Threadnocchio – Einsatz von Visualisierungstechniken zum spielerischen Erlernen der parallelen Programmierung mit Java-Threads

Dietrich Boles, Universität Oldenburg

Abstract

Die parallele Programmierung, d.h. die Entwicklung nebenläufiger Systeme, gewinnt auch außerhalb von Hochleistungsrechenzentren immer mehr an Bedeutung. Insbesondere wird die volle Leistungsstärke moderner Multicore-Rechner nur dann erreicht werden können, wenn die Programme den angebotenen Parallelismus auch nutzen. Allerdings pflegt die parallele Programmierung in heutigen Curricula eher ein Nischendasein. Viele Programmierer sind nicht in der Lage, Software zu entwickeln, die parallel verarbeitet werden kann.

In diesem Artikel wird Threadnocchio vorgestellt, ein Tool zum spielerischen Erlernen der Konzepte der parallelen Programmierung, insbesondere der Programmierung mit Java-Threads. In Threadnocchio werden Threads durch Bilder bzw. Icons visualisiert. Dadurch lässt sich die Ausführung paralleler Programme sehr gut nachvollziehen und die Auswirkungen von bspw. Kommunikations- und Sychronisationsmechanismen werden dem Programmierer unmittelbar vor Augen geführt.

1 Einleitung

Die parallele Programmierung - genauer die Entwicklung paralleler Anwendungen - führte in den vergangenen Jahrzehnten eher ein Nischendasein. Parallele Anwendungen wurden vor allem für numerische Probleme auf Hochleistungsrechnern entwickelt. Durchaus Nutzen findet der Einsatz paralleler Programmierkonzepte allerdings auch heute schon in gängigen Anwendungen, wie etwa Multimedia-Anwendungen, Simulationen, Web-Anwendungen oder Computerspielen [Boles 2008].

In bereits naher Zukunft wird die parallele Programmierung jedoch immens an Bedeutung gewinnen müssen, denn auf modernen Multicore-Rechnern mit vielen Prozessoren mit Multithreading-Fähigkeiten (siehe bspw. [Bode 2006] oder [Bengel et al. 2008]) werden zwar auch herkömmliche sequentielle Programme weiterhin laufen. Die volle Leistungsfähigkeit der Multicore-Technologien wird allerdings nur dann ausgeschöpft werden können, wenn auch Standard-PC-Anwendungen parallel implementiert sind.

Problem ist, dass die Entwicklung korrekter paralleler Programme deutlich komplizierter ist als die Entwicklung sequentieller Programme. Insbesondere bei der Synchronisation der Prozesse kommt es häufig zu Programmierfehlern, die schwer zu finden sind, weil sie nicht immer, sondern nur gelegentlich auftreten, aber zu fehlerhaften Ergebnissen oder Deadlocks führen können. Weiterhin wird der parallelen Programmierung in den Curricula der Hochschulen bisher zu wenig Platz eingeräumt. David Cearly, Vice President des Analystenhauses Gartner, schätzt daher, „dass weniger als die Hälfte der Programmierer in der Lage ist, Software zu schreiben, die parallel verarbeitet werden kann“ [Cearly 2008] und Erhard Plöderer, Universität Stuttgart, konstatiert: „Manche Informatikabsolventen, die ganz gut sequentiell programmieren können, haben von Parallelität keine Ahnung.” [Killer 2008]

Notwendig ist es also zum einen, die Forschung im Bereich des Software-Engineering für parallele Systeme voranzutreiben, um die Entwicklung paralleler Anwendungen zu vereinfachen. In aktuellen Projekten wird bspw. nach weniger komplexen Programmiermechanismen gesucht, es werden neuartige Debugger für die Entdeckung von Synchronisationsfehlern entwickelt und Pattern für die Strukturierung paralleler Anwendungen definiert (siehe auch [Pankratius et al. 2008]). Walter Tichy, Universität Karlsruhe, sieht in der Parallelisierung „die große neue Herausforderung für die Softwaretechnik” [Tichy 2008].

Zum anderen muss die parallele Programmierung in den Curricula der Hochschulen in Zukunft sehr viel stärkere Berücksichtigung finden. Für die Lehre müssen neuartige didaktische Hilfsmittel konzipiert und entwickelt werden, die das Erlernen der parallelen Programmierung vereinfachen.

Arndt Bode, TU München, fasst diese Forderungen wie folgt zusammen: „In weniger als 10 Jahren werden Standardmikroprozessoren mehr als 128 Prozessoren auf dem Chip aufweisen. Die effiziente Nutzung solcher Hardwarearchitekturen stellt neue Anforderungen an die Programmierung. Programmentwickler werden in Zukunft parallele Programme entwickeln, testen und warten müssen. Neue Programmiersprachen und –modelle sind gefordert, aber auch die Ausbildung für künftige Informatiker muss das Thema Parallelismus stärker berücksichtigen als das in bisherigen Curricula der Fall war.“ [Bode 2006]

In diesem Artikel wird mit *Threadnocchio* - auch *Thread-Theater* genannt - ein Tool vorgestellt, das der zweiten der beiden oben angeführten Forderungen gerecht werden soll. Threadnocchio unterstützt durch den Einsatz spielerischer und visualisierender Elemente das Erlernen der Basiskonzepte der parallelen Programmierung. Es nutzt dabei das Thread-Konzept der Programmiersprache Java. Das Tool, das aus einem graphischen Simulator und einer Klassenbibliothek bzw. API besteht, kann als Open-Source-Software von der Website `http://www.programmierkurs-java.de/threadnocchio` herunter geladen werden.

Grundlegende Idee von Threadnocchio ist die Visualisierung von Threads durch Bilder bzw. Icons. Schaut man nämlich in gängige Lehrbücher zur parallelen Programmierung werden in den Programmbeispielen die Aktivitäten von Prozessen im

Allgemeinen durch Ausgaben auf die Konsole (`System.out.println`) dargestellt. Lösungen bspw. für das bekannte Philosophenproblem dadurch vor Augen geführt zu bekommen, dass auf der Konsole die Ausgabe „Philosoph 1 isst gerade“ erscheint, ist für die Lernenden zum einen wenig motivierend und zum anderen durch die Eindimensionalität der Ausgabe auch unübersichtlich. In Threadnocchio dahingegen kann man den Philosophen im Simulator auf einer graphischen, zweidimensionalen Oberfläche beim Essen zuschauen. Und dabei muss sich der Programmierer des Philosophenproblems durch die Nutzung der Threadnocchio-API nur unwesentlich mit der Programmierung der graphischen Ausgabe beschäftigen, sondern kann sich voll und ganz der korrekten Synchronisation der Philosophen widmen.

Dieser Artikel ist so aufgebaut, dass nach dieser motivierenden Einleitung im zweiten Abschnitt zunächst die Grundidee von Threadnocchio sowie der Threadnocchio-Simulator vorgestellt werden. Die Beschreibung der Threadnocchio-API ist Gegenstand von Abschnitt 3. Durch ein konkretes Beispielszenario wird der Einsatz und Nutzen von Threadnocchio in Abschnitt 4 verdeutlicht. Abschnitt 5 beschließt den Artikel mit einer Zusammenfassung und einer Übersicht über weitere Aktivitäten rund um Threadnocchio.

2 Threadnocchio

Threadnocchio führt in die parallele Programmierung anhand des Thread-Konzeptes der Programmiersprache Java ein. In Java ist das Thread-Konzept sehr harmonisch in die objektorientierten Konzepte der Sprache integriert worden. Neue Threads werden ausgehend von Objekten einer vordefinierten Klasse `Thread` gestartet, wobei die Objekte zum Kapseln des Zustand des ihnen zugeordneten Threads genutzt werden können. Kommunikation betreiben Threads in Java über global definierte Objekte bzw. Variablen. Zur Absperrung kritischer Abschnitte gibt es die `synchronized`-Anweisung. Zum Warten auf die Erfüllung von Bedingungen durch andere Threads bzw. zum Signalisieren des Erfülltseins von Bedingungen kennt jedes Objekt in Java die Methoden `wait`, `notify` und `notifyAll`. Seit der Version 1.5 stellt Java über diese Basissynchronisationsmechanismen hinaus weitergehende Konstrukte wie explizite Locks, Semaphore und spezielle Collection-Klassen zur Verfügung (siehe auch [Boles 2008] oder [Oechsle 2007]).

2.1 Analogien

Zumindest andeutungsweise lässt sich die objektorientierte Programmierung – nicht nur in Java – mit einem Marionettentheater vergleichen:

- Die Objekte sind die Marionetten.
- Ein objektorientiertes Programm ist ein Marionettentheaterstück.
- Der Programmierer ist der Autor des Marionettentheaterstücks.
- Die Ausführung des Programms entspricht der Aufführung des Stücks.

- Der Marionettenspieler fungiert als Prozessor. Er führt die im Theaterstück gegebenen Handlungssequenzen nacheinander aus.

Bei einem objektorientierten Programm sind die Objekte, was ihre Handlungsfreiheiten angeht, genauso passiv wie die Marionetten im Marionettentheater. Bereits der Programmierer steuert ihr Zusammenspiel. Er gibt vor, welche Aktionen in welcher Reihenfolge ausgeführt werden sollen, auch wenn prinzipiell durch Benutzerinteraktionen die Ausführung in unterschiedliche Richtungen gelenkt werden kann. Der Programmierer hält im wahrsten Sinne des Wortes „alle Fäden in der Hand“.

Im Unterschied dazu können in der Thread-Programmierung in Java die Threads in Bezug auf ihre Koordination als aktive Objekte angesehen werden. Der Programmierer erschafft sie zwar, gibt jedoch bei der Ausführung eines parallelen Programms die Fäden aus der Hand, die Threads agieren nach ihrem Start selbstständig und müssen sich mit anderen Threads koordinieren. Threads sind also vergleichbar mit Pinocchio, der bekannten Kinderbuchfigur des italienischen Autors Carlo Collodi, die - vom Holzschnitzer Geppetto geschaffen - plötzlich lebendig wird und phantastische Abenteuer erlebt. Daher hat das hier vorgestellte Tool auch seinen Namen: *Threadnocchio* ist ein Marionettentheater für Java-Threads als selbstständige Marionetten.

2.2 Threadnocchio-Simulator

Diese Analogien spiegeln sich auch im Aufbau und Erscheinungsbild des Threadnocchio-Simulators wider (siehe auch Abbildung 1):

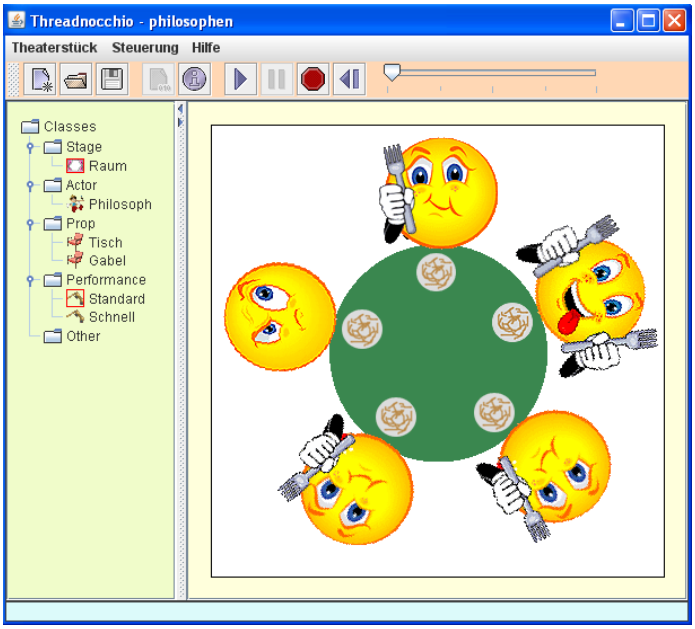

Abb. 1: Threadnocchio-Simulator mit Lösung des Philosophenproblems

- Ein paralleles objektorientiertes Programm entspricht einem Theaterstück (play).
- Handlungsumfeld für die Threads ist eine Bühne (stage). Dabei kann es in Threadnocchio zu einem Stück durchaus mehrere verschiedene Bühnenaufbauten geben.
- Eine Menge an Threads agieren in einem Threadnocchio-Theaterstück wie selbstständige Marionetten. Sie werden als Akteure (actor) bezeichnet.
- Neben den selbstständigen Threads existieren normale passive Objekte, die als Requisiten (prop) auf der Bühne platziert und von den Marionetten genutzt werden können.
- Die Aufführung (performance) eines Threadnocchio-Theaterstücks, also die Ausführung des Programms, kann in bestimmten Details variieren.

Im linken Bereich des Threadnocchio-Simulators ist es möglich, entsprechende Klassen auf der Grundlage der in Abschnitt 3 vorgestellten Threadnocchio-API zu definieren. In der Lösung des Philosophenproblems aus Abbildung 1 definiert eine von der Threadnocchio-Klasse `Stage` abgeleitete Klasse `Raum` den Aufbau und die Gestaltung der Bühne. Einer Bühne kann ein Bild zugeordnet werden, das als Hintergrund im rechten Teil des Simulators angezeigt wird.

Akteure, also Threads, in dem Beispiel sind die Philosophen. Ihr Verhalten wird in der Klasse `Philosoph` definiert, die von der Threadnocchio-Klasse `Actor` abgeleitet ist. Die Klasse `Actor` selbst ist wiederum (indirekt) von der Java-Klasse `Thread` abgeleitet. Akteuren kann ein Bild oder Icon zugeordnet werden, das bei einer Instanziierung der entsprechenden Actor-Klasse auf der Bühne im rechten Teil des Simulators erscheint (hier die Gesichter).

Als Requisiten, also Objekte ohne eigenen Handlungsstrang, werden in dem Beispiel ein Tisch und mehrere Gabeln benötigt, denen als Objekte von den als Unterklassen der Threadnocchio-Klasse `Prop` definierten Klassen `Tisch` und `Gabel` ebenfalls ein Bild zugeordnet und auf der Bühne platziert werden kann.

In dem Beispiel werden weiterhin die zwei Klassen `Standard` und `Schnell` als von der Threadnocchio-Klasse `Performance` abgeleitete Klassen definiert. In Performance-Klassen können Variationen der Ausführung des Programms wie bspw. die Geschwindigkeit festgelegt werden.

Das Vorgehen bei der Entwicklung eines Threadnocchio-Theaterstücks hat folgende Form: Zunächst werden alle benötigten Klassen auf der Grundlage der Threadnocchio-API definiert. Anschließend werden über Popup-Menüs interaktiv Akteure und Properties instanziiert und deren Icons auf der Bühne in der gewünschten Form und an der gewünschten Stelle platziert (alternativ kann das auch fest in einer entsprechenden Stage-Klasse programmiert werden). Danach kann mit den Steuerungsbuttons in der Toolbar des Simulators die Aufführung des Theaterstücks bzw. die Ausführung des Programms gesteuert und verwaltet werden, wobei die im linken Teil selektierte Performance-Klasse die genaue Form der Aufführung definiert (hier die Klasse `Standard`).

3 Threadnocchio-API

Threadnocchio nutzt als Klassenbibliothek bzw. API die so genannte *Theater-API*, die auch noch in anderen Werkzeugen eingesetzt wird (siehe Abschnitt 5). Die Theater- bzw. Threadnocchio-API umfasst eine Menge an vordefinierten Klassen, die das Grundverhalten aller beteiligten Objekte implementieren und weitere nützliche Features zur Verfügung stellen. Bei der Konzeption der API standen dabei die Grundsätze „so klein wie möglich“, aber „so mächtig wie nötig“ und „einfach erweiterbar“ im Vordergrund. Ziel der API ist die schnelle und einfache Entwicklung „kleiner“ Anwendungen, um einzelne Konzepte der parallelen Programmierung zu visualisieren. Sie ist nicht in erster Linie für die Entwicklung aus Sicht der Softwaretechnik sauber entworfener komplexer paralleler Anwendungen gedacht. Die wichtigsten Klassen der API sind: `Stage`, `Component`, `Actor`, `Prop` und `Performance`.

3.1 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 (siehe Abschnitt 3.4) 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.

3.2 Component, Actor und Prop

Die Klasse `Component` ist von der Java-Klasse `Thread` abgeleitet, erlaubt also die Erzeugung von Threads. Sie definiert Methoden zur Verwaltung von Akteuren und

Requisiten, die sie an die von ihr abgeleiteten Klassen `Actor` und `Prop` vererbt. Die Klassen `Actor` und `Prop` unterscheiden sich nur dadurch, dass von `Actor` abgeleitete Klassen die Thread-Methode `run` überschreiben, also Threads definieren können. Bei Unterklassen von `Prop` wird dies dadurch verhindert, dass die `run`-Methode in der Klasse `Prop` als leere `final`-Methode definiert wird.

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.

3.3 Performance

Die Klasse `Performance` definiert insbesondere Methoden zur Steuerung und Verwaltung der Ausführung von Threadnocchio-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 Methode überschreiben.

3.4 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`, `keyRealeased`, `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.

3.5 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 Lauf-

zeit 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.

4 Beispiel-Theaterstück Startrek

In dem im Folgenden vorgestellten Theaterstück namens *Startrek* wird der Einsatz und Nutzen von Threadnocchio demonstriert. Threads werden in dem Beispiel durch Raumschiffe visualisiert, die gleichzeitig durch den Weltraum gleiten (siehe Abbildung 2). Treffen die Raumschiffe auf Felsbrocken, werden diese „weggebeamt“, d.h. entfernt. Über Mausklicks auf die Raumschiffe kann der Nutzer die Priorität der entsprechenden Threads verändern und unmittelbar sehen, welche Auswirkungen dies auf die Ausführung eines Programms hat.

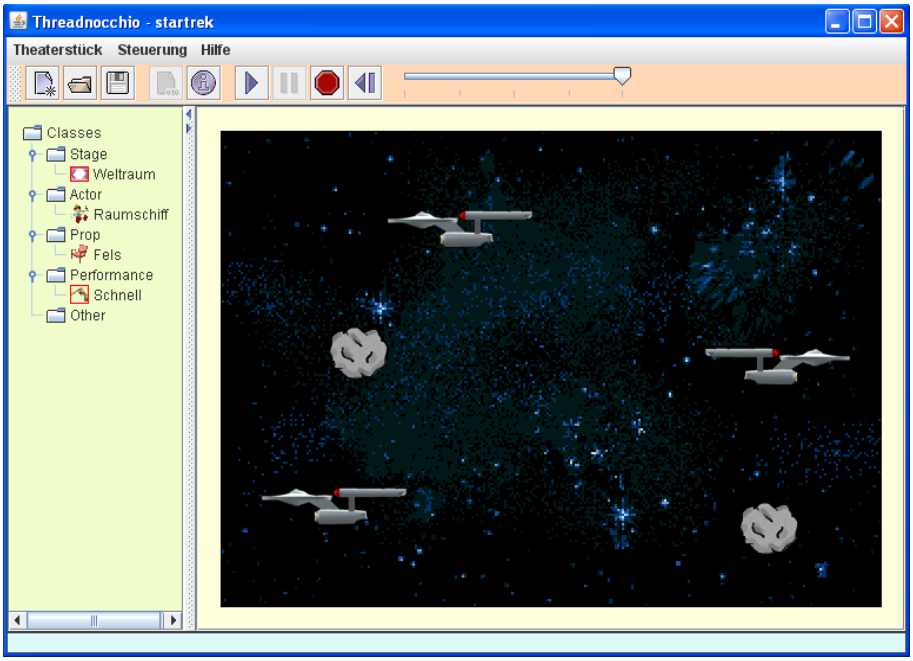

Abb. 2: Threadnocchio-Simulator mit Startrek-Theaterstück

Die Bühne im Startrek-Theaterstück wird über die folgende Klasse `Weltraum` realisiert:

```
import java.util.List;
import theater.*;

public class Weltraum extends Stage {

   public Weltraum() {
      // Buehne entspricht Groesse des Bildes, Zellen 1 Pixel
      super(560, 400, 1);
      // Zuordnung des Hintergrundbildes
      setBackground("weltraum.gif");
   }

   public void mousePressed(MouseInfo info) {
      // bei Mausklick auf Buehne wird allen Raumschiffe wieder die
      // Standard-Thread-Prioritaet zugeordnet
      List<Component> schiffe = getComponents(Raumschiff.class);
      for (Component raumschiff : schiffe)
           raumschiff.setPriority(Thread.NORM_PRIORITY);
   }
}
```

Raumschiffe sind Instanzen der Actor-Klasse `Raumschiff`. Wird der Start-Button in der Toolbar des Simulators gedrückt, werden alle Raumschiffe – also Threads –, die der Nutzer vorher über ein spezielles Menü auf der Bühne platziert hat, gestartet und führen ihre `run`-Methode aus. Auch während der Ausführung lassen sich interaktiv weitere Raumschiffe erzeugen. In der Methode `ueberpruefeFelsKollision` wird eine Synchronisation vorgenommen, um sicherzustellen, dass die bei der Kollisionsentdeckung ermittelten Felsbrocken noch nicht von anderen Raumschiffen weggebeamt worden sind. Das könnte ansonsten theoretisch passieren, wenn zwei Raumschiffe gleichzeitig auf einen Felsbrocken treffen und der Scheduler nach der Kollisionsentdeckungsanweisung einen Thread-Wechsel anstößt.

```
import java.awt.event.MouseEvent;
import java.util.List;
import theater.*;

public class Raumschiff extends Actor {

   public Raumschiff() {
      // Icon zuordnen
      setImage("raumschiff.gif");
   }

   public void run() {

      // Raumschiff gleitet im Weltraum hin und her
      int step = 1; // Richtung Ost
```

```
      while (true) {
           // befindet sich das Icon vollstaendig auf der Buehne?
           if (!isInside(getStage())) {
              step *= -1; // Richtunsgwechsel
              getImage().mirrorHorizontally(); // Icon-Spiegelung
           }
           // eine Celle in der aktuellen Richtung weiter platzieren
           setLocation(getColumn() + step, getRow());
           ueberpruefeFelsKollision();
      }
   }

   private void ueberpruefeFelsKollision() {
      synchronized (Fels.class) {
           List<Component> felsen =
            getStage().getIntersectingComponents(this, Fels.class);
           for (Component fels : felsen) {
              getStage().remove(fels); // Fels wegbeamen
           }
      }
   }

   public void mousePressed(MouseInfo event) {
      // Veraenderung der Thread-Prioritaet bei Mausklick auf Icon
      event.consume();
      if (event.getButton() == MouseEvent.BUTTON1) {
           setPriority(Math.min(getPriority() + 1,
                                Thread.MAX_PRIORITY));
      } else {
           setPriority(Math.max(getPriority() - 1,
                                Thread.MIN_PRIORITY));
      }
   }
}
```

Felsbrocken werden als Requisiten, d.h. als passive Objekte der von der Klasse `Prop` abgeleiteten Klasse `Fels` umgesetzt. Genauso wie Raumschiffe können auch Felsbrocken durch den Nutzer interaktiv im Weltraum, also auf der Bühne, platziert werden.

```
import theater.*;

public class Fels extends Prop {

   public Fels() {
      // Icon zuordnen
      setImage("fels.gif");
   }
}
```

Von der Klasse `Performance` wird die folgende Klasse `Schnell` abgeleitet, die dafür sorgt, dass nach dem Start die maximale Ausführungsgeschwindigkeit eingestellt wird.

```
import theater.*;

public class Schnell extends Performance {

   public void started() {
      // beim Start wird die Geschwindigkeit hochgesetzt
      setSpeed(Performance.MAX_SPEED);
   }
}
```

5 Schlussbemerkungen

Der parallelen Programmierung wird bereits in naher Zukunft bedingt durch die Multicore-Technologien moderner Rechner eine stark wachsende Bedeutung zukommen. Festzustellen ist, dass mit der parallelen Programmierung zusätzliche Programmierprobleme wie die Synchronisation einhergehen, denen viele Programmierer (noch) nicht gewachsen sind. Gefordert werden damit zum einen neuartige Konzepte, die die Komplexität paralleler Programmiermechanismen an sich reduzieren, sowie Werkzeuge die den Umgang mit den Konzepten der parallelen Programmierung sowie das Erlernen dieser Konzepte vereinfachen.

Letzteres ist das Ziel des in diesem Artikel vorgestellten Tools *Threadnocchio*. In Threadnocchio werden Threads durch Icons visualisiert, so dass Programmierer, die die parallele Programmierung mit Java-Threads erlernen wollen, die Ausführung ihrer Programme sehr gut nachvollziehen können und ihnen die Auswirkungen der eingesetzten Parallelitätsmechanismen unmittelbar vor Augen geführt werden.

Threadnocchio wurde im Rahmen eines Programmierpraktikums an der Universität Oldenburg konzipiert und anschließend weiterentwickelt. Einige Ideen und Konzepte des Tools *Greenfoot* sind dabei in die Entwicklung eingeflossen. Greenfoot unterstützt das Erlernen der objektorientierten Programmierung, indem Objekte durch Icons repräsentiert werden (siehe www.greenfoot.org). Ebenfalls Einfluss hatte das parallele Hamster-Modell [Boles 2008]. Während Java-Threads im Hamster-Modell durch (visuelle) Hamster repräsentiert werden, ist die Visualisierung von Threads in Threadnocchio frei wählbar.

Threadnocchio ist nur eine Instanz einer Reihe von Werkzeugen für die Programmierausbildung, die derzeit an der Universität Oldenburg entwickelt werden. Ziel aller dieser Werkzeuge ist die Visualisierung der Ausführung von Programmen. Sie bedienen sich dabei alle der Theatermetapher und nutzen die Theater-API.

So ermöglicht das Tool *Solist* die Generierung von Simulatoren für so genannte *Miniwelten* wie Karel, Niki, Hamster, Kara oder Turtle (siehe [Freiberger 2002]). Aufbauend auf der Theater-API muss dabei ein Programmierer lediglich die Eigenschaften der entsprechenden Miniwelt implementieren, was für die genannten Miniwelten innerhalb weniger Stunden möglich ist.

Ein weiteres Werkzeug namens *Kolosseum* erlaubt die Generierung von Simulatoren für so genannte *Educational Programming Games* wie Robocode, Robocup-Varianten oder CodeRally (siehe [Thevissen 2008]), bei denen Programme entwickelt werden müssen, die Roboter, Fußballspieler oder Rennfahrer implementieren und gegen andere Programme Wettkämpfe austragen.

Letztendlich existiert mit dem Werkzeug *Objekt-Theater* auch eine Alternative zu Greenfoot, bei der die Greenfoot-API durch die Theater-API ersetzt ist.

Literatur

[Akhter/Roberts 2006] S. Akhter, J. Roberts: Multi-Core Programming. Intel Press, 2006

[Bengel et al. 2008] G. Bengel, C. Baun, M. Kunze, K.-U. Stucky: Masterkurs Parallele und Verteilte Systeme: Grundlagen und Programmierung von Multicoreprozessoren, Multiprozessoren, Cluster und Grid, Vieweg+Teubner-Verlag, Wiesbaden, 2008

[Bode 2006] A. Bode: Multicore-Architekturen. In: Informatik-Spektrum, Band 29, Heft 5, Springer-Verlag, Heidelberg, Oktober 2006

[Boles 2008] D. Boles: Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell: Programmierung mit Java-Threads, Vieweg+Teubner-Verlag, Wiesbaden, 2008

[Cearly 2008] D. Cearly: Clients haben zuerst Probleme. In: Computerzeitung, Konradin IT-Verlag, Leinfelden-Echterdingen, 11. August 2008

[Freiberger 2002] U. Freiberger: Karel – Eine Übersicht über verschiedene Entwicklungen, die auf der Idee von „Karel, the Robot“ basieren. Luitpold Gymnasium München, 2002. http://www.schule.bayern.de/-karol/data/uebersicht.pdf

[Killer 2008] A. Killer: Software-Industrie verheddert sich in vielen Programmfäden. In: Computerzeitung, Konradin IT-Verlag, Leinfelden-Echterdingen, 11. August 2008

[Oechsle 2007] R. Oechsle: Parallele und verteilte Anwendungen in Java, Hanser-Verlag, München, 2007

[Pankratius et al. 2008] V. Pankratius, C. Schaefer, A. Jannesari, W. F. Tichy: Software Engineering for Multicore Systems – An Experience Report. In Proceedings of the 1st international Workshop on Multicore Software Engineering (Leipzig, Germany, May 11 - 11, 2008). IWMSE '08. ACM, New York, NY, 53-60, 2008. DOI= http://doi.acm.org/10.1145/1370082.1370096

[Thevissen 2008] C. Thevissen: Ein visuelles Framework zur Entwicklung von Educational Programming Games. Diplomarbeit, Universität Oldenburg, Januar 2008.

[Tichy 2008] W. Tichy: Multicore fordert die Informatik. In: Computerzeitung, Konradin IT-Verlag, Leinfelden-Echterdingen, 19. Mai 2008