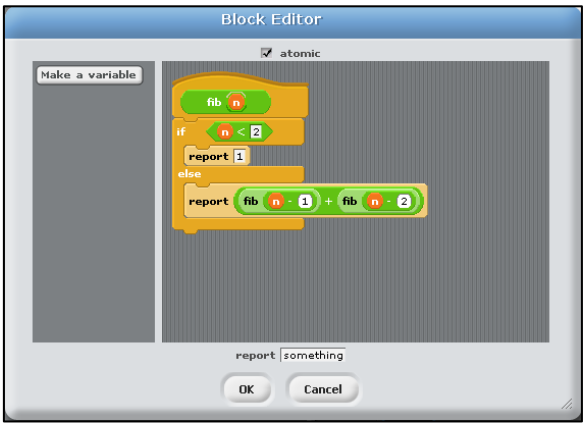
3. BYOB als Beispiel grafischer Programmierung

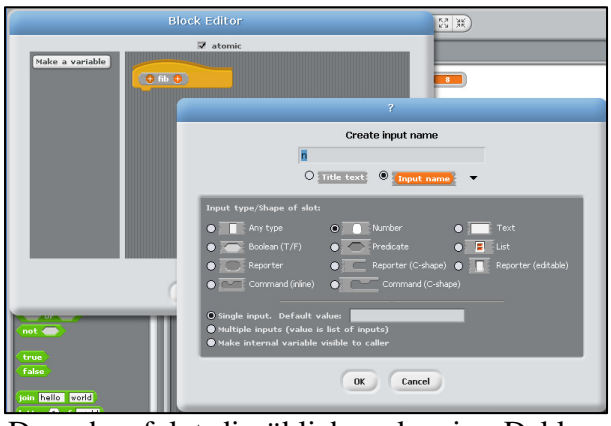
Die Einschränkungen von *Scratch* sind offensichtlich. Weder sind in der Algorithmik geschachtelte Methoden und Rekursion noch bei den Datenstrukturen Listen von Listen möglich – und so fort. *Jens Mönig* hat dieses zum Anlass genommen, Scratch um entsprechende Features zu erweitern. Die veränderte Version ist als *BYOB 2.0* (*build your own blocks*) schon einigermaßen bekannt (BYOB2, 2009). Für die gerade erschienene Version 3.0 (BYOB3, 2010) hat er sich zusammen mit *Brian Harvey* von der Universität Berkeley noch weit anspruchsvollere Ziele gesetzt: mit *BYOB 3* ist es möglich, die informatischen Konzepte des Lehrbuchklassikers „*Struktur und Interpretation von Computerprogrammen*" von *Abelson* und *Sussman* (ABSUS, 2001) komplett zu realisieren, also auf der Ebene von Berkeley-*Scheme* zu arbeiten. Damit wird die konzeptionelle Ebene von Sprachen wie *Java* deutlich überschritten. Für uns ist aber etwas anderes wichtig: wenn Berkeley seine Informatikvorlesung CS10 im Rahmen des AP-Curriculums (AP Principles 2010) mit BYOB als einziger Programmiersprache durchführen kann, dann wird das System auch für die deutsche Sekundarstufe II geeignet sein. Eine Diskussion in dieser Hinsicht erübrigt sich also.

Ausgangspunkt des Projekts ist die Erkenntnis, dass man keine klare Grenze zwischen dem Einstiegssystem *Scratch* und fortgeschritteneren Werkzeugen definieren kann. Egal, wo man ansetzt: man findet immer einen weiten Überlappungsbereich. *BYOB* soll deshalb die Oberfläche von Scratch möglichst wenig verändern, sich quasi 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".

An dieser Stelle können wir keine systematische Einführung in *BYOB* geben. Wir wollen auch nicht zeigen, dass und wie für den üblichen Unterricht neue informatische Konzepte mit diesem System realisiert werden können – das wäre ein anderer Artikel. Stattdessen wollen wir zeigen, dass sich die tradierten Informatikinhalte der Oberstufe mit *BYOB* leicht und oft besser als mit textbasierten Systemen unterrichten lassen. Beginnen wir mit den einfachen Erweiterungen. Mithilfe eines Blockeditors können Konzepte wie strukturierte Zerlegung, geschachtelten Operationen, lokale Variable, Rekursion usw. anschaulich umgesetzt werden. Bei einem Block kann es sich 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 so schön kurz, Fibonacci-Zahlen berechnen. Dazu erzeugen wir einen Reporter mit dem Parameter n, den wir als Zahl deklarieren. Lokale Variable werden nicht gebraucht.





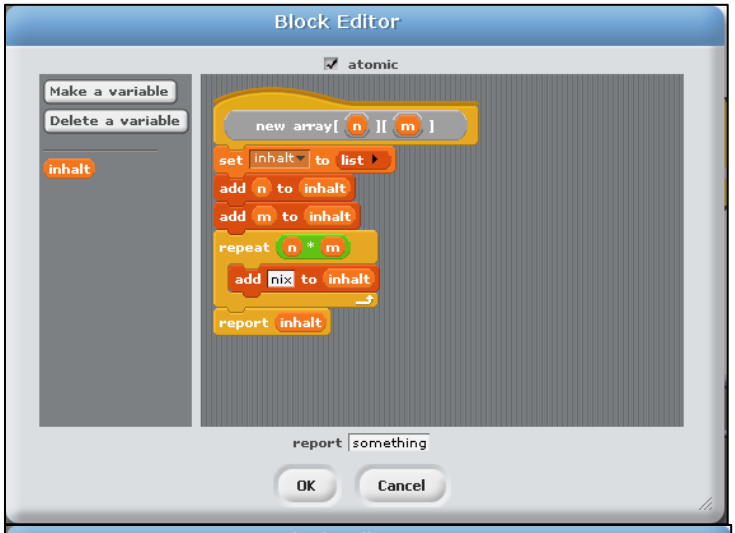
Danach erfolgt die übliche rekursive Deklaration.

Wenn das funktioniert, dann klappt es natürlich auch mit allen darauf aufbauenden Konzepten. Zerlegen wir ein Problem in Teilprobleme, dann können wir diesen Operationen zuordnen, die als Blöcke realisiert werden. Demonstrieren wollen wir das im Zusammenhang mit Datenstrukturen.

BYOB kennt wie *Scratch* Zeichenketten und Listen. In *BYOB*-Listen können allerdings beliebige Elemente eingefügt werden, also auch weitere Listen. Damit können wir jede beliebige Datenstruktur erzeugen. Als Beispiel wollen wir die Datenstruktur „zweidimensionale Reihung (*array*)“ implementieren, hier ganz einfach ohne jede Zugriffskontrolle. Als Datencontainer wählen wir eine einfache Liste, die Dimensionen n und m tragen wir ganz vorne ein. Zuerst müssen wir mithilfe eines Reporter-Blocks namens

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

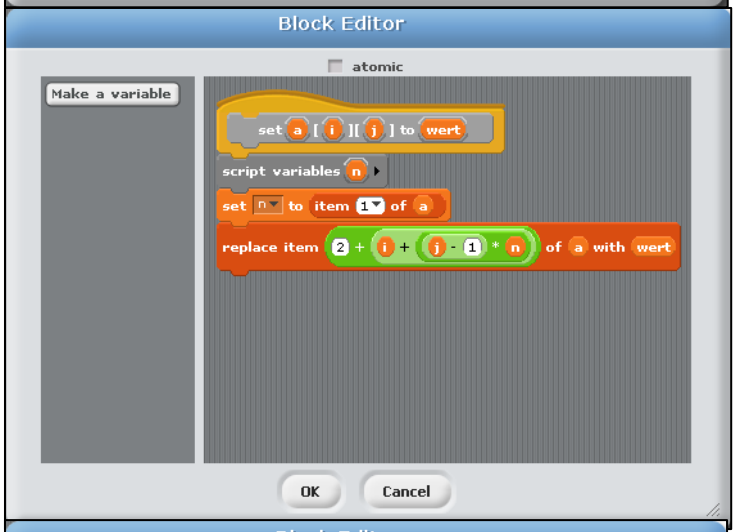
eine Reihung erzeugen. Dazu kreieren wir eine anfangs leere Liste namens inhalt, fügen die Dimensionen ein und danach n*m Elemente mit dem Wert „*nix*“. Dann wird diese Liste zurück gegeben.



In diese Liste können wir an geeigneter Stelle Inhalte einfügen. Dazu erzeugen wir in einer Methode

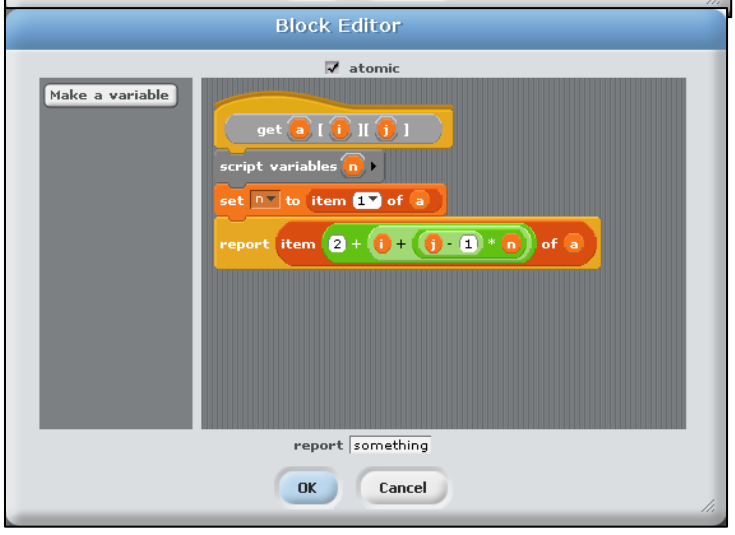
set<reihung>[<index1>][<index2>]
to <wert>

eine lokale Script-Variable, um die Breite der Reihung aufzunehmen, und ersetzen den Wert an der richtigen Stelle. Die Bezeichnungen sind natürlich frei gewählt.

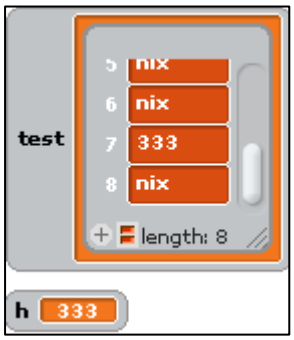


Werte der Reihung erhält man zurück mit

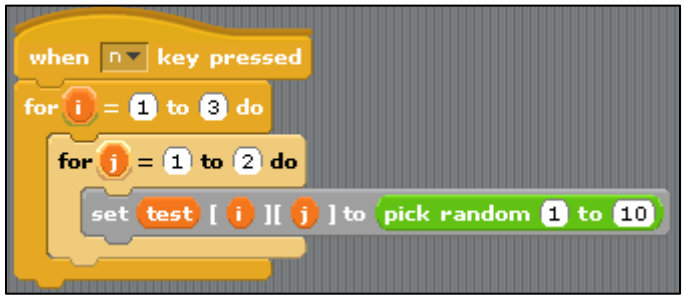
get<reihung>[<index1>][<index2>]



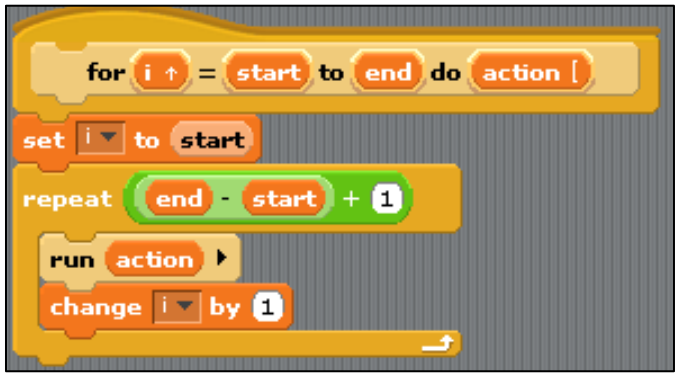
Die Benutzung solch einer Reihung zeigen die folgenden Anweisungen.



Nun werden viele meinen, dass die zugeordnete Kontrollstruktur einer Reihung die Zählschleife ist, und über sowas verfügt *BYOB* nicht, jedenfalls nicht in der üblichen Form. Wir möchten aber so etwas haben.



Na, dann bauen wir eine Zählschleife, indem wir eine Kontrollstruktur (wieder ohne jede Fehlerkontrolle) entwickeln, die *BYOB*-Code ausführen kann. Man beachte dabei den run<action> Abschnitt.



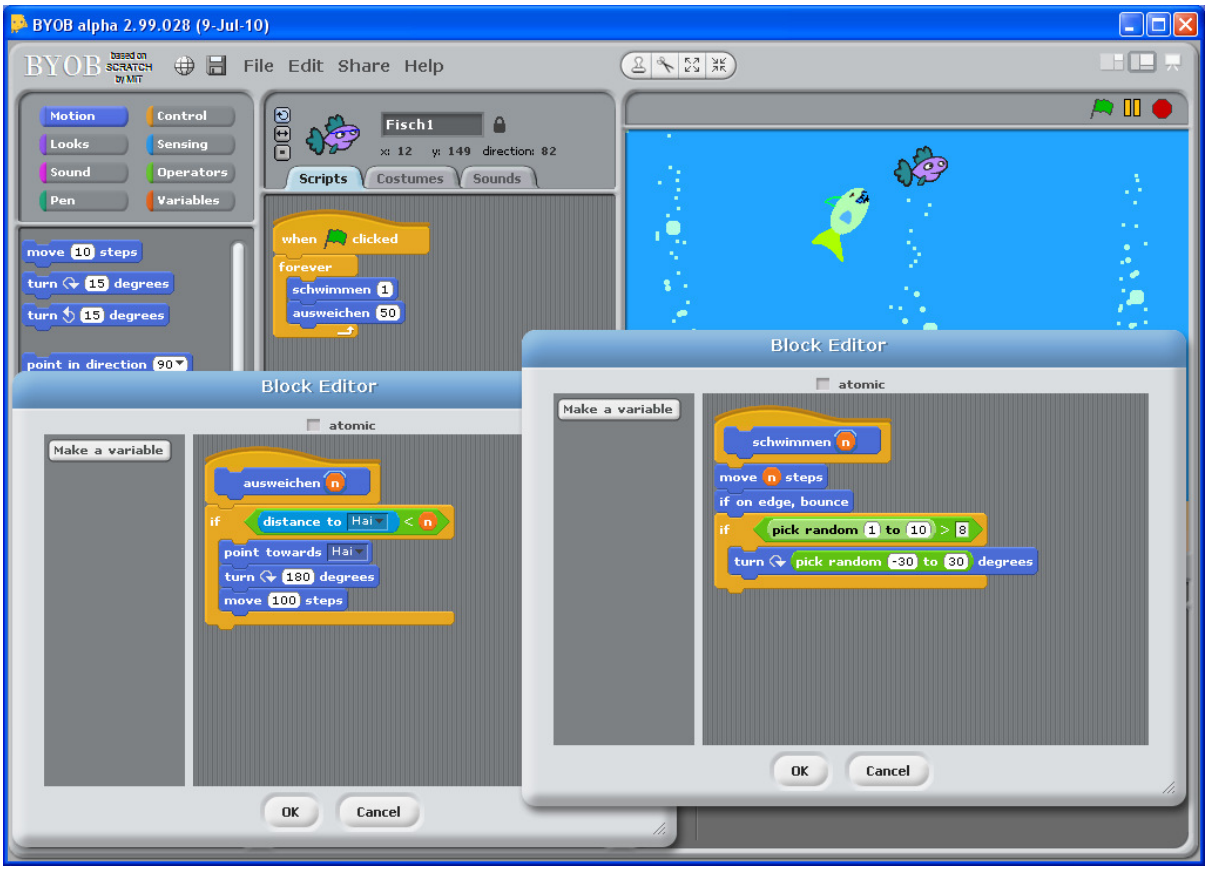
Das Beispiel illustriert kurz die weitergehenden Möglichkeiten des Systems, hier die „*Lambdafizierung*“ von *BYOB*-Strukturen, die es gestattet, diese wahlweise als Code oder Daten zu interpretieren.

Die *BYOB*-Listen lassen sich trivialerweise als Schlangen oder Stapel nutzen. Geschachtelte Listen bilden Bäume, Dictionaries, Graphen usw. Im Bereich der Datenstrukturen gib es hier keinerlei Beschränkungen.

4. BYOB und OOP

Ein wesentliches Konzept moderner Programmiersprache ist die Möglichkeit, die Welt durch Objekte und Klassen zu beschreiben. Das sind für Lernende zwei sehr unterschiedliche Dinge. Beachtet man, dass der normale Weg der Erkenntnis vom Konkreten zum Abstrakten verläuft, dann ist es etwas seltsam, mit dem abstrakten Konzept der Klassen zu beginnen, um erste, einfache Objekte z. B. in *Java* erzeugen zu können – und das auch noch als enormen Vorteil zu verkaufen. In *BYOB* haben wir beides: Objekte mit ihren Attributen und Methoden (Skripten) sowieso, geerbt von Scratch, und Klassen natürlich auch. Für den Anfangsunterricht in allen Jahrgangsstufen incl. der Universität erscheint es uns am natürlichsten zu sein, Objekte mit den gewünschten Eigenschaften auszustatten und dann zu klonen. Abstraktere Konzepte folgen, wenn sie benötigt werden, also in der Situation den Lernenden als sinnvoll erscheinen.

Als Beispiel wählen wir einen Schwarm von Fischen. Der besteht aus einzelnen, sehr ähnlichen Tieren, zumindest aus einem. Dieser Fisch schwimmt etwas ziellos in der Gegend herum und flieht ggf. panisch vor einem Hai. Wer mag diese Viecher schon.



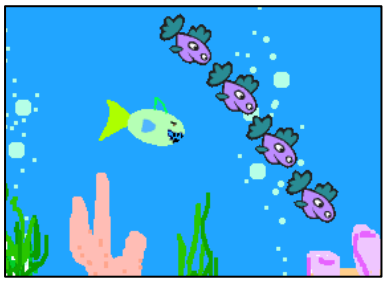
Ein Fisch bildet natürlich noch keinen Schwarm. Wir klonen deshalb unseren Fisch mehrfach, dann schwimmen alle Fische unabhängig von einander in der Gegend herum. Ein trauriges Bild von einem Schwarm! Unsere Fische haben nur die Methoden des Original-Objekts *geerbt*. Neu lernen müssen alle zusätzliches Verhalten, das zur Schwarmbildung führt. Z. B. können sie jeweils auf einen anderen, bestimmten Fisch zu schwimmen – aber 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, … Alle diese Verhaltensmuster führen zu unterschiedlichen Schwärmen. Ein Schwarm besteht natürlich aus einzelnen Fischen, einzelnen Objekten. Trotzdem handelt es sich bei ihm um etwas Neues, Abstrakteres. Spätestens, wenn viele Fische ins Spiel kommen, können wir nicht mehr das Verhalten der Fische *Hansi*, *Petra*, und *Herbertchen* betrachten. Wir benötigen neue Strukturen, die den Schwarm als Ganzes beschreiben, und schreiten so „forschend" im Lernprozess fort.

Da in *BYOB* Blöcke als Daten interpretiert werden können, impliziert das Verfahren eine elegante Form, Klassen zu implementieren: Methoden sind spezielle Attribute. Unterschiedliche Objekte einer Klasse unterscheiden sich in den „normalen" Attributen wie Position, Größe, Farbe, …, besitzen aber den gleichen Satz von „Methoden-Attributen", die man bei Bedarf über eine Nachricht aktiviert, indem sie ähnlich wie bei der for-Schleife über *call-* oder *run-*Anweisungen evaluiert werden. Leitet man von solch einer 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 findet man z. B. in (BYOB-REF, 2010).

5. BYOB und Abitur

Sichtet man die Zentralabituraufgaben 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 auf dem Papier ziemlich unpraktisch, *BYOB*-Programme zu zeichnen. Da diese aber sowieso keine Syntaxfehler zulassen und Algorithmen abbilden, die ebenso durch Struktogramme oder UML-Diagramme beschrieben werden können, muss man fragen, ob die Entwicklung von Algorithmen auf dem Papier nicht sowieso besser in dieser Form erfolgt.

6. 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, 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 er konstituierend für das Fach und kann nicht einfach klammheimlich gestrichen werden. Mit grafischen Programmiersystemen, speziell *BYOB*, stehen uns heute Werkzeuge zur Verfügung, den Anspruch „Informatik für alle“ auch im Bereich der Algorithmik erfolgreich umzusetzen. Inhaltlich sind uns durch die Verwendung solcher Systeme keine Grenzen gesetzt, im Gegenteil. Fassen wir den Begriff „Programmieren“ weit, beschreiben damit den gesamten Prozess von der Modellierung über die algorithmische Beschreibung zur Implementation, dann bilden zwar die beiden ersten Schritte den anspruchsvollen, im eigentlichen Sinn „bildenden“ Teil des Prozesses. Die Motivation, diesen steinigen Pfad 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 spezielle Wert der Schulinformatik. Systeme wie *BYOB* gestatten es nun, diesen letzten Schritt 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 aber wesentlich mehr Zeit als früher zur Verfügung. Damit sind wir in einer ähnlichen Situation wie bei der Einführung der GUI-Builder: der für die Gestaltung der Benutzeroberfläche erforderliche Zeitbedarf wurde auch dort marginalisiert, die gewonnene Zeit steht für inhaltlich anspruchsvollere Aufgaben zur Verfügung. Interessanterweise gibt es aber auch heute noch zahlreiche Kolleginnen und Kollegen, die meinen, GUI-Builder wie Delphi, NetBeans oder Eclipse als „Klicki-Bunti-Programmierung“ verspotten zu müssen.

Damit stellen sich einige für die Schulinformatik 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 BYOB als Werkzeug implementiert werden können? Ist die damit 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 Programmierunterricht? Soll er beibehalten oder vielleicht gestoppt, 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“ benutzt, 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*, … im Unterricht benutzt werden? Braucht der Informatikunterricht Ausbildungswerkzeuge ähnlich wie die Naturwissenschaften, die auch fast ausschließlich Gerätschaften benutzen, die man außerhalb der Schule kaum findet.

Diese Fragen sind ernsthaft gemeint. Sie stellen sich erst jetzt aufgrund der neu zur Verfügung stehenden Technologie, sie können jetzt präziser gestellt werden, weil es Alternativen gibt. Die Antworten werden unsere Sicht auf einen wichtigen Bereich der Schulinformatik und damit unsere Argumentation schärfen. Wir werden sie nicht am Schreibtisch, sondern durch Erfahrungen in der Schulpraxis finden, wahrscheinlich jeder etwas anders. Die Unterrichtser-

fahrungen sind aber wichtig, weil erst auf dieser Basis ein sachlicher Diskurs geführt werden kann. Also: probieren Sie es aus!

Literatur und Internetquellen

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.

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

STR – Strecker, Kerstin: Informatik für Alle - wie viel Programmierung braucht der Mensch?, Dissertation 2009, http://webdoc.sub.gwdg.de/diss/2009/strecker/strecker.pdf

BYOB2 – Moenig, Jens: BYOB 2.0, August 2009, http://chirp.scratchr.org/dl/BYOB%202.0.pdf

BYOB3 – Moenig, J., Harvey, B.: BYOB 3.0 – Build Your Own Blocks, August 2010, http://byob.berkeley.edu/

BYOB-REF – Harvey, Brian: BYOB Reference Manual, 2010, http://byob.berkeley.edu/BYOBManual.pdf

ABSUS – Abelson, H., Sussman, G.: Struktur und Interpretation von Computerprogrammen, Springer 2001

AP Principles: http://www.csprinciples.org/CSPrinciples0310.pdf

Alle Internetquellen wurden zuletzt am 27.8.2010 geprüft.

Prof. Dr. Eckart Modrow
Max-Planck-Gymnasium
Theaterplatz 10
37073 Göttingen
E-Mail: emodrow@informatik.uni-goettingen.de

Jens Mönig
Lange Str. 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