Ira Diethelm (Hrsg.): Informatische Bildung zum Verstehen und Gestalten der digitalen Welt,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 11

Herausforderungen durch neue Programmierkonzepte in blockbasierten Programmiersprachen

Sven Jatzlau[1], Ralf Romeike[2]

Abstract: Blockbasierte Programmiersprachen stehen für einen Zugang zur Programmierung, der von Programmieranfängern erfolgreich genutzt wird und zunehmend auch als Möglichkeit gesehen wird, nicht-professionellen Programmierern das Gestalten von Informatiksystemen zu ermöglichen. Als Gründe hierfür werden u. a. die intuitive Bedienung, schnelle Erfolgserlebnisse und ein breites und kontextualisiertes Anwendungsspektrum genannt, die zu einer großen Beliebtheit unter jungen Programmierern führen und sich als Kernmerkmale in den populären Programmierumgebungen Scratch und BYOB/Snap! wiederfinden. Diese grafische, blockbasierte Programmierung unterscheidet sich jedoch von klassischer textbasierter Programmierung nicht nur auf der Bedienebene, sondern bringt gegenüber gängigen im Unterricht genutzten Programmiersprachen auch verschiedene konzeptuelle Unterschiede mit sich. Solche ergeben sich zum einen aus der visuellen Darstellung von Objekten auf der Bühne, zum anderen aus der historischen Genese sowie didaktischen und pragmatischen Entscheidungen. Für Lehrerinnen und Lehrer ist ein konzeptuelles Verständnis wichtig, um Möglichkeiten und Probleme beim Programmierenlernen mit Scratch-ähnlichen Programmiersprachen zu erkennen und didaktisch darauf einzugehen. Im vorliegenden Beitrag werden verschiedene zentrale Konzepte, wie z. B. Nesting von Grafikobjekten, herausgearbeitet und verdeutlicht. Es zeigt sich, dass der Ansatz des „Programmierens für alle“ mit passenden Werkzeugen in greifbare Nähe rückt, eine didaktische Aufarbeitung zum Erreichen eines informatischen Grundverständnisses aber unerlässlich ist.

Keywords: visuelle Programmierung, Scratch, Snap!, BYOB, GP, blockbasierte Programmiersprachen

1 Motivation und Ziele

Blockbasierte Programmierung ist spätestens seit *Scratch* ein relevantes Thema für Programmieranfänger, ob in Schule oder Freizeit. Programmiersprachen, die sich auf das Konzept der blockbasierten Programmierung stützen, unterscheiden sich von textbasierten Sprachen sowohl auf konzeptioneller Ebene als auch in ihrer Interaktion mit dem Benutzer. Aufgrund ihrer Einfachheit und niedrigen Einstiegshürde (u.a. durch wegfallende Syntaxfehler) sind besonders die blockbasierten Sprachen, zu denen etwa *Scratch* und *Snap!* gehören, unter Anfängern und Einsteigern beliebt (vgl. [LKG17]).

Aber auch abseits der Syntaxreduzierung wurden weitere Entscheidungen getroffen, die das Verständnis der Schüler fördern: Im Folgenden soll herausgearbeitet werden, welche

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg, Didaktik der Informatik, Martensstr. 3, 91058 Erlangen, Germany sven.jatzlau@fau.de
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg, Didaktik der Informatik, Martensstr. 3, 91058 Erlangen, Germany ralf.romeike@fau.de

Unterschiede blockbasierte Programmiersprachen mit sich bringen und welche neuen Konzepte in diesem Zusammenhang wichtig sind. Zu diesem Zweck werden zunächst die historischen Ursprünge blockbasierter Programmierung analysiert und dann die wichtigsten neuen Konzepte anhand von Sprachen wie *Scratch*, *Snap!* und einem neuen Vertreter der Familie, *GP*, verdeutlicht. Die Konzepte nichtatomarer Interpreter, Klassen und Objekte, Nested Sprites, First-class-Strukturen, Entwickler-Modus, Block-to-Text-Slider und Class Browser wurden ausgewählt, da sie entweder in ihrer Darstellung neuartig oder aber für Neulinge in der blockbasierten Programmierung noch gänzlich unbekannt sind, weshalb es wichtig ist, dass sie didaktisch behandelt und aufgearbeitet werden.

2 Forschungsstand

Zum Thema Didaktik der blockbasierten Programmierung wurde bis heute nur relativ wenig Forschungsarbeit geleistet. Mitchel Resnick, verantwortlich für die Entwicklung der Sprache *Scratch*, kritisierte bereits 2002, dass Computer zwar einerseits immer zugänglicher und verbreiteter werden und dadurch eine Revolution der Lerngewohnheiten möglich ist, aber oftmals digitale Medien nur dazu verwendet werden, veraltete Herangehensweisen an den Lernprozess zu unterstützen [Re02]. Resnick et al. [RMMH09] machen darauf aufmerksam, dass digitale Kompetenz nicht nur Interaktion und Kommunikation beinhalten darf, sondern auch das Erschaffen und die Weiterentwicklung von Inhalten. Durch Modrow et al. [Mo11, MMS11] wurden die Einstellungen von Schülern zu *Scratch* untersucht. Dabei war klar erkennbar, dass Sprachen wie *Scratch* an Schulen zwar gerne von ihnen verwendet werden, jedoch als reine Einsteigerprogramme verstanden werden. Für komplexere Aufgabenstellungen oder im Berufsleben wird *Scratch* etwa als nicht angemessen empfunden. Gleichzeitig jedoch sind beispielsweise alle untersuchten Abituraufgaben (Niedersachsen, 2011) problemlos in *Scratch* oder *Snap!* lösbar, sodass eine Abwendung von textbasierter Programmierung hin zu blockbasierter durchaus denkbar wäre. Die Forschungsergebnisse von Strecker [St15] unterstützen diese These: Sie verglich die Leistungen von Schülern, die mit *Snap!* auf das Abitur vorbereitet wurden, mit denen, die reine Java-Kurse besuchten. Im Durchschnitt erbrachten Erstere bessere Leistungen, v.a. im Hinblick auf Teillösungen. Außerdem waren sie den Herausforderungen besser gewachsen, da ihnen weniger Konzepte oder Kompetenzen fehlten. Besonders im Bereich der Algorithmik konnten dadurch bessere Ergebnisse erzielt werden. Price et al. [PB15] erforschten das Verhalten von Schülern bei der Verwendung blockbasierter Programmiersprachen. Laut ihren Forschungen waren diese Schüler in der Lage, Programmieraufgaben deutlich schneller zu lösen als solche, die textbasierte Programmiersprachen verwendeten. Dieses Ergebnis deutet darauf hin, dass blockbasierte Programmiersprachen und Benutzeroberflächen leistungsfördernd wirken. Dies könnte mit der Motivation der Lerner zusammenhängen: Ruf et al. [RMH14] verglichen intrinsische Motivation und Programmierfähigkeiten von zwei Gruppen von Lernern. Eine Gruppe verwendete *Scratch* zum Erlernen der Programmierung, die andere die textbasierte Programmiersprache *Karol*. Hierbei zeigte sich, dass sowohl Motivation als auch Programmierkompetenz in der *Scratch* Klasse stärker vertreten waren. Die Forschungsergebnisse von

Weintrop et al. [WW15] zeigen ähnliche Ergebnisse: Schülern wurden Aufgaben vorgelegt, die sie bearbeiten sollten. Der zu bearbeitende Code wurde jeweils in Blöcken bzw. in Textform vorgelegt, doch die blockbasierten Aufgaben wurden deutlich häufiger korrekt beantwortet. Dwyer et al. [DHH15] erforschten, wie Informationen in blockbasierten Userinterfaces aufgenommen und verarbeitet werden und zeigten, dass die visuelle Natur solcher Programmierumgebungen oftmals hilfreich für Lerner ist. Trotzdem besteht die Gefahr, dass Nutzer Funktionen der Umgebung übersehen oder visuelle Hinweise falsch interpretieren (etwa wenn die Position oder das Aussehen der Objekte auf der Bühne fälschlicherweise als Hinweise für ihr Verhalten gedeutet werden). Aber auch auf motivationaler Ebene zeigen sich nicht ausschließlich Vorteile: Lewis [Le10] untersuchte die Wahrnehmung von Lernern bei der Verwendung von *Scratch* und *Logo*. Zwischen den Gruppen zeigten sich keine bedeutenden Unterschiede hinsichtlich ihrer Motivation, zukünftig zu programmieren.

Es wird deutlich, dass blockbasierte Programmiersprachen besonders für Programmieranfänger eine große Chance darstellen, schnell und sicher Kompetenzen im Bereich der Algorithmik und Programmierung zu erlangen. Dies wirkt sich positiv auf deren intrinsische Motivation und Selbsteinschätzung aus.

3 Meilensteine in der Entwicklung der blockbasierten Programmiersprachen

Ein großer Teil der konzeptionellen Neuerungen, die sich in blockbasierten Programmiersprachen finden, lassen sich auf die historische Entwicklung zurückführen. Daher ist es bedeutsam, die Meilensteine in der Entwicklung dieser Programmiersprachenfamilie zu kennen. Hierzu sind drei essentielle Programmierumgebungen als Meilensteine zu nennen, die neben *Scratch* und *Snap!* die Genese visueller Sprachen beeinflussen: *Morphic*, *Smalltalk* und *GP*. Diese sollen im Folgenden kurz erläutert und ihre Bedeutung für die blockbasierte Programmierung herausgestellt werden. Diese Umgebungen wurden gewählt, da sich ihre Konzepte schließlich in den aktuellen Programmierumgebungen wiederfinden, sodass sie ihre Entwicklung maßgeblich beeinflussten bzw. Ansätze der zukünftigen Weiterentwicklung aufzeigen (*GP*).

Morphic ist ein Framework, das es dem Benutzer ermöglicht, auf einfache Art und Weise ein grafisches Userinterface zu erstellen und mit diesem zu interagieren. Das Framework basiert dabei sowohl auf einer direkten Manipulation der jeweiligen Objekte (anhand von Kontextmenüs) als auch einer Manipulation innerhalb von Programmen. Ursprünglich wurde es als Teil der Sprache Self bei Sun entworfen, die erstmals in den 80ern und 90ern als experimentell verwendetes Testsystem für Sprachendesign eingesetzt wurde. Grundlegend für diese Umgebung ist, dass alle sichtbaren Objekte sog. „Morphs“ sind und somit etwa auf Ereignisse reagieren (wie etwa Mausklicks), eine Überlappung mit anderen Objekten erkennen oder aus Teilobjekten bestehen können [SWM13]. Dadurch besteht die Möglichkeit, Objekte zu verschachteln, sodass diese in einer Teil-Ganzes-Beziehung stehen, wodurch jeder „Morph“ eine bestimmte Stelle in einer Hierarchie einnimmt (an deren Spitze

die „World“ bzw. die Bühne steht). Einige der Konzepte finden sich etwa in *Scratch* wieder: So dürfte das Prinzip der Verschachtelung als Vorlage für „Nested Sprites“ in *Snap!* gedient haben.

Smalltalk ist das Resultat langwieriger Forschung, deren Ziel es war, dem Benutzer die Interaktion mit einem Computer auf einer funktionalen Ebene möglich zu machen [GR83, 8]. Es handelt sich hier um eine objektorientierte Programmiersprache, die gezielt zum Einstieg in die Programmierung konzipiert wurde. Wichtig war laut Entwickler Alan Kay die Bereitstellung einer Umgebung, in der Erkundung belohnt wird (nach Montessori), die enaktives, ikonisches und symbolisches Lernen ermöglicht und fördert (nach Piaget und Bruner), in der die Magie im Bekannten steckt (Negroponte) und die als verstärkender Spiegel für die Intelligenz des Benutzers dient (Coleridge, [Ka96, 33]). Die Sprache basiert auf dem Kernkonzept „Alles ist ein Objekt“. Ausgehend von diesem Konzept gelten bestimmte Regeln für Objekte, wie etwa, dass sie untereinander mithilfe von Nachrichten (die Objekte sind) kommunizieren oder, dass Objekte über ein eigenes Gedächtnis verfügen (das wiederrum ein Objekt ist). *Smalltalk* und seine spätere Weiterentwicklung, bzw. Modifikation *Squeak* sind in ihrer Entwicklungslaufbahn zur Basis von diversen visuellen Programmierumgebungen geworden, wie z. B. *Scratch* oder *Snap!*. Die objektorientierte Natur dieser Sprache, in der mit grafischen Objekten anhand von Nachrichten kommuniziert wird (etwa eine Nachricht an eine Box, sich zu drehen oder zu vergrößern) wurde als sehr motivierend aufgefasst, sodass bereits Kinder in der Lage waren, Malprogramme oder Notenblatt-Lesesysteme zu entwerfen. Diese Tatsache könnte ein Grund dafür sein, dass sich das Prinzip des Nachrichtenversands auch in *Scratch* oder *Snap!* wiederfindet.

GP ist der neueste Vertreter der Sprachenfamilie von *Scratch* und befindet sich momentan noch in der Entwicklungsphase. Es wird derzeit u.A. von Jens Mönig und John Maloney entwickelt und basiert auf der Vision, dass dem Benutzer weniger Grenzen vorgegeben werden sollen als etwa in *Scratch*. Das bedeutet, dass komplexe Programmierkonzepte, die in den verwandten Sprachen aus didaktischen Gründen fehlen, umsetzbar sein sollen. Außerdem war es das Ziel, eine erweiterbare Plattform zu erschaffen - der interne Code der gesamten Programmierumgebung ist einsehbar und manipulierbar, sodass Modifikationen inhärent unterstützt werden. Die Programmierumgebung ist *Scratch* und *Snap!* sehr ähnlich, weist jedoch einige Unterschiede auf: Sie ist beinahe vollständig in sich selbst implementiert, d.h. wenn Codeblöcke untersucht werden, um deren interne Funktionsweise sichtbar zu machen, enthüllt sich wiederrum Code aus grafischen Blöcken. Dieses Konzept findet sich durchgehend bis zur untersten Ebene, sog. „primitives“, also primitiven Funktionen, die nicht weiter einsehbar sind, da sie in C implementiert sind. Neben diesem Konzept finden sich in *GP* jedoch noch weitere konzeptionelle Neuerungen: Wie in den anderen Vertretern der Sprachenfamilie verfügt es über einen nichtatomaren Interpreter. Außerdem ist es in der Lage, „Nested Sprites“ zu erschaffen, mit Dateisystemen zu interagieren und grafische Codeblöcke als Text darzustellen. Alle diese Phänomene werden im folgenden Kapitel näher erläutert.

4 Neue Konzepte blockbasierter Programmierung

Im Folgenden sollen nun fünf der wichtigsten konzeptionellen Unterschiede und Neuerungen aufgezeigt werden. Die Konzepte sind bedeutsam, da sie klare Neuerungen darstellen, die durch visuelle, blockbasierte Programmiersprachen eingeführt wurden: Sie existieren in dieser Form nicht in den bekannten textbasierten Programmierumgebungen, die momentan noch etwa in der Schule Verwendung finden. Daher ist es wichtig, dass Lehrer sich dieser Neuerungen bewusst sind, sodass sie didaktisch behandelt werden können. Diese Konzepte finden sich in den aktuellen Vertretern blockbasierter Programmierumgebungen: *Scratch*, *Snap!* und *GP*.

Nichtatomare Interpreter und Debugging Die wichtigsten Neuerungen haben eins gemeinsam: Sie sind auf didaktische Gründe zurückzuführen, d.h. die Motivation war es, die Programmiersprache intuitiver zu gestalten. Hierzu gehört z. B. die Eigenschaft, dass Codeblöcke verzögert ausgeführt werden. Das bedeutet, dass bestimmte Skripte langsamer ausgeführt werden als die Umgebung (und der Computer) es eigentlich zulassen würde(n). Ohne diese Verzögerung könnten z. B. Sprites sofort nach Programmstart die Grenzen der Leinwand verlassen, ohne dass für den Benutzer ersichtlich wäre, warum. Um Anfänger vor diesem Phänomen zu bewahren, wurde die Verzögerung für bestimmte Block-Arten eingeführt: Alle Schleifen-, Warte- und Bewegungsblöcke haben diese Verzögerung. Diese Verzögerung kann etwa in *Snap!* mithilfe zweier verschiedener Maßnahmen umgangen werden: Entweder durch die Verwendung eines speziellen „Warp“-Blocks oder durch Aktivierung des “Turbo”-Modus.

Dieser Unterschied in der Ausführung des Codes ist ein wichtiges Konzept, das für fortgeschrittene Lerner u.U. zu großer Verwirrung führen kann. Daher ist es wichtig, bei der Vermittlung der Programmierung auf die neuen Anforderungen einzugehen. Besonders für Debugging ist das Konzept der verzögerten Ausführung nützlich: *Snap!* bietet beispielsweise die Möglichkeit, die Ausführungsgeschwindigkeit von Blöcken manuell zu regeln. Die Option „visible stepping“ erweitert die Benutzeroberfläche um einen Schieberegler, der die Ausführungsgeschwindigkeit des Codes stufenlos reduziert, sodass der Programmablauf einfach verfolgt und nachvollzogen werden kann. Der Verstehensprozess wird zusätzlich durch Hervorhebung des momentan ausgeführten Codeblocks unterstützt. Hier wird deutlich, wie die Visualisierung des Codes (nicht nur auf der Leinwand, sondern auch in der direkten Darstellung) verständnisfördernd aufbereitet werden kann.

Klassen-/Objekt-Darstellung Auch die Darstellung von Klassen und Objekten (als Instanzen von Klassen) unterliegt einigen Änderungen im Kontext der blockbasierten Programmierung: Hier wird der „Prototyping“-Ansatz verwendet, nach dem Objekte („Sprites“) geklont werden und dadurch neue Objekte erzeugt werden, die ihrerseits exakte Kopien sind und alle Codeblöcke und Attribute des ursprünglichen Objektes übernehmen. Dadurch „beschreibt“ der Benutzer ein Beispiel einer Instanz, keinen abstrakten Bauplan (Klasse). Dieser Prototyp kann dann weitere Instanzen von sich selbst erzeugen, die seinen Code teilen. Diese Form der Darstellung wurde von der Arbeit von Henry Lieberman inspiriert,

der verdeutlichte, dass durch das Prinzip von Prototypen „Standards“ und Abweichungen von solchen intuitiv verständlich sind [Li86]. Der verständnisfördernde Charakter des Prototypen-Konzepts wurde als didaktisch sinnvoll erfasst, sodass es in vielen blockbasierten Programmiersprachen, wie etwa *Scratch* und *Snap!*, Verwendung findet. Modrow zeigt auf, wie hier das objektorientiete Konzept der Vererbung über das Delegations-Modell realisiert werden kann [Mo13]. Prinzipiell sind zwei Arten des Klonens zu unterscheiden: Das Klonen zur Laufzeit (durch Ausführung der entsprechenden Codeblöcke) und das Klonen zur Programmierzeit (durch „duplicate“-Befehl im Kontextmenü eines Objekts). Ersteres erschafft eine exakte Kopie des Originals, die bei Programmende entfernt wird, während Letzteres einen permanenten Klon erschafft, dessen Aussehen und Verhalten danach verändert werden können ohne das ursprüngliche Objekt zu beeinflussen (Abb. 1).

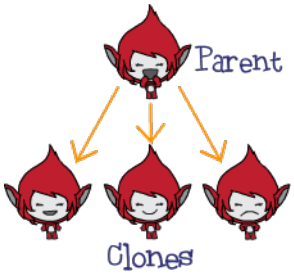

Abb. 1: Visuelle Darstellung des Klon-Konzepts in *Scratch* [Sc17a]

Abb. 2: Klasse und ihre Instanzen in *GP*

Auch GP folgt der Idee des Prototyping, greift aber für die Darstellung auf das von Smalltalk bekannte und aus der textbasierten Programmierung verbreitete Klasse-Objekt-Konzept zurück, wie es beispielsweise in Java oder C++ implementiert wird. Die Abb. 2 ist ein Ausschnitt aus der Programmieroberfläche. Es ist eine Auflistung aller Klassen zu sehen, die im laufenden Projekt Verwendung finden und rechts davon eine Übersicht über die Instanzen dieser Klasse (und ihrem momentanen Aussehen). Für beide Bereiche gibt es +-Knöpfe, deren Betätigung eine neue Klasse bzw. eine Instanz von ihr erzeugt. Objekte derselben Klassen teilen sich alle den Code, der für diese Klasse definiert wurde - sie reagieren auf dieselben Ereignisse und führen dieselben Codeblöcke aus. Jedes Objekt ist dadurch die grafische Instanz einer Klasse. Wird eine neue Klasse erzeugt, so wird gleichzeitig die erste Instanz dieser Klasse auf der Leinwand erschaffen, die somit als Prototyp fungiert. Existiert nur eine Instanz pro Klasse, so ist das Verhalten der Programmierumgebung funktional ähnlich zu *Scratch* – mit dem Unterschied, dass eine explizite Auswahl des Klassennamens aus der Liste (vgl. Abb. 2) notwendig ist, um die konkrete Instanz anwählbar zu machen. Werden von einer Klasse mehrere Instanzen erzeugt, machen sich die Unterschiede zu *Scratch* und *Snap!* bemerkbar: In *GP* bezieht sich der Code immer auf eine ganze Klasse. Wird dieser verändert, so ändert er sich für alle Instanzen dieser Klasse (analog zum Laufzeit-Klonen in *Scratch/Snap!*). *Scratch* erlaubt hingegen zusätzlich, nach dem Klonen den Code eines Objektes zu ändern, ohne dabei den anderer Klone oder des „Originals“ zu beeinflussen (durch Klonen zum Programmierzeitpunkt).

Es wird deutlich, dass die Frage, in welcher Form das Klassen/Objekt-Prinzip vermittelt werden soll, zentral ist. Die Unterschiede in der Darstellung von Vererbung und Klassen/Objekten müssen für eine didaktisch wertvolle Vermittlung beachtet und reflektiert werden.

„Nested Sprites“ Das Konzept der „Nested Sprites“ ist ein gutes Beispiel dafür, dass visuelle Programmiersprachen neue Problemlöseansätze eröffnen (die im Unterricht thematisiert werden müssen). Es handelt sich hierbei um eine örtliche Verbindung von Objekten bzw. von „Sprites“ auf der Leinwand: Sie stehen in einer Teil-Ganzes-Beziehung zueinander und nehmen dadurch einen Platz in einer Objekthierarchie ein. Ein Objekt kann aus mehreren Teilen bestehen und selbst Teil eines Objektes sein. Die Rotationen und Bewegungen der beteiligten Objekte sind demnach voneinander abhängig (bewegt sich der Besitzer, so bewegen sich alle seine Teilobjekte mit ihm).

Das Konzept ist charakteristisch für visuelle Sprachen, da es nur durch die visuelle Repräsentation von Objekten möglich gemacht wird (gäbe es keine Leinwand, um Objekte zu visualisieren, gäbe es auch das Konzept der verschachtelten Objekte nicht). Es kann beispielsweise verwendet werden, um Simulationen zu implementieren, wie etwa die mechanische Funktionsweise eines Verbrennungsmotors: Ein Teil bewegt sich, wodurch sich wiederrum andere Teile bewegen, sodass schließlich eine Verbrennung stattfindet (Abb. 3).

Abb. 3: Vereinfachte Simulation eines Verbrennungsmotors aus GP

In *Snap!* kann eine solche Beziehung zwischen Objekten durch Drag-and-drop erzeugt werden, sodass Objekte an andere Objekte angefügt werden. In *GP* muss hierzu ein Objekt mit einem Anderen überlappen, woraufhin per Kontextmenü eine Zuweisung „attach“ verfügbar wird.

„First class Strukturen“ Smalltalk, Morphic, *Snap!* und *GP* setzen ein aus der funktionalen Programmierung bekanntes und mächtiges Konzept um, das durch die visuelle Programmierung intuitiv verständlich und damit gut handhabbar umgesetzt wird: Alle vorhandenen Strukturen und Objekte sind „first class“. Strukturen und Objekte werden als „First class objects“ bezeichnet, wenn sie folgende Eigenschaften erfüllen (vgl. [HM17]):

- sie können Werte von Variablen sein,
- sie können Parameter und Rückgabewerte von Funktionen sein,
- sie sind anonym (werden bei Erstellung nicht benannt),
- sie können Teil eines Aggregats sein.

Geprägt wurde der Begriff von Christopher Strachey, der um 1960 forderte, dass jeder

Datentyp einer Sprache „first class data“ sein sollte [Sc17b], sodass jeder Datentyp völlig uneingeschränkt benutzbar ist. Dies ist eines der fundamentalen Prinzipien von *Snap!*. Durch die daraus resultierenden Anwendungsmöglichkeiten (wie etwa Listen von Listen) bieten sich neue Herangehensweisen und Lösungsmöglichkeiten für bekannte Probleme an. Beispielsweise können mapping-Funktionen über Listen durchgeführt werden oder Blöcke als Eingabeparameter für andere Blöcke fungieren.

Entwicklermodus, Block-to-Text-Slider, Class Browser Diese Konzepte stehen bislang nur in der Umgebung *GP* zur Verfügung. Der Entwicklermodus stellt eine optional zuschaltbare Erweiterung der Funktionalität der Programmierumgebung dar. Bei aktivem Entwicklermodus werden einige Paletten erweitert, wie z. B. Variablen um den Typ *script* oder die „New Class“-Funktion um die „helper“-Variante. Auch der „Class Browser“ wird zugänglich: Hier findet sich der Systemcode von *GP* (näheres hierzu weiter unten).

Die wohl interesanteste Erweiterung offenbart sich als Schieberegler in der Menüleiste: ein stufenloser Regler der Codedarstellung von Blöcken zu Text. Dieser soll die Brücke zu textbasierten Programmiersprachen repräsentieren und die Äquivalenz der beiden Repräsentationen verdeutlichen – je nach subjektiver Präferenz kann der Benutzer seine Bedienung entweder blockbasiert oder textbasiert einstellen (vgl. Abb. 4).

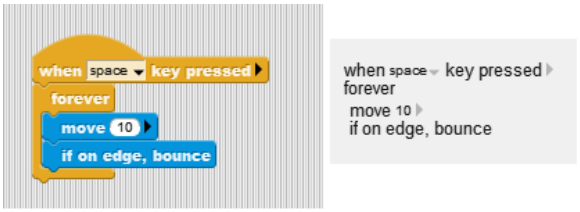

Abb. 4: Beispielcode in Block- und in Textmodus

Fließende Übergänge ergeben sich aus der alternativen Bedienung mit der Tastatur. Wie *Snap!* bietet *GP* die Möglichkeit, Codeblöcke durch Tastatureingaben (mit Autovervollständigung) zu erzeugen: Die Umgebung reagiert auf neu eingegebene Zeichen damit, dass alle verfügbaren Kategorien nach passenden Blöcken durchsucht und in einer Liste dargestellt werden. Der passendste Block wird stets ganz oben in der Liste angezeigt, und eine Bestätigung per Enter-Knopf fügt diesen an die momentan ausgewählte Stelle im Skript an. Die Kategorie, der der angefügte Block angehört, wird zudem direkt angewählt, sodass ähnliche Blöcke sofort sichtbar sind – eine Funktionalität, die *Scratch* nicht bietet. Dadurch ist eine vollständige Bedienung der Oberfläche durch die Tastatur möglich.

Zweifellos ist dieser Eingabemodus, v.a. gepaart mit dem Block-zu-Text-Slider auch als Brücke zu textbasierten Programmiersprachen intendiert. Wenn beide Modi aktiviert sind, gibt die Benutzung das Gefühl, eine textbasierte Programmierumgebung mit starker Syntaxunterstützung zu verwenden (wie beispielsweise Strides in *Greenfoot*). Hier zeigt sich erneut, dass *GP* als „Lösung“ dafür erschaffen wurde, dass das visuelle Programmieren

nicht „echt“ erscheint, zu unprofessionell präsentiert wird und nicht wirklich übertragbar auf andere Programmiersprachen scheint.

Das letzte Konzept, der Class Browser, verkörpert die Vision von *GP*, dass selbst komplexeste Projekte realisierbar sind – „high ceiling“ nach Seymour Papert. Im Class Browser sind alle Skripte gelistet, die in der Palette zu finden sind und darüber hinaus alle Skripte, die das interne Verhalten der Umgebung beschreiben. Der Class Browser macht es möglich nachzuvollziehen, wie die Funktionsweisen der Programmierumgebung bis zur untersten Ebene hin implementiert sind (Abb. 5); so lässt sich die Implementierungshierarchie von Skripten bis zur primitiven Ebene hin verfolgen. Nutzbar ist er dadurch beispielsweise, um vordefinierte Skripte zu ändern, ergänzen oder zu löschen und somit eine eigene *GP*-Version völlig den eigenen Ansprüchen entsprechend zu erstellen.

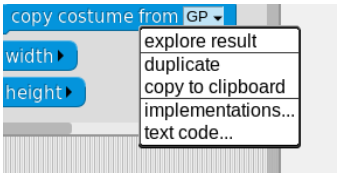

Abb. 5: Verwendung des Class Browsers: „implementations...“

Bei diesen Konzepten ist eine besondere Bedeutung für Schüler sichtbar: Anfangs werden komplexe Funktionen und Blöcke verborgen, sodass eine niedrige Einstiegshürde gegeben ist. Bei aktiviertem Entwickler-Modus wird dann die selbstständige Erkundung der Programmierumgebung und ihres Codes durch den Class Browser möglich, sodass interessierten Schülern kaum Grenzen gesetzt sind. Gleichzeitig ermöglicht der Block-zu-Text-Schalter eine subjektive Regelung der Programmieroberfläche, sodass Lerner sie je nach Präferenz, Kenntnisstand oder Programmierfähigkeit individuell anpassen können.

5 Fazit

Die wachsende Präsenz blockbasierter Programmiersprachen kann für Schüler große Vorteile mit sich bringen. Dennoch birgt diese neuartige Form der Programmierung auch Herausforderungen durch neue Konzepte, die verstanden und erforscht werden müssen. Prinzipien wie der nichtatomare Interpreter, der Skripte verlangsamt ausführt, sodass diese intuitiver ablaufen oder Nested Sprites, die beinahe ausschließlich auf der visuellen Repräsentation von Objekten basieren, sind neu und aus didaktischer Sicht noch praktisch unerforscht. Trotz der intuitiven Natur visueller und insbesondere blockbasierter Programmiersprachen ist hier also noch viel Forschungsarbeit zu leisten - bisher wurden erst Ansätze für eine Didaktik visueller Programmiersprachen geliefert.

Literaturverzeichnis

[DHH15] Dwyer, H.; Hill, C.; Hansen, A. et al.: Fourth Grade Students Reading Block-Based Programs: Predictions, Visual Cues, and Affordances. In: Proceedings of the eleventh annual International Conference on International Computing Education Research. ACM, S. 111–119, 2015.

[GR83] Goldberg, A.; Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., 1983.

[HM17] Snap! Reference manual online, snap.berkeley.edu/SnapManual.pdf, Stand: 09.06.2017.

[Ka96] Kay, A.: The early history of Smalltalk. In: History of programming languages—II. ACM, S. 511–598, 1996.

[Le10] Lewis, C.: How programming environment shapes perception, learning and goals: logo vs. scratch. In: Proceedings of the 41st ACM technical symposium on Computer science education. ACM, S. 346–350, 2010.

[Li86] Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: ACM Sigplan Notices. Jgg. 21. ACM, S. 214–223, 1986.

[LKG17] Scratch: Statistics, scratch.mit.edu/statistics/, Stand: 09.06.2017.

[Mo11] Modrow, E.: Visuelle Programmierung – oder: Was lernt man aus Syntaxfehlern? In (Thomas, M., Hrsg.): Informatik in Bildung und Beruf. Jgg. 14 in Lecture Notes in Informatics (LNI), Köllen, Bonn, S. 27–36, 2011.

[Mo13] Informatik mit BYOB / Snap!, www.uni-goettingen.de/de/informatik-mit-byob/423680.html, Stand: 09.06.2017.

[PB15] Price, T.; Barnes, T.: Comparing textual and block interfaces in a novice programming environment. In: Proceedings of the eleventh annual International Conference on International Computing Education Research. ACM, S. 91–99, 2015.

[Re02] Resnick, M.: Rethinking learning in the digital age. The Global Information Technology Report: Readiness for the Networked World. Oxford University Press, 2002.

[RMH14] Ruf, A.; Mühling, A.; Hubwieser, P.: Scratch vs. Karel: impact on learning outcomes and motivation. In: Proceedings of the 9th Workshop in Primary and Secondary Computing Education. ACM, S. 50–59, 2014.

[RMMH09] Resnick, M.; Maloney, J.; Monroy-Hernández, A. et al.: Scratch: programming for all. Communications of the ACM, 52(11):60–67, 2009.

[Sc17a] Scratch-Wiki: Cloning, wiki.scratch.mit.edu/wiki/Cloning, Stand: 09.06.2017.

[Sc17b] Scratch-Wiki: Snap!, wiki.scratch.mit.edu/wiki/Snap, Stand: 09.06.2017.

[St15] Strecker, K.: Grafische Programmiersprachen im Abitur. In (Jens Gallenbacher, Hrsg.): INFOS 2015: Informatik allgemeinbildend begreifen. Jgg. 16 in Lecture Notes in Informatics (LNI), Köllen, Bonn, S. 293–300, 2015.

[SWM13] Squeak-Wiki: Morph, wiki.squeak.org/squeak/1820, Stand: 09.06.2017.

[WW15] Weintrop, D.; Wilensky, U.: Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In: ICER. Jgg. 15, S. 101–110, 2015.