Visuelle Programmierung - oder: Was lernt man aus Syntaxfehlern?

Eckart Modrow

Didaktik der Informatik
Universität Göttingen
Goldschmidtstrasse 7
37077 Göttingen
emodrow@gmx.de

Abstract: Visuelle Entwicklungsumgebungen gibt es schon lange, weil sich durch die Vermeidung von Syntaxfehlern einerseits Entwicklungszeiten verringern lassen, andererseits Programmieranfängern die Möglichkeit gegeben wird, sich schnell auf die inhaltlichen Aspekte des Programmierens zu konzentrieren. Allerdings waren diese Systeme meist auf enge Anwendungsbereiche beschränkt. Mit neuen, sehr viel universeller einsetzbaren Werkzeugen wie BYOB[1] lassen sich alle Bereiche der Schulinformatik oder einer Anfängervorlesung abdecken. Damit stellt sich die Frage, ob sie nicht in absehbarer Zeit die klassischen textbasierten Sprachen ablösen können – und welche Konsequenzen das hätte. Der Beitrag beruht auf Unterrichtserfahrungen mit visuellen Entwicklungsumgebungen in den Klassenstufen 5 bis 13 sowie Vorlesungen für Programmieranfänger in den Naturwissenschaften.

1 Einleitung

In der Informatik gibt es zwei weitgehend unbestrittene Aussagen zum Bereich Algorithmik/Programmieren: 1) „Wichtig sind die Algorithmen, nicht deren Codierung." und 2) „Richtiges Programmieren ist textbasiert". Lange Zeit konnte nur textbasiert programmiert werden. Es wurde deshalb auch kaum hinterfragt, welche Bedeutung die Codierung selbst für die Allgemeinbildung haben könnte, denn wenn die Implementierung von Algorithmen einen Bildungswert besitzt, dann war die Codierung nicht zu vermeiden. Leider ist das so wenig angesehene Codieren aber ausgerechnet die Haupthürde für Programmieranfänger.

[1] [MH10]

Fordert man, dass nach Abschluss eines wie auch immer gearteten Programmierlehrgangs eine Programmiersprache als Werkzeug zur Formulierung und Implementierung eigener Ideen und Problemlösungen genutzt werden kann, dann scheitert der überwiegende Teil der Anfänger, bis zu 80%, an dieser Hürde sowohl im Informatikunterricht der Schulen[2] wie auch an den Hochschulen. Man muss das wohl fordern, denn das reine Lesen und Nachvollziehen fertiger Programmtexte erscheint sinnlos, weil Algorithmen genauso gut, meist besser, in anderen Notationsformen formulier- und vermittelbar sind – und auf die kommt es ja angeblich nur an. Es ist also kein Zufall, dass auf vielen Ebenen versucht wird, diesen Zustand zu verbessern. Reduzierte Modellsysteme mit eingeschränkten Befehlssätzen, Programmgeneratoren, vereinfachte Sprachen wurden und werden eingeführt – und eben visuelle Entwicklungssysteme. Deren Nutzung wollen wir etwas genauer betrachten.

2 Zur Akzeptanz visueller Entwicklungssysteme

Visuelle Entwicklungsumgebungen gibt es sehr unterschiedlicher Art: als Vertreter der Zustandsmodellierung mögen die Komponente AutoEdit aus AtoCC, Kara oder der schöne USBomat[3] von Helmar Becker dienen. Sie steuern ebenso wie blockorientierte Systeme den Kontrollfluss, aber diese kommen der traditionellen Programmierung sehr viel näher, egal ob sie struktogrammartige Strukturen wie die erste LEGO-Mindstorms-Version und Scratch oder Flussdiagramme benutzen, wie sie besonders im technischen Bereich verbreitet sind. Sie scheinen damit auch geeigneter, auf die textbasierte Programmierung vorzubereiten – wenn das denn erforderlich ist. Etwas aus dem Rahmen fallen Systeme wie LabView und die daraus abgeleiteten neueren LEGO-NXT-Systeme, die eher datenflussorientiert arbeiten, sowie Sonderwege wie das Smalltalk-basierte eToys oder das 3D-Entwicklungssystem Alice[4], das die Erfolgsquote an der Carnegie-Mellon-Universität bei den Programmieranfängern in etwa verdoppelt hat. Dass es sich nicht um Spielzeuge handeln muss, zeigt die Steuerung des südafrikanischen Großteleskops SALT, die komplett über LabView erfolgt. Entsprechendes lässt sich über die Aktivitätsdiagramme von UML sagen, die ebenfalls für den professionellen Einsatz gedacht sind und dort auch zunehmend Verwendung finden.

Man soll nur nicht glauben, dass neue Entwicklungen immer auf ungeteilte Begeisterung stoßen! Neue Techniken entwerten altes Expertenwissen – und das gefällt meist nur den Laien; die Experten finden das gar nicht witzig. Neu ist diese Beobachtung nicht: schon der Übergang von Assemblern zu universellen Programmiersprachen wurde kritisch beäugt, und ältere Kolleginnen und Kollegen werden sich an den nicht nur ironisch gemeinten Aufsatz „Echte Programmierer meiden Pascal“ erinnern, der das Aufkommen der strukturierten Programmierung als „Müslifresser-Programmierung“ kommentierte[5]. Folgerichtig gehören die GUI-Builder wie Delphi, NetBeans oder Eclipse in den Bereich des „Klicki-Bunti“.

---

[2] z. B. in [Hu00]
[3] [HW04], [NHR99], [Be08]
[4] [MI07], z. B. [Fi10], [CM99]
[5] z. B. in [Le83]

Wir können also nicht erwarten, dass die Abkoppelung der Implementierung schultypischer Algorithmen vom Gebrauch universeller Programmiersprachen von den Experten begrüßt wird. Wird sie auch nicht. Eine hübsche Reaktion eines hier ungenannt bleibenden Kollegen auf die Vorstellung von BYOB war: „Lernen muss weh tun!“ Besser kann man es nicht ausdrücken.

Was sagen nun die Betroffenen dazu?

Erstaunlicherweise sehen sie das sehr ähnlich. Programmieranfänger aus den Naturwissenschaften des dritten Semesters, die sowohl Java wie Scratch[6] kennen gelernt hatten, äußerten überwiegend[7], dass Scratch für den Einstieg außerordentlich wichtig gewesen wäre (m: 83%, w: 100%) und auch weiterhin parallel zu Java eingesetzt werden sollte, weil es für das Verständnis der Strukturen und Abläufe wichtig sei (m: 67%, w: 100%), außerdem mache es viel Spaß (m: 67%, w: 83%). Trotzdem sei es nicht „richtiges Programmieren“ (m: 33%, w: 67%), es fehlten wichtige Strukturen (m: 83%, w: 50%) und sei nur für den Anfang geeignet (m: 50%, w: 33%). Ihre persönliche Präferenz liege nach der Veranstaltung bei Java (m: 100%, w: 50%).

Dazu einige Zitate:

„Nach dem Arbeiten mit Java stellt sich allerdings das Gefühl ein, dass man mit Scratch nicht viel wirkliches Programmieren lernt. Man lernt bei Scratch weder etwas über Klassen oder Methoden, noch hinterblickt man die wirkliche Strukturierung eines Programms.“

„Scratch eignet sich … um …ein Verständnis für Algorithmik zu bekommen. …Dennoch empfinde ich es als wichtig, auch Java zu lernen, da viele Programme …damit geschrieben sind und es somit beim Verständnis von diesen hilft.“

„Ich finde Scratch gut, um die Denkweise …und den Aufbau eines Programmes kennenzulernen. Die Programmiersprache Java ist sehr wichtig, da mit ihr die meisten Programme geschrieben sind ….“

„(Zu Scratch:) Man versteht ziemlich schnell die Grundgedanken, die man fürs Programmieren braucht und alles ist sehr übersichtlich. Zudem macht Scratch es einem fast unmöglich Fehler zu machen. … (Zu Java:) Zudem ist es wahnsinnig frustrierend, wenn man fast alles richtig gemacht hat, das Programm aber dennoch nicht läuft, weil sich irgendwo ein kleiner Fehler eingeschlichen hat. …., aber danach …muss auch Java gezeigt werden, da es doch sehr viel mehr Möglichkeiten bietet.“

„Ich denke, dass Scratch … auch nicht … in Frage kommt, da es meiner Meinung nach für eine Präsentation durch den Lehrer äußerst unseriös wirkt.“

Fazit: Java is the real thing!

---

[6] BYOB war im Sommersemster 2010 noch in der Entwicklung.

[7] Es handelt sich um 12 Interviews (m:6, w:6), die typisch für die Reaktion der Teilnehmenden aus zwei Veranstaltungen waren. Die Zahlen sollen nur eine Tendenz illustrieren.

Die Schülerinnen und Schüler sehen das genauso. Einerseits arbeiten sie sehr gerne und erfolgreich mit Scratch, andererseits wollen sie nach einer Weile „richtiges Programmieren“ kennen lernen. Eine in beiden Systemen relativ erfahrene Schülergruppe der Klassenstufe 11 schilderte sehr eindringlich, dass sie z. B. bei der Einführung von Variablen und einfachen Datentypen mit Java das Gefühl hätten, der Maschine „viel näher“ zu sein und sich deshalb auch viel sicherer fühlten als bei visueller Programmierung. Sie schlugen sogar vor, aus diesem Grund mit Java zu beginnen (!) und danach Scratch für die „richtigen Arbeiten“ zu benutzen, weil das viel effizienter sei. Das Gefühl, etwas gut verstanden zu haben, ist entscheidend für den Lernerfolg. Schon deshalb sind diese Äußerungen sehr ernst zu nehmen. Die Einschätzungen ähneln der Diskussion in der Linux-Gemeinde zwischen den Anhängern der Konsolennutzung und den Freunden der grafischen Oberflächen. Beides sind „Shells“ unterschiedlichen Aussehens, aber gleicher Funktionalität – mehr nicht. Man mag das eine mögen oder das andere. Größere Unterschiede gibt es nicht, schon gar nicht in der „Nähe“ zur Maschine.

Betrachten wir einmal die sachlichen Grundlagen der Aussagen. Die entwickelten Java-Programme waren weder hinsichtlich ihrer Funktionalität noch in den algorithmischen Strukturen komplexer als ihre Scratch-Pendants – eher deutlich umgekehrt. Das hat die Schülergruppe auch richtig erkannt. Der Aufwand für ihre Erstellung war dagegen wesentlich umfangreicher. Die benutzten Datentypen waren in beiden Systemen vorhanden, Anfänger benötigen nun mal keine anspruchsvollen Strukturen. Das Argument, dass sehr viel Software in Java erstellt wurde, zählt natürlich nicht für die Anwender – denen kann die Entwicklungsumgebung egal sein. Da es keine Anschlussveranstaltung für die Studierenden gab, werden fast alle auch keine umfangreicheren Programme mehr schreiben. Wir müssen also wohl akzeptieren, dass Programmieranfänger, obwohl sie erfahren haben, dass z. B. Scratch für entsprechend gewählte Beispiele einfacher, verständlicher und fehlertoleranter, auch im Gebrauch viel schneller ist als ein Java-Entwicklungssystem, Scheinargumente heranziehen, um die textbasierte Programmierung zu rechtfertigen. Sie wollen textbasiert arbeiten – unabhängig von dessen Sinn. Wenn das so ist, dann haben wir ein echtes Lernhindernis, das die Vorteile der visuellen Programmierung völlig kompensieren kann. Können wir es überwinden? Sollen wir es überhaupt überwinden?

Ziehen wir eine erste Zwischenbilanz:

1. Wenn sich die Lernenden algorithmisches Problemlösen und die dafür erforderlichen Methoden und Erfahrungen aneignen sollen, dann kommt es nach einhelliger Auffassung auf die Algorithmik an, nicht auf die Codierung – und genau dafür sind visuelle Entwicklungssysteme wie Scratch/BYOB die geeigneteren Werkzeuge, zumindest in einem Anfängerkurs.

2. Praktisch alle Beteiligten sind sich gefühlsmäßig darüber einig, dass es „irgendwie doch“ auf die Codierung ankommt, die Experten vielleicht, weil sie ihr Expertentum gefährdet sehen oder schlicht aus Gewohnheit, die Lernenden vielleicht, weil sie Experten nach den vorliegenden Kriterien werden wollen, also Experten im Codieren.

3. Die langjährigen Erfahrungen zeigen, dass das Codieren zu einer eigentlich völlig inakzeptablen Misserfolgsquote führt, die aber trotzdem akzeptiert wird, oft sogar gewollt ist. Die neuen Möglichkeiten, diese Situation zu verbessern und zu denen wir noch kommen, werden von beiden Seiten kaum angenommen – beide Seiten wollen sie eigentlich nicht.

4. Die vorhandenen Vorbehalte, die wir allerdings auch schon in der Vergangenheit bei strukturellen Änderungen bei den Entwicklungssystemen vorfanden, mögen ihre Berechtigung haben. Diese ist bisher aber überhaupt nicht untersucht, insbesondere ist der eigenständige allgemeinbildende Wert textbasierter Codierung bisher nicht begründet, weil keinerlei Notwendigkeit für eine Begründung vorlag. Die Codierung hat sich ihre Bedeutung einfach von der Algorithmik, mit der sie – bisher – unzertrennbar verknüpft war, entliehen.

Vergleichen wir die Situation einmal mit der Motivierung für die Naturwissenschaften. Schülerinnen und Schüler werden in ein Schülerlabor gekarrt, einen außerschulischen Lernort, extrahieren dort Farbstoffe aus Gemüseteilen und tragen dabei einen weißen Kittel und – natürlich – eine Schutzbrille, obwohl die gar nicht nötig ist. Sie fühlen sich dadurch aber als kleine Chemiker, es kommt nicht auf das Extraktionsverfahren an, sondern auf die Schutzbrille, die äußeren Insignien des Wissenschaftlers. Die motivierende Wirkung der Aktion ist deutlich spürbar.

Vielleicht spielen die textbasierten Programmiersprachen eine ähnliche Rolle wie die Schutzbrille als eine Art Geheimschrift, die niemand lesen kann außer den Experten. Wenn dem so sein sollte, dann kommen wir allerdings mit rationalen Argumenten nicht mehr weiter. Wir hätten es mit Symbolen zu tun, mit Bildern für einen Berufszweig. Wenn kryptische Geheimschrift das Symbol für die Informatik ist, was – nebenbei – ziemlicher Unsinn wäre, dann wäre die Ansicht der Lernenden verständlich, dass die Beherrschung dieser Geheimschrift zum Initiationsritus der Informatik gehört, dass das Erlernen von z. B. Java unabhängig von dessen Sinnhaftigkeit den Informatiker kennzeichnet, dass erst das „richtige“ Informatik wäre. Weiterhin erklärte es auch wenigstens teilweise die Ablehnung des Fachs in weiten Kreisen. Geheimschriftexperten sind Sonderlinge, grenzen sich durch ihre Geheimnistuerei ab, liefern kein positives Bild für überwiegend offen und optimistisch in die Zukunft blickende Jugendliche.

Die Wertschätzung der textuellen Codierung besonders im Schulbereich liefert ein Bild des Faches, das der Wirklichkeit widerspricht. Statt die Breite der informatischen Anwendungsbereiche – von ‚A‘ wie Archäologie über ‚L‘ wie Lebenswissenschaften zu ‚Z‘ wie Zukunftsfragen – zu betonen, versperrt ein in der Berufswirklichkeit der meisten Informatiker nachrangiger Aspekt den Blick auf einen der vielfältigsten Berufe, der Kommunikations- und Teamfähigkeit, analytisches Denken, Anwendungsorientierung sowie Methoden- und Werkzeugbeherrschung erfordert und in fast allen Bereichen der Gesellschaft präsent ist. Nicht umsonst klagen die universitären Fachbereiche nicht nur über zu wenig Nachwuchs, sondern auch über den falschen. Wir sollten vielleicht besser am Bild der Informatik arbeiten als an immer neuen fachlichen Spezialitäten – und das können wir am besten, wenn all die genannten Aspekte unseren Unterricht prägen.

Um nicht falsch verstanden zu werden: Die Implementierung von eigenen Lösungsideen, das schrittweise Verbessern, auch der experimentelle Zugang zur Problemlösung, der daraus erwachsende Produktstolz und die Förderung selbstständigen Arbeitens sind zentrale Elemente eines lebendigen Informatikunterrichts. Wenn all dieses ohne Syntaxprobleme möglich ist – umso besser!

3 BYOB: Build Your Own Blocks

Der Erfolg von Scratch im Ausbildungssystem ist eigentlich verblüffend: ein für Kindergarten und Grundschule konzipiertes und designtes System wird nicht nur in den Sekundarstufen, sondern sogar in der universitären Grundausbildung erfolgreich eingesetzt – und das nicht in irgendwelchen Klitschen, sondern am MIT oder in Berkeley. Die Not muss also groß sein! Ein „Kindergartensystem" sollte eigentlich für die universitäre Ausbildung ungeeignet sein. Wird es trotzdem benutzt, dann hält ein Teil der Ausbilder die traditionellen Ausbildungswerkzeuge offensichtlich für noch ungeeigneter.

Die Einschränkungen von Scratch sind im den Bereichen Modularisierung und Datenstrukturen offensichtlich, obwohl lokale und globale Operationen und Variable sowie Zeichenketten und lineare Listen zur Verfügung stehen. Das wurde zum Anlass genommen, ein für die Grundausbildung an Universitäten geeignetes Werkzeug auf der Grundlage von Scratch zu entwickeln. Die Ziele sind im Artikel „Bringing 'No Ceiling' to Scratch: Can One Language Serve Kids and Computer Scientists?" [8] von Brian Harvey und Jens Mönig beschrieben. Mit BYOB ist es möglich, die informatischen Konzepte des Lehrbuchklassikers „Struktur und Interpretation von Computerprogrammen" von Abelson und Sussman[9] 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[10] 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.

Zentrales Werkzeug von BYOB ist ein Blockeditor, mit dessen Hilfe Konzepte wie strukturierte Zerlegung, geschachtelten Operationen, lokale Variable, Rekursion usw. anschaulich umgesetzt werden. Bei einem Block kann es sich einen neuen Befehl, eine Funktion oder ein Prädikat handeln. Parameter können an beliebiger Stelle eingefügt und bei Bedarf typisiert werden. Mehrere Blockeditoren können gleichzeitig offen sein. Im Bereich der Objektorientierten Programmierung bietet BYOB einen sehr viel verständlicheren Weg als z. B. Java: man kommt von den Objekten zu Klassen, vom Konkreten zum Abstrakten – und nicht umgekehrt. Da BYOB Lisp-Strukturen enthält, ist z. B. die Erzeugung neuer Kontrollstrukturen sehr einfach. Dass die gängigen deutschen Abiturthemen problemlos mit BYOB behandelbar sind, wurde an anderer Stelle gezeigt[11].

---

[8] [HM10]
[9] [AS01]
[10] [AP10]
[11] [MSM11]

BYOB stellt mit diesen und darüber hinausgehenden Features eine echte Alternative zu textbasierten Entwicklungssystemen dar und ist keineswegs auf den Anfangsunterricht beschränkt. Wir können mit diesem Werkzeug in einem sehr weiten Bereich implementieren, ohne uns um textuelle Syntaxprobleme zu kümmern. Da diese einen Großteil der Unterrichtszeit beanspruchen, hätten wir also sehr viel mehr Zeit als bisher – z. B. zum Implementieren! Wir können unterrichtsbegleitend immer dann schnell lauffähige Programme erzeugen, wenn algorithmisches Problemlösen gefragt ist. Das Werkzeug würde aus dem Fokus verschwinden – so, wie es sein soll.

Wir wollen hier nicht behaupten, dass man deshalb jetzt das Werkzeug wechseln muss, aber wir denken schon, dass man ernsthaft darüber nachdenken kann. BYOB ist also ein Anlass, darüber zu reflektieren, was und weshalb wir etwas in diesem Bereich tun – und das ist doch schön!

4 Was ist nun mit der Syntax?

Natürlich sind Systeme wie BYOB derzeit noch Modellsysteme, anhand derer die Konzepte und Methoden der Informatik übersichtlich und inhaltsorientiert vermittelt und erprobt werden können, und zwar im gesamten Bereich der Schulinformatik. Dafür sind sie gemacht. Nicht gedacht sind sie als echte Produktionssysteme. Für Anwendungsprobleme benötigt man wohl auf absehbare Zeit Entwicklungssysteme wie Eclipse oder Visual Studio mit den entsprechenden textuellen Sprachen, und für einige Gebiete sind diese Werkzeuge natürlich auch einfach besser geeignet.

Wir sind der Meinung, dass die Informatikdidaktik zu begründen hätte, weshalb sich alle Informatikschülerinnen und –schüler mit den Syntaxproblemen textbasierter Programmierung herumschlagen müssen, wenn es dazu Alternativen gibt, die den Schwerpunkt der Arbeit drastisch von der Codierung der Lösung hin zur Entwicklung und Verbesserung von Ideen verlagern. Es wäre zu zeigen, wo der allgemeinbildende Wert textueller Codierung liegt, wenn die Vorteile dieses Verfahrens für die Masse der Lernenden niemals zum Tragen kommen, und es wäre zu zeigen, dass dieser Wert die dadurch zumindest mitverursachte unglaubliche Ausstiegsquote der fast immer freiwillig zu uns Kommenden rechtfertigt.

Andererseits hat die Schule neben der Vermittlung inhaltlicher Kompetenzen auch eine Orientierungsfunktion. Die Lernenden sollten im Schulsystem die Gelegenheit bekommen, sich auf unterschiedlichen Gebieten zu erproben und begründete Perspektiven für ihren Lebensweg zu entwickeln, z. B. bezüglich ihrer beruflichen Orientierung[12]. Wenn die textbasierte Programmierung in der fachlichen Informatikausbildung auch eine wesentliche Rolle spielt, dann sollten diejenigen Schülerinnen und Schüler, die sich für diesen Weg entscheiden, wissen, worauf sie sich einlassen. Sie sollten erproben können, ob ihnen diese Tätigkeit liegt. Dieser Aspekt ähnelt dem Vorgehen im Physikunterricht der Oberstufe: die Physik wird weitgehend ohne große mathematische Vorkenntnisse vermittelt, aber die Lernenden sollten an geeigneten Stellen erfahren, dass Physiker später Mathematik brauchen, sie beherrschen und ihre Anwendung auch mögen müssen.

[12] [Mo05a]

Weiterhin geben die informatischen Entwicklungswerkzeuge einigen Schülerinnen und Schülern die Möglichkeit, sich unabhängig von der beruflichen Entwicklung auf diesem Gebiet zu entfalten. Auch diese brauchen ihre Chance!

Wann und wie also kommen wir zu den textuellen Sprachen? Ein sequentielles Vorgehen, also erst BYOB, dann z. B. Java, erscheint uns ungeeignet. Gegenüber der Mächtigkeit und den Anwendungsmöglichkeiten von BYOB kann eine nach anspruchsvollen BYOB-Modellierungen eingeführte textbasierte Sprache nur verlieren, weil die Bedeutung der dann wieder herangezogenen Anfängerproblemchen hinter dem Berg von Syntax- und Hantierungsproblemen verschwindet und sie eigentlich nur demonstrieren, dass textbasierte Programmierung dafür völlig unangemessen ist.

Die Benutzung reduzierter Sprachen mit vereinfachter textueller Syntax und/oder vereinfachtem Befehlssatz scheint auch nicht der rechte Weg zu sein, wenn wir zu universellen Produktionssystemen hin wollen, und für den Einsatz von voll entwickelten Sprachen innerhalb virtueller Miniwelten wie bei JavaKara oder Greenfoot gilt das Gleiche, weil sie den Spracherwerb in den Mittelpunkt stellen und nicht die Algorithmen – wenn diese an anderer Stelle wesentlich übersichtlicher implementiert werden können. So schön z. B. Greenfoot auch ist – es wird genauso wenig und aus den gleichen Gründen nicht als „richtige Informatik“ akzeptiert wie BYOB.

Wenn wir textbasierte Sprachen im Unterricht parallel zu visuellen nutzen, dann muss deren Gebrauch aus der Situation heraus den Lernenden sinnvoll erscheinen – und das ist ein hartes Kriterium! Weiterhin müssen die ersten Beispiele so einfach sein, dass die Syntaxprobleme nicht überwiegen. Gehen wir davon aus, dass anfangs nur visuell und am Ende der Ausbildung in nennenswertem Umfang textbasiert programmiert wird, dann muss sich die textuelle Form langsam „einschleichen“, also an sinnvollen Stellen eingeführt werden, an denen ihre Vorteile augenfällig sind – und die gibt es natürlich:

- Wenn es auf das „Look-and-feel“ echter Anwendungen ankommt, also den Entwurf von Oberflächen, die den gewohnten gleichen, dann liefern GUI-Builder wie gehabt einen einfachen, leicht gangbaren Weg, der bei den ersten Arbeitsschritten der visuellen Programmierung gleicht. Der Übergang ist fließend und die Ausstattung der Eventhandler mit Funktionalität kann auf elementarstem Niveau geschehen, etwa bei der Rekonstruktion vorgefundener Elemente.[13]
- Wenn es auf den Umgang mit „richtigen“ Formeln ankommt, also z. B. bei mathematischen Problemen und ggf. ihrer Verknüpfung mit grafischer Visualisierung, die zusätzlich erheblichen Rechenaufwand erfordert, zeigt sich die Leistungsfähigkeit textueller Hochsprachen sofort. Das Gleiche gilt z. B. für durch zahlreiche Fallunterscheidungen umfangreiche Programme, die nicht anspruchsvoller als ihre kürzeren Verwandten, kaum modularisierbar, aber bei grafischer Darstellung irgendwann völlig unübersichtlich sind. Hier zeigen sich dann langsam die Vorteile der textuellen Repräsentation.

[13] Beispiele dazu z. B. in [Mo05b]

- Wenn es auf die Nutzung externer Expertise etwa durch die Einbindung von Bibliotheken, Zugriff auf externe Ressourcen wie Datenbanken usw. ankommt, benötigt man Systeme, die diese Integration erlauben. Python mit seinen zahlreichen Schnittstellen nach außen ist dafür ein gutes Beispiel.[14]

Für dieses Vorgehen gibt es noch einen weiteren guten Grund. Wir haben in der Vergangenheit mehrfach Situationen gehabt, in denen das in den Schulen vorhandene informatische Wissen durch technische Umwälzungen schlagartig entwertet wurde. Ein Grund dafür war, dass es kaum eine didaktische Begründung für das tradierte Vorgehen gab, für das neue natürlich auch nicht, und deshalb die letzten informatischen Entwicklungen sofort als neues „Paradigma" verkündet wurden. Die fehlende Didaktik verhinderte ein Bewerten und Abwägen der alten gegen die neuen Chancen und einen geordneten Übergang. Die Schulinformatik hat bisher weitaus mehr Kolleginnen und Kollegen durch Frustration verloren als durch Pensionierung! Führen wir also eine für die Unterrichtenden gewohnte textuelle Programmiersprache behutsam an geeigneten Stellen ein, dann bleibt die Expertise auf diesem Gebiet den Schulen erhalten und die neuen Möglichkeiten stehen dazu nicht im Gegensatz, sondern sollten als sinnvolle Ergänzung erscheinen. Wie sich das Verhältnis dann weiter entwickelt, mag die Zukunft entscheiden. In jedem Fall brauchen wir keinen schmerzhaften Bruch.

5 Fazit

Obwohl einiges für einen Übergang zu visuellen Entwicklungsumgebungen spricht, wobei BYOB als ein möglicher Vertreter für diese anzusehen ist, steht diesem der Widerwille vieler am Lernprozess Beteiligter entgegen. Ohne hier eine Lösung zu finden, können wir den Übergang nicht vollziehen, denn es gibt durchaus abschreckende Beispiele für solch ein Vorgehen. Im Physikunterricht werden z. B. Schülerübungsgeräte benutzt, die man nirgends sonst als in der Schule findet. Diese Abkoppelung von der Erfahrungswelt der Unterrichteten könnte mit ein Grund für die Unbeliebtheit des Faches sein. BYOB sollte nicht in eine „Schülerübungsgerät-Rolle" gedrängt werden.

Wenn es stimmt, dass das Bild von Informatikern durch das Codieren geprägt wird, dann scheint uns hier der Schlüssel zu liegen. Der zumindest anfängliche Verzicht auf das Codieren schafft Zeit im Unterricht – sehr viel Zeit. Nutzen wir diesen Freiraum doch bitte nicht, um noch mehr „Stoff" in das Fach hinein zu quetschen. Nutzen wir die Zeit lieber für eine an die Lebenswelt angebundene Informatik, für eine Art „Informatik im Kontext". Die Erfahrungen in der Mittelstufe mit einer im Fächerverbund unterrichteten Informatik, hier: Astronomie und Informatik, die an anderer Stelle geschildert wurden[15], legen es nahe, dass so ein breites Interesse erreicht werden kann. Nutzen wir die Zeit für projektartiges Arbeiten, für Erfahrungen in Teamarbeit, für die Modellierung von „interessanten" Bereichen – und damit für vertieften Einblick in diese –, die so einer Algorithmisierung zugänglich gemacht werden. Nutzen wir die vorhandenen Werkzeuge klug, mit allen ihren Einschränkungen, um ihren angemessenen Einsatz zu planen.

---

[14] siehe dazu z. B. [Ol10]
[15] [Mo10]

Und zu dieser Angemessenheit der Werkzeuge sollte es zuerst an einigen Stellen, dann zunehmend gehören, dass textbasierte Systeme dort benutzt werden, wo dieses offensichtlich sinnvoll ist – und sonst eben nicht. Wenn dadurch das Bild des Informatikers vielfältiger und interessanter – eben realistischer – wird, wenn textuelle Programmierung als mächtig und für Spezialisten sinnvoll, aber nicht als alles überschattend erfahren wird, dann hätten wir wirklich etwas erreicht.

Literaturverzeichnis

Alle Internetquellen wurden zuletzt am 9.1.2011 geprüft.

[AP10] AP Principles: http://www.csprinciples.org/CSPrinciples0310.pdf

[AS01] Abelson, H.; Sussman, G.: Struktur und Interpretation von Computerprogrammen, Springer 2001

[Be08] Becker, H.: USBomat, 2008, http://hbecker.sytes.net/usbomat/

[CM99] Carnegie Mellon University, Alice 1999, http://www.alice.org/

[Fi10] Fischertechnik RoboPro Software, http://www.fischertechnik.de/desktopdefault.aspx/tabid-39/

[HM10] Harvey, B.; Moenig, J.: Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists?, Constructionism 2010, Paris

[Le83] Leeds, B.: Echte Programmierer meiden Pascal., DATAMATION 1983, http://www.bytecruncher.de/witze/compprg2.html

[HuU00] Humbert, L.: Bericht zur Lehrerausbildung Informatik, Königstein 2000 http://koenigstein.inf.tu-dresden.de/00/humbert2.html

[MH10] Moenig, J.; Harvey, B.: BYOB 3.0 – Build Your Own Blocks, August 2010, http://byob.berkeley.edu/

[MI07] MIT MediaLab, Scratch 2007, http://scratch.mit.edu/

[Mo05a] Modrow, E.: Informatikunterricht mit technischen Aspekten, MNU 58/7 2005

[Mo05b] Modrow, E.: Einführung in die Algorithmik, 2005, http://vlin.de/index.php?p=sek

[Mo10] Modrow, E.: Informatik als technisches Fach, LOG IN 163/164 2010

[MSM11] Modrow, E.: Strecker, K.; Mönig, J.: Wozu Java?, LOG IN 168, 2011 (im Druck)

[NHR99] Nievergelt, J.;Hartmann, W.; Reichert, R.: Kara, 1999, http://www.swisseduc.ch/informatik/karatojava/kara/

[Ol10] Oldenburg, R.: Programmieren mit Python in der Sek.1, 2010, http://www.hrpi.gi-ev.de/fileadmin/gliederungen/fg-hrpi/texte/python.pdf