Erfahrungen zur Individualisierung im Programmierunterricht

Peter K. Antonitsch
Alpen-Adria Universität Klagenfurt
und HTL Mössingerstraße
Peter.Antonitsch@uni-klu.ac.at

Individualisierung von Unterricht ist ein iterativer Prozess, in dem eine Vielzahl von Details zu beachten ist. Speziell für den Programmierunterricht besteht eine komplexe Abhängigkeit der Aufgabenkultur von der erst in Entwicklung befindlichen »Kultur der Kompetenzen«. Anhand von Teilerfolgen eines vom Autor selbst durchgeführten Individualisierungsversuches wird auf Möglichkeiten, Fallstricke und Werkzeuge bei der Individualisierung von Programmierunterricht hingewiesen.

1 Vom Programmieren zum Individualisieren

1.1 Der Lernbereich »Programmieren«

Nach dem derzeit gültigen österreichischen AHS-Lehrplan soll das Pflichtfach Informatik Schülerinnen und Schüler *„befähigen, [informatische und informationstechnische Grundkenntnisse] zur Lösung einer Problemstellung sicher und kritisch einzusetzen*.“ [LA04] Konkret im Hinblick auf das Programmieren findet sich im (noch) gültigen Lehrplan für das Pflichtfach Angewandte Informatik an Höheren Technischen Lehranstalten, Abteilung für Elektrotechnik als „Bildungs- und Lehr(!)- Aufgabe“: *„Der Schüler soll mit Hilfe einer höheren Programmiersprache einfache Probleme der Berufspraxis lösen können*.“ [LB97]. Ein wesentliches Ziel des Informatikunterrichts im Allgemeinen und des Programmierunterrichts im Speziellen ist also, die Problemlösekompetenz der Lernenden zu fördern.

»Problemlösen durch Programmieren« ist aber nicht gleich »Problemlösen«! In [Du92] wird darauf hingewiesen, dass es bei ersterem eben nicht darauf ankommt, das Problem selbst zu lösen, sondern die Problemlösung so aufzubereiten, dass diese durch einen »Akteur« (d.h. durch ein animiertes Graphikobjekt, durch ein abstraktes Programm bzw. in letzter Instanz durch eine »Maschine«) abgearbeitet werden kann. »Problemlösen durch Programmieren«, kurz: Programmieren, umfasst daher neben dem Verstehen des zu lösenden Problems und dem Finden einer Lösung vor allem auch die Kenntnis der »Fähigkeiten« bzw. »Beschränkungen« des Akteurs, der das Problem lösen soll. Daraus ergibt sich bei Berücksichtigung moderner »Programmierumgebungen« für den Lernbereich Programmieren die in Abb.1 dargestellte Grobstruktur.

1.2 Didaktische Reduktion im Programmier-Anfangsunterricht

Für das Weitere ist die Unterscheidung von Problem und Aufgabe wichtig. Nach Renate Girmes markiert eine Aufgabe eine Lücke, die sich dem/der Einzelnen *„als Spannung zwischen Sein und Sollen*“ darstellt und die er/sie schließen möchte. Dabei enthält *„eine Lücke, die als sich stellende Aufgabe wahrgenommen werden kann, [...] immer schon eine Idee oder Vorstellung von Möglichkeiten der Lückenschließung*.“, während *„[...] beim Problem die*

Vorstellung einer möglichen und sinnvollen Lösungsperspektive noch fehlt oder zu diffus ist [...]“ [Gi07, S. 19f].

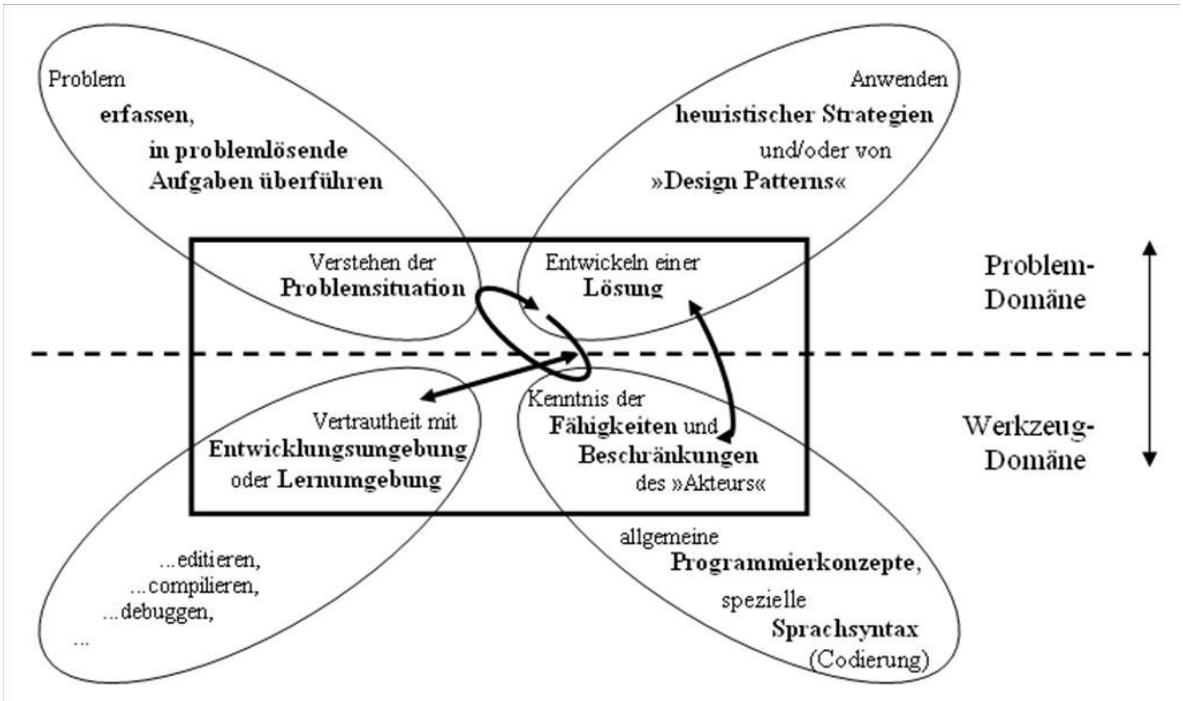


Abbildung 1: Grobstruktur des Lernbereichs »Programmieren« unter Hervorhebung des Problemlösekreislaufs sowie der Abhängigkeit von Lösung(sansatz) und dem jeweiligen Werkzeug. Die vier zentralen Aspekte (Kompetenzen ?) des Programmierens sind durch den rechteckigen Rahmen gekennzeichnet.

Im typischen Problemlöseprozess wird ein zu lösendes Problem in lösbare Aufgaben aufgeteilt (vgl. Abbildung 1). Im Programmier-Anfangsunterricht wird dieser Übergang vom Problem zur Aufgabe verkehrt: Dort haben die Lernenden das Problem (!) zu lösen, die ursprüngliche (»eigentliche«) (Programmier-)Aufgabe (!) einem »Akteur« zur Lösung zu übergeben. Das Problem liegt darin begründet, dass die Lösung der Programmieraufgabe durch die Sprache bestimmt ist, die der jeweilige Akteur »versteht«, andererseits aber die (Struktur dieser) Sprache nur erlernt werden kann, indem Lösungen in dieser Sprache formuliert werden! In Unkenntnis der (Programmier-) Sprache haben Programmieranfänger aber zunächst meist keine „Vorstellung einer möglichen und sinnvollen Lösungsperspektive“ – die Aufgabe wird dadurch zum Problem.

Im Programmierunterricht muss folglich früh gelernt werden, mit dem jeweiligen »Akteur« in der »Sprache« zu kommunizieren, die dieser »versteht«. Der zuvor geschilderten Pattsituation wird in der didaktischen Praxis »klassischen« Programmierunterrichts dadurch begegnet, dass der Problemlöseprozess zunächst auf den Aspekt der Sprache reduziert wird und dabei die zu lösenden Aufgaben bewusst »einfach« gewählt werden.

1.3 Individualisierung und das Problem der »einfachen« Aufgaben

Individualisierung wird nachfolgend verstanden als „*Unterricht, der darauf abzielt, den jeweils unterschiedlichen Lernbedürfnissen der einzelnen Schülerinnen und Schüler Rechnung zu tragen [...]*“ [BS09, S. 114]. Individualisierung verlangt daher (unter anderem) nach Lern-

aufgaben, die einerseits von den Lernenden individuell als sinnvoll wahrgenommen werden. Andererseits müssen die Lernenden über das zum Lösen der jeweiligen Aufgabe benötigte Vorwissen verfügen. Diese beiden Bedingungen definieren nach Ansicht des Autors ein Kernproblem von Individualisierung im Programmierunterricht:

Gemäß der zuvor skizzierten didaktischen Reduktion des Problemlöseprozesses werden Lernende im Programmierunterricht zunächst mit Aufgaben konfrontiert, die sie für sich meist nicht als ernstzunehmende Aufgaben wahrnehmen: Das Bestimmen der größten von drei Zahlen oder auch das Sortieren einer Liste (moderater Länge) sind zwar für den Anfangsunterricht ernstzunehmende Codierungsprobleme, die allerdings ohne größeren Aufwand SELBST gelöst werden können und als solche für die Lernenden „*banale und langweilige*" Aufgaben sind (vgl. [Du92]): Die durch die Aufgabenstellung »vorgetäuschte« „*Lücke zwischen Sein und Sollen*" kann von den Lernenden leicht geschlossen werden, ohne dass die vom Lehrenden (!) intendierte Aufgabe – die Formulierung eines entsprechenden Programms – von den Lernenden gelöst wird. Wenn diese die Aufgabe dennoch übernehmen, hat dies meist wenig mit selbstgesteuertem Lernen, sondern eher mit »Programmieren der Lernenden« zu tun. Nachhaltiges Lernen setzt aber die individuelle Lernbereitschaft voraus, sodass die von den Lernenden als sinnarm empfundenen Aufgaben des Anfangsunterrichts häufig fehlendes Vorwissen zur individuellen Bearbeitung von Aufgaben im weiteren Programmierunterricht nach sich ziehen.

1.4 Individualisierung im Programmierunterricht?

Aus pädagogischer Sicht basiert Individualisierung von Unterricht auf kooperativen Lernarrangements (vgl. [BS08], [BS09]), obige Überlegungen betonen zudem die Bedeutung geeigneter Lernaufgaben. Im Informatikunterricht werden Lernarrangements (meist) durch die verwendete Software mitbestimmt. Demnach liegt eine fachdidaktische Herausforderung bei der Individualisierung im Programmierunterricht darin, Programmier-Lernwerkzeuge auszuwählen, die helfen, lernzielspezifische Lücken für die Lernenden als Aufgaben sichtbar zu machen, und mit denen die Lernenden auf Basis ihrer jeweiligen Vorkenntnisse die wahrgenommenen Aufgaben kooperativ lösen können.

Lernzielspezifische Lücken im Programmierunterricht manifestieren sich in Fragen der Art: „Wie bringe ich [...den jeweiligen Akteur...] dazu, dies oder jenes zu tun?" und sollten idealerweise in explorativem Ausloten der Lösungsmöglichkeiten münden. Einen Startpunkt für Individualisierung stellen daher Programmier-Lernwerkzeuge dar, die

- die Identifikation des/der Lernenden mit dem jeweiligen Akteur fördern,
- die heuristische Strategie von »Versuch und Irrtum« unterstützen und
- in der Anfangsphase allgemeine Programmierkonzepte von den Syntaxproblemen der Programmiersprache trennen.

2 Individualisierung im Programmierunterricht!

Richard Bornat et al. beschreiben die Lücke, die Lehrende (und wohl auch Lernende) im Programmierunterricht wahrnehmen können, wie folgt: „*Learning to program is notoriously difficult. Substantial failure rates plague introductory programming courses the world over […]*" ([BDS08]). Die daraus ableitbare Aufgabe, den Programmierunterricht so zu verändern, dass die angestrebten Grundkompetenzen von den Lernenden tatsächlich erworben werden können, ist ein Kernthema fachdidaktischer Forschung (vgl. z.B. [Pap85], [Pat95], [Kö03],

[RNH04] oder [An05]) und eine ständige Herausforderung in der Unterrichtspraxis des Fachs Informatik. Nachfolgende Reflexionen beziehen sich auf eigene Erfahrungen des Autors, dieser Herausforderung in einem ersten Jahrgang der Höheren Technischen Lehranstalt (9. Schulstufe) mit dem im zuvor skizzierten Ansatz der Individualisierung zu begegnen.

2.1 Rahmenbedingungen

Die vom Autor im Schuljahr 2009/10 im ersten Jahrgang (mit einer Doppelstunde Informatik) unterrichtete Lerngruppe bestand aus 14 Schülern und 2 Schülerinnen, die keine vorherige Programmiererfahrung hatten. Da Planung individualisierten Unterrichts jedenfalls von den Voraussetzungen der Lerngruppe ausgehen soll, wurde der im Lehrplan genannte Lehrstoff „*Lösung einfacher Probleme durch Algorithmen. Umsetzung in Programme*." wie folgt konkretisiert: Die Schülerinnen und Schüler sollen lernen,

- Aufgaben wahrzunehmen, die mit Hilfe der verwendeten Programmierumgebung gelöst werden können;
- grundlegende Programmierkonzepte wie das Variablenkonzept, Zuweisung, Verzweigung, Wiederholung oder Modularisierung anwenden können, um Aufgaben mit Hilfe eines Programms zu lösen;
- Problemsituationen in problemlösende Aufgaben aufzuteilen und diese mit Hilfe eines Programms zu lösen.

Bei der Auswahl der Programmier-Lernumgebung(en) waren einerseits die Überlegungen aus 1.4 zu berücksichtigen. Schulspezifisch wird andererseits erwartet, dass Schülerinnen und Schüler nach dem ersten Informatik-Lernjahr einfache Programme mit Java oder C# codieren können. Aus diesem Grund erschien zunächst die Verwendung der »Greenfoot«-Umgebung sinnvoll, in der Graphikobjekte als »Akteure« durch Java-Code zum Lösen von Aufgaben innerhalb einer frei gestaltbaren »Mikrowelt« programmiert werden können (vgl. [Gr06]). Allerdings erlaubt dieser Zugang nicht die für den Anfangsunterricht als sinnvoll erachtete Trennung von allgemeinen Programmierkonzepten und der Sprachsyntax. Daher wurde ein »hybrider« Zugang gewählt: Der Erstkontakt der Lernenden mit dem Programmieren erfolgte mit der Lernumgebung »Scratch«, da diese die in 1.4 formulierten Anforderungen ideal erfüllt (vgl. [Sc07]). Diese Umgebung hat jedoch den Nachteil, dass der für die Programmausführung erzeugte »Code« nicht in textueller Form verfügbar ist, weswegen nach vier Monaten der Umstieg auf »Java mit Greenfoot« für sinnvoll erachtet wurde. Für die didaktische Konzeption von Scratch und Greenfoot sei z.B. auf [Re09] bzw. [Kö10] verwiesen.

2.2 Individualisieren mit Scratch: Lücken machen Programmieren interessant!

Programmieren mit Scratch bedeutet das Kombinieren graphisch repräsentierter Programmblöcke, die wie »Puzzleteile« zusammengefügt werden. Rückmeldung darüber, ob die programmierte Lösung die jeweilige Aufgabe tatsächlich löst, erhalten die Lernenden unmittelbar durch das Verhalten eines Akteurs/ mehrerer Akteure (»Sprites«), der/die durch das jeweilige Programm gesteuert wird/werden (Abbildung1). Die Lernenden können so durch „Versuch und Irrtum" das Programmverhalten studieren, anhand »codefreier« Beispiele die Semantik von (Kontroll- und auch Daten-) Strukturen erfahren und durch wahrgenommene »Lücken« eigene (!) Lernaufgaben formulieren.

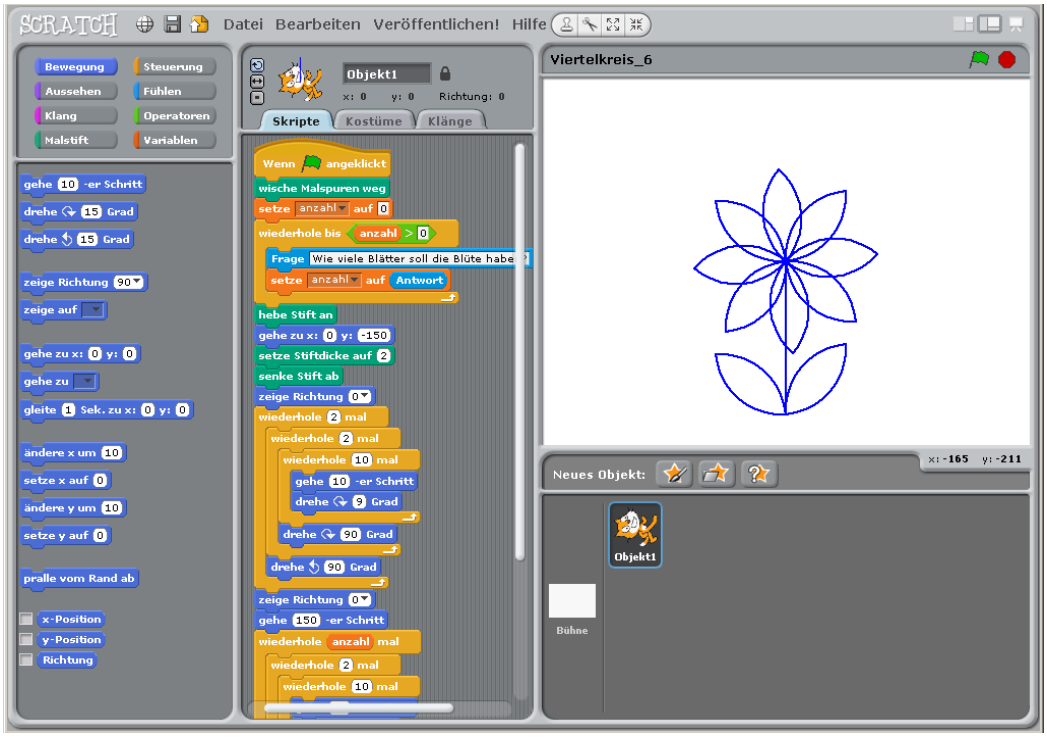
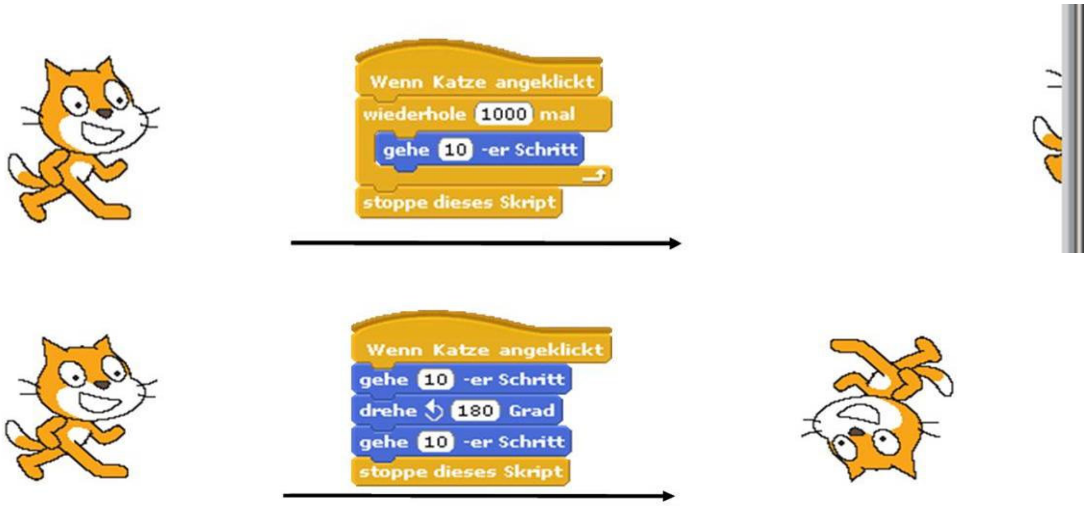


Abbildung 2: Die Scratch-Lernumgebung mit den »Programmier-Puzzleteilen« (links), dem Programmeditor (mitte) und dem Ausgabefenster, in dem das »Verhalten« des »Akteurs« sichtbar gemacht werden kann (rechts).

Beispielhaft für das Entstehen von Lücken steht die folgende Erfahrung der Lernenden beim »Erstkontakt« mit Programmieren: Beim Versuch, den Akteur sich horizontal über den Ausgabebildschirm bewegen zu lassen, bleibt dieser am Bildschirmrand »stecken«; dreht er sich – programmgesteuert – um (damit er nicht »stecken bleibt«), so steht er plötzlich »am Kopf« (Abbildungen 3 und 4).



Abbildungen 3, 4: Lücken zwischen „*Sein und Sollen*“, die sich bei Scratch aus einfachsten Programmen ergeben: Der »Akteur« (hier eine Katze) bleibt unerwartet am Bildschirmrand »stecken« oder steht plötzlich Kopf. Zudem wird der Befehl »gehe 10-er Schritt« mit einem »großen Schritt« anstelle von 10 »kleinen Schritten« ausgeführt. Der »Akteur« »verhält« sich anders als erwartet!

Versierte Programmierer mögen diese »Lücken« für trivial befinden, die Lernenden formulierten aber durch Fragen der Art „Wie kann die Katze nach Erreichen des rechten Bildschirmrandes am linken Rand wiedererscheinen?“, „Wie kann ich es erreichen, dass die Katze nach dem Umdrehen nicht am Kopf steht?“ oder: „Wie bringe ich die Katze dazu, 10 kleine Schritte zu machen?“ individuell für sinnvoll und herausfordernd erachtete Aufgaben. Die Aufarbeitung dieser Fragen führte auf die Themen »Verzweigungen«, »Schleifen«, »Bildschirm-Koordinatensystem«, sowie zur Behandlung des integrierten Graphik-Editors und konnte beispielgestützt innerhalb der restlichen Unterrichtseinheit abgeschlossen werden.

2.3 Individualisieren mit Scratch: Provozieren von Lücken

Das neu erworbene Werkzeugwissen motivierte die Schülerinnen und Schüler zum »weiterprobieren«, woraus sie für die nächstwöchige Unterrichtseinheit selbst (!) die Problemstellung entwickelten, die Katze »als Akteur« über den Bildschirm zu steuern. Die Rolle des/der Lehrenden wandelt sich in diesem Szenario zu der eines Coaches, der von den Schülerinnen und Schülern eingeforderte Informationen zur Verfügung stellt, oder besser noch, der die Lernenden anleitet, diese Informationen selbst zu erschließen. In traditionell verstandenem Unterricht fällt ihm/ihr auch die Aufgabe zu, zum Festigen der neu erworbenen »Fähigkeiten« Übungen anzubieten und auch durch das »Provozieren weiterer Lücken« das Erreichen der Lernziele zu verfolgen (vgl. Abbildungen 5 bis 7).

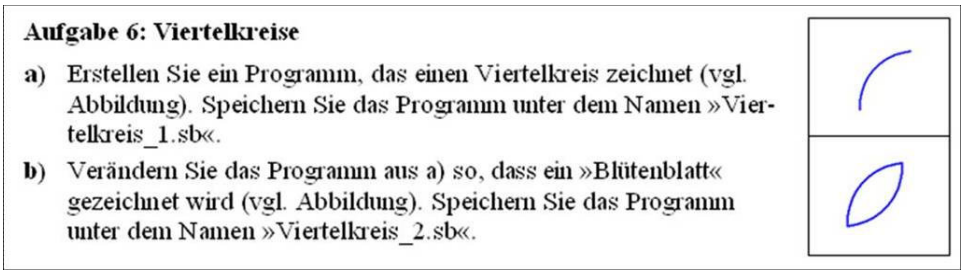


Abbildung 5: Aus einem Arbeitsblatt zur Festigung des Programmierkonzepts »Schleife«. Der »Akteur« soll im Stil der Logo Turtle ([Pap85]) geometrischer Figuren (Polygone, Kreis, Kreisteile) zeichnen (vgl. Abbildung 2 zur Realisierung einer weiteren Aufgabe dieses Arbeitsblattes)

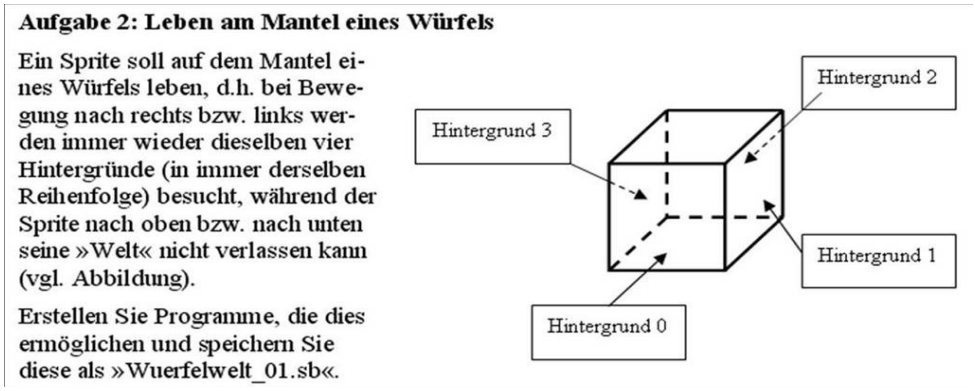


Abbildung 6: Beispielaufgabe zur Einführung von Variablen. Wenn der Akteur auch auf die Grund- und Deckfläche wechseln können soll, wird das Programm durch Codieren der Hintergründe mit Hilfe einer Variablen wesentlich vereinfacht.

Abbildung 7: Provozieren einer Lücke durch Erweiterung der Aufgabenstellung. Die Lösung der Aufgabe »um die Ecke laufen« wird den Lernenden zur Verfügung gestellt, die Aufgabe »an einer dreiarmigen Kreuzung zufällig abbiegen« erzeugt Fragen wie: „Wie bringe ich den »Akteur« dazu, bereits beim Erreichen des Kreuzungsmittelpunktes die Richtung zu ändern, anstatt zuerst mit der gegenüberliegenden »Wand« zu kollidieren?“. Weiterführende Aufgaben zum Durchschreiten eines Labyrinths mit »Mitzählen« der Richtungsänderungen festigen (u. a.) das Variablenkonzept.

Bei diesen vorgegebenen Aufgaben wurden neben den »intendierten« Lücken für die Lernenden auch individuelle Lücken bedeutsam: Beispielsweise inspirierte die Aufgabe aus Abbildung 5 eine Schülerin dazu, den »Akteur« (zusätzlich) so zu programmieren, dass ein Herz gezeichnet wurde, ein anderer Schüler stellte sich bei der Aufgabe aus Abbildung 6 die Zusatzaufgabe, die Hintergründe zufällig anzeigen zu lassen.

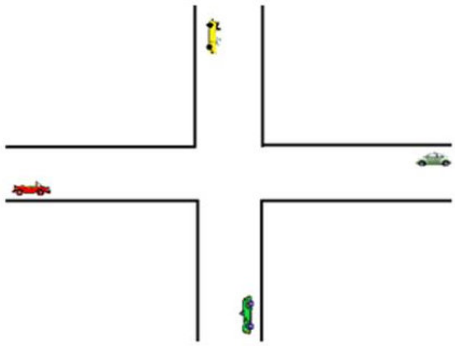
2.4 Individualisierung mit Scratch: Leistungsbeurteilung?

Die „*[Zuwendung der Lernenden zu] einer selbst gewählten Aufgabe*“ ist ein wesentliches Kennzeichen von Individualisierung (vgl. [BS09], S. 114). Die hohe Motivation der Lernenden beim »Schließen der wahrgenommenen Lücken«, gepaart mit der unkomplizierten Anwendbarkeit allgemeiner Programmierkonzepte durch eine intuitiv verständliche Sprache förderte bei Verwendung von Scratch die produktive (!) Selbsttätigkeit der Schülerinnen und Schüler. Dadurch verschob sich in den skizzierten Unterrichtseinheiten die Aktivität vom Lehrenden zu den Lernenden, wodurch in weiterer Folge konstruktive individuelle Interaktionen zwischen Lehrendem und Lernenden erst möglich wurden.

Diese beobachtende und beratende Begleitung der einzelnen Lernprozesse stellt aus Sicht des Autors/Lehrenden einen weiteren wichtigen Aspekt von Individualisierung dar: Lehrenden obliegt es ja nicht nur, die Rahmenbedingungen für den Wissens- und Kompetenzerwerb der Lernenden zu schaffen, es muss auch der Fortschritt der Lernenden festgestellt und nachvollziehbar beurteilt werden. Die Struktur von Scratch macht punktuelle Überprüfungen, bei denen Lösungen auf Papier codiert werden müssen, unmöglich. Andererseits können punktuelle Überprüfungen, bei denen die Lernenden die gestellten Aufgaben in der gewohnten Programmier-Lernumgebung lösen, aufgrund elektronischer Austauschmöglichkeiten stark an Aussagekraft verlieren.

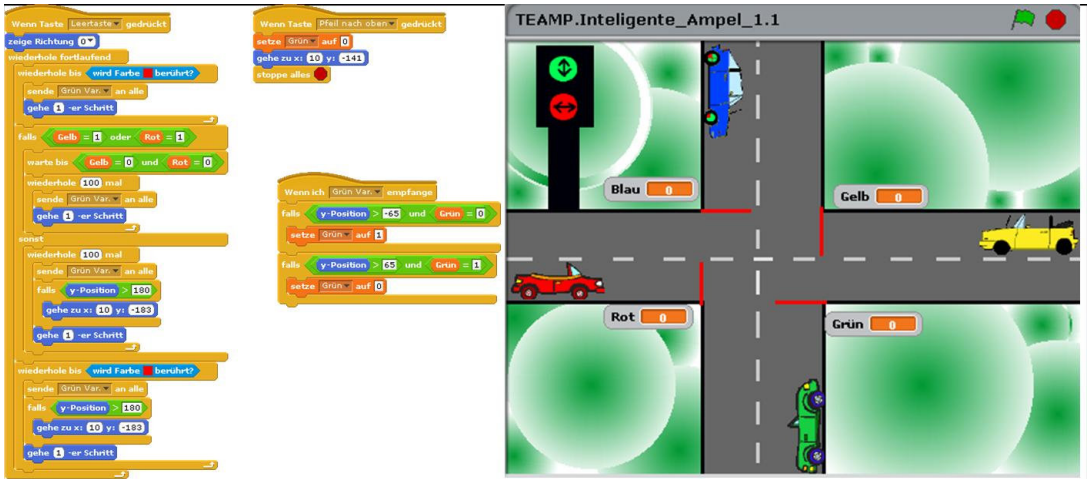
Die Möglichkeit, mit den Lernenden immer wieder individuell an Lösungsansätzen arbeiten zu können, bietet in dieser Situation eine gute Beurteilungsgrundlage. Zusätzlich wurde durch die bearbeiteten Aufgabenblätter aber auch eine problemhaltige Situation vorbereitet, die zum Abschluss der Beschäftigung mit Scratch in Form einer »Gruppenarbeit« (4 Gruppen zu jeweils 4 Lernenden) bearbeitet wurde (vgl. Abbildungen 8, 9).

Situationsbeschreibung und mögliche Teamziele



An der Kreuzung zweier Straßen kommen »zufällig« Autos an, queren die Kreuzung oder biegen ab, beachten Vorrangregeln (und verursachen tunlichst keine Kollisionen). Folgende Varianten sind vorstellbar:

- Ungeregelte Kreuzung mit Vorrangstraße und entsprechender Beschilderung: Die Autos mit Nachrang bleiben an der Kreuzung jedenfalls stehen und fahren los, wenn kein Querverkehr (oder – im Links-Abbiegefall – kein geradeaus fahrender Gegenverkehr) »in unmittelbarer Kreuzungsnähe« ist. Abbieger auf der Vorrangstraße haben nur den Gegenverkehr zu beachten (und auch das nur beim Abbiegen nach links).



Abbildungen 8, 9: Auszug aus der Situationsbeschreibung und Lösung einer Gruppe zur Problemstellung »Simulation einer vierarmigen Kreuzung, wahlweise mit Rechtsregel, Vorrangstraße oder Ampelsteuerung«. Der dargestellte Programmcode steuert nur eines der 5 Objekte!

Die vorgegebenen Schritte zur Strukturierung des Gruppenprozesses

- Teambildung und Aufgabendefinition/Festlegen der Teilaufgaben,
- Einzelarbeit – Formulieren einer Lösungsidee für einen der »Akteure«,
- Teamphase – Diskussion der Lösungsideen/Festlegen der Schnittstellen,
- Einzelphase – Programmieren der Teillösungen,
- Teamphase – »Zusammenbauen« der Gesamtlösung und
- Teamphase – Präsentation der Gesamtlösung

enthalten das »Grundprinzip des Kooperativen Lernens«: Think – Pair – Share (Denken – Austauschen – Vorstellen, vgl. [BS08], S. 17f) und gewährleisteten die selbständige Auseinandersetzung der Gruppen mit dem selbstgewählten Simulationsproblem.

Auf Basis vorheriger Erfahrungen mit traditionellen Programmierumgebungen war es beeindruckend mitzuerleben, dass alle Gruppen ihr selbst gewähltes Simulationsproblem mit nur drei Monaten »Programmier-Vorerfahrung« efolgreich lösen konnten, die definierten Lernziele daher offenbar erreicht wurden. Für die Leistungsbeurteilung war diese Arbeitsphase wichtig, weil hier die unterschiedliche Leistungsfähigkeit der Lernenden sichtbar und auch den Lernenden selbst bewusst wurde!

2.5 Individualisierung mit Greenfoot: Ein »sanfter Umstieg«?

Die Greenfoot-Lernumgebung ist auf ersten Blick jener von Scratch sehr ähnlich: Animierte Graphikobjekte können programmiert werden, um als »Akteure« in unterschiedlichen Szenarien Aufgaben zu lösen. Zum Unterschied von Scratch erfordert Programmieren in Greenfoot jedoch textuelles Codieren (vgl. Abbildung 10).

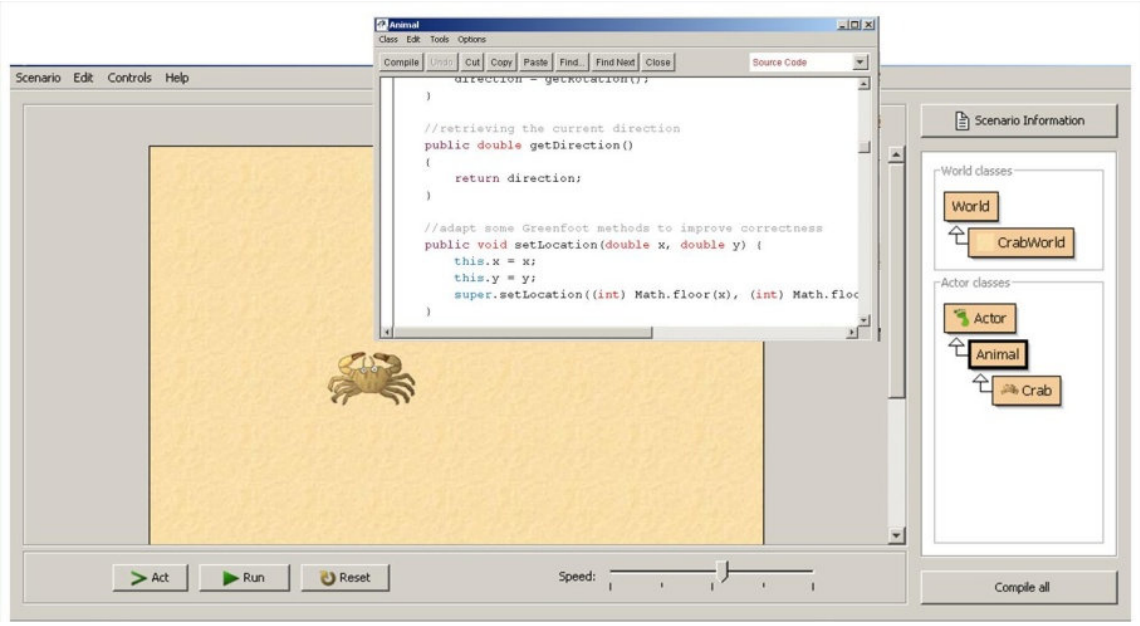


Abbildung 10: Die Greenfoot-Lernumgebung mit Klassenübersicht (rechts), Ausgabefenster, in dem das »Verhalten« des »Akteurs« sichtbar gemacht weren kann (links) und Programmeditor, über den »Akteure« mit »klassischem« Java-Code programmiert werden können/müssen.

Neben weiteren »codetechnischen« Unterschieden, wie

- Scratch kennt »Methoden« ohne explizite Parameterübergabe und ohne Rückgabewerte, die über Broadcast-Nachrichtenaustausch »aufgerufen« werden, während Greenfoot die »gewohnten« typisierten und parametrisierten Java-Methoden nutzt;
- in Scratch müssen/können Variable nicht typisiert werden – demgemäß ist auch keine explizite Typumwandlung von Benutzereingaben notwendig;
- in Scratch können neben vorgefertigten Zählschleifen nur Schleife programmiert werden, die so lange ausgeführt werden, BIS eine Bedingung erfüllt ist, während »Java-Schleife« so lange ausgeführt werden, SOLANGE eine Bedingung erfüllt ist.

erwiesen sich für die Fortsetzung des mit Scratch begonnenen Individualisierungsansatzes auch die folgenden Unterschiede als wesentlich:

- In Greenfoot sind die verwendbaren Codeblöcke nicht graphisch in der Lernumgebung repräsentiert. Befehle wie »if{...} else {...}«, »while{...}«, »Greenfoot.getRandomNumber()« oder auch die Namen der in Szenarien zur Verfügung gestellten Methoden müssen gemerkt bzw. in der jeweiligen Dokumentation oder im Quellcode »nachgeschlagen« werden.
- In Greenfoot können/müssen über das Konzept der Subklassen »Akteure« mit unterschiedlichen »Fähigkeiten« bzw. »Beschränkungen« erzeugt werden, während in Scratch jeder »Akteur« über dieselben Grundbefehle programmiert werden kann/muss.

- In Greenfoot kann der Hintergrund des Szenarios nicht über einen integrierten Graphikeditor einfach »gemalt« werden, sondern muss als Graphikdatei in die jeweilige »Welt« integriert werden.

Die letzten beiden Punkte bedingen, dass das für die Unterrichtseinheit benötigte Szenario (Bereitstellen spezifischer Superklassen für den/die Akteure, Festlegen des Hintergrundes) zunächst sinnvollerweise von dem/der Lehrenden »vorgefertigt« wird, während sich die Lernenden in jedes Szenario neu »einarbeiten« müssen.

Für den Umstieg von Scratch auf Greenfoot wurde daher folgende Vorgangsweise gewählt (vgl. auch [Kö03] bzw. [Kö10]):

- In einem ersten Schritt wurde das Arbeiten mit Greenfoot und das Codieren der grundlegenden Programmierkonzepte mit Java anhand eines Beispielszenarios gezeigt. Die Lernenden konnten durch das Aufrufen von Methoden »Akteure« steuern und wurden zu ersten kleinen Veränderungen des Codes angeleitet.
- Um lange Einarbeitungszeiten zu vermeiden, wurde das »Little Crab«-Szenario aus [Kö10] um einige der aus Scratch bekannten und als wichtig erachteten Abstraktionen erweitert, sodass dieses Szenario längerfristig verwendet werden konnte. Zu den Erweiterungen zählen:
  - Bereitstellen von in ihrer Funktionalität bekannten Methoden (»makeSteps«, »turn«, »penUp«, »penDown«);
  - Bereitstellen von Methoden, deren »Wirkung« aus Scratch bekannt ist, die aber »anders zu verwenden« sind (»InputBox«, »MessageBox«)
- Das Codieren wurde zunächst an »kleinen« Methoden eingeübt, die dann – ähnlich wie bei Scratch – zu Programmen »zusammengebaut« wurden. Die Aufgaben wurden aus den Scratch-Arbeitsblättern übernommen und angepasst und verlangten z.B. das Codiern von Methoden zur Steuerung des Verhaltens des »Akteurs« beim Erreichen eines Bildschirmrandes oder von Methoden zum Zeichnen geometrischer Figuren (vgl. Abbildungen 2 und 5).

2.6 Individualisierung mit Greenfoot: Wahrgenommene Probleme

Wenngleich durch den skizzierten »sanften Umstieg« die (meisten) Schülerinnen und Schüler auch beim Programmieren mit Greenfoot/Java schnelle Erfolgserlebnisse hatten, traten bekannte Codierungsprobleme (Unterscheidung von Groß- und Kleinschreibung, Vergessen des Strichpunktes, Unklarheiten bei der Formulierung elementarer Kontrollstrukturen) immer wieder auf und führten zu Phasen intensiver Hilfstellung durch den Lehrenden. Problematischer aber erscheint, dass sich durch den Wechsel der Lernumgebung die »Qualität« der Lücke geändert hatte, die die Lernenden wahrnahmen: Während bei Scratch das Verhalten des »Akteurs« Anlass zu neuen Fragestellungen gab, bestand nunmehr die Aufgabe für die Lernenden darin, die vom Programmieren mit Scratch her prinzipiell bekannte Lösung dem »Akteur« in einer neuen Sprache mitzuteilen, weil dies vom Lehrenden durch vorbereiteten Aufgabenblätter (in bester didaktischer Absicht) eingefordert (»provoziert«) wurde! Dieser letzte Aspekt wurde durch einen zusätzlichen, neuen Typ von Aufgaben noch verstärkt:

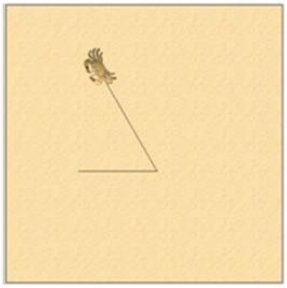
Micheal Kölling et al. weisen in [Kö03] darauf hin, dass bei Betonung von Objektorientierung im Programmier-Anfangsunterricht die Gefahr besteht, dass »traditionelle« Inhalte algorithmischer Natur vernachlässigt werden. Im technischen Schulwesen sind aber auch diese Inhalte wichtig. Daher wurden beim Programmieren mit Greenfoot zunehmend Aufgaben, die

durch Bewegung des »Akteurs« gelöst werden mussten, mit solchen gemischt, bei denen »nur« (einfache) Berechnungen durchzuführen waren. Zwar wurden auch diese Aufgaben von den Schülerinnen und Schülern bearbeitet, doch wurde in der Interaktion mit den Lernenden spürbar, dass hier für sie keine Lücke entstanden war, die sie für wichtig befanden, geschlossen zu werden!

2.7 »Individualisierung« mit Greenfoot: Feedback durch Leistungsfeststellung

Da Java-Programmcode als Text geschrieben werden muss, sind beim Arbeiten mit der Greenfoot-Lernumgebung auch »Papier-und-Bleistift-Tests« geeignet, um festzustellen, ob Programmcode verstanden wird, insbesondere aber, inwieweit aktives Codieren beherrscht wird. Nach etwa zwei Monaten Programmiererfahrung der Lernenden mit Greenfoot/Java, wurden derartige Leistungsfeststellungen durchgeführt, die einen Mix aus »Bewegungsaufgaben« und »Berechnungsaufgaben« enthielten. In Anlehnung an den Wettbewerb »Biber der Informatik« ([BI09]) wurden neben Offene-Antwort Aufgaben auch Multiple-Choice Aufgaben gestellt (vgl. Abbildungen 11 bis 13).

Durch welche Befehlsfolge könnte die nebenstehende Figur gezeichnet werden?



☐
```
penDown();
makeSteps(40);
turn(120);
makeSteps(60);
```

☐
```
penDown();
makeSteps(40);
turn(120);
makeSteps(40);
```

☐
```
penDown();
makeSteps(40);
turn(-120);
makeSteps(40);
```

☐
```
penDown();
makeSteps(40);
turn(-120);
makeSteps(60);
```

In einer Methode finden Sie die folgenden Befehlszeilen:

```
int a;
int b;
int c;
a = 10;
b = 20;
c = 0;
a = b;
c = a + b;
b = a;
c = c + b;
```

Welche Werte sind zum Schluss in den Variablen a, b und c gespeichert?

Abbildungen 11, 12: Aufgaben zum Codeverständnis aus »Papier-und-Bleistift-Tests«. Die »Bewegungsaufgabe« wurde in Anlehnung an Aufgaben aus dem Wettbewerb »Biber der Informatik« kreiert, die »Berechnungsaufgabe« zum Nachvollziehen von Zuweisungen wurde von [DB09] inspiriert; dort findet sich auch eine Analyse möglicher Denkmodelle, die bei der Analyse von Zuweisungen verwendet werden können.

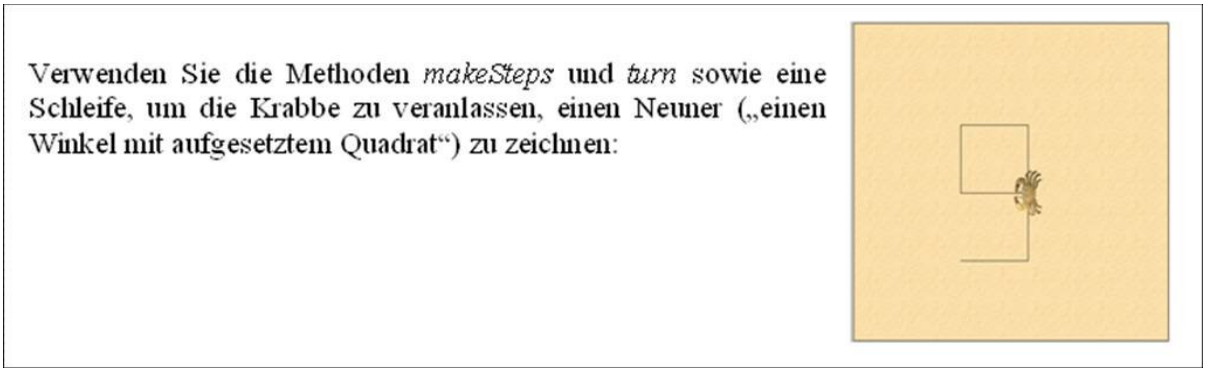


Abbildung 13: Aufgabe aus »Papier-und-Bleistift-Tests«, bei der ohne »Unterstützung« der Lernumgebung selbst Code zu schreiben ist.

Diese informellen Leistungsfeststellungen ergaben ein differenziertes Bild: Nur eine Schülerin und zwei Schüler konnten die meisten Aufgaben (fast) fehlerfrei lösen. Generell waren die Ergebnisse bei der Analyse von vorgegebenem Code signifikant besser als beim eigenen Codieren, Codierungsaufgaben mit Berechnungen wurden häufig überhaupt ausgelassen.

Bei der darauffolgenden Diskussion über die Probleme beim Lösen derartiger Tests zeigte sich einerseits, dass die Schülerinnen und Schüler beim individuellen Lösen von Aufgaben mit Scratch die Methode von »Versuch und Irrtum« als zielführend erlebt hatten und diese auch nach dem Wechsel zu Greenfoot anwendeten. Viele Schülerinnen und Schüler verließen sich daher auf die »Antwort« des »Akteurs«, der durch seine »Reaktion« mitteilte, ob das Programm die Lösung der jeweiligen Aufgabe liefert. Aus diesem Grund wären auch »Bewegungsaufgaben« viel leichter zu lösen als »Berechnungsaufgaben«: Bei letzteren bleibt »das Tun des Akteurs« weitgehend unsichtbar, der Ablauf des Programms ist daher „nicht vorstellbar“.

2.7 Zwischenresümee

Der Wechsel der Lernumgebung bedeutete für die Schülerinnen und Schüler eine doppelte Abstraktion: Zum einen wurde die graphische Repräsentation des Programmcodes durch Codetext (kombiniert mit einer graphischen Repräsentation der Klassenstruktur) ersetzt. Dadurch wurde das Codieren um die visuelle und die taktile Komponente (»Codeblock sehen und in den Editierbereich ziehen) beraubt und zu einer primär kognitiven Tätigkeit. Zum anderen wurde diese Verschiebung in der »Sinnlichkeit des Programmierens« durch vermehrtes Stellen von »Berechnungsaufgaben« verstärkt, deren Lösungen ohne sichtbare Aktion des »Akteurs« blieben. Nur einige wenige der Lernenden hatten von selbst mentale Modelle für den Ablauf eines Programms konstruiert und hatten mit diesem Abstraktionsschritt nur geringe Probleme.

Die Notwendigkeit der Herausbildung mentaler Modelle, um Programmieren zu können, wird in der Literatur vielfach beschrieben (vgl. [DB06] oder [Ho90]). Andererseits weist die »Unbeschwertheit«, mit der die Lernenden in der Lernumgebung Scratch Aufgaben unterschiedlichster Schwierigkeitsgrade aufgreifen und erfolgreich bewältigen, darauf hin, dass das Erlernen des Programmierens ein mehrphasiger Prozess ist, in dem mentale Modelle gegenüber dem aktiven Tun zunächst eine untergeordnete Rolle spielen können.

Zur Beschreibung dieses Prozesses erscheint ein Modell geeignet, das auf Überlegungen zur Professionalität des Lehrerhandelns im Unterricht zurückgeht und in [Sch83] bzw. auch in

[AP98] beschrieben ist. Dort werden drei typische Formen des Zusammenspiels von »Wissen« und »Handeln« unterschieden:

Handeln „auf der Basis unausgesprochenen Wissens-in-der-Handlung“ (knowing in action) ist dadurch gekennzeichnet, dass „sich der Handelnde oft nicht [des für die Handlung benötigten Wissens] bewusst ist [...]“ und „üblicherweise nicht ohne weiteres in der Lage ist, dieses Wissen verbal zu beschreiben“.

Der zweite Handlungstyp benötigt „Reflexion-in-der-Handlung“ (reflection in action) und „beginnt mit dem Erleben einer Diskrepanz zwischen den Erwartungen, die man hinsichtlich des Ablaufs einer Situation [hegt], und dem realen Ablauf dieser Situation [... Der Handelnde] trennt Mittel und Ziele nicht, sondern bestimmt sie interaktiv, wenn er eine problematische Situation definiert.“.

Im dritten Handlungstyp (Reflexion-über-die-Handlung, reflection on action) tritt „die Reflexion aus dem Handlungsfluss heraus [...]., wodurch „die Bearbeitung besonders komplexer Handlungsaufgaben und [...] die Lösung besonders schwieriger Handlungsprobleme“ möglich wird. Reflexion-über-die Handlung „schafft erst die Möglichkeit für einen geordneten sprachlichen Ausdruck des Wissens, das der eigenen Handlung zugrundeliegt“.

Professionelle Tätigkeit braucht das Zusammenspiel von »reflection in action« und „nicht [....] reflektierte Handlungsselbstverständlichekeiten“ (»knowing in action«), ergänzt um »reflection on action« wenn es darum geht, das eigene Wissen zu kommunizieren (nach [AP98], S. 329).

Programmieren ist eine professionelle Tätigkeit, in der die Kommunikation der gefundenen Problemlösung ein wesentlicher Schritt ist. Klassischer Programmierunterricht fokussiert diesen Aspekt und scheint den dritten Handlungstyp nach obigem Modell zu betonen, während die ersten beiden vernachlässigt werden. Programmierunterricht, der die Möglichkeiten der Lernumgebung Scratch nützt, scheint andererseits Gefahr zu laufen, in den ersten beiden Handlungstypen »stecken zu bleiben«. Die Frage, wie ein Wechsel z.B. von Scratch zu einer anderen Programmierumgebung/Programmiersprache sinnvoll zu vollziehen ist, wird aber in der Literatur bislang kaum und nur am Rande thematisiert (vgl. z.B. [Re09], S. 66f).

3. Die Lücke: Ein konzeptioneller Rahmen für die Lernenden?

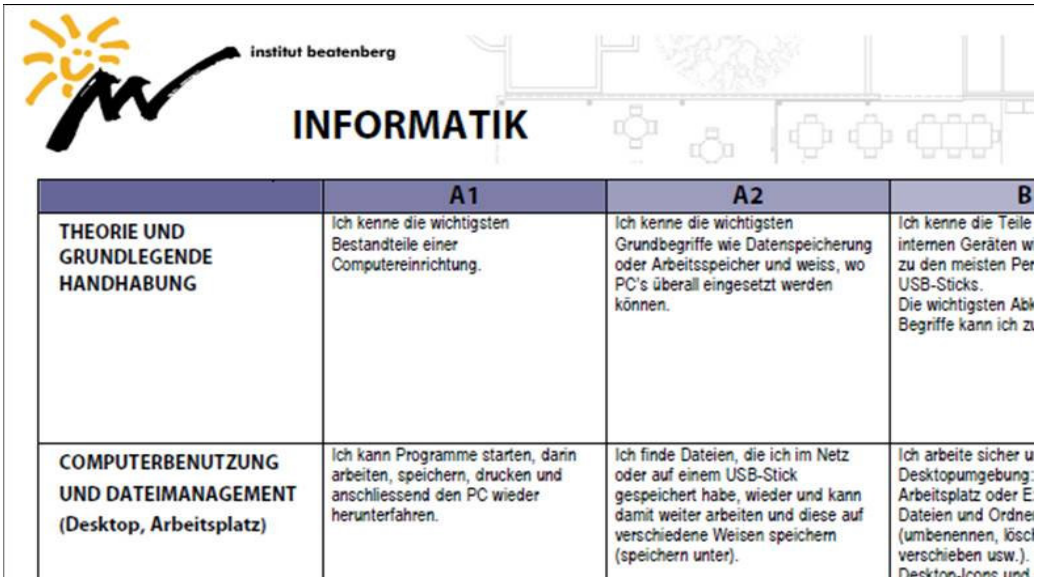
Ist vielleicht der Blickwinkel der Informatik-Fachdidaktik nicht ausreichend, um eine Antwort auf das »Wie« eines Wechsels von Scratch zu einer anderen Lern- und/oder Programmierumgebung zu geben? In der reflexiven Betrachtung des beschriebenen Ansatzes zur Individualisierung im Programmierunterricht irritiert vor allem, dass sich nach dem vollzogenen Wechsel die Aktivität zunehmend wieder von den Lernenden zum Lehrenden verschoben hat. Die selbstkritische Analyse zeigt noch mehr: Genau genommen war die Steuerung der Lernprozesse stets in der Verantwortung des Lehrenden, die Lernenden hatten über die aktuellen Aufgabenstellungen hinaus nicht die Möglichkeit, ihren Lernprozess an langfristigen Lernzielen (wie z.B. dem Entwickeln von »mentalen Modellen«) zu orientieren.

3.1 Kompetenzraster...

Individuelles Fördern durch den Lehrenden und selbsttätiges Bearbeiten von (idealerweise selbstgewählten) Aufgaben mögen wichtige Komponenten von Individualisierung sein. Wenn aber die Lernenden auf ihren unterschiedlichen Lernpfaden auch die gleichen, z.B. durch

Bildungsstandards festgelegten Ziele erreichen sollen, muss der Lernprozess durch diesbezügliche Verbindlichkeiten vorab geregelt werden. Und: „Eine individuelle Förderung verlangt [...] nach individuellen Verbindlichkeiten.“ ([Mü03]).

Die moderne Pädagogik kennt Kompetenzraster als Werkzeuge zur Festlegung derartiger individueller Verbindlichkeiten: Es geht darum „[...] individuelle Leistungen mit einem [vorher definierten] Referenzwert in Beziehung zu bringen. Diesen Referenzwert und damit die inhaltliche Basis bilden so genannte Kompetenzraster. Die Kompetenzraster definieren die Inhalte und die Qualitätsmerkmale der verschiedenen Fachgebiete in präzisen „Ich-kann“-Formulierungen.“ ([Mü03]). Kompetenzraster informieren die Lernenden VOR Beginn des Lernprozesses an darüber, welche Inhalte gelernt werden sollen. Für jeden dieser Inhalte werden Kompetenzsstufen angegeben, die von den Lernenden nach individuellem Vermögen angestrebt werden können (vgl. Abbildung 14).



institut beatenberg

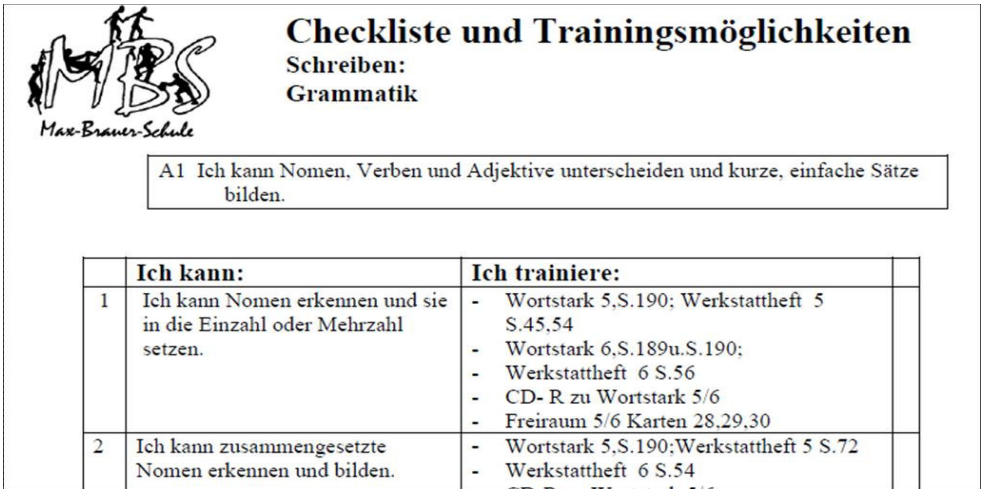
INFORMATIK

	A1	A2	B
THEORIE UND GRUNDLEGENDE HANDHABUNG	Ich kenne die wichtigsten Bestandteile einer Computereinrichtung.	Ich kenne die wichtigsten Grundbegriffe wie Datenspeicherung oder Arbeitsspeicher und weiss, wo PC's überall eingesetzt werden können.	Ich kenne die Teile internen Geräten w zu den meisten Per USB-Sticks. Die wichtigsten Abl Begriffe kann ich z
COMPUTERBENUTZUNG UND DATEIMANAGEMENT (Desktop, Arbeitsplatz)	Ich kann Programme starten, darin arbeiten, speichern, drucken und anschliessend den PC wieder herunterfahren.	Ich finde Dateien, die ich im Netz oder auf einem USB-Stick gespeichert habe, wieder und kann damit weiter arbeiten und diese auf verschiedene Weisen speichern (speichern unter).	Ich arbeite sicher u Desktopumgebung: Arbeitsplatz oder E Dateien und Ordne (umbenennen, lösc verschieben usw.). Desktop-Icons und

Abbildung 14: Kompetenzraster grundlegender Inhalte informatischer Bildung (Auszug, Quelle: http://www.institut-beatenberg.ch/xs_daten/Materialien/kompetenzraster.pdf; 30. Mai 2010). Die Inhalte stehen in der ersten Spalte, die zugehörigen Kompetenzstufen in der jeweiligen Zeile.

3.2 ...und Kompetenz-Checklisten ....

Die Lernenden sollen aber nicht nur ihren Lernprozess auf Ziele hin auszurichten, sie sollen in individualisiertem Unterricht auch in der Lage sein, ihren Lernfortschritt eigenständig zu überprüfen. Dazu reichen Kompetenzraster allein nicht aus, vielmehr muss für jede Kompetenz durch Angabe von operationalisierbaren und operationalisierten Teilkompetenzen erklärt werden, was damit gemeint ist, und durch welche Aufgaben diese Teilkompetenzen erlernt bzw. deren Erreichen überprüft werden kann. Diese Ergänzungen zum Kompetenzraster werden in Kompetenz-Checklisten zusammengefasst (vgl. Abbildung 15).



Checkliste und Trainingsmöglichkeiten
Schreiben:
Grammatik

Max-Braun-Schule

A1 Ich kann Nomen, Verben und Adjektive unterscheiden und kurze, einfache Sätze bilden.

	Ich kann:	Ich trainiere:	
1	Ich kann Nomen erkennen und sie in die Einzahl oder Mehrzahl setzen.	- Wortstark 5,S.190; Werkstattheft 5 S.45,54 - Wortstark 6,S.189u.S.190; - Werkstattheft 6 S.56 - CD- R zu Wortstark 5/6 - Freiraum 5/6 Karten 28,29,30	
2	Ich kann zusammengesetzte Nomen erkennen und bilden.	- Wortstark 5,S.190;Werkstattheft 5 S.72 - Werkstattheft 6 S.54	

Abbildung 15: Kompetenz-Checkliste aus Deutsch/Sekundarstufe 1 (Auszug, Quelle: [MBS08])

3.3 .... im Programmierunterricht?

Die Abbildungen 14 und 15 weisen darauf hin, dass Kompetenzraster und Kompetenz-Checklisten in der Informatik-Fachdidaktik und im Informatikunterricht weitestgehend unbekannt sind. Sie geben aber den Lehrenden ein geeignetes Instrument in die Hand, um individuelle Lernprozesse auf ein inhaltliches (und auch ein zeitliches) Ziel hin zu orientieren.

Kompetenzraster und Kompetenz-Checklisten sind nach eigenem Ermessen auch geeignete Werkzeuge, um einen Wechsel von Programmier-Lernumgebungen zu steuern, der für einen motivierenden Einstieg in das Programmieren wesentlich erscheint: Die Lernenden können z.B. zeitgerecht zum Bilden mentaler Modelle für den Programmablauf bzw. zur metasprachlichen Beschreibung von Programmen »provoziert« werden. Erste eigene diesbezügliche Überlegungen lassen aber vermuten, dass dies ein weites Feld fachdidaktischer Forschung ist, geht es ja nicht nur darum, schülerverständliche „Ich kann.....“-Formulierungen zu sinnvollen Inhalten zu finden, sondern auch, durch genügend Aufgaben individuelle Lernpfade zum Erreichen der einzelnen Kompetenzstufen zu ermöglichen. Dabei ist zudem stets zu bedenken, dass bei der Formulierung der Aufgaben auf die »Fähigkeiten und Beschränkungen des Akteurs« bzw. auf das kreative Potential, das die Lernumgebung (z.B. eben Scratch) bietet, bedacht zu nehmen ist.

4 Resümee

Die selbst gemachten Erfahrungen mit Individualisierung im Programmierunterricht spannen einen weiten Bogen von traditionell fachdidaktischen Fragestellungen bis hin zum »Design individueller Lernprozesse« und weisen darauf hin, dass (Programmier-) Unterricht auch fachdidaktisch, aber eben nicht ausschließlich fachdidaktisch verstanden und geplant werden kann. Wenn es um die Vermittlung langfristiger Kompetenzen geht, erscheinen Überlegungen, bis zu welcher Grenze diese Lernumgebung bzw. jene Softwareumgebung einsetzbar ist, zu kurz zu greifen, wenn sie nicht gleichzeitig Wege aufzeigen, wie diese Grenzen so überschritten werden können, dass kontinuierliche Lernprozesse für (möglichst) alle Schülerinnen und Schüler möglich werden.

Die selbst gemachten Erfahrungen mit Individualisierung im Programmierunterricht lassen eine Lücke für die Lehrenden erahnen, die durch Kompetenzraster und Kompetenz-

Checklisten geschlossen werden könnte. Die wahrgenommene Aufgabe besteht darin, den Lernenden einen Orientierungsrahmen zur Verfügung zu stellen, der Lernen zu einem selbstwirksamen Prozess macht: Erst wenn die Lernenden das Ziel des Lernprozesses vorab kennen, können Sie über die individuelle Bearbeitung von Aufgaben hinaus tätig werden. Individualisierung meint auch die individuelle und doch zielgerichtete Wahl der Aufgaben(reihenfolge) und die (Mit-) Beurteilung der Lösungen seitens der Lernenden. Beim Fehlen dieses Orientierungsrahmens besteht die Gefahr, dass gewollte Individualisierung sich in Phasen offenen Lernens erschöpft und unter den Rahmenbedingungen schulischer Praxis schnell wieder in alte Muster des Unterrichtens zurückfällt.

Die selbst gemachten Erfahrungen zeigen auch, dass die Kombination fachdidaktischer und pädagogischer Aspekte unter den gegebenen schulorganisatorischen Rahmenbedingungen Individualisierung von Programmierunterricht zu einem intensiven individuellen Lernprozess des/der Lehrenden macht. Das In-Beziehung-Setzen der eigenen Erfahrungen mit Ergebnissen fachdidaktischer und pädagogischer Forschung ist ein Bestandteil dieses Lernprozesses. Der Austausch über eigene Erfolge, Halb-Erfolge oder auch Misserfolge mag die Bereitschaft fördern, sich individuell auf Individualisierung einzulassen.

Literatur und Referenzen

[An05] Antonitsch P.: Standard Software as Microworld? In: Mittermeir R.: From Computer Literacy to Informatics Fundamentals. Springer LNCS 3422, Berlin et al. 2005

[AP98] Altrichter H., Posch P.: Lehrer erforschen ihren Unterricht, Klinkhardt, Bad Heilbronn, 3. Auflage 1998

[BDS08] Bornat R., Dehnadi S., Simon: Mental Models, Consistency and Programming Aptitude. In: Hamilton S. and M.: Conferences in Research and Practice in Information Technology (CRPIT), Vol. 78, 2008
URL: http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper3.pdf

[BI09] österreichische Homepage zum Wettbewerb »Biber der Informatik«, 2009, URL: http://www.biber.ocg.at/

[BS08] Brüning L, Saum T.: Erfolgreich Unterrichten durch Kooperatives Lernen 1, Neue Deutsche Schule, Essen 2008

[BS09] Brüning L, Saum T.: Erfolgreich Unterrichten durch Kooperatives Lernen 2, Neue Deutsche Schule, Essen 2009

[DB06] Dehnadi S., Bornat R.: The Camel Has Two Humps, draft paper, 2006
URL: http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

[Du92] Duchâteau C.: From "DOING IT…" to "HAVING IT DONE BY…": the heart of programming. Some didactical thoughts. In: Preproceedings NATO ARW Cognitive Models and Intelligent Environments for Learning Programming. S. Margherita Ligure, Genova 1992

[Gi04] Girmes R.: [Sich] Aufgaben stellen, Kallmeyer, Seelze 2004

[Gr06] Homepage zur Programmier-Lernumgebung »Greenfoot«; verfügbar seit 2006, URL: http://www.greenfoot.org/

[Ho90]: Hoc J.M., Green T.R.G., et. al.. (eds.): Psychology of Programming, Academic Press, San Diego, 1990

[Kö03] Kölling M., Quig B., Patterson A., Rosenberg, J.: The BlueJ System and Its Pedagogy, Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, No 4, Dec 2003.
URL: http://www.bluej.org/papers/2003-12-CSEd-bluej.pdf

[Kö10] Kölling M.: Introduction to Programming with Greenfoot, Pearson Higher Education, New Jersey, 2010

[LA04] Lehrplan für das Pflichtfach Informatik der AHS-Oberstufe, Österreich
URL: http://www.bmbwk.gv.at/medienpool/11866/lp_neu_ahs_14.pdf

[LB97] Lehrplan für das Pflichtfach Angewandte Informatik an der HTL/Abteilung für Elektrotechnik, Österreich
URL: http://www.htl.at/fileadmin/content/Lehrplan/HTL/ELEKTROTECHNIK_Anlage_1.1.3_302-97.pdf

[MBS08] Die neue Max Brauer Schule Hamburg. Beispiele aus dem Lernbüro, den Projekten und Werkstätten, 2008
URL: http://www.maxbrauerschule.de/mbs/downloads/2008_neue_mbs_bsp.pdf

[Mü03] Müller A.: Anstiftung zum Lernerfolg, Institut Beatenberg – sprit of learning, 2003
URL: http://www.institut-beatenberg.ch/xs_daten/Materialien/Artikel/artikel_lerncoach.pdf

[Mü06] Müller A.: Das Lernen gestaltbar machen, Institut Beatenberg – spirit of learning, 2006
URL: http://www.institut-beatenberg.ch/xs_daten/home/artikel_selbstgestaltungSCREEN. pdf

[Pap85] Papert S.: Gedankenblitze, rororo, Reinbek bei Hamburg, 1985

[Pat95] Pattis R.E.: Karel the Robot, Wiley, New York et al., 1995 (2nd ed.)

[Re09] M. Resnick, Y. Kafai et al.: Scratch: Programming for All. Comm. of the ACM, Vol. 52, Nr. 11; November 2009.
URL: http://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf

[RNH04] Reichert R., Nievergelt J., Hartmann W.: Programmieren mit Kara, Springer, Berlin et al., 2004

[Sc07] Homepage der Programmier-Lernumgebung »Scratch«; verfügbar seit 2007
URL: http://scratch.mit.edu/[Sch83]

[Sc08] Schön D.A.: The Reflecitve Practitioner, Temple Smith, London, 1983