Solist – eine Entwicklungsumgebung für Miniprogrammierwelt-Simulatoren

Dietrich Boles

Universität Oldenburg
Fakultät II, Department für Informatik
Escherweg 2
26121 Oldenburg
boles@informatik.uni-oldenburg.de

Abstract: Miniprogrammierwelten sind spezielle didaktische Modelle, die dazu dienen, Programmieranfängern einen vereinfachten Einstieg in die komplexe Welt der Programmierung (Algorithmik) zu ermöglichen. Die Programmieranfänger entwickeln Programme, in denen sie auf der Grundlage einiger weniger vorgegebener Befehle virtuelle Akteure durch virtuelle Landschaften steuern und gegebene Aufgaben lösen lassen. Zu den Modellen existieren im Allgemeinen einfach zu handhabende Programmierwerkzeuge (Simulatoren) zum Erstellen, Ausführen und Testen von entsprechenden Programmen. In diesem Artikel wird die Entwicklungsumgebung Solist vorgestellt, die es Lehrerinnen und Lehrern mit normalen Programmierkenntnissen ermöglicht, innerhalb kurzer Zeit eigene Miniprogrammierwelten und dazugehörige Simulatoren zu entwickeln bzw. existierende Miniprogrammierwelten an spezifische Anforderungen anzupassen.

1 Miniprogrammierwelten

Das Erlernen der Programmierung ist keine einfache Angelegenheit. Sicher gibt es Schülerinnen und Schüler, die sehr schnell damit zurechtkommen. Viele haben aber Probleme mit der Komplexität der Thematik und sind schnell frustriert. Gefordert sind daher Hilfsmittel bzw. Werkzeuge, die den Einstieg in die Welt der Programmierung vereinfachen und motivierend gestalten.

Ein weit verbreitetes Werkzeug für Programmieranfänger mit dem Fokus auf die objektorientierte Modellierung ist bspw. *BlueJ* [BK09]. Durch die Möglichkeit der Interaktion mit Objekten werden Schülerinnen und Schülern mit Hilfe von BlueJ sehr schön Konzepte wie Objekte, Klassen oder der Aufruf von Methoden für Objekte vor Augen geführt.

Für einen eher algorithmisch orientierten Einstieg in die Programmierung existieren seit vielen Jahren so genannte „micro worlds“ oder Miniprogrammierwelten (MPWs). Sie basieren auf dem Prinzip, dass die Programmieranfänger Programme entwickeln, mit denen sie auf der Basis einiger weniger vorgegebener Befehle virtuelle Akteure durch virtuelle Landschaften steuern und gegebene Aufgaben lösen lassen.

Bekannte Beispiele[1] für MPWs sind:

- Hamster-Modell [Bo08]: Ein Hamster wird durch ein virtuelles Kornfeld gesteuert, wobei er Körner aufnehmen und ablegen kann (Befehle: `vor`, `linksUm`, `nimm`, `gib`). Er kann dabei überprüfen, ob der Weg gegebenenfalls durch Mauern versperrt ist, ob Körner vorhanden sind und ob er aktuell Körner im Maul hat (Test-Befehle: `vornFrei`, `kornDa`, `maulLeer`).
- Kara, der Marienkäfer [RNH04]: Ein Marienkäfer namens Kara kann Kleeblätter pflücken und Pilze verschieben und muss dabei Wurzeln umgehen.
- Turtle-Graphics [Pa80]: Eine Schildkröte wird über ein Zeichenblatt gesteuert und kann mit Hilfe eines Stiftes, den sie bei sich trägt, Zeichnungen erstellen.

MPWs setzen sich aus 3 Bestandteilen zusammen: dem (didaktischen) Modell, der Programmiersprache und einer Lernumgebung. Im didaktischen Modell einer MPW werden der generelle Aufbau und die Bestandteile der virtuellen Welt festgelegt. Weiterhin wird definiert, welche Befehle den Programmieranfängern zur Steuerung des Akteurs zur Verfügung stehen sollen.

Das didaktische Modell ist prinzipiell unabhängig von der Wahl der Programmiersprache. Hierzu existieren 3 Alternativen (vgl. [Br94]): Es wird eine eigene Programmiersprache definiert, es wird eine existierende Programmiersprache gewählt oder von einer existierenden Programmiersprache wird eine Auswahl an Konzepten übernommen.

Für den Umgang mit einer MPW bedarf es einer Lernumgebung, häufig als „Simulator“ bezeichnet, die es den Programmieranfängern ermöglicht, Programme zu entwickeln und zu testen. Solche Simulatoren sollten, um den Programmieranfängern das Leben nicht noch zusätzlich zu erschweren, einfach und intuitiv bedienbar sein und mindestens folgende Komponenten beinhalten: einen Editor zum Eingeben und Verwalten von Programmen, einen Compiler[2], einen „Welt-Gestalter“ zum interaktiven Gestalten und Verwalten von virtuellen Welten, in denen die Akteure dann agieren können, eine Simulationskomponente zum Ausführen und Steuern von Programmen und einen Debugger[3].

Einige MPWs, wie der Hamster und Kara, unterstützen dabei auch mehrere Programmiersprachen und es gibt auch MPWs, wie die Turtle-Graphics, zu denen mehrere unterschiedliche Lernumgebungen existieren.

Für den Einsatz von MPWs in Kursen für Programmieranfänger lassen sich folgende Vorteile nennen:

---

[1] Übersichten über bzw. Links zu existierenden MPWs finden sich bspw. unter http://www.schule.bayern.de-/karol/data/uebersicht.pdf bzw. http://www-is.informatik.uni-oldenburg.de/~dibo/hamster/links.html.
[2] nicht notwendig bei Skriptsprachen wie Python oder Ruby
[3] wobei ein Debugger bei einigen existierenden MPWs leider fehlt

- Die Komplexität der Programmierung wird durch den reduzierten Befehlssatz deutlich verringert[4].
- Die Ausführung der Befehle wird auf dem Bildschirm durch entsprechende Aktionen des jeweiligen Akteurs visualisiert. Programmierfängern wird also unmittelbar vor Augen geführt, was ihr Programm bewirkt (und ob es korrekt ist). Die visuelle (teilweise sogar multimediale) Repräsentation der Welt auf dem Bildschirm bedingt eine wesentlich größere Motivation als die bloße Ausgabe von Zahlenkolonnen auf einer Konsole (vgl. etwa [Ha08]).
- Programmieranfänger können in wenigen Minuten ihr erstes Programm erstellen. Sie müssen sich nicht zuvor, wie in vielen Lehrbüchern zu Programmiersprachen üblich, zunächst mit komplexen (und durch die Nähe zur Mathematik häufig abschreckenden) Konzepten, wie Variablen, Typen oder Werten auseinandersetzen.
- Es gibt unzählige Möglichkeiten für Aufgabenstellungen, die sich die Programmieranfänger bei Bedarf auch selbst ausdenken können.

In der Literatur wird durchgängig von guten Erfahrungen beim Einsatz von MPWs in Kursen für Programmieranfänger berichtet. Es wird jedoch bspw. in [Bo05] darauf hingewiesen, dass es bei einem zu langen Einsatz einer MPW insbesondere den besseren Schülerinnen und Schülern schnell langweilig werden kann, und auch im WWW finden sich durchaus kritische Anmerkungen, wie bspw. beim Einsatz von Kara in einer 11. Klasse[5] („fehlender Realitätsbezug“, „mit der Zeit wird es langweilig und eintönig“, „kindisch“, „immer nur Kleeblätter und Bäume“).

Die Entwicklung neuer MPWs zu fördern, dabei die Vorteile von MPWs nutzen zu können und gleichzeitig zu versuchen, die genannte Nachteile zu beseitigen, das ist das Ziel des im weiteren Verlauf dieses Artikels vorgestellten Werkzeuges Solist.

2 Solist

Solist ist ein Werkzeug für die Entwicklung neuer MPWs inklusive dazugehöriger Simulatoren. Es besteht aus einer graphisch-interaktiven Entwicklungsumgebung, dem Solist-Simulator, und einer vordefinierten Java-Klassenbibliothek, der Theater-Klassenbibliothek bzw. Theater-API. Zielgruppe von Solist sind bspw. Informatik-Lehrerinnen und -Lehrer, die Simulatoren für selbst entwickelte MPWs implementieren und ihren Schülerinnen und Schülern zur Verfügung stellen wollen. Voraussetzung zur Benutzung von Solist ist die Beherrschung der Basissprachkonzepte von Java (Klassen, Objekte, Attribute, Methoden, Anweisungen, Variablen, Schleifen, …). Nicht notwendig sind Kenntnisse der JDK-Klassenbibliothek bspw. zur Entwicklung von graphischen Oberflächen (GUIs) mit Java-AWT oder Java-Swing.

[4] daher wird in der Literatur auch von „mini languages“ gesprochen[Br97]
[5] siehe http://www.oberstufeninformatik.de/info11/rueckmeldungen.html

2.1 Solist-Simulator

Solist ist ein Werkzeug des Programmier-Theaters (siehe auch Abschnitt 3). Ihm liegt die Metapher der Theaterwelt zugrunde:

- Projekte werden in Solist als Theaterstücke (play) bezeichnet.
- Handlungsumfeld ist eine Bühne (stage). Dabei kann es zu einem Stück durchaus mehrere verschiedene Bühnenaufbauten geben.
- Auf der Bühne agiert in Solist ein einzelner Schauspieler (actor) als Solist (daher der Name des Werkzeugs).
- Es können weitere Requisiten (prop) auf der Bühne platziert werden.
- Die Ausführung eines Programms entspricht einer Aufführung (performance) eines Theaterstücks.

Die Theater-Metapher spiegelt sich auch im Erscheinungsbild des Solist-Simulators wider (siehe Abbildung 1).

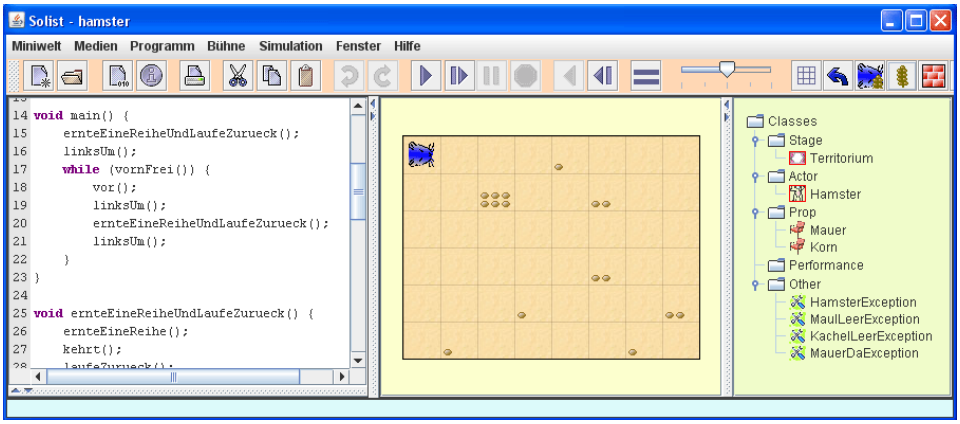

Abbildung 1: Solist-Simulator mit Hamster-Theaterstück

Im rechten Bereich des Solist-Simulators können Klassen auf der Grundlage der im kommenden Abschnitt vorgestellten Theater-API definiert werden. Im Theaterstück *Hamster* aus Abbildung 1, das als Beispiel fungieren soll, definiert eine von der Theater-Klasse `Stage` abgeleitete Klasse `Territorium` den Aufbau und die Gestaltung der Bühne. Die aktuelle Bühne wird im mittleren Teil des Fensters angezeigt.

Der Solist ist in diesem Beispiel ein Hamster. Die Befehle, die er kennen soll, werden in einer Klasse `Hamster` definiert, die von der Theater-Klasse `Actor` abgeleitet ist. Dem Solist kann ein Bild oder Icon zugeordnet werden, das ihn auf der Bühne repräsentiert. Als Requisiten existieren im Hamster-Modell Mauern und Körner, denen als Objekte von den als Unterklassen der Theater-Klasse `Prop` definierten Klassen `Mauer` und `Korn` ebenfalls ein Bild zugeordnet und auf der Bühne platziert werden kann.

Weiterhin ist es möglich, Hilfsklassen mit einzubeziehen (im Beispiel die Exception-Klassen) sowie von der Theater-Klasse `Performance` Klassen abzuleiten und hierin Variationen der Ausführung des Programms wie bspw. die Anfangsgeschwindigkeit festzulegen.

Im linken Teil des Solist-Simulators befindet sich ein Editor, mit dem Test-Programme geschrieben werden können, die den Solist mittels der für ihn in der entsprechenden `Actor`-Klasse definierten Befehle auf der Bühne agieren lassen. Die Ausführung bzw. Steuerung dieser Programme erfolgt dabei über die Steuerung-Buttons der Toolbar. Weiter ist es für den Entwickler möglich, in die Toolbar eigene Buttons zu integrieren, über die sich mit Hilfe entsprechend definierter Klassen die Bühne gestalten lässt.

Nach Abschluss der Entwicklungsarbeiten kann ein MPW-Entwickler über das Miniwelt-Menü einen Simulator für seine MPW generieren lassen und diesen Programmieranfängern zur Verfügung stellen (siehe Abbildung 2). Der generierte Simulator enthält dabei automatisch alle in Abschnitt 1 aufgelisteten Komponenten (Editor, Compiler, Bühnengestalter, Simulationskomponente, Debugger).

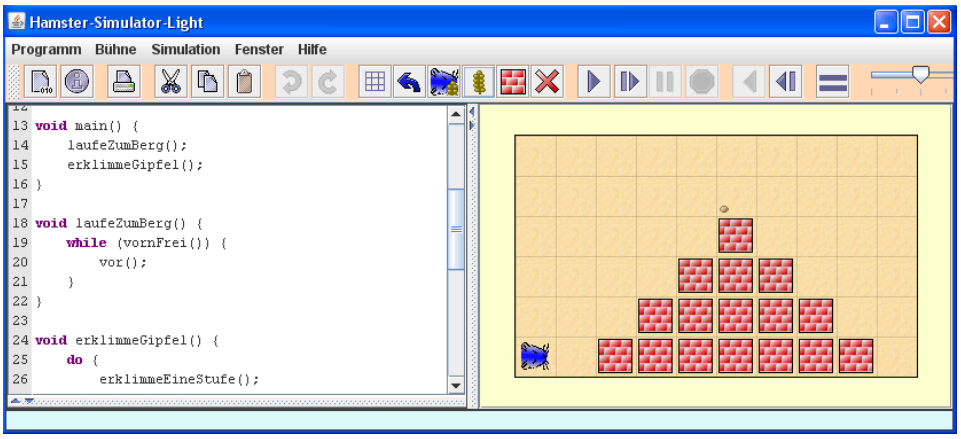

Abbildung 2: Generierter Hamster-Simulator

2.2 Theater-API

Die Theater-API umfasst eine Menge an vordefinierten Klassen, die das Grundverhalten aller beteiligten Objekte implementieren und weitere nützliche Features zur Verfügung stellen. Bezüglich der angebotenen Funktionalitäten ist sie ähnlich aufgebaut wie die API von *Greenfoot* [Kö09]. Die wichtigsten Klassen der API sind: `Stage`, `Component`, `Actor`, `Prop` und `Performance`.

Stage Die Gestaltung einer konkreten Bühne kann durch die Definition einer von der Klasse `Stage` abgeleiteten Klasse erfolgen. Eine Bühne besteht dabei aus einem rechteckigen Gebiet, das sich aus gleichförmigen quadratischen Zellen zusammensetzt. Die Größe der Bühne wird über einen Konstruktor durch die Anzahl an Spalten und

Reihen sowie die Größe der Zellen in Pixeln festgelegt. Hierdurch wird ein Koordinatensystem definiert, das zum Platzieren von Akteuren und Requisiten (im Folgenden durch den Begriff *Komponente* zusammengefasst) auf der Bühne dient. Das Koordinatensystem ist nicht endlich, so dass sich Komponenten auch außerhalb der Bühne („backstage“) befinden können, also (zwischenzeitlich) nicht sichtbar sind.

Neben einer Menge von Getter-Methoden zum Abfragen des Zustands einer Bühne sowie Methoden zur Verwaltung von Maus- und Tastatur-Events lassen sich die Methoden der Klasse `Stage` einteilen in Methoden zur Gestaltung der Bühne und Methoden zur „Kollisionserkennung“.

Zu den Gestaltungsmethoden gehören `add`- und `remove`-Methoden zum Platzieren und Entfernen von Komponenten auf bzw. von der Bühne. Neben der Spalte und Reihe, in die eine Komponente platziert werden soll, kann bei den `add`-Methoden zusätzlich eine z-Koordinate angegeben werden, die eine dritte Dimension auf der eigentlich zweidimensionalen Bühne simuliert. Weiterhin existieren Methoden zum Festlegen eines Hintergrundbildes für die Bühne.

Über die Kollisionserkennungsmethoden lässt sich zur Laufzeit u.a. ermitteln, welche Komponenten sich aktuell in bestimmten Bereichen der Bühne aufhalten oder welche Komponenten (genauer gesagt deren Icons) sich berühren oder überlappen.

Component, Actor und Prop Die Klasse `Component` definiert Methoden zur Verwaltung von Akteuren und Requisiten, die sie an die von ihr abgeleiteten Klassen `Actor` und `Prop` vererbt. Die wichtigsten Methoden der Klasse `Component` sind Methoden, um Akteuren und Requisiten ein Icon zuzuordnen und sie auf der Bühne bewegen, also umplatzieren oder bspw. rotieren zu können. Weiterhin sind eine Menge von Getter-Methoden definiert, um zum einen ihren Zustand abfragen und zum anderen das aktuelle Bühnen- sowie Performance-Objekt ermitteln zu können.

Wie die Klasse `Stage` enthält die Klasse `Component` darüber hinaus Kollisionserkennungsmethoden zum Entdecken von Kollisionen der entsprechenden Komponente mit anderen Komponenten sowie Methoden zur Verwaltung von Maus- und Tastatur-Events.

Performance Die Klasse `Performance` definiert insbesondere Methoden zur Steuerung und Verwaltung der Ausführung von Solist-Programmen: `stop`, `suspend`, `setSpeed`, `started`, `stopped`, `suspended`, `resumed` und `playSound` sowie Getter-Methoden zur Zustandsabfrage. Möchte ein Programmierer zusätzliche Aktionen implementieren, wenn die entsprechenden Steuerungsbuttons in der Toolbar angeklickt werden, kann er eine Unterklasse der Klasse `Performance` definieren und hierin die entsprechende Methoden überschreiben.

Maus- und Tastatur-Events Sowohl die Klasse `Stage` als auch die Klasse `Component` definieren die von der Java-GUI-Programmierung bekannten Methoden zur Verarbeitung von Maus- und Tastatur-Events (`keyPressed`, `keyReleased`, `mouseClicked`, `mouseEntered`, …). Soll ein Akteur, eine Requisite oder die

Bühne darauf reagieren, wenn während der Programmausführung bspw. eine bestimmte Taste gedrückt oder das Icon des Akteurs mit der Maus angeklickt wird, kann der Programmierer die Methode in der jeweiligen Klasse entsprechend überschreiben. Den Methoden werden Objekte der Theater-Klassen `KeyInfo` bzw. `MouseInfo` übergeben, über die weitergehende Informationen zu dem Event abgefragt werden können.

Weitere Klassen `TheaterImage` ist eine Theater-Klasse mit vielfältigen Methoden zum Erzeugen und Manipulieren von Bildern bzw. Icons, die dann Akteuren, Requisiten oder der Bühne zugeordnet werden können. Die Bilder lassen sich dabei auch noch zur Laufzeit verändern, so dass mit Hilfe der Klasse `TheaterImage` bspw. Punktezähler für kleinere Spiele implementiert werden können.

`PixelArea` ist ein Interface, das die Grundlage der Kollisionserkennungsmethoden darstellt. Es definiert hierzu entsprechende Methoden `contains`, `isInside` und `intersects`. Neben einigen zur Verfügung gestellten Standardklassen (`Point`, `Rectangle`, `Cell`, `CellArea`) implementieren auch die Klassen `Stage` und `Component` das Interface. Dadurch sind nur sehr wenige Methoden zur Kollisionserkennung notwendig, die jedoch sehr flexibel und umfassend eingesetzt werden können.

Die Implementierung des `kornDa`-Testbefehls des Hamsters im Hamster-Simulator soll einen Eindruck von der Nutzung der Theater-API geben:

```
public boolean kornDa() {
    return this.getStage()
        .getComponentsAt(this.getColumn(), this.getRow(),
                Korn.class).size() > 0;
}
```

2.3 Weitere Features

Die MPWs bzw. Simulatoren, die mittels Solist generiert werden, erlauben den Programmieranfängern die Entwicklung von Programmen in Java-Syntax[6]. Dabei werden jedoch das in Java obligatorische Klassengerüst und die public-static-void-main-Prozedur durch einen Precompiler automatisch hinzugefügt, so dass sich Programmieranfänger um diese für sie häufig „obskuren“ Konstrukte nicht mehr kümmern müssen. Startpunkt eines Programms ist immer eine zu definierende parameterlose Prozedur *main* (siehe auch das Programm in Abbildung 2). Über das Optionen-Menü kann der Precompiler jedoch auch ausgeschaltet werden.

[6] In zukünftigen Versionen von Solist ist die Integration bzw. Berücksichtigung weiterer Programmiersprachen, wie Python oder Ruby, und auch einer visuellen Programmiersprache in Anlehnung an Scratch (http://scratch.mit.edu/) geplant.

Weiterhin kann dem Solisten bei der Definition der entsprechenden `Actor`-Klasse ein Name zugeordnet und hierüber eine eher objektorientierte Variante des Befehlsaufrufs unterstützt werden, beim Kara-Modell bspw. `kara.move();`

3 Potential von Solist

Mit Hilfe von Solist ist es möglich, mit durchschnittlichen Programmierkenntnissen in kurzer Zeit neue MPWs zu entwickeln. Während Informatik-Lehrerinnen und -Lehrer bisher beim Einsatz von MPWs im Unterricht auf die Simulatoren angewiesen waren, die von Programmierprofis implementiert und zur Verfügung gestellt wurden, unterstützt sie Solist nun dabei, diesbezüglich eigene Ideen zu verwirklichen und auszuprobieren.

Weiterhin bietet sich Lehrerinnen und Lehrern die Möglichkeit, existierende MPWs selbstständig an eigene Anforderungen anzupassen. Auf der Website von Solist (http://www.programmierkurs-java.de/solist) stehen dazu die bekanntesten MPWs (Hamster, Kara, Turtle, …) als Solist-Theaterstücke zum Runterladen zur Verfügung. Soll bspw. der Hamster sich standardmäßig auch rechtsum drehen können, kann in der Klasse `Hamster` des Hamster-Theaterstücks eine Methode (und damit der Befehl) `rechtsUm` ergänzt werden. Der anschließend generierte Hamster-Simulator unterstützt dann automatisch auch diesen Grundbefehl. Oder störten einen bei Kara schon immer die Wurzeln, lassen sich diese durch Änderung des Kara-Theaterstückes innerhalb von wenigen Minuten entfernen.

Die Handhabung aller mit Solist generierten Simulatoren ist vollkommen identisch. Damit bietet sich die Möglichkeit, nicht immer nur dieselbe MPW sondern mehrere MPWs abwechselnd im Unterricht einzusetzen. Damit lässt sich das Problem der „Eintönigkeit“ (in gewissen Grenzen) beseitigen.

Beim Einsatz des Hamster-Modells in meinen Programmierkursen habe ich die Erfahrung gemacht, dass für die besseren Schülerinnen und Schüler ein großer Reiz darin besteht, den Quellcode des Hamster-Simulators selbst zu verstehen und dem Hamster neue Grundbefehle beizubringen. Mit Solist ist die Erfüllung dieses Wunsches durchaus umsetzbar, womit auch der Gefahr einer Unterforderung entgegengewirkt werden kann.

Über die Umsetzung klassischer MPWs hinaus bietet Solist aber auch die Möglichkeit zur Realisierung von MPWs, die der Visualisierung von Algorithmen dienen. Auf der Website von Solist stehen inzwischen eine große Menge von entsprechenden Theaterstücken und Simulatoren zur Verfügung, die direkt im Unterricht eingesetzt werden oder auch als Vorlage für ähnliche MPWs dienen können.

Ein Beispiel ist das bekannte Spiel „Türme von Hanoi“ (siehe Abbildung 3 (links)). Als Solist fungiert hier ein virtueller Spieler, der die Scheiben der Türme umlegen kann. Programmieranfängern werden lediglich zwei Befehle `int anzahlScheiben()` und `void verschieben(int vonTurm, int nachTurm)` vorgegeben, mit denen

sie eine allgemeingültige Lösung für das Spiel entwickeln müssen. Die Ausführung des `verschieben`-Befehls wird dabei auf der Bühne graphisch animiert dargestellt.

Zum Erlernen der Implementierung von Spielstrategien dient das TicTacToe-Theaterstück (siehe Abbildung 3 (rechts)). Hier werden den Programmieranfängern die zwei Befehle `void ziehen(Spielzug zug)` und `Spielzug warten(int farbe)` vorgegeben. Mit dem `ziehen`-Befehl kann eine Spielfigur auf dem Spielbrett platziert werden, mit dem `warten`-Befehl wird auf einen menschlichen Spielzug (Klick auf das entsprechende Feld) gewartet. Das TicTacToe-Theaterstück kann als Vorlage für die Umsetzung anderer 2-Personen-Strategiespiele (Schach, 4-Gewinnt, Reversi, …) genutzt werden, bei denen die Programmieranfänger letztendlich auf der Grundlage von wenigen vorgegebenen Befehlen Programme entwickeln können, die gegen andere Programme oder Menschen das entsprechende Spiel spielen können.

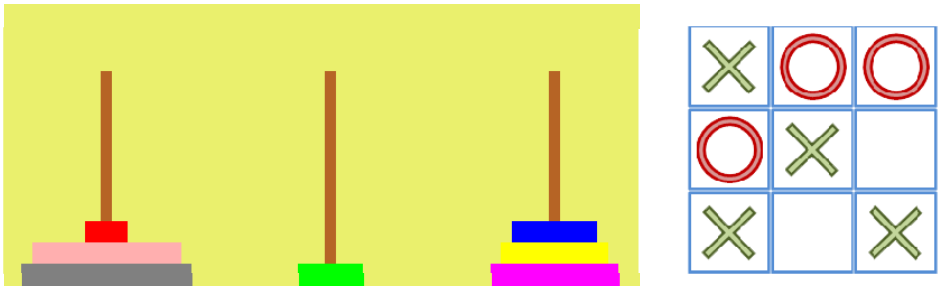

Abbildung 3: Solist-Theaterstücke Türme-von-Hanoi und TicTacToe

Beim Sudoku-Theaterstück können Programmieranfänger mit wenigen vorgegebenen Befehlen Programme entwickeln, die beliebige Sudoku-Rätsel lösen. Beim Damen- und Springer-Theaterstück (siehe Abbildung 4 (links)) gilt es für die Programmieranfänger das Backtracking-Verfahren kennenzulernen und einzusetzen, um die beiden bekannten Probleme aus dem Schachumfeld zu lösen. Weitere bereits realisierte Theaterstücke sind das Game-of-Life als Beispiel für die Umsetzung von Simulationen und das sogenannte Früchte-Theaterstück (siehe Abbildung 4 (rechts)) als Beispiel für die Lösung von Optimierungsproblemen.

Solist gehört zu den Werkzeugen des Programmier-Theaters. Dies ist eine Familie von Werkzeugen für Programmieranfänger, die alle die in Abschnitt 2 vorgestellte Theater-API nutzen. Aktuell gibt es mit *Objekt-Theater* und *Threadnocchio* zwei weitere Werkzeuge im Programmier-Theater[7]. In Objekt-Theater, das die Einführung in die objektorientierte Programmierung unterstützt, werden Objekte durch Icons repräsentiert und die Benutzer können interaktiv darauf zugreifen. Vom zugrunde liegenden Prinzip ist Objekt-Theater mit Greenfoot [Kö09] vergleichbar. Threadnocchio dient zur Einführung in die parallele Programmierung mit Java-Threads. In Threadnocchio werden Threads durch Icons repräsentiert, wodurch sich die Auswirkungen von

[7] siehe http://www.programmierkurs-java.de/theater

Kommunikations- und Synchronisationsmechanismen sehr schön visualisieren lassen [Bo09].

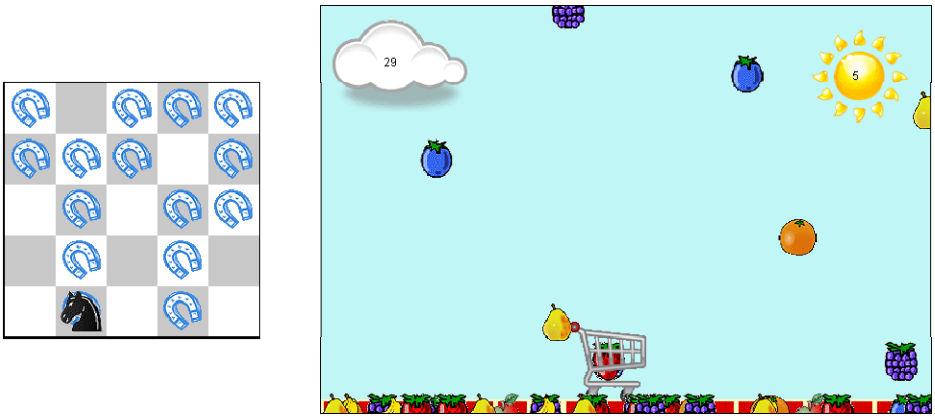

Abbildung 4: Solist-Theaterstücke Springer und Früchte

Literaturverzeichnis

[BK09] Barnes, D. J.; Kölling, M.: Java lernen mit BlueJ: Eine Einführung in die objektorientierte Programmierung. Pearson Studium; 4. Auflage, 2009.

[Bo05] Boles, D.: Spielerisches Erlernen der Programmierung mit dem Java-Hamster-Modell In: Lecture Notes in Informatics (LNI) - Proceedings, Volume P-60, Unterrichtskonzepte für informatische Bildung, INFOS 2005, Hrsg. Steffen Friedrich. Köllen Druck+Verlag GmbH Bonn, Seiten 243-252, September 2005.

[Bo08] Boles, D.: Programmieren spielend gelernt mit dem Java-Hamster-Modell. Vieweg-Teubner-Verlag, 4. Auflage, 2008. (www.java-hamster-modell.de)

[Bo09] Boles, D.: Threadnocchio - Einsatz von Visualisierungstechniken zum spielerischen Erlernen der parallelen Programmierung mit Java-Threads. In: Ulrike Jaeger, Kurt Schneider (Hrsg.): Software Engineering im Unterricht der Hochschulen, SEUH 11 - Hannover 2009, dpunkt.verlag, Heidelberg, Seiten 131-143, Februar 2009.

[Br94] Brusilovsky, P. et.al.: Teaching programming to novices: A review of approaches and tools. In Educational Multimedia and Hypermedia. Proceedings of ED-MEDIA'94 – World Conference on Educational Multimedia and Hypermedia, Seiten 103–110, 1994.

[Br97] Brusilovsky, P. et.al.: Mini-languages: A way to learn programming principles. Education and Information Technologies, 2(1), Seiten 65-83, 1997.

[Ha08] Hardy, D.: Programmieren lernen für Kinder - Eine verständliche Einführung. Markt+Technik, 2008.

[Kö09] Kölling, M.: Introduction to Programming with Greenfoot: Object-Oriented Programming in Java with Games and Simulations. Prentice Hall, 2009. (www.greenfoot.org)

[Pa80] Papert, S.: Mindstorms: Children, Computers, and Powerful Ideas. Basic Books, New York, NY, 1980.

[RNH04] Reichert, R.; Nievergelt, J.; Hartmann, W.: Programmieren mit Kara. Ein spielerischer Zugang zur Informatik. Springer, 2. Auflage, 2004. (www.swisseduc.ch/informatik/-karatojava/)