Benutzungshandbuch

Threadnocchio

Version 1.0 (12.12.2008)

Dietrich Boles

Universität Oldenburg

Inhalt

1 Einleitung 5

1.1 Motivation 5

1.2 Voraussetzungen 6

1.3 Aufbau des Benutzerhandbuch 7

2 Installation 8

2.1 Voraussetzungen 8

2.2 Download und Installation 8

2.3 Erste Schritte 8

3 Grundlagen 13

3.1 Analogien 13

3.2 Theaterstücke 14

3.3 Bühnen 14

3.4 Akteure 15

3.5 Requisiten 15

3.6 Aufführungen 16

3.7 Hilfsklassen 16

4 Threadnocchio-Simulator 17

4.1 Menüs 17

4.2 Toolbar 19

4.3 Klassenbereich 20

4.4 Bühnenbereich 21

4.5 Meldungsbereich 21

4.6 Editor 21

5 Threadnocchio-API 23
5.1 Stage 23
5.1.1 Gestaltungsmethoden 24
5.1.2 Getter-Methoden 24
5.1.3 Kollisionserkennungsmethoden 24
5.1.4 Event-Methoden 25
5.2 Component, Actor und Prop 25
5.2.1 Manipulationsmethoden 25
5.2.2 Getter-Methoden 26
5.2.3 Kollisionserkennungsmethoden 26
5.2.4 Event-Methoden 26
5.2.5 Akteure 27
5.3 Performance 27
5.4 Maus- und Tastatur-Events 28
5.4.1 KeyInfo 29
5.4.2 MouseInfo 29
5.5 TheaterImage 29
5.6 Kollisionserkennung 30
6 Referenzhandbuch 32
6.1 Stage 32
6.2 Component, Actor, Prop 40
6.2.1 Component 40
6.2.2 Actor 48
6.2.3 Prop 49
6.3 Performance 49

6.4 TheaterImage 53
6.5 Ereignisse 57
6.5.1 KeyInfo 57
6.5.2 MouseInfo 59
6.6 Kollisionserkennung 62
6.6.1 PixelArea 62
6.6.2 Rectangle 63
6.6.3 Point 65
6.6.4 Cell 66
6.6.5 CellArea 68
7 Literatur zur parallelen Programmierung 70
7.1 Allgemeine Lehrbücher für parallele Programmierung 70
7.2 Lehrbücher für die parallele Programmierung mit Java-Threads 70
8 Beispiel-Theaterstücke 71
8.1 Beispiel-Theaterstück Startrek 71
8.2 Beispiel-Theaterstück Philosophen 73
8.3 Beispiel-Theaterstück Hamster 78
8.3.1 Klasse Territorium 79
8.3.2 Klasse Hamster 82
8.3.3 Klasse Mauer 87
8.3.4 Klasse Korn 88
8.3.5 Exception-Klassen 89
8.4 Beispiel-Theaterstück Leser-Schreiber-Hamster 92

1 Einleitung

Threadnocchio ist ein Werkzeug zum Erlernen der parallelen Programmierung mit Java-Threads. Durch den Einsatz spielerischer Elemente erhöht es für entsprechende Programmieranfänger die Motivation und veranschaulicht durch den Einsatz visualisierender Elemente das Ablaufverhalten der entwickelten parallelen Programme.

1.1 Motivation

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.

In bereits naher Zukunft wird die parallele Programmierung jedoch immens an Bedeutung gewinnen. Auf modernen Multicore-Rechnern mit vielen Prozessoren mit Multithreading-Fähigkeiten 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“ und Prof. Erhard Plöderer, Universität Stuttgart, konstatiert: „Manche Informatikabsolventen, die ganz gut sequentiell programmieren können, haben von Parallelität keine Ahnung.”

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. Prof. Walter Tichy, Universität Karlsruhe, sieht in der Parallelisierung „die große neue Herausforderung für die Softwaretechnik”.

Zum anderen muss die parallele Programmierung in den Curricula der Hochschulen in Zukunft sehr viel stärkere Berücksichtigung finden. Für die Lehre müssen neu-

artige didaktische Hilfsmittel konzipiert und entwickelt werden, die das Erlernen der parallelen Programmierung vereinfachen.

Prof. 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."

Threadnocchio ist ein Tool, 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 besteht dabei aus einem graphischen Simulator und einer Klassenbibliothek bzw. API.

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 Threadnocchio-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.

1.2 Voraussetzungen

Zielgruppe von Threadnocchio sind Programmierer, die die Entwicklung paralleler Programme mit Java-Threads erlernen wollen. Vorausgesetzt wird, dass der Programmierer die Basissprachkonzepte von Java (Klassen, Objekte, Attribute, Methoden, Anweisungen, Variablen, Schleifen, …) beherrscht. Nicht notwendig sind Kenntnisse zur Entwicklung von graphischen Oberflächen (GUIs) mit Java-AWT oder Java-Swing.

Threadnocchio ist ein Werkzeug, das das Erlernen der parallelen Programmierung unterstützt. Es ist kein Lehrbuch. Auf gute Lehrbücher im Bereich der parallelen Programmierung wird in Kapitel 7 (Literatur zur parallelen Programmierung) hingewiesen.

1.3 Aufbau des Benutzerhandbuch

Das Benutzungshandbuch ist in 8 Kapitel gegliedert. Nach dieser Einleitung folgen in Kapitel 2 Hinweise zur Installation von Threadnocchio. Kapitel 3 führt in die Grundlagen von Threadnocchio ein und gibt einen Überblick über die einzelnen Bestandteile. Der Threadnocchio-Simulator wird in Kapitel 4 vorgestellt. Kapitel 5 widmet sich der Threadnocchio-API. Während diese in Kapitel 5 nur überblicksartig vorgestellt wird, enthält Kapitel 6 die komplette Referenz. Kapitel 7 listet begleitende Literatur zum Erlernen der parallelen Programmierung auf und Kapitel 8 demonstriert an einigen Beispielen den Einsatz und Nutzen von Threadnocchio.

2 Installation

2.1 Voraussetzungen

Voraussetzung zum Starten von Threadnocchio ist ein Java Development Kit SE (JDK) der Version 6 oder höher. Ein Java Runtime Environment SE (JRE) reicht nicht aus. Das jeweils aktuelle JDK kann über die Website http://java.sun.com/javase/downloads/index.jsp bezogen werden und muss anschließend installiert werden.

2.2 Download und Installation

Threadnocchio hat eine eigene Website: http://www.programmierkurs-java.de/threadnocchio. Von dieser Website muss eine Datei *threadnocchio-1.0.zip* herunter geladen und entpackt werden. Es entsteht ein Ordner namens *threadnocchio-1.0* (der so genannte *Threadnocchio-Ordner*), in dem sich eine Datei *threadnocchio.jar* befindet. Durch Doppelklick auf diese Datei wird der Threadnocchio-Simulator gestartet.

Beim ersten Start sucht Threadnocchio nach der JDK-Installation auf Ihrem Computer. Sollte diese nicht gefunden werden, werden Sie aufgefordert, den entsprechenden Pfad einzugeben. Der Pfad wird anschließend in einer Datei namens *theater.properties* im Threadnocchio-Ordner gespeichert, wo er später wieder geändert werden kann, wenn Sie bspw. eine aktuellere JDK-Version auf Ihrem Rechner installieren sollten. In der Property-Datei können Sie weiterhin die Sprache angeben, mit der der Threadnocchio-Simulator arbeitet. In der Version 1.0 wird allerdings nur deutsch unterstützt.

2.3 Erste Schritte

Lesen oder überfliegen Sie zumindest zunächst dieses Handbuch, um die Grundlagen von Threadnocchio zu verstehen. Spielen Sie danach am besten zunächst einmal mit ein paar Beispielszenarien herum, die sich im Unterordner *plays* im Threadnocchio-Ordner befinden. Gehen Sie dabei folgendermaßen vor:

Doppelklicken Sie die Datei *threadnocchio.jar* im Threadnocchio-Ordner. Damit wird der Threadnocchio-Simulator gestartet. Zunächst erscheint ein Startbild (siehe Abbildung 2.1), dann das Hauptfenster des Threadnocchio-Simulators (siehe Abbildung 2.2).

Wählen Sie im Menü *Theaterstück* den Eintrag *Öffnen…* Begeben Sie sich über die Dateiauswahl in den Unterordner *plays* und wählen Sie ein Theaterstück aus, bspw. das Theaterstück *startrek* (siehe Abbildung 2.3). Es öffnet sich ein neues Hauptfenster mit dem Startrek-Theaterstück. Das andere Hauptfenster können Sie schließen.

Öffnen Sie nun im Dateibaum des Startrek-Fensters den Ordner *Actor*. Aktivieren Sie oberhalb vom Eintrag *Raumschiff* das Popup-Menü und wählen Sie im Popup-Menü

den Eintrag *new Raumschiff()* . Klicken Sie dann irgendwo in das Weltall-Bild (die Bühne). Dort wird dann ein Raumschiff platziert. Wiederholen Sie dies ein paar Mal.

Führen Sie dieselbe Aktion im Dateibaumordner *Prop* für den Eintrag *Fels* aus und platzieren Sie Felsen im Weltall (siehe auch Abbildung 2.4).

Starten Sie dann eine Simulation (Aufführung eines Theaterstücks) durch Anklicken des *Start*-Buttons in der Toolbar (alternativ den Eintrag *Start* im Menü *Steuerung* wählen). Resultat: Die Raumschiffe fliegen los. Immer wenn sie einen Fels berühren, wird dieser „weggebeamt“.

Sie können auch während einer Simulation weitere Raumschiffe und Felsen im Weltall platzieren.

Durch Doppelklick auf Raumschiff oder Fels im Dateibaum öffnen Sie jeweils einen Editor und können sich die Implementierung der beiden Klassen auf der Basis der Threadnocchio-API anschauen (siehe auch Abbildung 2.5). Führen Sie vielleicht ein paar einfache Änderungen durch. Anschließend müssen Sie die Änderungen abspeichern und compilieren. Das Compilieren erfolgt dabei zentral im Hauptfenster durch Anklicken des nach der Speicherung rot dargestellten Buttons in der Toolbar. Nach erfolgreicher Compilation verschwindet das Rot wieder. Anschließend können Sie wieder Raumschiffe und Felsen erzeugen und eine neue Simulation starten.

Abb. 2.1: Startbild des Threadnocchio-Simulators

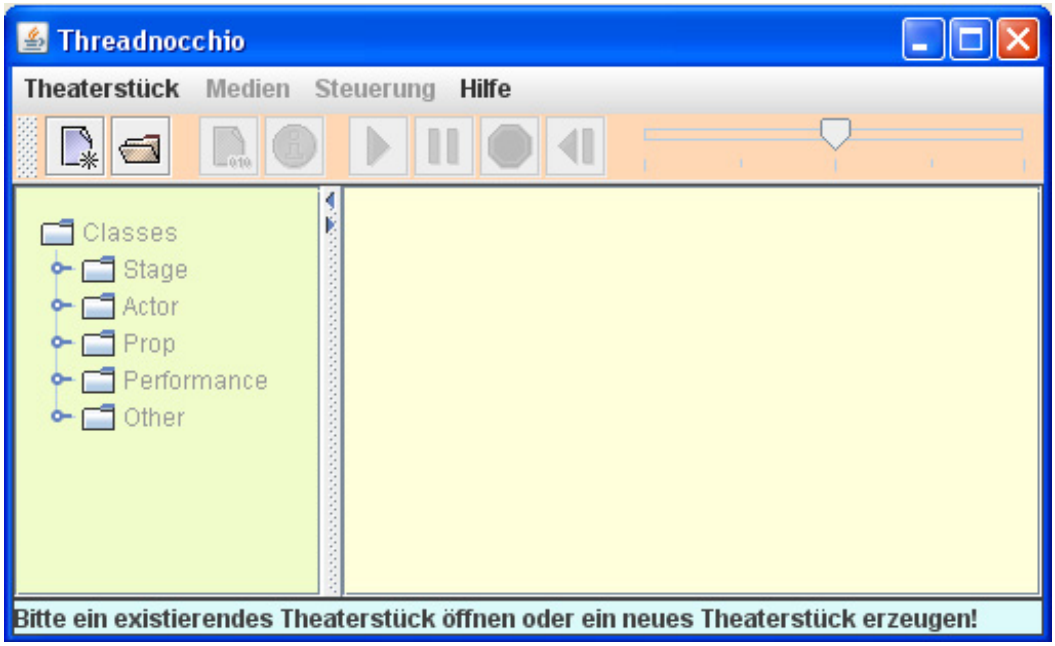

Abb. 2.2: Hauptfenster des Threadnocchio-Simulators

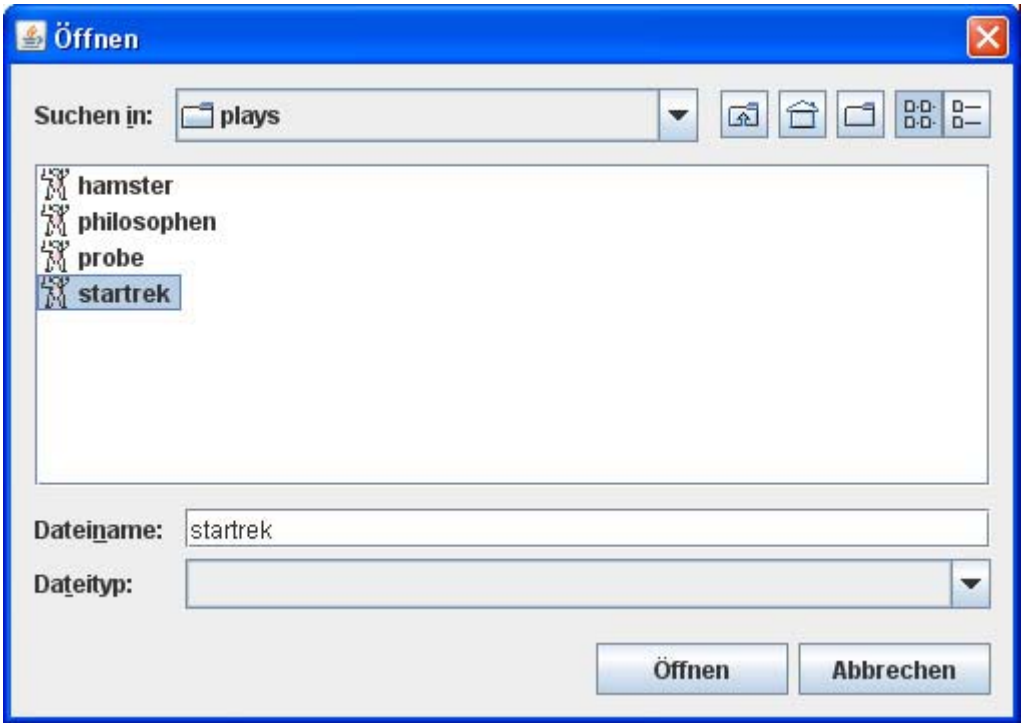

Abb. 2.3: Auswahl eines existierenden Theaterstückes

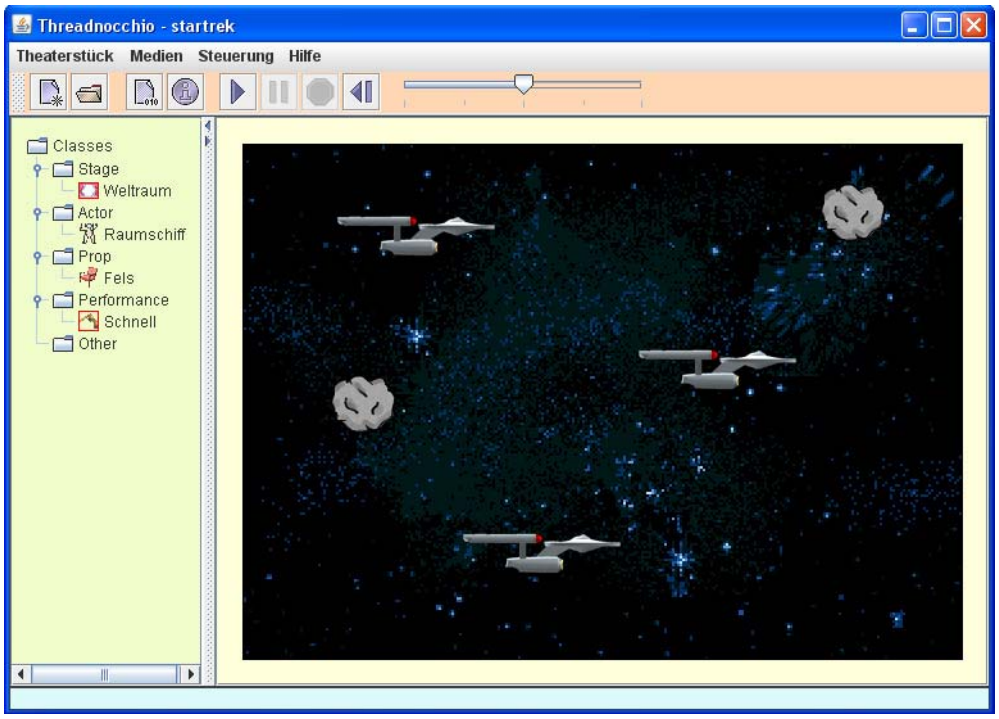

Abb. 2.4: Hauptfenster mit Startrek-Theaterstück

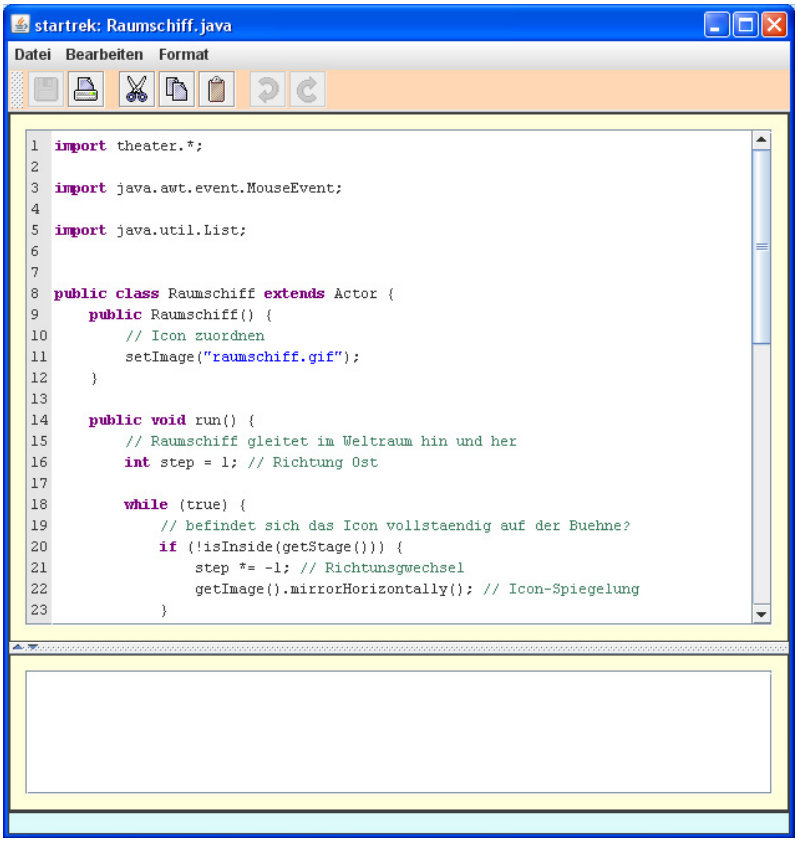

Abb. 2.5: Editor mit der Actor-Klasse Raumschiff

3 Grundlagen

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.

3.1 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.

3.2 Theaterstücke

Ein paralleles Programm wird in Threadnocchio als *Theaterstück* (Play) bezeichnet. Ein Theaterstück ist dabei gleichbedeutend mit dem Begriff „Projekt“, der in vielen Programmentwicklungsumgebungen benutzt wird. Ein Theaterstück setzt sich aus einer Menge von Klassen zusammen, die im Allgemeinen von durch Threadnocchio zur Verfügung gestellten Klassen abgeleitet werden.

3.3 Bühnen

Handlungsumfeld in Threadnocchio ist die *Bühne* (Stage). Bei der Bühne handelt es sich um ein rechteckiges Gebiet, das sich aus einzelnen quadratischen Zellen zusammensetzt. Die Größe, d.h. Breite und Höhe der Zellen wird in Pixeln angegeben (siehe Abbildung 3.1).

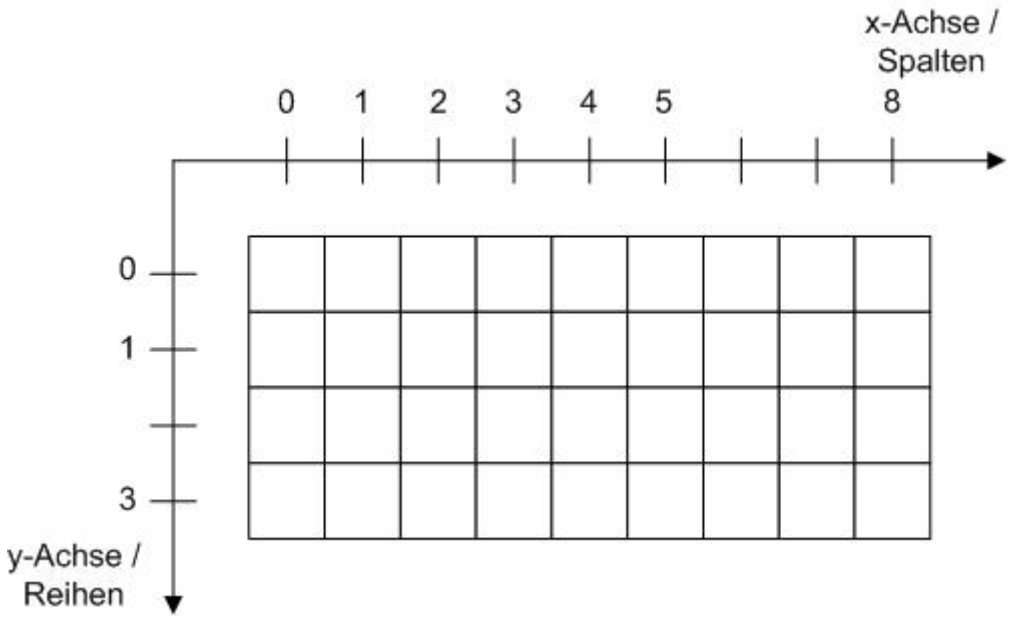

Abb. 3.1: Bühnenaufbau

Der Bühne ist ein Koordinatensystem zugeordnet. Die linke obere Ecke der Bühne besitzt die Koordinaten (x=0 / y=0). Die x-Achse wächst nach rechts, die y-Achse wächst nach unten. Prinzipiell ist das Koordinatensystem sowohl in den positiven als auch in den negativen Bereich nicht begrenzt, d.h. wenn von der Bühne gesprochen wird, ist meistens nur der sichtbare Bereich gemeint. Es ist durchaus jedoch auch möglich, dass sich Akteure „hinter der Bühne“, also im unsichtbaren Bereich der Bühne befinden.

Einer Bühne lässt sich ein Bild („Bühnenbild“) zuordnen. Dieses wird mit seiner linken oberen Ecke in die linke obere Ecke der Bühne platziert. Ist das Bühnenbild kleiner als die Bühne, wird es entsprechend oft repliziert, so dass die Bühne immer komplett von dem entsprechenden Bild ausgefüllt ist.

Bühnen werden in Threadnocchio repräsentiert als Klassen, die von der Threadnocchio-Klasse `Stage` abgeleitet werden. Ein Threadnocchio-Theaterstück muss immer mindestens eine, darf jedoch durchaus auch mehrere Bühnen bzw. Bühnenaufbauten, also entsprechende Klassen definieren. Welche Bühne dann tatsächlich beim Start einer Aufführung genutzt wird, kann vor dem Start der Aufführung durch einen einfachen Mausklick auf die entsprechende Klasse ausgewählt werden. Die ausgewählte Bühne wird als aktive Bühne bezeichnet. Auch während einer Aufführung ist ein Wechsel der Bühne durch Aufruf einer entsprechenden Methode möglich.

3.4 Akteure

Schauspieler bzw. selbstständige Marionetten, die auf der Bühne agieren, werden in Threadnocchio als *Akteure* (Actor) bezeichnet. Akteure sind dabei Objekte von Klassen, die von der Threadnocchio-Klasse `Actor` abgeleitet sind.

Akteuren kann ein Ikon zugeordnet werden, durch das die Akteure auf der Bühne repräsentiert werden. Die Platzierung von Akteuren auf der Bühne erfolgt durch Angabe der entsprechenden Zelle bzw. deren Spalte und Reihe. Optisch wird der Mittelpunkt des Ikons dabei auf den Mittelpunkt der Zelle gesetzt.

Programmiertechnisch handelt es sich bei den Akteuren um Threads. Die Klasse `Actor` ist dazu von der Java-Klasse `Thread` abgeleitet. Das Eigenleben eines Akteurs wird daher in einer Methode `public void run` der entsprechenden Actor-Klasse beschrieben.

Akteure können durch entsprechende Anweisungen im Konstruktor der aktiven Stage-Klasse initial auf einer Bühne auf der Bühne platziert werden. Sie können auch durch den Nutzer vor Beginn der Aufführung durch eine Drag-and-Drog-Aktion erzeugt und an beliebiger Stelle auf der Bühne platziert werden. Weiterhin ist es möglich, Akteure zur Laufzeit durch das Programm zu erzeugen, auf der Bühne zu platzieren und zu starten.

Die wichtigsten Aktionen, die Akteure ausführen können, sind Aktionen zum Bewegen auf der Bühne, zum Zuordnen von (neuen) Ikons oder auch Rotationen. Darüber hinaus können Kollisionen mit anderen Akteuren oder Gegenständen auf der Bühne überprüft werden. Eine interaktive Beeinflussung von Akteuren durch den Benutzer ist durch die Definition entsprechender Tastatur- bzw. Maus-Event-Handler möglich.

3.5 Requisiten

Neben Akteuren können sich noch *Requisiten* (Prop) – genauer gesagt ein ihnen jeweils zugeordnetes Ikon - auf der Bühne befinden. Im Unterschied zu Akteuren

sind Requisiten jedoch passive Objekte. Ihnen kann kein eigener Thread zugeordnet werden. Und das ist auch schon der einzige Unterschied zwischen Akteuren und Requisiten. Programmiertechnisch sind Requisiten nämlich Objekte von Klassen, die von der Threadnocchio-Klasse `Prop` abgeleitet sind. Sowohl die Klasse `Actor` als auch die Klasse `Prop` sind aber von der Threadnocchio-Klasse `Component` abgeleitet, die alle aufrufbaren Methoden für Akteure und Requisiten definiert. Daher werden Akteure und Requisiten im Folgenden auch häufig unter dem Begriff *Komponenten* zusammengefasst.

Genauso wie Akteure können auch Requisiten initial, per Drag-and-Drop-Aktion oder durch das Programm erzeugt und auf der Bühne platziert werden. Sie lassen sich verschieben und rotieren und ihr Ikon kann ausgetauscht werden. Es lassen sich Kollisionen zwischen Akteuren und Requisiten überprüfen und auch Requisiten können Tastatur- und Maus-Event-Handler zugeordnet werden.

3.6 Aufführungen

Die Ausführung eines Programms wird in Threadnocchio mit der *Aufführung* des Theaterstücks (Performance) assoziiert. Normalerweise wird dabei eine Standard-Aufführung genutzt. Threadnocchio-Theaterstücken können jedoch auch zusätzliche Klassen zugeordnet werden, die die Art der Aufführung in bestimmten Eigenschaften variieren. Dies geschieht durch die Ableitung einer Klasse von der Threadnocchio-Klasse `Performance`. Welche Aufführung dann tatsächlich genutzt wird, kann vor dem Start der Aufführung durch einen einfachen Mausklick auf die entsprechende Klasse ausgewählt werden. Die ausgewählte Aufführung wird als aktive Aufführung bezeichnet.

Die Variation einer Ausführung ergibt sich aus dem Überschreiben bestimmter Methoden der Klasse Performance. Dies sind insbesondere die Methoden `started`, `stopped`, `suspended`, `resumed` und `speedChanged`. Die Methoden werden unmittelbar nach der entsprechenden Steuerungsaktion ausgeführt, auch wenn diese durch die Steuerungselemente im Threadnocchio-Simulator ausgelöst wurden. Soll bspw. beim Start einer Aufführung die Ausführungsgeschwindigkeit des Programms maximiert werden, kann dies durch ein entsprechendes Überschreiben der `started`-Methode umgesetzt werden.

3.7 Hilfsklassen

Neben den von den Threadnocchio-Klassen abgeleiteten Klassen lassen sich weitere Hilfsklassen definieren und in die Programme einbinden. Diese Klassen werden als *Hilfsklassen* bezeichnet.

4 Threadnocchio-Simulator

Der Threadnocchio-Simulator ist ein graphisch-interaktives Werkzeug zum Entwickeln und Ausführen von Threadnocchio-Programmen. Abbildung 4.1 skizziert seinen Aufbau.

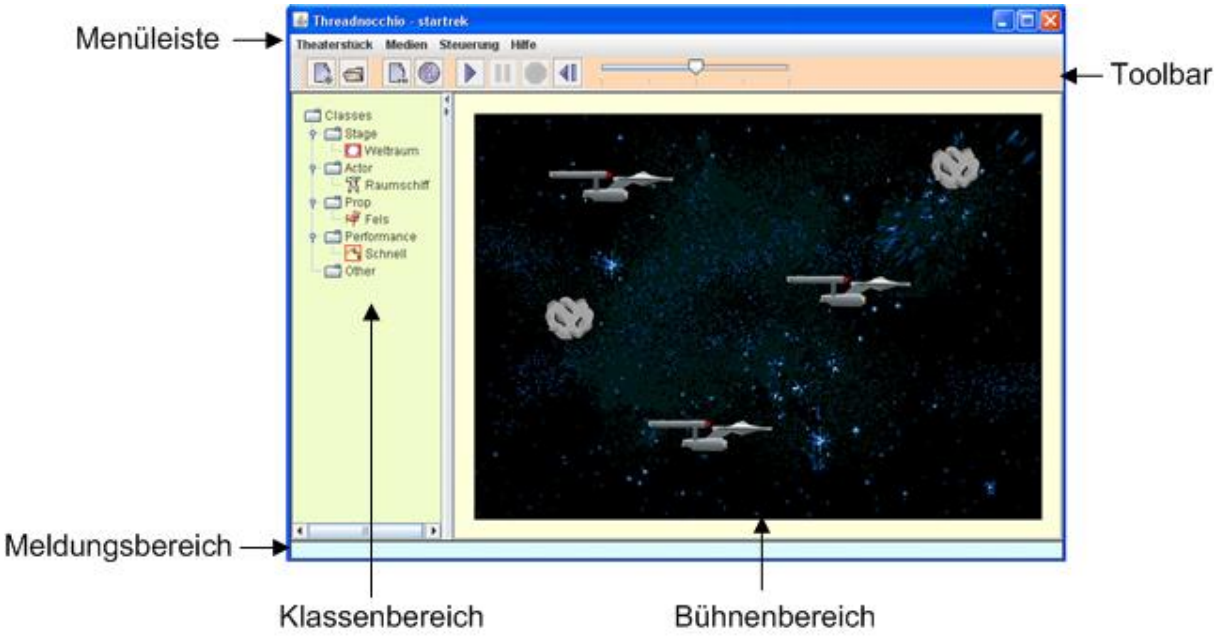

Abb. 4.1: Aufbau des Threadnocchio-Simulators

Ganz oben befindet sich die Menüleiste mit den vier Menüs *Theaterstück*, *Medien*, *Steuerung* und *Hilfe*. In der darunter liegenden Toolbar sind die wichtigsten Funktionalitäten des Simulators durch einen einzelnen Mausklick ausführbar. Im mittleren Bereich sieht man links den Klassenbereich, in dem die existierenden Klassen des bearbeiteten Theaterstücks angezeigt werden. Rechts im Bühnenbereich wird die aktuelle Bühne angezeigt. Ganz unten im Meldungsbereich werden Hinweise und Fehlermeldungen ausgegeben.

4.1 Menüs

Die vier Menüs des Threadnocchio-Hauptfensters stellen zahlreiche Funktionen zur Bearbeitung und Ausführung von Theaterstücken Verfügung. Es sei jedoch anzumerken, dass Sie die am häufigsten benutzten Funktionen auch schnell und direkt über entsprechende Buttons der Toolbar auslösen können (siehe Abschnitt 4.2).

Über das MenüTheaterstück können Sie neue Theaterstücke erzeugen, existierende Theaterstücke öffnen sowie die Bearbeitung des angezeigten Threadnocchio-Theaterstücks beenden (siehe Abbildung 4.2).

Abb. 4.2: Theaterstück-Menü

Das Menu *Medien* unterstützt den Import von Bild- und Ton-Dateien in die Unterordner *images* bzw. *sounds* des entsprechenden Theaterstücks (siehe Abbildung 4.3). Nach der Auswahl des entsprechenden Eintrags erscheint eine Dateiauswahlbox, über die Sie die zu importierende Datei auswählen können. Alternativ können Sie natürlich auch auf der Ebene des Betriebssystems die Dateien in die Unterordner kopieren.

Abb. 4.3: Medien-Menü

Das Menu *Steuerung* stellt Funktionen zur Steuerung der Aufführung eines Theaterstücks zur Verfügung (siehe Abbildung 4.4). Hier werden Sie aber höchstwahrscheinlich eher die entsprechenden Buttons in der Toolbar benutzen.

Abb. 4.4: Steuerung-Menü

Über das Menü *Hilfe* haben Sie insbesondere die Möglichkeit, dieses Benutzerhandbuch zu öffnen. Weiterhin haben Sie Zugriff auf die detaillierte Referenz der Threadnocchio-API im javadoc-Format (siehe auch Abbildung 4.5).

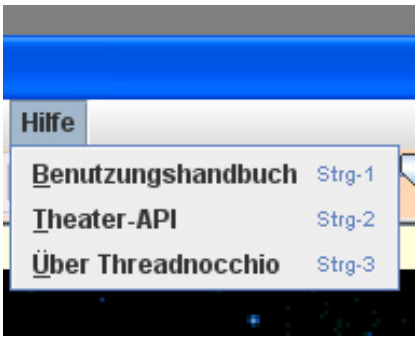

Abb. 4.5: Hilfe-Menü

4.2 Toolbar

Über die Toolbar sind die wichtigsten Funktionalitäten des Simulators durch einen einfachen Mausklick ausführbar. Abbildung 4.6 skizziert den Aufbau der Toolbar.

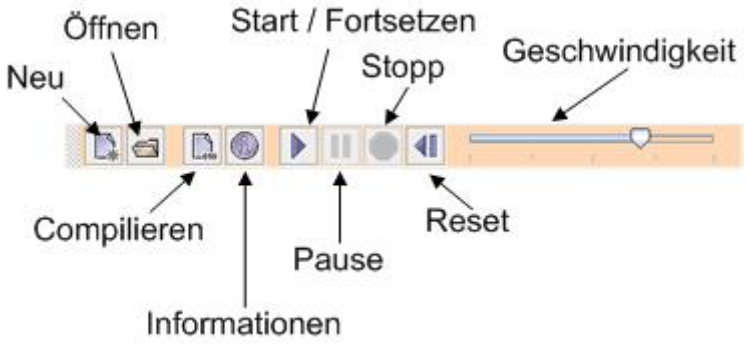

Abb. 4.6: Aufbau der Toolbar

- *Neu*: Button zum Erzeugen und Öffnen eines neuen Theaterstücks. Nach dem Anklicken öffnet sich eine Dialogbox, in der der Name des neuen Theaterstücks angegeben werden muss. Nach Klick auf „OK“ in der Dialogbox öffnet sich ein weiteres Simulatorfenster für das (noch leere) neue Theaterstück. Anzumerken ist, dass mehrere Theaterstücke gleichzeitig geöffnet und bearbeitet werden können. Einzelne Aktionen (Speichern, Kompilieren, …) beziehen sich aber nur jeweils auf das entsprechende Theaterstück.
- *Öffnen*: Button zum Öffnen eines bereits existierenden Theaterstücks. Nach dem Anklicken öffnet sich eine Dialogbox, in der das existierendeTheaterstück ausgewählt werden kann.
- *Compilieren*: Button zum Compilieren aller Klassen des Theaterstücks. Enthalten Klassen syntaktische Fehler wird pro Klasse ein Editorfenster geöffnet und die Fehler werden dort angezeigt. Der Compilierbutton erscheint mit rotem Hintergrund, wenn Klassen geändert und gespeichert aber noch nicht neu compiliert wurden oder wenn noch nicht beseitigte Compilierfehler bestehen.
- *Informationen*: Button zum Editieren bzw. Abrufen genereller Informationen zum Theaterstück. Beim Anklicken des Buttons öffnet sich ein Fenster, in dem

Informationen zum Theaterstück (Entwickler, Version, Anleitung, …) eingegeben bzw. abgerufen werden können.

- *Start / Fortsetzen*: Steuerungsbutton zum Starten einer Aufführung des Theaterstücks. Beim Anklicken des Buttons wird eine Aufführung gestartet oder fortgesetzt, falls sie pausiert war.
- *Pause*: Steuerungsbutton zum Pausieren einer Aufführung. Beim Anklicken des Buttons wird eine laufende Aufführung pausiert bzw. angehalten. Durch anschließendes Klicken auf den Start/Fortsetzen-Button kann sie anschließend wieder fortgesetzt werden.
- *Stopp*: Steuerungsbutton zum Stoppen einer Aufführung. Beim Anklicken des Buttons wird eine Aufführung endgültig gestoppt, d.h. alle laufenden Threads abgebrochen. Da gestoppte Threads in Java nicht erneut aktiviert werden können, ist eine anschließende Fortsetzung des Theaterstücks nicht mehr möglich. Stattdessen muss das Theaterstück über den Reset-Button komplett auf den initialen Zustand zurückgesetzt werden.
- *Reset*: Steuerungsbutton zum Zurücksetzen einer Aufführung. Beim Anklicken dieses Buttons wird das Theaterstück in seinen Ausgangszustand zurückgesetzt, d.h. aktive Akteure werden gestoppt und alle Komponenten werden gelöscht. Anschließend wird der Konstruktor der aktiven Bühne neu ausgeführt.
- *Geschwindigkeit*: Steuerungsregler zum Regeln der Geschwindigkeit einer Aufführung. Je weiter der Regler rechts steht, desto schneller ist die Ausführungsgeschwindigkeit.

4.3 Klassenbereich

Im Klassenbereich wird ein Klassenbaum angezeigt, der die aktuellen Klassen des Theaterstücks enthält bzw. anzeigt. Die Klassen sind dabei gegliedert in Stage-, Actor-, Prop-, Performance- und Other-Klassen.

Möchte man eine neue Klasse erzeugen, muss man über dem entsprechenden Ordner ein Popup-Menü aktivieren (rechte Maustaste) und dort das Item „Neue Unterklasse …“ auswählen. Anschließend wird man aufgefordert, den Namen der neuen Klasse einzugeben. Wichtig: Bei dem Namen muss es sich um einen korrekten Bezeichner für Java-Klassen handeln. Nach Anklicken des „OK“-Buttons wird eine neue Klasse erzeugt und in den Klassenbaum eingefügt. Durch Doppelklick auf den Namen der Klasse öffnet sich ein Editor, in dem man die Klasse editieren kann.

Für jede Klasse existiert ein Popup-Menu. Nach dessen Aktivierung (rechte Maustaste) hat man die Möglichkeit, den Editor aufzurufen oder die Klasse zu löschen. Weiterhin werden im Popup-Menü alle public-static-Methoden der Klasse aufgeführt, die interaktiv durch Mausklick aufgerufen werden können.

In den Popup-Menüs von Stage-, Actor- und Prop-Klassen erscheinen ferner auch für jeden public-Konstruktor der Klasse Items der Form „new <Konstruktor>“. Hierüber ist es möglich, interaktiv neue Objekte der entsprechenden Klassen zu erzeugen.

Nach der Ausführung eines new-Items für eine Bühnenklasse wird der entsprechende Konstruktor ausgeführt und unter Umständen eine neue Bühne in den Bühnenbereich geladen. (Hinweis: In der aktuellen Version wird nur der Default-Konstruktor unterstützt, der unbedingt vorhanden und als public definiert sein muss).

Nach der Ausführung eines new-Items für eine Actor- oder Prop-Klasse wird ein Objekt der entsprechenden Klasse erzeugt. Durch Klicken mit der Maus auf die Bühne wird die Komponente an der entsprechenden Stelle auf die Bühne platziert. Ist aktuell eine Aufführung gestartet, wird im Falle eines Akteurs der neue Akteur auch unmittelbar gestartet.

In Threadnocchio ist es möglich, für ein Theaterstück mehrere Bühnen und mehrere Aufführungen zu definieren. Die aktive Bühne bzw. Aufführung lässt sich durch Mausklick auf die entsprechende Klasse auswählen. Man erkennt sie an einer roten Umrandung des jeweiligen Klassen-Ikons im Klassenbaum. Nach Wechsel der aktiven Bühnenklasse wird unmittelbar der Default-Konstruktor der neuen aktiven Klasse ausgeführt und die neue Bühne im Bühnenbereich angezeigt.

4.4 Bühnenbereich

Im Bühnenbereich wird die aktuell aktive Bühne angezeigt. Hier agieren während einer Aufführung die Akteure.

Wenn keine Aufführung aktiv ist, lassen sich Komponenten durch Anklicken mit der Maus beliebig auf der Bühne verschieben. Auch ist es möglich, für jede Komponente ein Popup-Menu zu aktivieren. In diesem Popup-Menu erscheinen alle public-Methoden der entsprechenden Klasse und können interaktiv für das entsprechende Objekt ausgeführt werden.

4.5 Meldungsbereich

Im Meldungsbereich werden wichtige Hinweise und Fehlermeldungen bei der Benutzung des Simulators ausgegeben. Die Ausgaben im Meldungsbereich verschwinden nach einer gewissen Zeitspanne wieder.

4.6 Editor

Im Editor können Klassen editiert werden. Abbildung 4.7 skizziert den Aufbau des Editors.

In der Titelzeile erscheint der Name der Datei. Ein Sternchen hinter dem Namen deutet an, dass Änderungen im Sourcecode vorgenommen, die Datei aber noch nicht abgespeichert worden ist. Unter der Titelzeile befindet sich die Menüleiste mit den Menüs Datei und Bearbeiten. Die wichtigsten Funktionalitäten des Editors sind über die Toolbar per Mausklick aktivierbar. Im Programmbereich unter der Toolbar kann der Sourcecode bearbeitet werden. Nach dem Compilieren (erfolgt immer zentral im Simulatorfenster) erscheinen Compilierfehler im Compilierfehlerbereich. Im Meldungsbereich ganz unten im Editorfenster werden Hinweise und Benutzungsfehler angezeigt.

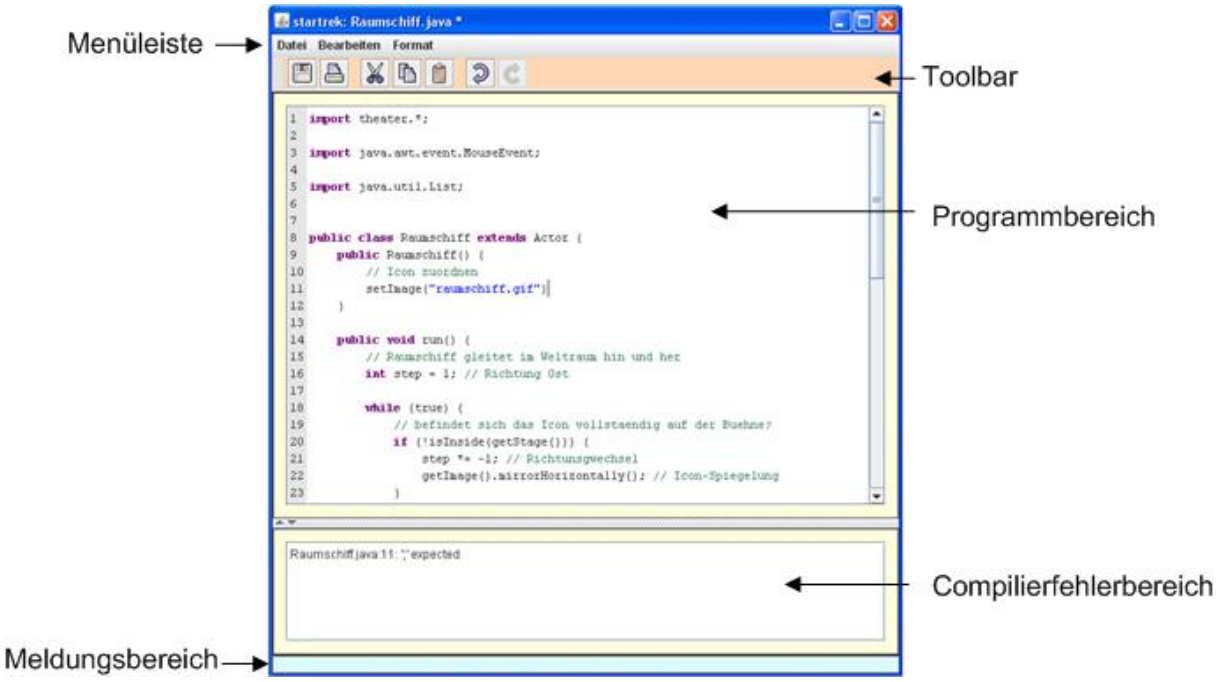

Abb. 4.7: Aufbau des Editors

5 Threadnocchio-API

Die Threadnocchio-API oder –Klassenbibliothek besteht aus einer Menge von Klassen und Interfaces, auf deren Grundlage Threadnocchio-Programme entwickelt werden können:

- **Stage**: Basisklasse zur Definition einer Bühne.
- **Component**: Oberklasse der Klassen Actor und Stage.
- **Actor**: Basisklasse zur Definition eines Akteurs.
- **Prop**: Basisklasse zur Definition einer Requisite.
- **Performance**: Basisklasse zur Definition einer Aufführung.
- **TheaterImage**: Klasse zum Erzeugen und Manipulieren von Bildern und Icons
- **MouseInfo**: Klasse, die Informationen zu Maus-Ereignissen verwaltet
- **KeyInfo**: Klasse, die Informationen zu Tastatur-Ereignissen verwaltet
- **PixelArea**: Interface zur Umsetzung der Kollisionsentdeckung
- **Rectangle**: Klasse, die das Interface PixelArea für rechteckige Gebiete implementiert
- **Point**: Klasse, die das Interface PixelArea für einzelne Pixel implementiert
- **Cell**: Klasse, die das Interface PixelArea für einzelne Zellen einer Bühne implementiert
- **CellArea**: Klasse, die das Interface PixelArea für rechteckige Gebiete mit mehreren Zellen einer Bühne implementiert

5.1 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 Komponenten, d.h. Akteuren und Requisiten, auf der Bühne dient. Das Koordinatensystem ist nicht endlich, so dass sich Akteure und Requisiten auch außerhalb der Bühne befinden können, also (zwischenzeitlich) nicht sichtbar sind. Der Größe der Bühne lässt sich später nicht mehr ändern.

Die Methoden der Klasse `Stage` lassen sich in vier Kategorien einteilen:

- Gestaltungsmethoden (Methoden zur Gestaltung der Bühne)
- Getter-Methoden (Methoden zum Abfragen des Zustandes der Bühne)
- Kollisionserkennungsmethoden (Methoden zur Ermittlung bestimmter Komponenten)
- Event-Methoden (Methoden zur Reaktion auf Maus- und Tastaturevents)

5.1.1 Gestaltungsmethoden

Für das Platzieren von Komponenten auf der Bühne existiert eine `add`-Methode. Dieser wird neben der Komponente die Spalte und Reihe der Zelle übergeben, auf der die Komponente platziert werden soll. Platzieren bedeutet dabei, dass das der Komponente zugeordnete Icon oberhalb der Bühne angezeigt wird, und zwar wird der Mittelpunkt des Icons auf den Mittelpunkt der entsprechenden Zelle gesetzt. Über eine `remove`-Methode können Komponenten wieder von der Bühne entfernt werden.

Zwei setBackground-Methoden erlauben die Zuordnung eines Hintergrundbildes zu einer Bühne. Das Bild wird dabei in der linken oberen Ecke der Bühne platziert. Ist es größer als die Bühne, wird es rechts und unten abgeschnitten. Ist es kleiner als die Bühne, wird es repliziert dargestellt, bis die Bühne vollständig überdeckt ist. Zulässig sind Bilder im gif-, jpg und png-Format. Bilddateien müssen im Unterverzeichnis „images“ des entsprechenden Theaterstück-Verzeichnisses gespeichert werden.

Bei jedweder Threadnocchio-internen Änderung der Bühne während einer Aufführung wird unmittelbar automatisch die geänderte Bühne im Threadnocchio-Simulator neu gezeichnet. Für Änderungen, auf die Threadnocchio selbst keinen Einfluss hat (bspw. Änderungen des AWT-Images eines TheaterImages) steht die Methode `paint` zur Verfügung hat, die die Bühne explizit neu zeichnet.

5.1.2 Getter-Methoden

Über die Getter-Methoden kann der Zustand der Bühne abgefragt wird. Dies beinhaltet die Anzahl an Spalten der Bühne, die Anzahl an Reihen, die Größe der Zellen und das zugeordnete Hintergrundbild.

5.1.3 Kollisionserkennungsmethoden

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.

Über eine Methode `getComponents` können alle Komponenten ermittelt werden, die sich aktuell auf der Bühne befinden. Die Methode `getComponentsAt` schränkt die Abfrage auf eine bestimmte Zelle ein.

Die Methode `getComponentsInside` liefert alle Komponenten innerhalb einer bestimmten PixelArea und die Methode `getIntersectingComponents` liefert alle Komponenten, die eine bestimmte PixelArea berühren oder schneiden.

Allen Kollisionserkennungsmethoden kann optional eine Menge von Klassenobjekten übergeben werden. In diesem Fall werden nur Objekte der entsprechenden Klassen bei der Kollisionserkennung berücksichtigt.

Die Klasse `Stage` implementiert übrigens das Interface `PixelArea` und kann damit unmittelbar selbst in die Kollisionserkennung mit einbezogen werden. Die Methode `contains` überprüft, ob ein bestimmter Punkt innerhalb der Bühne liegt, über die

Methode `isInside` lässt sich ermitteln, ob die Bühne vollständig innerhalb einer bestimmten PixelArea liegt und die Methode `intersects` liefert true, falls die Bühne eine bestimmte PixelArea schneidet.

5.1.4 Event-Methoden

Soll eine Bühne auf Maus- und Tastatur-Events reagieren, können entsprechend des genauen Event-Typs folgende Methoden überschrieben werden: `keyTyped`, `keyPressed`, `keyRealeased`, `mousePressed`, `mouseReleased`, `mouseClicked`, `mouseDragged`, `mouseMoved`, `mouseEntered`, `mouseExited`. Die Events entsprechen dabei den Events des Java-AWT in den Klassen `java.awt.event.KeyListener`, `java.awt.event.MouseListener` bzw. `java.awt.event.MouseMotionListener`. Den Methoden werden Objekte vom Typ `KeyInfo` bzw. `MouseInfo` übergeben, über die genauere Informationen über das entsprechende Event abgefragt werden können.

Ob eine Bühne überhaupt über Tastatur- und Maus-Events informiert werden soll, lässt sich über die Methoden `setHandlingKeyEvents` bzw. `setHandlingMouseEvents` festlegen und über die Methoden `isHandlingKeyEvents` bzw. `isHandlingMouseEvents` abfragen. Standardmäßig wird die Bühne über Tastatur- und MouseEvents informiert.

5.2 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 (im Folgenden zusammengefasst als *Komponenten* bezeichnet), 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 Methoden der Klasse Component lassen sich in vier Kategorien einteilen:

- Manipulationsmethoden (Methoden zur Veränderung von Komponenten)
- Getter-Methoden (Methoden zum Abfragen des Zustandes von Komponenten)
- Kollisionserkennungsmethoden (Methoden zur Ermittlung bestimmter Komponenten)
- Event-Methoden (Methoden zur Reaktion auf Maus- und Tastaturevents)

5.2.1 Manipulationsmethoden

Mit Hilfe zweier `setImage`-Methoden ist es möglich, einer Komponente ein Bild oder Icon zuzuordnen, über das sie auf der Bühne repräsentiert wird. Zulässig sind Bilder im gif-, jpg und png-Format. Bilddateien müssen im Unterverzeichnis „images“ des entsprechenden Theaterstück-Verzeichnisses gespeichert werden.

Die Methode `setLocation` erlaubt das Umplatzieren einer Komponente auf der Bühne. Angegeben werden die neue Spalte und Reihe. Das Ändern der z-Koordinate ist über die Methode `setZCoordinate` möglich. Die z-Koordinate bestimmt die Zeichenreihenfolge und damit u. U. Sichtbarkeit von Komponenten. Es gilt: Je höher die z-Koordinate einer Komponente ist, desto später wird sie gezeichnet, d.h. umso weiter gelangt sie in den Vordergrund. Bei gleicher z-Koordinate zweier Komponenten ist deren Zeichenreihenfolge undefiniert.

Komponenten kann ein Rotationswinkel zugeordnet werden. Dazu dient die Methode `setRotation`. Der übergebene Parameter definiert den absoluten Rotationswinkel des Icons der Komponente im Uhrzeigersinn. Standardmäßig ist der Rotationswinkel einer Komponente 0. Es ist zu beachten, dass der Rotationswinkel keinen Einfluss auf die Weite und Höhe eines einer Komponente zugeordneten Icons hat. Diese werden immer auf der Grundlage eines Rotationswinkels von 0 berechnet.

Über die Methode `setVisible` kann die Sichtbarkeit einer Komponente verändert werden. Standardmäßig sind Komponenten sichtbar. Es ist zu beachten, dass auch unsichtbare Komponenten in Kollisionsberechnungen mit einbezogen werden.

Alle Initialisierungen einer Komponente, die unabhängig von der Bühne sind, können im entsprechenden Konstruktor erfolgen. Für alle Bühnen-abhängigen Initialisierungen muss eine Methode `addedToStage` überschrieben werden, die aufgerufen wird, sobald die Komponente auf der Bühne platziert wurde.

5.2.2 Getter-Methoden

Über Getter-Methoden lässt sich der Zustand einer Komponente abfragen. Es sind entsprechende Methoden für die Abfrage des zugeordneten Icons, der aktuellen Position der Komponente auf der Bühne, der z-Koordinate, des Rotationswinkels und der Sichtbarkeit definiert.

Weiterhin existieren Methoden zur Ermittelung des aktuellen Bühnen- und Performance-Objektes.

5.2.3 Kollisionserkennungsmethoden

Die Klasse `Component` implementiert das Interface `PixelArea`. Damit können Komponenten unmittelbar selbst in die Kollisionserkennung mit einbezogen werden. Die Methode `contains` überprüft, ob ein bestimmter Punkt innerhalb des Icons einer Komponente liegt, über die Methode `isInside` lässt sich ermitteln, ob das Icon einer Komponente vollständig innerhalb einer bestimmten PixelArea liegt, und die Methode `intersects` liefert true, falls das Icon einer Komponente eine bestimmte PixelArea schneidet.

5.2.4 Event-Methoden

Soll eine Komponente auf Maus- und Tastatur-Events reagieren, können entsprechend des genauen Event-Typs folgende Methoden überschrieben werden: `keyTyped`, `keyPressed`, `keyRealeased`, `mousePressed`, `mouseReleased`,

mouseClicked, mouseDragged, mouseMoved, mouseEntered, mouseExited. Die Events entsprechen dabei den Events des Java-AWT in den Klassen java.awt.event.KeyListener, java.awt.event.MouseListener bzw. java.awt.event.MouseMotionListener. Den Methoden werden Objekte vom Typ KeyInfo bzw. MouseInfo übergeben, über die genauere Informationen über das entsprechende Event abgefragt werden können.

Ob eine Komponente überhaupt über Tastatur- und Maus-Events informiert werden soll, lässt sich über die Methoden setHandlingKeyEvents bzw. setHandlingMouseEvents festlegen und über die Methoden isHandlingKeyEvents bzw. isHandlingMouseEvents abfragen. Standardmäßig wird eine Komponente über Tastatur- und MouseEvents informiert.

Die Reihenfolge, in der Komponenten bzw. die Bühne über Maus- und Tastatur-Events benachrichtigt werden, ergibt sich aus der Zeichenreihenfolge. Je weiter eine Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des Events (Aufruf der Methode consume) kann die Benachrichtigungskette abgebrochen werden.

5.2.5 Akteure

Das Eigenleben eines Akteurs wird in einer Methode „public void run“ der entsprechenden Klasse festgelegt. Dabei können u. a. die von der Klasse Component und Thread geerbten Methoden eingesetzt werden. Bei Aufruf der Methode start für ein Objekt einer von der Klasse Actor abgeleiteten Klasse, wird ein neuer Thread gestartet, der die Methode run ausführt.

5.3 Performance

Die Klasse Performance definiert Methoden zur Steuerung und Verwaltung der Ausführung von Threadnocchio-Programmen:

- stop: Stoppt eine Aufführung.
- suspend: Pausiert eine Aufführung.
- setSpeed: Ändert die Ausführungsgeschwindigkeit einer Aufführung.
- freeze: Friert die Ausgabe der Bühne ein, die Akteure agieren jedoch weiter
- unfreeze: Das Einfrieren der Bühne wird wieder aufgehoben

Weiterhin werden folgende Methoden definiert, die unmittelbar nach entsprechenden Steuerungsaktionen aufgerufen werden, und zwar auch, wenn die Aktionen durch die Steuerungsbuttons des Simulators ausgelöst wurden:

- started
- stopped
- suspended
- resumed
- speedChanged

Möchte ein Programmierer zusätzliche Aktionen mit den entsprechenden Steuerungsaktionen einhergehen lassen, kann er eine Unterklasse der Klasse `Performance` definieren und hierin die entsprechende Methode überschreiben.

Zusätzlich stellt die Klasse `Performance` eine Methode `playSound` zur Verfügung, mit der eine Audio-Datei abgespielt werden kann. Die Datei muss sich im Unterverzeichnis „sounds“ des entsprechenden Theraterstücks befinden. Unterstützt werden die Formate wav, au und aiff.

Über die Methode `setActiveStage` ist es möglich, die aktuell aktive Bühne gegen eine andere Bühne auszutauschen, d.h. das Bühnenbild zu wechseln. Unter Umständen aktive Akteure der alten Bühne werden dabei nicht automatisch gestoppt. Die Methode `getActiveStage` liefert die gerade aktive Bühne.

Das während einer Aufführung aktuelle Performance-Objekt kann mit Hilfe der statischen Methode `getPerformance` ermittelt werden. Ein Wechsel des Performance-Objektes ist während einer Aufführung nicht möglich.

5.4 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: `keyTyped`, `keyPressed`, `keyRealeased`, `mousePressed`, `mouseReleased`, `mouseClicked`, `mouseDragged`, `mouseMoved`, `mouseEntered`, `mouseExited`. Die Events entsprechen dabei den Events des Java-AWT in den Klassen `java.awt.event.KeyListener`, `java.awt.event.MouseListener` bzw. `java.awt.event.MouseMotion-Listener`. Zur Dokumentation sei auf die Java-AWT-Dokumentation verwiesen. Den Methoden werden Objekte vom Typ `KeyInfo` bzw. `MouseInfo` übergeben, über die genauere Informationen über das entsprechende Event abgefragt werden können.

Soll ein Akteur, eine Requisite oder eine 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.

Ob eine Komponente bzw. eine Bühne überhaupt über Tastatur- und Maus-Events informiert werden soll, lässt sich über die Methoden `setHandlingKeyEvents` bzw. `setHandlingMouseEvents` festlegen und über die Methoden `isHandlingKeyEvents` bzw. `isHandlingMouseEvents` abfragen. Standardmäßig werden Komponenten und Bühnen über Tastatur- und MouseEvents informiert.

Die Reihenfolge, in der Komponenten bzw. die aktive Bühne über Maus- und Tastatur-Events benachrichtigt werden, ergibt sich aus der Zeichenreihenfolge. Je weiter eine Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die aktive Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des

Events (Aufruf der Methode consume) kann die Benachrichtigungskette abgebrochen werden. Das den Methoden übergebene KeyInfo- bzw. MouseInfo-Objekt ist dabei pro Event immer das gleiche. Zur Kommunikation zwischen den benachrichtigten Komponenten stellen die Klassen KeyInfo und MouseInfo eine Methode setUserObject zur Verfügung, über die eine Komponente dem KeyInfo- bzw. MouseInfo-Objekt ein beliebiges anwendungsspezifisches Objekt zuordnen kann, das sich durch eine später benachrichtigte Komponente durch Aufruf der Methode getUserObject wieder abfragen lässt.

5.4.1 KeyInfo

Die Klasse KeyInfo ist von der Klasse java.awt.event.KeyEvent abgeleitet, stellt also alle deren geerbte Methoden zur Verfügung. An dieser Stelle sei auf die entsprechende Dokumentation der Klasse java.awt.event.KeyEvent verwiesen. Überschrieben ist die Methode getSource, die das aktuelle Bühnenobjekt liefert.

5.4.2 MouseInfo

Die Klasse MouseInfo ist von der Klasse java.awt.event.MouseEvent abgeleitet, stellt also alle deren geerbte Methoden zur Verfügung. An dieser Stelle sei auf die entsprechende Dokumentation der Klasse java.awt.event.MouseEvent verwiesen.

Überschrieben ist die Methode getSource. Wird das MouseInfo-Objekt einer Event-Handler-Methode des Bühnen-Objektes übergeben, liefert die Methode das aktuelle Bühnen-Objekt. Wird das MouseInfo-Objekt einer Event-Handler-Methode eines Komponenten-Objektes übergeben, liefert die Methode das entsprechende Komponenten-Objekt.

Ebenfalls überschrieben sind die Methoden getX, getY und getPoint. Im Falle, dass das Source-Objekt das Bühnen-Objekt ist, liefern die Methoden die Pixel-Koordinaten des Maus-Events relativ gesehen zur Bühne. Im Falle, dass das Source-Objekt eine Komponente ist, werden die Pixel-Koordinaten des Maus-Events relativ gesehen zu dem der Komponente zugeordneten Ikon geliefert. Über die Methoden getColumn bzw. getRow kann abgefragt werden, über welcher Zelle sich der Mauszeiger während des Auftretens der Events befunden hat..

5.5 TheaterImage

TheaterImage ist eine Threadnocchio-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.

Initialisiert wird ein TheaterImage über entsprechende Konstruktoren entweder mittels einer Bilddatei, eines bereits existierenden TheaterImages oder leer in einer gewünschten Größe. Im ersteren Fall sind Bilder im gif-, jpg und png-Format

zulässig. Die Bilddateien müssen im Unterverzeichnis „images“ des entsprechenden Theaterstück-Verzeichnisses gespeichert werden.

Die wichtigsten Bild-Manipulations-Methoden der Klasse `TheaterImage` sind: `setColor`(legt die Zeichenfarbe fest), `setFont` (legt den Zeichenfont fest), `draw`-Methoden zum Zeichnen von anderen Images, Linien, Rechtecken, Ovalen, Polygonen in der aktuellen Zeichenfarbe, `fill`-Methoden zum Zeichnen von gefüllten Rechtecken, Ovalen und Polygonen in der aktuellen Zeichenfarbe, `drawString` zum Zeichnen eines Textes im aktuellen Font sowie `mirrorHorizontally`, `mirrorVertically`, `rotate` und `scale` zum Spiegeln, Rotieren und Skalieren des Bildes. Dabei ist zu beachten, dass bei den `mirror`- und `rotate`-Methoden die Größe des Bildes nicht verändert wird, so dass unter Umständen Teile des Bildes nicht mehr sichtbar sind. Weiterhin kann über die Methoden `clear` bzw. `fill` ein TheaterImage gelöscht oder komplett mit der aktuellen Farbe gefüllt werden.

Eine Menge von Getter-Methoden erlaubt die Abfrage von Eigenschaften eines TheaterImage, wie Größe, aktuelle Farbe oder aktueller Font. Intern wird ein TheaterImage mit Hilfe der Klasse `java.awt.image.BufferedImage` realisiert. Das entsprechende Objekt kann über die Methode `getAWTImage` abgefragt und direkt angesprochen werden.

5.6 Kollisionserkennung

`PixelArea` ist ein Interface, das die Grundlage der Kollisionserkennungsmethoden darstellt. Eine PixelArea kann man sich dabei als ein beliebiges Gebiet auf der Bühne vorstellen.

Das Interface `PixelArea` definiert genau drei Methoden. Die Methode `contains` überprüft, ob ein angegebener Punkt innerhalb der PixelArea liegt. Die Methode `isInside` überprüft, ob die aufgerufene PixelArea komplett innerhalb einer anderen PixelArea liegt, und die Methode `intersects` überprüft, ob die aufgerufene PixelArea eine andere PixelArea schneidet.

Threadnocchio stellt vier Standardklassen zur Verfügung, die das Interface PixelArea implementieren: `Point`, `Rectangle`, `Cell` und `CellArea`. Mit Hilfe der Klasse `Point` und `Rectangle` können Kollisionen von Punkten bzw. rechteckigen Gebieten mit anderen Bereichen der Bühne überprüft werden. Objekte der Klassen `Cell` bzw. `CellArea` repräsentieren einzelne Zellen bzw. rechteckige Gebiete von Zellen der Bühne, die somit auch auf Kollisionen überprüft werden können.

Die Klassen `Stage` und `Component` implementieren ebenfalls das Interface `PixelArea`. Bezüglich der Klasse `Stage` wird dabei der sichtbare Bereich als PixelArea angesehen. Bei Komponenten, d.h. Akteure und Requisiten repräsentiert das ihnen zugeordnete Ikon die PixelArea.

Durch das Konzept der PixelArea ist eine Kollisionserkennung sehr flexibel ohne eine größere Menge an Methoden realisierbar. So kann sehr einfach abgefragt werden,

ob sich bspw. zwei Akteure überlagern oder ob sich ein Akteur komplett im sichtbaren Bereich der Bühne befindet. Bei Bedarf kann ein Programmierer weitere Klassen definieren (bspw. Kreise oder Polygone), die das Interface `PixelArea` implementieren, womit sich dann ohne weitere Änderungen überprüfen ließe, ob sich bspw. ein Akteur in einem bestimmten durch ein Polygon definierten Bereich der Bühne aufhält.

6 Referenzhandbuch

Dieses Kapitel enthält die genaue Beschreibung der einzelnen Klassen und ihrer Methoden.

6.1 Stage

```
package theater;

/**
 * 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
 * Komponenten, d.h. Akteuren und Requisiten, auf der Bühne dient. Das
 * Koordinatensystem ist nicht endlich, so dass sich Akteure und Requisiten auch
 * außerhalb der Bühne befinden können, also (zwischenzeitlich) nicht sichtbar
 * sind.
 * <p>
 * </p>
 * 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".
 * <p>
 * </p>
 * Zu den Gestaltungsmethoden gehören add- und remove-Methoden zum Platzieren
 * und Entfernen von Komponenten auf bzw. von der Bühne. Weiterhin existieren
 * Methoden zum Festlegen eines Hintergrundbildes für die Bühne.
 * <p>
 * </p>
 * 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. Die Klasse Stage implementiert das Interface PixelArea und
 * kann damit unmittelbar selbst in die Kollisionserkennung mit einbezogen
 * werden.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Stage implements PixelArea {

   /**
    * Über den Konstruktor wird die Größe der Bühne festgelegt. Eine Bühne
    * besteht dabei aus einem rechteckigen Gebiet, das sich aus gleichförmigen
    * quadratischen Zellen zusammensetzt. Die Größe der Bühne kann nachträglich
    * nicht mehr geändert werden.
    *
    * @param noOfCols
    *            Anzahl der Spalten der Bühne
    * @param noOfRows
    *            Anzahl der Reihen der Bühne
    * @param cellSize
    *            Größe einer Zelle in Pixeln
    */
   public Stage(int noOfCols, int noOfRows, int cellSize)

   /**
    * Liefert die Anzahl an Spalten der Bühne.
    *
    * @return die Anzahl an Spalten der Bühne
```

```
 */
public int getNumberOfColumns()

/**
 * Liefert die Anzahl an Reihen der Bühne.
 *
 * @return die Anzahl an Reihen der Bühne
 */
public int getNumberOfRows()

/**
 * Liefert die Größe der Zellen der Bühne in Pixeln
 *
 * @return die Größe der Zellen der Bühne in Pixeln
 */
public int getCellSize()

/**
 * Platziert eine neue Komponente auf der Bühne. Angegeben werden die Spalte
 * und Reihe der Zelle, auf der die Komponente platziert werden soll. Unter
 * Platzierung ist dabei zu verstehen, dass der Mittelpunkt des Icons, das
 * die Komponente repräsentiert, auf den Mittelpunkt der Zelle gesetzt wird.
 * Da das Koordinatensystem der Bühne nicht begrenzt ist, kann eine
 * Komponente auch außerhalb der Bühne platziert werden. Wenn die Komponente
 * bereits auf der Bühne platziert ist, passiert nichts.
 * <p>
 * </p>
 * Seiteneffekt: Nach erfolgreicher Platzierung der Komponente auf der Bühne
 * wird die Methode addedToStage der Komponente mit dem entsprechenden
 * Stage-Objekt als Parameter aufgerufen.
 * <p>
 * </p>
 * Wird bei laufender Simulation die Methode add für einen Akteur
 * aufgerufen, wird anschließend automatisch dessen start-Methode
 * aufgerufen, d.h. der Thread gestartet.
 * <p>
 * </p>
 * Achtung: Jede Komponente darf maximal einer Bühne zugeordnet sein. Ist
 * bei Aufruf der Methode add die Komponente bereits einer anderen Bühne
 * zugeordnet, muss sie bei dieser Bühne zunächst mittels remove entfernt
 * werden. Ansonsten bleibt der Aufruf von add wirkungslos.
 *
 * @param comp
 *            die Komponente, die auf der Bühne platziert werden soll
 * @param col
 *            die Spalte, in der die Komponente platziert werden soll
 * @param row
 *            die Reihe, in der die Komponente platziert werden soll
 */
public void add(Component comp, int col, int row)

/**
 * Entfernt eine Komponente von der Bühne. Wenn die Komponente nicht auf der
 * Bühne platziert ist, passiert nichts.
 * <p>
 * </p>
 * Achtung: Wird die Methode für einen aktiven Akteur aufgerufen, wird
 * dieser nicht automatisch gestoppt!
 *
 * @param comp
 *            die Komponente, die von der Bühne entfernt werden soll
 */
public void remove(Component comp)

/**
 * Entfernt alle in der Liste enthaltenen Komponenten von der Bühne.
 * <p>
 * </p>
 * Achtung: Aktive Akteure der Liste werden nicht automatisch gestoppt!
 *
```

```
 * @param components
 *            enthält die Komponenten, die von der Bühne entfernt werden
 *            sollen
 */
public void remove(java.util.List<Component> components)

/**
 * Ordnet der Bühne ein Hintergrundbild zu. Erlaubt sind Bilder der Formate
 * gif, jpg und png. Das Bild wird mit der linken oberen Ecke auf die linke
 * obere Ecke der Bühne platziert. Ist das Bild größer als die Bühne, wird
 * rechts und/oder unten abgeschnitten. Ist das Bild kleiner als die Bühne,
 * wird es repliziert dargestellt, bis die komplette Bühne überdeckt ist.
 *
 * @param filename
 *            Name der Bilddatei; die Datei muss sich im Unterverzeichnis
 *            images des Theaterstück-Verzeichnisses befinden
 * @throws IllegalArgumentException
 *             wird geworfen, wenn die angegebene Datei keine gültige
 *             lesbare Bilddatei ist
 */
public void setBackground(String filename) throws IllegalArgumentException

/**
 * Ordnet der Bühne ein TheaterImage als Hintergrundbild zu. Das Bild wird
 * mit der linken oberen Ecke auf die linke obere Ecke der Bühne platziert.
 * Ist das Bild größer als die Bühne, wird rechts und/oder unten
 * abgeschnitten. Ist das Bild kleiner als die Bühne, wird es repliziert
 * dargestellt, bis die komplette Bühne überdeckt ist.
 *
 * @param image
 *            das TheaterImage, das als Hintergrundbild verwendet werden
 *            soll
 */
public void setBackground(TheaterImage image)

/**
 * Liefert das Hintergrundbild der Bühne alsTheaterImage-Objekt. Wurde kein
 * Hintergrundbild gesetzt, wird null geliefert.
 *
 * @return das Hintergrundbild der Bühne als TheaterImage-Objekt
 */
public TheaterImage getBackground()

/**
 * Zeichnet die Bühne. Normalerweise ist ein Aufruf dieser Methode nicht
 * notwendig, da die Bühne bei Änderungen automatisch aktualisiert wird.
 * Wenn allerdings mittels der Methode getAWTImage der Klasse TheaterImage
 * ein java.awt.image.BufferedImage-Objekt erfragt und geändert wird, muss
 * diese Methode aufgerufen werden, wenn die Änderungen des Image
 * unmittelbar sichtbar gemacht werden sollen.
 */
public void paint()

/**
 * Liefert eine Liste mit allen Komponenten bestimmter Klassen, die aktuell
 * auf der Bühne platziert sind. Fehlt der Parameter, werden alle
 * Bühnen-Komponenten geliefert.
 *
 * @param classes
 *            Menge von Klassen, die bei der Suche berücksichtigt werden
 *            sollen
 * @return Liste mit allen Bühnen-Komponenten bestimmter Klassen
 */
public java.util.List<Component> getComponents(Class<?>... classes)

/**
 * Liefert eine Liste mit allen Komponenten bestimmter Klassen, die aktuell
 * auf einer bestimmten Zelle der Bühne platziert sind. Werden keine
 * Klassenobjekte übergeben, so werden alle Bühnen-Komponenten auf der
 * entsprechenden Zelle geliefert.
```

```
 *
 * @param column
 *            Spalte der Zelle
 * @param row
 *            Reihe der Zelle
 * @param classes
 *            Menge von Klassen, die bei der Suche berücksichtigt werden
 *            sollen
 * @return Liste mit allen Bühnen-Komponenten bestimmter Klassen auf der
 *         angegebenen Zelle
 */
public java.util.List<Component> getComponentsAt(int column, int row,
   Class<?>... classes)

/**
 * Liefert eine Liste mit allen Komponenten bestimmter Klassen, deren
 * zugeordnetes Icon vollständig innerhalb einer bestimmten PixelArea liegt.
 * Werden keine Klassenobjekte übergeben, so werden alle Bühnen-Komponenten
 * innerhalb der PixelArea geliefert.
 *
 * @param area
 *            das Gebiet, in dem die Komponenten liegen sollen (darf nicht
 *            null sein)
 * @param classes
 *            Menge von Klassen, die bei der Suche berücksichtigt werden
 *            sollen
 * @return Liste mit allen Bühnen-Komponenten bestimmter Klassen innerhalb
 *         der angegebenen PixelArea
 */
public java.util.List<Component> getComponentsInside(PixelArea area,
   Class<?>... classes)

/**
 * Liefert eine Liste mit allen Komponenten bestimmter Klassen, deren
 * zugeordnetes Icon eine bestimmte PixelArea berührt oder schneidet. Werden
 * keine Klassenobjekte übergeben, so werden alle Bühnen-Komponenten
 * geliefert, die die PixelArea berühren oder schneiden.
 *
 * @param area
 *            das Gebiet, das die Komponenten berühren oder schneiden sollen
 *            (darf nicht null sein)
 * @param classes
 *            Menge von Klassen, die bei der Suche berücksichtigt werden
 *            sollen
 * @return Liste mit allen Bühnen-Komponenten bestimmter Klassen, die die
 *         angegebene PixelArea berühren oder schneiden
 */
public java.util.List<Component> getIntersectingComponents(PixelArea area,
   Class<?>... classes)

/**
 * Überprüft, ob der angegebene Punkt mit den Koordinaten x und y innerhalb
 * der Bühne liegt.
 *
 * @param x
 *            x-Koordinate des Punktes
 * @param y
 *            y-Koordinate des Punktes
 * @return true, falls der angegebene Punkt innerhalb der Bühne liegt
 *
 * @see theater.PixelArea#contains(int, int)
 */
public boolean contains(int x, int y)

/**
 * Überprüft, ob die Bühne vollständig innerhalb der angegebenen PixelArea
 * liegt.
 *
 * @param area
 *            das Gebiet, das überprüft werden soll (darf nicht null sein)
```

```
 * @return true, falls die Bühne vollständig innerhalb der angegebenen
 *         PixelArea liegt
 *
 * @see theater.PixelArea#isInside(theater.PixelArea)
 */
public boolean isInside(PixelArea area)

/**
 * Überprüft, ob die Bühne eine angegebene PixelArea berührt oder schneidet.
 *
 * @param area
 *            das Gebiet, das überprüft werden soll (darf nicht null sein)
 * @return true, falls die Bühne die angegebenen PixelArea berührt oder
 *         schneidet
 *
 * @see theater.PixelArea#intersects(theater.PixelArea)
 */
public boolean intersects(PixelArea area)


/**
 * Legt fest, ob die Bühne Tastatur-Ereignisse behandeln soll. Standardmäßig
 * ist dies der Fall.
 *
 * @param handlingKeyEvents
 *            true, falls die Bühne Tastatur-Ereignisse behandeln soll;
 *            false andernfalls.
 */
public void setHandlingKeyEvents(boolean handlingKeyEvents)

/**
 * Überprüft, ob die Bühne Tastatur-Ereignisse behandelt. Standardmäßig ist
 * dies der Fall.
 *
 * @return true, falls die Bühne Tastatur-Ereignisse behandelt
 */
public boolean isHandlingKeyEvents()

/**
 * Wird aufgerufen, wenn während die Bühne den Focus besitzt ein
 * keyTyped-Event eingetreten ist. Soll eine Bühne auf keyTyped-Events
 * reagieren, muss sie diese Methode entsprechend überschreiben. Genauere
 * Informationen zu keyTyped-Events finden sich in der Klasse
 * java.awt.event.KeyListener. Übergeben wird der Methode ein
 * KeyInfo-Objekt, über das Details zum eingetretenen Event abgefragt werden
 * können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingKeyEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl die Bühne als auch Komponenten können auf keyTyped-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void keyTyped(KeyInfo e)

/**
 * Wird aufgerufen, wenn während die Bühne den Focus besitzt ein
 * keyPressed-Event eingetreten ist. Soll eine Bühne auf keyPressed-Events
 * reagieren, muss sie diese Methode entsprechend überschreiben. Genauere
 * Informationen zu keyPressed-Events finden sich in der Klasse
 * java.awt.event.KeyListener. Übergeben wird der Methode ein
```

```
 * KeyInfo-Objekt, über das Details zum eingetretenen Event abgefragt werden
 * können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingKeyEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl die Bühne als auch Komponenten können auf keyPressed-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void keyPressed(KeyInfo e)

/**
 * Wird aufgerufen, wenn während die Bühne den Focus besitzt ein
 * keyReleased-Event eingetreten ist. Soll eine Bühne auf keyReleased-Events
 * reagieren, muss sie diese Methode entsprechend überschreiben. Genauere
 * Informationen zu keyReleased-Events finden sich in der Klasse
 * java.awt.event.KeyListener. Übergeben wird der Methode ein
 * KeyInfo-Objekt, über das Details zum eingetretenen Event abgefragt werden
 * können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingKeyEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl die Bühne als auch Komponenten können auf keyReleased-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void keyReleased(KeyInfo e)

/**
 * Legt fest, ob die Bühne Maus-Ereignisse behandeln soll. Standardmäßig ist
 * dies der Fall.
 *
 * @param handlingMouseEvents
 *            true, falls die Bühne Maus-Ereignisse behandeln soll; false
 *            andernfalls.
 */
public void setHandlingMouseEvents(boolean handlingMouseEvents

/**
 * Überprüft, ob die Bühne Maus-Ereignisse behandelt. Standardmäßig ist dies
 * der Fall.
 *
 * @return true, falls die Bühne Maus-Ereignisse behandelt
 */
public boolean isHandlingMouseEvents()

/**
 * Wird aufgerufen, wenn ein mousePressed-Event auf der Bühne eingetreten
 * ist, d.h. eine Maustaste gedrückt wird, während sich der Mauszeiger
 * oberhalb der Bühne befindet. Soll eine Bühne auf mousePressed-Events
 * reagieren, muss sie diese Methode entsprechend überschreiben. Genauere
 * Informationen zu mousePressed-Events finden sich in der Klasse
```

```
 * java.awt.event.MouseListener. Übergeben wird der Methode ein
 * MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
 * werden können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl die Bühne als auch Komponenten können auf mousePressed-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void mousePressed(MouseInfo e)

/**
 * Wird aufgerufen, wenn ein mouseReleased-Event auf der Bühne eingetreten
 * ist, d.h. eine gedrückte Maustaste losgelassen wird, während sich der
 * Mauszeiger oberhalb der Bühne befindet. Soll eine Bühne auf
 * mouseReleased-Events reagieren, muss sie diese Methode entsprechend
 * überschreiben. Genauere Informationen zu mouseReleased-Events finden sich
 * in der Klasse java.awt.event.MouseListener. Übergeben wird der Methode
 * ein MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
 * werden können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl die Bühne als auch Komponenten können auf mouseReleased-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void mouseReleased(MouseInfo e)

/**
 * Wird aufgerufen, wenn ein mouseClicked-Event auf der Bühne eingetreten
 * ist, d.h. eine Maustaste geklickt wurde, während sich der Mauszeiger
 * oberhalb der Bühne befindet. Soll eine Bühne auf mouseClicked-Events
 * reagieren, muss sie diese Methode entsprechend überschreiben. Genauere
 * Informationen zu mouseClicked-Events finden sich in der Klasse
 * java.awt.event.MouseListener. Übergeben wird der Methode ein
 * MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
 * werden können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl die Bühne als auch Komponenten können auf mouseClicked-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
```

```
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseClicked(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseDragged-Event auf der Bühne eingetreten
     * ist, d.h. die Maus bei gedrückter Maustaste bewegt wurde, während sich
     * der Mauszeiger oberhalb der Bühne befindet. Soll eine Bühne auf
     * mouseDragged-Events reagieren, muss sie diese Methode entsprechend
     * überschreiben. Genauere Informationen zu mouseDragged-Events finden sich
     * in der Klasse java.awt.event.MouseMotionListener. Übergeben wird der
     * Methode ein MouseInfo-Objekt, über das Details zum eingetretenen Event
     * abgefragt werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl die Bühne als auch Komponenten können auf mouseDragged-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseDragged(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseMoved-Event auf der Bühne eingetreten ist,
     * d.h. die Maus bewegt wurde, während sich der Mauszeiger oberhalb der
     * Bühne befindet. Soll eine Bühne auf mouseMoved-Events reagieren, muss sie
     * diese Methode entsprechend überschreiben. Genauere Informationen zu
     * mouseMoved-Events finden sich in der Klasse
     * java.awt.event.MouseMotionListener. Übergeben wird der Methode ein
     * MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
     * werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl die Bühne als auch Komponenten können auf mouseMoved-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseMoved(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseEntered-Event auf der Bühne eingetreten
     * ist, d.h. der Mauszeiger auf die Bühne gezogen wird. Soll eine Bühne auf
     * mouseEntered-Events reagieren, muss sie diese Methode entsprechend
     * überschreiben. Genauere Informationen zu mouseEntered-Events finden sich
     * in der Klasse java.awt.event.MouseListener. Übergeben wird der Methode
     * ein MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
     * werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
```

```
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl die Bühne als auch Komponenten können auf mouseEntered-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *             enthält Details zum eingetretenen Event
     */
    public void mouseEntered(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseExited-Event auf der Bühne eingetreten
     * ist, d.h. der Mauszeiger die Bühne verlässt. Soll eine Bühne auf
     * mouseExited-Events reagieren, muss sie diese Methode entsprechend
     * überschreiben. Genauere Informationen zu mouseExited-Events finden sich
     * in der Klasse java.awt.event.MouseListener. Übergeben wird der Methode
     * ein MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
     * werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl die Bühne als auch Komponenten können auf mouseExited-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *             enthält Details zum eingetretenen Event
     */
    public void mouseExited(MouseInfo e)

}
```

6.2 Component, Actor, Prop

6.2.1 Component

```
package theater;

/**
 * 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.
 * <p>
 * </p>
 * 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 ist eine Menge von
 * Getter-Methoden definiert, um zum einen ihren Zustand abfragen und zum
 * anderen das Bühnen-Objekt ermitteln zu können, dem sie u.U. zugeordnet sind.
 * <p>
 * </p>
```

```
 * 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. Die Klasse Component implementiert das Interface
 * PixelArea, so dass Akteure und Requisiten unmittelbar selbst in die
 * Kollisionserkennung mit einbezogen werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Component extends Thread implements PixelArea {

   /**
    * Standardmäßiger Wert der Z-Koordinate
    */
   public final static int DEF_Z_COORDINATE = 0;

   /**
    * Konstruktor, der eine neue Komponente als Akteur oder Requisite
    * initialisiert.
    *
    * @param isActor
    *            true, falss es sich um einen Akteur handelt; false, falls es
    *            sich um eine Requisite handelt
    */
   protected Component(boolean isActor)

   /**
    * Muss überschrieben werden und die Akteur-spezifischen Aktionen enthalten
    */
   public void run()

   /**
    * Liefert das Objekt der Bühne, auf dem sich die Komponente gerade befindet
    * oder null, falls sich die Komponente aktuell auf keiner Bühne befindet
    *
    * @return die Bühne, auf der sich die Komponente gerade befindet
    */
   public Stage getStage()

   /**
    * Ordnet der Komponente ein Icon zu, durch das sie auf der Bühne
    * repräsentiert wird. Erlaubt sind Bilder der Formate gif, jpg und png.
    *
    * @param filename
    *            Name der Bilddatei; die Datei muss sich im Unterverzeichnis
    *            images des Theaterstück-Verzeichnisses befinden
    * @throws IllegalArgumentException
    *             wird geworfen, wenn die angegebene Datei keine gültige
    *             lesbare Bilddatei ist
    */
   public void setImage(String filename) throws IllegalArgumentException

   /**
    * Ordnet der Komponente ein TheaterImage als Icon zu, durch das sie auf der
    * Bühne repräsentiert wird.
    *
    * @param image
    *            das TheaterImage, das als Icon verwendet werden soll
    */
   public void setImage(TheaterImage image)

   /**
    * Liefert das Icon der Komponente als TheaterImage-Objekt. Wurde kein Icon
    * zugeordnet, wird null geliefert.
    *
    * @return das Icon der Komponente als TheaterImage-Objekt
    */
   public TheaterImage getImage()
```

```
/**
 * Mit Hilfe der add-Methode der Klasse Stage kann eine Komponente auf einer
 * Bühne platziert werden. Nach der erfolgreichen Platzierung wird diese
 * Methode addedToStage für die Komponente aufgerufen. Als Parameter wird
 * dabei das Bühnenobjekt übergeben. Sollen für eine Komponente bestimmte
 * Aktionen ausgeführt werden, sobald sie einer Bühne zugeordnet wird, muss
 * die Methode entsprechend überschrieben werden.
 *
 * @param stage
 *            das Objekt, das die Bühne repräsentiert, auf die die
 *            Komponente platziert wurde
 */
public void addedToStage(Stage stage)

/**
 * Mit Hilfe der add-Methode der Klasse Stage können Komponente in einer
 * bestimmten Spalte und Reihe auf einer Bühne platziert werden. Die Methode
 * setLocation ermöglicht die Umplatzierung der Komponente auf der Bühne.
 *
 * @param newCol
 *            die Spalte, in die die Komponente umplatziert werden soll
 * @param newRow
 *            die Reihe, in die die Komponente umplatziert werden soll
 * @throws IllegalStateException
 *             wird geworfen, wenn die Komponente aktuell nicht auf einer
 *             Bühne platziert ist
 */
public void setLocation(int newCol, int newRow)
   throws IllegalStateException

/**
 * Liefert die Spalte, in der sich die Komponente aktuell auf einer Bühne
 * befindet.
 *
 * @return die Spalte, in der sich die Komponente aktuell auf einer Bühne
 *         befindet
 * @throws IllegalStateException
 *             wird geworfen, wenn die Komponente aktuell nicht auf einer
 *             Bühne platziert ist
 */
public int getColumn() throws IllegalStateException

/**
 * Liefert die Reihe, in der sich die Komponente aktuell auf einer Bühne
 * befindet.
 *
 * @return die Reihe, in der sich die Komponente aktuell auf einer Bühne
 *         befindet
 * @throws IllegalStateException
 *             wird geworfen, wenn die Komponente aktuell nicht auf einer
 *             Bühne platziert ist
 */
public int getRow() throws IllegalStateException

/**
 * Liefert die Zelle, in der sich die Komponente aktuell auf einer Bühne
 * befindet.
 *
 * @return die Zelle, in der sich die Komponente aktuell auf der Bühne
 *         befindet
 * @throws IllegalStateException
 *             wird geworfen, wenn die Komponente aktuell nicht auf einer
 *             Bühne platziert ist
 */
public Cell getCell() throws IllegalStateException

/**
 * Ändert die z-Koordinate der Komponente, mit der die Zeichenreihenfolge
 * von Komponenten beeinflusst werden kann. Je höher der Wert der
```

```
 * z-Koordinate einer Komponente ist, umso weiter gelangt das Icon der
 * Komponente auf der Bühne in den Vordergrund. Die Zeichenreihenfolge von
 * Komponenten mit gleicher z-Koordinate ist undefiniert. Standardmäßig hat
 * die z-Koordinate einer Komponente den Wert DEF_Z_COORDINATE.
 *
 * @param newZ
 *            die neue z-Koordinate der Komponente
 */
public void setZCoordinate(int newZ)

/**
 * Liefert die aktuelle z-Koordinate der Komponente.
 *
 * @return die aktuelle z-Koordinate der Komponente
 */
public int getZCoordinate()

/**
 * Legt den Rotationswinkel fest, mit der das Icon der Komponente gezeichnet
 * werden soll. Die Drehung erfolgt im Uhrzeigersinn. Standardmäßig beträgt
 * der Rotationswinkel 0.
 * <p>
 * </p>
 * Achtung: Der Rotationswinkel hat keinen Einfluss auf die Weite und Höhe
 * eines einer Komponente zugeordneten Icons. Diese werden immer auf der
 * Grundlage eines Rotationswinkels von 0 berechnet.
 *
 * @param rotation
 *            der neue Rotationswinkel der Komponente
 */
public void setRotation(int rotation)

/**
 * Liefert den aktuellen Rotationswinkel der Komponente.
 *
 * @return der aktuelle Rotationswinkel der Komponente
 */
public int getRotation()

/**
 * Über diese Methode können Komponenten sichtbar bzw. unsichtbar gemacht
 * werden. In möglichen Kollisionsabfragen werden allerdings auch
 * unsichtbare Komponenten mit einbezogen.
 *
 * @param visible
 *            falls true, wird die Komponente sichtbar; falls false, wird
 *            sie unsichtbar
 */
public void setVisible(boolean visible)

/**
 * Liefert die Sichtbarkeit der Komponente.
 *
 * @return true, falls die Komponente sichtbar ist; ansonsten false
 */
public boolean isVisible()

/**
 * Überprüft, ob der angegebene Punkt mit den Koordinaten x und y innerhalb
 * des Icons der Komponente liegt.
 *
 * @param x
 *            x-Koordinate des Punktes
 * @param y
 *            y-Koordinate des Punktes
 * @return true, falls der angegebene Punkt innerhalb des Icons der
 *         Komponente liegt
 * @throws IllegalStateException
 *             wird geworfen, wenn die Komponente aktuell nicht auf einer
 *             Bühne platziert ist
```

```
 *
 * @see theater.PixelArea#contains(int, int)
 */
public boolean contains(int x, int y) throws IllegalStateException

/**
 * Überprüft, ob das Icon der Komponente vollständig innerhalb der
 * angegebenen PixelArea liegt.
 *
 * @param area
 *            das Gebiet, das überprüft werden soll (darf nicht null sein)
 * @return true, falls das Icon der Komponente vollständig innerhalb der
 *         angegebenen PixelArea liegt
 * @throws IllegalStateException
 *             wird geworfen, wenn die Komponente aktuell nicht auf einer
 *             Bühne platziert ist
 *
 * @see theater.PixelArea#isInside(theater.PixelArea)
 */
public boolean isInside(PixelArea area) throws IllegalStateException


/**
 * Überprüft, ob das Icon der Komponente eine angegebene PixelArea
 * schneidet.
 *
 * @param area
 *            das Gebiet, das überprüft werden soll (darf nicht null sein)
 * @return true, falls das Icon der Komponente die angegebenen PixelArea
 *         schneidet
 * @throws IllegalStateException
 *             wird geworfen, wenn die Komponente aktuell nicht auf einer
 *             Bühne platziert ist
 *
 * @see theater.PixelArea#intersects(theater.PixelArea)
 */
public boolean intersects(PixelArea area) throws IllegalStateException

/**
 * Legt fest, ob die Komponente Tastatur-Ereignisse behandeln soll.
 * Standardmäßig ist dies der Fall.
 *
 * @param handlingKeyEvents
 *            true, falls die Komponente Tastatur-Ereignisse behandeln soll;
 *            false andernfalls.
 */
public void setHandlingKeyEvents(boolean handlingKeyEvents)

/**
 * Überprüft, ob die Komponente Tastatur-Ereignisse behandelt. Standardmäßig
 * ist dies der Fall.
 *
 * @return true, falls die Komponente Tastatur-Ereignisse behandelt
 */
public boolean isHandlingKeyEvents()

/**
 * Wird aufgerufen, wenn während die Bühne den Focus besitzt ein
 * keyTyped-Event eingetreten ist. Soll eine Komponente auf keyTyped-Events
 * reagieren, muss sie diese Methode entsprechend überschreiben. Genauere
 * Informationen zu keyTyped-Events finden sich in der Klasse
 * java.awt.event.KeyListener. Übergeben wird der Methode ein
 * KeyInfo-Objekt, über das Details zum eingetretenen Event abgefragt werden
 * können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingKeyEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
```

```
 * Sowohl Komponenten als auch die Bühne können auf keyTyped-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void keyTyped(KeyInfo e)

/**
 * Wird aufgerufen, wenn während die Bühne den Focus besitzt ein
 * keyPressed-Event eingetreten ist. Soll eine Komponente auf
 * keyPressed-Events reagieren, muss sie diese Methode entsprechend
 * überschreiben. Genauere Informationen zu keyPressed-Events finden sich in
 * der Klasse java.awt.event.KeyListener. Übergeben wird der Methode ein
 * KeyInfo-Objekt, über das Details zum eingetretenen Event abgefragt werden
 * können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingKeyEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl Komponenten als auch die Bühne können auf keyTyped-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void keyPressed(KeyInfo e)

/**
 * Wird aufgerufen, wenn während die Bühne den Focus besitzt ein
 * keyReleased-Event eingetreten ist. Soll eine Komponente auf
 * keyReleased-Events reagieren, muss sie diese Methode entsprechend
 * überschreiben. Genauere Informationen zu keyReleased-Events finden sich
 * in der Klasse java.awt.event.KeyListener. Übergeben wird der Methode ein
 * KeyInfo-Objekt, über das Details zum eingetretenen Event abgefragt werden
 * können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingKeyEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl Komponenten als auch die Bühne können auf keyTyped-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void keyReleased(KeyInfo e)

/**
 * Legt fest, ob die Komponente Maus-Ereignisse behandeln soll.
 * Standardmäßig ist dies der Fall.
 *
 * @param handlingMouseEvents
 *            true, falls die Komponente Maus-Ereignisse behandeln soll;
```

```
 *            false andernfalls.
 */
public void setHandlingMouseEvents(boolean handlingMouseEvents)

/**
 * Überprüft, ob die Komponente Maus-Ereignisse behandelt. Standardmäßig ist
 * dies der Fall.
 *
 * @return true, falls die Komponente Maus-Ereignisse behandelt
 */
public boolean isHandlingMouseEvents()

/**
 * Wird aufgerufen, wenn ein mousePressed-Event auf der Komponente
 * eingetreten ist, d.h. eine Maustaste gedrückt wird, während sich der
 * Mauszeiger oberhalb des Icons der Komponente befindet. Soll eine
 * Komponente auf mousePressed-Events reagieren, muss sie diese Methode
 * entsprechend überschreiben. Genauere Informationen zu mousePressed-Events
 * finden sich in der Klasse java.awt.event.MouseListener. Übergeben wird
 * der Methode ein MouseInfo-Objekt, über das Details zum eingetretenen
 * Event abgefragt werden können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl Komponenten als auch die Bühne können auf keyTyped-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void mousePressed(MouseInfo e)

/**
 * Wird aufgerufen, wenn ein mouseReleased-Event auf der Bühne eingetreten
 * ist, d.h. eine gedrückte Maustaste losgelassen wird, während sich der
 * Mauszeiger über dem Icon der Komponente befindet. Soll eine Komponente
 * auf mouseReleased-Events reagieren, muss sie diese Methode entsprechend
 * überschreiben. Genauere Informationen zu mouseReleased-Events finden sich
 * in der Klasse java.awt.event.MouseListener. Übergeben wird der Methode
 * ein MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
 * werden können.
 * <p>
 * </p>
 * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * <p>
 * </p>
 * Sowohl Komponenten als auch die Bühne können auf mouseReleased-Events
 * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
 * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
 * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
 * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
 * Events, kann die Benachrichtigungssequenz abgebrochen werden.
 *
 * @param e
 *            enthält Details zum eingetretenen Event
 */
public void mouseReleased(MouseInfo e)

/**
 * Wird aufgerufen, wenn ein mouseClicked-Event auf der Komponente
 * eingetreten ist, d.h. eine Maustaste geklickt wurde, während sich der
 * Mauszeiger auf dem Icons der Komponente befindet. Soll eine Komponente
```

```
     * auf mouseClicked-Events reagieren, muss sie diese Methode entsprechend
     * überschreiben. Genauere Informationen zu mouseClicked-Events finden sich
     * in der Klasse java.awt.event.MouseListener. Übergeben wird der Methode
     * ein MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
     * werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl Komponenten als auch die Bühne können auf mouseClicked-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseClicked(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseDragged-Event auf der Komponente
     * eingetreten ist, d.h. die Maus bei gedrückter Maustaste bewegt wurde,
     * während sich der Mauszeiger auf dem Icon der Komponente befindet. Soll
     * eine Komponente auf mouseDragged-Events reagieren, muss sie diese Methode
     * entsprechend überschreiben. Genauere Informationen zu mouseDragged-Events
     * finden sich in der Klasse java.awt.event.MouseMotionListener. Übergeben
     * wird der Methode ein MouseInfo-Objekt, über das Details zum eingetretenen
     * Event abgefragt werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl Komponenten als auch die Bühne können auf mouseDragged-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseDragged(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseMoved-Event auf der Komponente eingetreten
     * ist, d.h. die Maus bewegt wurde, während sich der Mauszeiger auf dem Icon
     * der Komponente befindet. Soll eine Komponente auf mouseMoved-Events
     * reagieren, muss sie diese Methode entsprechend überschreiben. Genauere
     * Informationen zu mouseMoved-Events finden sich in der Klasse
     * java.awt.event.MouseMotionListener. Übergeben wird der Methode ein
     * MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
     * werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl Komponenten als auch die Bühne können auf mouseMoved-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
```

```
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseMoved(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseEntered-Event auf der Komponente
     * eingetreten ist, d.h. der Mauszeiger auf das Icon der Komponente gezogen
     * wird. Soll eine Komponente auf mouseEntered-Events reagieren, muss sie
     * diese Methode entsprechend überschreiben. Genauere Informationen zu
     * mouseEntered-Events finden sich in der Klasse
     * java.awt.event.MouseListener. Übergeben wird der Methode ein
     * MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
     * werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl Komponenten als auch die Bühne können auf mouseEntered-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseEntered(MouseInfo e)

    /**
     * Wird aufgerufen, wenn ein mouseExited-Event auf der Komponente
     * eingetreten ist, d.h. der Mauszeiger das Icon der Komponente verlässt.
     * Soll eine Komponente auf mouseExited-Events reagieren, muss sie diese
     * Methode entsprechend überschreiben. Genauere Informationen zu
     * mouseExited-Events finden sich in der Klasse
     * java.awt.event.MouseListener. Übergeben wird der Methode ein
     * MouseInfo-Objekt, über das Details zum eingetretenen Event abgefragt
     * werden können.
     * <p>
     * </p>
     * Die Methode wird nur aufgerufen, wenn der HandlingMouseEvents-Status
     * gesetzt ist (was standardmäßig der Fall ist).
     * <p>
     * </p>
     * Sowohl Komponenten als auch die Bühne können auf mouseExited-Events
     * reagieren. Die Reihenfolge, in der eine Benachrichtigung erfolgt, richtet
     * sich nach der Zeichenreihenfolge der Komponenten: Je weiter eine
     * Komponente im Vordergrund ist, desto früher wird sie benachrichtigt. Die
     * Bühne wird nach allen Komponenten benachrichtigt. Durch Konsumieren des
     * Events, kann die Benachrichtigungssequenz abgebrochen werden.
     *
     * @param e
     *            enthält Details zum eingetretenen Event
     */
    public void mouseExited(MouseInfo e)

}
```

6.2.2 Actor

```
package theater;

/**
```

```
 * Die Klasse Actor ist die Basisklasse aller Akteure. Sie erbt als Unterklasse
 * der Klasse Component, die wiederum Unterklasse der Java-Klasse Thread ist,
 * alle Methoden dieser beiden Klassen.
 * <p>
 * </p>
 * Soll ein neuer Akteur definiert werden, muss eine entsprechende Klasse von
 * der Klasse Actor abgeleitet und die Methode "public void run()" überschrieben
 * werden. In dieser wird das Eigenleben von Akteuren der Klasse festgelegt,
 * wobei u. a. die geerbten Methoden der Klassen Component und Thread genutzt
 * werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Actor extends Component {

   /**
    * Default-Konstruktor der Klasse Actor
    */
   public Actor()

   /**
    * Muss überschrieben werden und die Actor-spezifischen Aktionen enthalten
    */
   public void run()

}
```

6.2.3 Prop

```
package theater;

/**
 * Die Klasse Prop ist die Basisklasse aller Requisiten. Sie ist Unterklasse der
 * Klasse Component und erbt alle deren Methoden.
 * <p>
 * </p>
 * Soll eine neue Requisite definiert werden, muss eine entsprechende Klasse von
 * der Klasse Prop abgeleitet werden. Zum Umgang mit den Requisiten können die
 * geerbten Methoden der Klassen Component genutzt werden.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Prop extends Component {

   /**
    * Default-Konstruktor der Klasse Prop
    */
   public Prop()

   /**
    * Überschreibt die Component-Methode run als leere Methode und definiert
    * sie als final. Damit wird verhindert, das die Requisiten als passive
    * Objekte selbstständig agieren können.
    *
    * @see theater.Component#run()
    */
   public final void run()

}
```

6.3 Performance

```
package theater;
```

```
/**
 * Die Klasse Performance definiert Methoden zur Steuerung und Verwaltung der
 * Ausführung von Threadnocchio-Programmen(stop, suspend, setSpeed, freeze,
 * unfreeze).
 * <p>
 * </p>
 * Weiterhin werden Methoden definiert, die unmittelbar nach entsprechenden
 * Steuerungsaktionen aufgerufen werden, und zwar auch, wenn die Aktionen durch
 * die Steuerungsbuttons des Simulators ausgelöst wurden(started, stopped,
 * suspended, resumed, speedChanged). Möchte ein Programmierer zusätzliche
 * Aktionen mit den entsprechenden Steuerungsaktionen einhergehen lassen, kann
 * er eine Unterklasse der Klasse Performance definieren und hierin die
 * entsprechende Methode überschreiben.
 * <p>
 * </p>
 * Zusätzlich stellt die Klasse Performance eine Methode playSound zur
 * Verfügung, mit der eine Audio-Datei abgespielt werden kann. Die Datei muss
 * sich im Unterverzeichnis "sounds" des entsprechenden Theraterstücks befinden.
 * Unterstützt werden die Formate wav, au und aiff.
 * <p>
 * </p>
 * Über die Methode setActiveStage ist es möglich, die aktuell aktive Bühne
 * gegen eine andere Bühne auszutauschen, d.h. das Bühnenbild zu wechseln. Unter
 * Umständen aktive Akteure der alten Bühne werden dabei nicht automatisch
 * gestoppt. Die Methode getActiveStage liefert die gerade aktive Bühne.
 * <p>
 * </p>
 * Das während einer Aufführung aktuelle Performance-Objekt kann mit Hilfe der
 * statischen Methode getPerformance ermittelt werden. Ein Wechsel des
 * Performance-Objektes ist während einer Aufführung nicht möglich.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Performance {

   /**
    * Minimale Geschwindigkeit
    */
   public static int MIN_SPEED = 0;

   /**
    * Maximale Geschwindigkeit
    */
   public static int MAX_SPEED = 100;

   /**
    * Default-Geschwindigkeit
    */
   public static int DEF_SPEED = 50;

   /**
    * Default-Konstruktor zur Initialisierung eines Performance-Objektes. Der
    * Zustand des Performance-Objektes wird auf STOPPED gesetzt, die
    * Geschwindigkeit auf DEF_SPEED.
    */
   public Performance()

   /**
    * Liefert das aktuell gesetzte Performance-Objekte
    *
    * @return das aktuell gesetzte Performance-Objekt
    */
   public static Performance getPerformance()

   /**
    * Stopped eine Performance und setzt sie in den Zustand STOPPED. Wenn sich
    * die Performance bereits im Zustand STOPPED befindet, passiert nichts.
```

```
 * <p>
 * </p>
 * Die Methode kann aus einem Programm heraus aufgerufen werden. Sie wird
 * auch aufgerufen, wenn der Nutzer im Theater-Simulator den Stopp-Button
 * anklickt.
 */
public void stop()

/**
 * Pausiert eine Performance und setzt sie in den Zustand PAUSED. Die
 * Ausführung dieser Methode bewirkt nur dann etwas, wenn sich die
 * Performance im Zustand RUNNING befindet.
 * <p>
 * </p>
 * Die Methode kann aus einem Programm heraus aufgerufen werden. Sie wird
 * auch aufgerufen, wenn der Nutzer im Threadnocchio-Simulator den
 * Pause-Button anklickt.
 */
public void suspend()

/**
 * Diese Methode wird unmittelbar nach dem Starten eines Theaterstücks
 * aufgerufen.
 * <p>
 * </p>
 * Sollen mit dem Start eines Theaterstücks weitere Aktionen einhergehen,
 * muss der Programmierer eine Klasse von der Klasse Performance ableiten
 * und diese Methode entsprechend überschreiben.
 */
public void started()

/**
 * Diese Methode wird unmittelbar nach dem Stoppen eines Theaterstücks
 * aufgerufen.
 * <p>
 * </p>
 * Sollen mit dem Stoppen eines Theater-Programms weitere Aktionen
 * einhergehen, muss der Programmierer eine Klasse von der Klasse
 * Performance ableiten und diese Methode entsprechend überschreiben.
 */
public void stopped()

/**
 * Diese Methode wird unmittelbar nach dem Anhalten/Pausieren eines
 * Theaterstücks aufgerufen.
 * <p>
 * </p>
 * Sollen mit dem Anhalten eines Theater-Programms weitere Aktionen
 * einhergehen, muss der Programmierer eine Klasse von der Klasse
 * Performance ableiten und diese Methode entsprechend überschreiben.
 */
public void suspended()

/**
 * Diese Methode wird unmittelbar nach dem Fortsetzen eines angehaltenen
 * Theaterstücks aufgerufen.
 * <p>
 * </p>
 * Sollen mit dem Fortsetzen eines Theater-Programms weitere Aktionen
 * einhergehen, muss der Programmierer eine Klasse von der Klasse
 * Performance ableiten und diese Methode entsprechend überschreiben.
 */
public void resumed()

/**
 * Diese Methode wird aufgerufen, wenn sich die Geschwindigkeit des
 * Theaterstücks ändert.
 * <p>
 * </p>
 * Sollen mit einer Geschwindigkeitsänderung weitere Aktionen einhergehen,
```

```
 * muss der Programmierer eine Klasse von der Klasse Performance ableiten
 * und diese Methode entsprechend überschreiben.
 *
 * @param newSpeed
 *            die neue Geschwindigkeit
 */
public void speedChanged(int newSpeed)

/**
 * Ändert die aktuell eingestellte Ausführungsgeschwindigkeit. Die minimale
 * Geschwindigkeit kann über die Konstante MIN_SPEED, die maximale
 * Geschwindigkeit über die Konstante MAX_SPEED abgefragt werden. Wird als
 * gewünschte Geschwindigkeit ein kleinerer bzw. größerer Wert als die
 * entsprechende Konstante übergeben, wird die Geschwindigkeit automatisch
 * auf MIN_SPEED bzw. MAX-SPEED gesetzt.
 * <p>
 * </p>
 * Die Methode kann aus einem Programm heraus aufgerufen werden. Sie wird
 * auch aufgerufen, wenn der Nutzer im Theater-Simulator den
 * Geschwindigkeitsregler benutzt.
 *
 * @param newSpeed
 *            die neue Ausführungsgeschwindigkeit, die zwischen MIN_SPEED
 *            und MAX_SPEED liegen sollte
 */
public void setSpeed(int newSpeed)

/**
 * Liefert die aktuelle Ausführungsgeschwindigkeit der Performance.
 *
 * @return die aktuelle Ausführungsgeschwindigkeit der Performance
 */
public int getSpeed()

/**
 * Spielt einen Sound ab, der aus einer Datei geladen wird. Erlaubt sind die
 * Formate au, aiff und wav.
 *
 * @param soundFile
 *            Name der Sounddatei; die Datei muss sich im Unterverzeichnis
 *            sounds des Theaterstück-Verzeichnisses befinden
 * @throws IllegalArgumentException
 *             wird geworfen, wenn die angegebene Datei keine gültige
 *             lesbare Sounddatei ist
 */
public void playSound(String soundFile) throws IllegalArgumentException

/**
 * Der Aufruf dieser Methode führt dazu, dass die Ansicht der Bühne
 * "eingeforen" wird, d.h. es werden keinerlei Zustandsänderungen mehr
 * sichtbar, bevor nicht die Methode unfreeze aufgerufen worden ist.
 */
public void freeze()

/**
 * Bei Aufruf dieser Methode wird der Eingefroren-Zustand wieder verlassen.
 * Befindet sich der aufrufene Thread nicht im Eingefroren-Zustand, bewirkt
 * ein Aufruf dieser Methode nichts.
 */
public void unfreeze()

/**
 * Überprüft ob sich das Theaterstück im Eingefroren-Zustand befindet
 *
 * @return true, falls sich das Theaterstück im Eingefroren-Zustand
 *         befindet; false, sonst
 */
public boolean isFrozen()

/**
```

```
    * Wechselt die Bühne. Wird ein Bühnenwechsel während der Aufführung eines
    * Theaterstücks angestoßen, werden alle Akteure der neuen Bühne automatisch
    * gestartet, insofern sie noch nicht gestartet sind. Die Akteure der alten
    * Bühne laufen (im Hintergrund) weiter.
    *
    * @param stage
    *            die neue Bühne
    */
   public void setActiveStage(Stage stage)

   /**
    * Liefert die aktuell dargestellte Bühne
    *
    * @return die aktuell dargestellte Bühne
    */
   public Stage getActiveStage()

}
```

6.4 TheaterImage

```
package theater;

/**
 * TheaterImage ist eine Theater-Klasse mit vielfältigen Methoden zum Erzeugen
 * und Manipulieren von Bildern bzw. Ikons, 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.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class TheaterImage {

   /**
    * Konstruktor zur Initialisierung eines TheaterImages mit dem Default-Image
    * der entsprechenden Klasse:
    * <ul>
    * <li>Stage: Bühne mit Vorhang</li>
    * <li>Actor: Marionette</li>
    * <li>Prop: Sessel</li>
    * <li>Performance: Flagge</li>
    * <li>ansonsten: Werkzeuge</li>
    * </ul>
    *
    * Die Bilder werden im Dateibereich des Theater-Simulators benutzt. Die
    * Default-Images werden auch benutzt, wenn eine Actor- bzw. Prop-Klasse
    * ihren Objekten keine eigenen Ikons zuordnet.
    *
    * @param father
    *            Klassenobjekt der entsprechenden Klasse (Stage.class,
    *            Actor.class, Prop.class, Performance.class)
    */
   public TheaterImage(Class<?> father)

   /**
    * Konstruktor zum Initialisieren eines TheaterImage mit einem Bild aus
    * einer Datei. Erlaubte Bildformate sind gif, jpq und png.
    *
    * @param filename
    *            Name der Bilddatei; die Datei muss sich im Unterverzeichnis
    *            "images" des Theaterstücks befinden
    * @throws IllegalArgumentException
    *             wird geworfen, wenn die Datei nicht existiert, keine gültige
    *             Bilddatei ist oder nicht lesbar ist
    */
```

```
public TheaterImage(String filename) throws IllegalArgumentException

/**
 * Konstruktor zum Erzeugen eines leeren TheaterImages in einer bestimmten
 * Größe
 *
 * @param width
 *            Breite des Bildes in Pixeln (> 0)
 * @param height
 *            Höhe des Bildes in Pixeln (> 0)
 */
public TheaterImage(int width, int height)

/**
 * Copykonstruktor zum Initialisieren eines TheaterImages mit einem bereits
 * existierenden TheaterImage
 *
 * @param im
 *            ein bereits existierendes TheaterImage (darf nicht null sein)
 */
public TheaterImage(TheaterImage im)

/**
 * Liefert die Breite des TheaterImages in Pixeln.
 *
 * @return die Breite des TheaterImages in Pixeln
 */
public int getWidth()

/**
 * Liefert die Höhe des TheaterImages in Pixeln.
 *
 * @return die Höhe des TheaterImages in Pixeln
 */
public int getHeight()

/**
 * Ordnet dem TheaterImage eine Farbe zu, in der bei Aufruf der draw- bzw.
 * fill-Methoden die entsprechenden Graphik-Primitiven gezeichnet werden.
 *
 * @param color
 *            die neue Zeichenfarbe
 */
public void setColor(java.awt.Color color)

/**
 * Liefert die aktuelle Zeichenfarbe des TheaterImages.
 *
 * @return die aktuelle Zeichenfarbe des TheaterImages
 */
public java.awt.Color getColor()

/**
 * Setzt den Font, in dem Texte durch nachfolgende Aufrufe der
 * drawString-Methode in dem TheaterImage gezeichnet werden sollen.
 *
 * @param f
 *            der neue Font
 */
public void setFont(java.awt.Font f)

/**
 * Liefert den aktuellen Font des TheaterImages.
 *
 * @return der aktuelle Font des TheaterImages
 */
public java.awt.Font getFont()


/**
```

```
 * Zeichnet im TheaterImage eine Linie in der aktuellen Zeichenfarbe.
 *
 * @param x1
 *            x-Koordinate, von der aus die Linie gezeichnet werden soll
 * @param y1
 *            y-Koordinate, von der aus die Linie gezeichnet werden soll
 * @param x2
 *            x-Koordinate, bis wohin die Linie gezeichnet werden soll
 * @param y2
 *            y-Koordinate, bis wohin die Linie gezeichnet werden soll
 */
public void drawLine(int x1, int y1, int x2, int y2)

/**
 * Zeichnet im TheaterImage ein Rechteck in der aktuellen Zeichenfarbe.
 *
 * @param x
 *            x-Koordinate der linken oberen Ecke des Rechtecks
 * @param y
 *            y-Koordinate der linken oberen Ecke des Rechtecks
 * @param width
 *            Breite des Rechtecks (in Pixeln)
 * @param height
 *            Höhe des Rechtecks (in Pixeln)
 */
public void drawRect(int x, int y, int width, int height)

/**
 * Zeichnet im TheaterImage ein Oval in der aktuellen Zeichenfarbe.
 *
 * @param x
 *            x-Koordinate der linken oberen Ecke des Ovals
 * @param y
 *            y-Koordinate der linken oberen Ecke des Ovals
 * @param width
 *            Breite des Ovals in Pixeln
 * @param height
 *            Höhe des Ovals in Pixeln
 */
public void drawOval(int x, int y, int width, int height)

/**
 * Zeichnet im TheaterImage ein Polygon in der aktuellen Zeichenfarbe. Es
 * wird automatisch ein Linie hinzugefügt, die das Polygon schließt.
 *
 * @param xPoints
 *            x-Koordinaten der Linien
 * @param yPoints
 *            y-Koordinaten der Linien
 * @param nPoints
 *            Anzahl der Liniensegmente
 */
public void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)

/**
 * Zeichnet im TheaterImage einen Text im aktuell gesetzten Font.
 *
 * @param string
 *            der zu zeichnende Text
 * @param x
 *            x-Koordinate, an der der Text beginnen soll
 * @param y
 *            y-Koordinate, an der der Text beginnen soll
 */
public void drawString(String string, int x, int y)

/**
 * Zeichnet ein existierendes TheaterImage an einer bestimmten Stelle in das
 * aufgerufene TheaterImage
 *
```

```
 * @param image
 *            das TheaterImage, das gezeichnet werden soll (darf nicht null
 *            sein)
 * @param x
 *            x-Koordinate, an der das Image gezeichnet werden soll
 * @param y
 *            y-Koordinate, an der das Image gezeichnet werden soll
 */
public void drawImage(TheaterImage image, int x, int y)

/**
 * Füllt das gesamte TheaterImage in der aktuellen Zeichenfarbe.
 */
public void fill()

/**
 * Zeichnet im TheaterImage ein gefülltes Rechteck in der aktuellen
 * Zeichenfarbe.
 *
 * @param x
 *            x-Koordinate der linken oberen Ecke des Rechtecks
 * @param y
 *            y-Koordinate der linken oberen Ecke des Rechtecks
 * @param width
 *            Breite des Rechtecks (in Pixeln)
 * @param height
 *            Höhe des Rechtecks (in Pixeln)
 */
public void fillRect(int x, int y, int width, int height)

/**
 * Zeichnet im TheaterImage ein gefülltes Oval in der aktuellen
 * Zeichenfarbe.
 *
 * @param x
 *            x-Koordinate der linken oberen Ecke des Ovals
 * @param y
 *            y-Koordinate der linken oberen Ecke des Ovals
 * @param width
 *            Breite des Ovals in Pixeln
 * @param height
 *            Höhe des Ovals in Pixeln
 */
public void fillOval(int x, int y, int width, int height)

/**
 * Zeichnet im TheaterImage ein gefülltes Polygon in der aktuellen
 * Zeichenfarbe. Es wird automatisch ein Linie hinzugefügt, die das Polygon
 * schließt.
 *
 * @param xPoints
 *            x-Koordinaten der Linien
 * @param yPoints
 *            y-Koordinaten der Linien
 * @param nPoints
 *            Anzahl der Liniensegmente
 */
public void fillPolygon(int[] xPoints, int[] yPoints, int nPoints

/**
 * Löscht ein TheaterImage.
 */
public void clear()

/**
 * Setzt ein bestimmtes Pixel des TheaterImages auf eine bestimmte Farbe.
 *
 * @param x
 *            x-Koordinate des Pixels
 * @param y
```

```
     *            y-Koordinate des Pixels
     * @param color
     *            neue Farbe des Pixels
     */
    public void setColorAt(int x, int y, java.awt.Color color)

    /**
     * Liefert die Farbe eines bestimmten Pixels des TheaterImages.
     *
     * @param x
     *            x-Koordinate des Pixels
     * @param y
     *            y-Koordinate des Pixels
     * @return die Farbe eines bestimmten Pixels des TheaterImages
     */
    public java.awt.Color getColorAt(int x, int y)

    /**
     * Spiegelt das TheaterImage horizontal. Achtung: Die Größe des Bildes wird
     * dabei nicht verändert!
     */
    public void mirrorHorizontally()

    /**
     * Spiegelt das TheaterImage vertikal. Achtung: Die Größe des Bildes wird
     * dabei nicht verändert!
     */
    public void mirrorVertically()

    /**
     * Dreht das TheaterImage um eine bestimmte Gradzahl. Achtung: Die Größe des
     * Bildes wird dabei nicht verändert!
     *
     * @param degrees
     *            Gradzahl der Drehung
     */
    public void rotate(int degrees)

    /**
     * Skaliert das TheaterImage auf einebestimmte Größe.
     *
     * @param width
     *            die neue Breite des TheaterImages
     * @param height
     *            die neue Höhe des TheaterImages
     */
    public void scale(int width, int height)

    /**
     * Intern wird ein TheaterImage durch ein
     * java.awt.image.BufferedImage-Objekt realisiert. Diese Methode liefert das
     * entsprechende Objekt. Achtung: Einige Methoden tauschen intern das
     * BufferedImage-Objekt aus!
     *
     * @return das aktuelle interne BufferedImage-Objekt
     */
    public java.awt.image.BufferedImage getAwtImage()

}
```

6.5 Ereignisse

6.5.1 KeyInfo

```
package theater;

/**
```

```
 * Sowohl die Klasse Stage als auch die Klasse Component definieren die von der
 * Java-GUI-Programmierung bekannten Methoden zur Verarbeitung von
 * Tastatur-Events: keyTyped, keyPressed und keyRealeased. Die Events
 * entsprechen dabei den Events des Java-AWT in der Klasse
 * java.awt.event.KeyListener. Den Methoden werden Objekte vom Typ KeyInfo
 * übergeben, über die genauere Informationen über das entsprechende Event
 * abgefragt werden können.
 * <p>
 * </p>
 * Die Klasse KeyInfo ist von der Klasse java.awt.event.KeyEvent abgeleitet, so
 * dass auch alle deren Methoden benutzt werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class KeyInfo extends java.awt.event.KeyEvent {


   /**
    * Konstruktor zur Initialisierung eines KeyInfo-Objektes mit einem
    * KeyEvent-Objekt.
    * <p>
    * </p>
    * Der Konstruktor wird Theater-intern aufgerufen.
    *
    * @param e
    *            das eingetretene KeyEvent
    */
   public KeyInfo(java.awt.event.KeyEvent e)

   /**
    * Überschreibt die geerbte Methode und liefert das aktuelle Stage-Objekt.
    *
    * @return das aktuelle Stage-Objekt
    *
    * @see java.util.EventObject#getSource()
    */
   public Object getSource()

   /**
    * Tritt ein Tastatur-Event ein, so werden alle Komponenten und die Bühne
    * darüber informiert, insofern sie eine entsprechenden Handler-Methode
    * definiert und die Tastatur-Event-Benachrichtigung aktiviert haben. Für
    * die Reihenfolge der Benachrichtigung gilt: Je weiter das Objekt auf der
    * Bühne im Vordergrund ist, desto eher wird es informiert. Die Bühne wird
    * als letzte informiert. Das KeyInfo-Objekt, das dabei den Methoden
    * übergeben wird, ist dabei immer das gleiche. Über die Methode
    * setUserObject bekommen die Komponenten die Möglichkeit zu kommunizieren,
    * indem sie dem KeyInfo-Objekt ein anwendungsspezifisches Objekt
    * zuzuordnen, das später benachrichtigte Objekte über die Methode
    * getUserObject abfragen können.
    *
    * @param userObject
    *            ein beliebiges anwendungsspezifisches Objekt
    */
   public void setUserObject(Object userObject)

   /**
    * Tritt ein Tastatur-Event ein, so werden alle Komponenten und die Bühne
    * darüber informiert, insofern sie eine entsprechenden Handler-Methode
    * definiert und die Tastatur-Event-Benachrichtigung aktiviert haben. Für
    * die Reihenfolge der Benachrichtigung gilt: Je weiter das Objekt auf der
    * Bühne im Vordergrund ist, desto eher wird es informiert. Die Bühne wird
    * als letzte informiert. Das KeyInfo-Objekt, das dabei den Methoden
    * übergeben wird, ist dabei immer das gleiche. Über die Methode
    * setUserObject bekommen die Komponenten die Möglichkeit zu kommunizieren,
    * indem sie dem KeyInfo-Objekt ein anwendungsspezifisches Objekt
    * zuzuordnen, das später benachrichtigte Objekte über die Methode
    * getUserObject abfragen können.
```

```
     *
     * @return das dem KeyInfo-Objekt mittels der Methode setUserObject
     *         zugeordnete Objekt oder null, falls kein Objekt zugeordnet wurde.
     */
    public Object getUserObject()

    /**
     * Überschreibt die geerbte Methode und liefert im Falle eines Aufrufs eine
     * RuntimeException, da ein Zugriff auf die Theater-interne
     * Java-AWT-Komponente nicht erlaubt ist.
     *
     * @return wirft immer eine RuntimeException
     * @throws RuntimeException
     *             wird bei jedem Aufruf der Methode geworfen
     *
     * @see java.awt.event.ComponentEvent#getComponent()
     */
    public java.awt.Component getComponents() throws RuntimeException

}
```

6.5.2 MouseInfo

```
package theater;

/**
 * Sowohl die Klasse Stage als auch die Klasse Component definieren die von der
 * Java-GUI-Programmierung bekannten Methoden zur Verarbeitung von Maus-Events:
 * mousePressed, mouseReleased, mouseClicked, mouseDragged, mouseMoved,
 * mouseEntered und mouseExited. Die Events entsprechen dabei den Events des
 * Java-AWT in den Klassen java.awt.event.MouseListener bzw.
 * java.awt.event.MouseMotionListener. Den Methoden werden Objekte vom Typ
 * MouseInfo übergeben, über die genauere Informationen über das entsprechende
 * Event abgefragt werden können.
 * <p>
 * </p>
 * Die Klasse MouseInfo ist von der Klasse java.awt.event.MouseEvent abgeleitet,
 * so dass auch alle deren Methoden benutzt werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class MouseInfo extends java.awt.event.MouseEvent {

    /**
     * Konstruktor zur Initialisierung eines MouseInfo-Objektes mit einem
     * MouseEvent-Objekt.
     * <p>
     * </p>
     * Der Konstruktor wird Theater-intern aufgerufen.
     *
     * @param e
     *            das eingetretene MouseEvent
     */
    public MouseInfo(java.awt.event.MouseEvent e)

    /**
     * Überschreibt die geerbte Methode und liefert das jeweilige Komponenten-
     * bzw. Bühnenobjekt, oberhalb dessen Ikon das Event aufgetreten ist.
     *
     * @return das jeweilige Komponenten- bzw. Bühnenobjekt, oberhalb dessen
     *         Ikon das Event aufgetreten ist
     *
     * @see java.util.EventObject#getSource()
     */
    public Object getSource()
```

```
/**
 * Überschreibt die geerbte Methode. In dem Fall, dass die Bühne über das
 * Maus-Event informiert wird, also das aktuelle Bühnenobjekt das
 * Source-Objekt ist, liefert die Methode die x-Koordinate des Mauszeigers
 * bezüglich des Hintergrundes. In dem Fall, dass eine Komponente über das
 * Maus-Event informiert wird, also die Komponente das Source-Objekt ist,
 * liefert die Methode die x-Koordinate des Mauszeigers bezüglich des der
 * Komponente zugeordneten Ikons.
 *
 * @return die x-Koordinate des Maus-Events relativ gesehen zum
 *         Source-Objekt
 *
 * @see java.awt.event.MouseEvent#getX()
 */
public int getX()

/**
 * Überschreibt die geerbte Methode. In dem Fall, dass die Bühne über das
 * Maus-Event informiert wird, also das aktuelle Bühnenobjekt das
 * Source-Objekt ist, liefert die Methode die y-Koordinate des Mauszeigers
 * bezüglich des Hintergrundes. In dem Fall, dass eine Komponente über das
 * Maus-Event informiert wird, also die Komponente das Source-Objekt ist,
 * liefert die Methode die y-Koordinate des Mauszeigers bezüglich des der
 * Komponente zugeordneten Ikons.
 *
 * @return die y-Koordinate des Maus-Events relativ gesehen zum
 *         Source-Objekt
 *
 * @see java.awt.event.MouseEvent#getY()
 */
public int getY()

/**
 * Überschreibt die geerbte Methode. In dem Fall, dass die Bühne über das
 * Maus-Event informiert wird, also das aktuelle Bühnenobjekt das
 * Source-Objekt ist, liefert die Methode die x- und y-Koordinate des
 * Mauszeigers bezüglich des Hintergrundes. In dem Fall, dass eine
 * Komponente über das Maus-Event informiert wird, also die Komponente das
 * Source-Objekt ist, liefert die Methode die x- und y-Koordinate des
 * Mauszeigers bezüglich des der Komponente zugeordneten Ikons.
 *
 * @return die x- und y-Koordinate des Maus-Events relativ gesehen zum
 *         Source-Objekt
 * @see java.awt.event.MouseEvent#getPoint()
 */
public java.awt.Point getPoint()

/**
 * Die Methode liefert die Spalte, über der sich der Mauszeiger aktuell
 * befindet.
 *
 * @return die Spalte, über der sich der Mauszeiger befindet
 */
public int getColumn()

/**
 * Die Methode liefert die Reihe, über der sich der Mauszeiger aktuell
 * befindet.
 *
 * @return die Reihe, über der sich der Mauszeiger befindet
 */
public int getRow()

/**
 * Tritt ein Maus-Event ein, so werden alle Komponenten und die Bühne
 * darüber informiert, insofern das Maus-Event oberhalb des ihnen
 * zugeordneten Icons erfolgte, sie eine entsprechenden Handler-Methode
 * definiert und die Maus-Event-Benachrichtigung aktiviert haben. Für die
 * Reihenfolge der Benachrichtigung gilt: Je weiter das Objekt auf der Bühne
 * im Vordergrund ist, desto eher wird es informiert. Die Bühne wird als
```

```
 * letzte informiert. Das MouseInfo-Objekt, das dabei den Methoden übergeben
 * wird, ist dabei immer das gleiche. Über die Methode setUserObject
 * bekommen die Komponenten die Möglichkeit zu kommunizieren, indem sie dem
 * MouseInfo-Objekt ein anwendungsspezifisches Objekt zuzuordnen, das später
 * benachrichtigte Objekte über die Methode getUserObject abfragen können.
 *
 * @param userObject
 *            ein beliebiges anwendungsspezifisches Objekt
 */
public void setUserObject(Object userObject)

/**
 * Tritt ein Maus-Event ein, so werden alle Komponenten und die Bühne
 * darüber informiert, insofern das Maus-Event oberhalb des ihnen
 * zugeordneten Icons erfolgte, sie eine entsprechenden Handler-Methode
 * definiert und die Maus-Event-Benachrichtigung aktiviert haben. Für die
 * Reihenfolge der Benachrichtigung gilt: Je weiter das Objekt auf der Bühne
 * im Vordergrund ist, desto eher wird es informiert. Die Bühne wird als
 * letzte informiert. Das MouseInfo-Objekt, das dabei den Methoden übergeben
 * wird, ist dabei immer das gleiche. Über die Methode setUserObject
 * bekommen die Komponenten die Möglichkeit zu kommunizieren, indem sie dem
 * MouseInfo-Objekt ein anwendungsspezifisches Objekt zuzuordnen, das später
 * benachrichtigte Objekte über die Methode getUserObject abfragen können.
 *
 * @return das dem MouseInfo-Objekt mittels der Methode setUserObject
 *         zugeordnete Objekt oder null, falls kein Objekt zugeordnet wurde.
 */
public Object getUserObject()

/**
 * Überschreibt die geerbte Methode und liefert im Falle eines Aufrufs eine
 * RuntimeException, da ein Zugriff auf die Theater-interne
 * Java-AWT-Komponente nicht erlaubt ist.
 *
 * @return wirft immer eine RuntimeException
 * @throws RuntimeException
 *             wird bei jedem Aufruf der Methode geworfen
 *
 * @see java.awt.event.ComponentEvent#getComponent()
 */
public java.awt.Component getComponent()

/**
 * Setzt die x-Koordinate. Die Methode wird Theater-intern aufgerufen.
 *
 * @param x
 *            die neue x-Koordinate
 */
public void setX(int x)

/**
 * Setzt die y-Koordinate. Die Methode wird Theater-intern aufgerufen.
 *
 * @param y
 *            die neue y-Koordinate
 */
public void setY(int y)

/**
 * Setzt die Spalte, über der sich der Mauszeiger aktuell befindet. Die
 * Methode wird Theater-intern aufgerufen.
 *
 * @param col
 *            die neue Spalte
 */
public void setColumn(int col)

/**
 * Setzt die Reihe, über der sich der Mauszeiger aktuell befindet. Die
 * Methode wird Theater-intern aufgerufen.
```

```
     *
     * @param row
     *            die neue Reihe
     */
    public void setRow(int row)

    /**
     * Setzt das Source-Objekt. Die Methode wird Theater-intern aufgerufen.
     *
     * @param source
     *            das neue Source-Objekt
     */
    public void setSource(Object source)

}
```

6.6 Kollisionserkennung

6.6.1 PixelArea

```
package theater;

/**
 * PixelArea ist ein Interface, das die Grundlage der
 * Kollisionserkennungsmethoden darstellt. Eine PixelArea kann man sich dabei
 * als ein beliebiges Gebiet auf der Bühne vorstellen. 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.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public interface PixelArea {

    /**
     * Überprüft, ob der Punkt mit den Koordinaten x und y innerhalb der
     * PixelArea liegt.
     *
     * @param x
     *            x-Koordinate des Punktes
     * @param y
     *            y-Koordinate des Punktes
     * @return genau dann true, wenn der Punkt mit den Koordinaten x und y
     *         innerhalb der PixelArea liegt
     */
    public boolean contains(int x, int y);

    /**
     * Überprüft, ob die aufgerufene PixelArea komplett innerhalb der als
     * Parameter übergebenen PixelArea liegt.
     *
     * @param area
     *            die zu vergleichende PixelArea
     * @return genau dann true, wenn die aufgerufene PixelArea komplett
     *         innerhalb der als Parameter übergebenen PixelArea liegt
     */
    public boolean isInside(PixelArea area);

    /**
     * Überprüft, ob die aufgerufene PixelArea die als Parameter übergebene
     * PixelArea schneidet.
     *
     * @param area
     *            die zu vergleichende PixelArea
```

```
    * @return genau dann true, wenn die aufgerufene PixelArea die als Parameter
    *         übergebene PixelArea schneidet
    */
   public boolean intersects(PixelArea area);
}
```

6.6.2 Rectangle

```
package theater;

/**
 * Die Klasse Rectangle repräsentiert ein rechteckiges Gebiet auf der Bühne. Sie
 * implementiert das Interface PixelArea, so dass mit dieser Klasse Kollisionen
 * von rechteckigen Gebieten mit anderen Gebieten der Bühne überprüft werden
 * können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Rectangle implements PixelArea {

   /**
    * x-Koordinate der linken oberen Ecke
    */
   protected int x;

   /**
    * y-Koordinate der linken oberen Ecke
    */
   protected int y;

   /**
    * Breite des Rechteckes
    */
   protected int width;

   /**
    * Höhe des Rechteckes
    */
   protected int height;

   /**
    * Konstruktor zum Initialisieren eines Rechteckes
    *
    * @param x
    *            x-Koordinate der linken oberen Ecke
    * @param y
    *            y-Koordinate der linken oberen Ecke
    * @param w
    *            Breite des Rechteckes
    * @param h
    *            Höhe des Rechteckes
    */
   public Rectangle(int x, int y, int w, int h)

   /**
    * Konstruktor zum Initialisieren eines Rechteckes mit einem
    * java.awt.Rectangle-Objekt
    *
    * @param r
    *            ein bereits existierendes java.awt.Rectangle-Objekt (draf
    *            nicht null sein)
    */
   public Rectangle(java.awt.Rectangle r)

   /**
    * Überprüft, ob der Punkt mit den als Parameter übergebenen Koordinaten
```

```
 * innerhalb des aufgerufenen Rechteckes liegt.
 *
 * @param x
 *            x-Koordinate des Punktes
 * @param y
 *            y-Koordinate des Punktes
 * @return genau dann true, wenn der Punkt mit den als Parameter übergebenen
 *         Koordinaten innerhalb des aufgerufenen Rechteckes liegt
 *
 * @see theater.PixelArea#contains(int, int)
 */
public boolean contains(int x, int y)

/**
 * Überprüft, ob das aufgerufene Rechteck komplett innerhalb der als
 * Parameter übergebenen PixelArea liegt.
 *
 * @param area
 *            die zu vergleichende PixelArea
 * @return genau dann true, wenn das aufgerufene Rechteck komplett innerhalb
 *         der als Parameter übergebenen PixelArea liegt
 *
 * @see theater.PixelArea#isInside(theater.PixelArea)
 */
public boolean isInside(PixelArea area)

/**
 * Überprüft, ob das aufgerufene Rechteck die als Parameter übergebene
 * PixelArea schneidet.
 *
 * @param area
 *            die zu vergleichende PixelArea
 * @return genau dann true, wenn das aufgerufene Rechteck die als Parameter
 *         übergebene PixelArea schneidet
 * @see theater.PixelArea#intersects(theater.PixelArea)
 */

public boolean intersects(PixelArea area)

/**
 * Liefert die x-Koordinate der linken oberen Ecke des Rechteckes.
 *
 * @return die x-Koordinate der linken oberen Ecke des Rechteckes
 */
public int getX()

/**
 * Liefert die y-Koordinate der linken oberen Ecke des Rechteckes.
 *
 * @return die y-Koordinate der linken oberen Ecke des Rechteckes
 */
public int getY()

/**
 * Liefert die Breite des Rechteckes.
 *
 * @return die Breite des Rechteckes
 */
public int getWidth()

/**
 * Liefert die Höhe des Rechteckes.
 *
 * @return die Höhe des Rechteckes
 */
public int getHeight()

/**
 * Wandelt das Rechteck um in ein Objekt der Klasse java.awt.Rectangle
 *
```

```
    * @return das in ein java.awt.Rectangle-Objekt umgewandelte Rechteck
    */
   public java.awt.Rectangle toAWTRectangle()
}
```

6.6.3 Point

```
package theater;

/**
 * Die Klasse Point repräsentiert ein Pixel auf der Bühne. Sie implementiert das
 * Interface PixelArea, so dass mit dieser Klasse Kollisionen von Pixeln mit
 * anderen Gebieten der Bühne überprüft werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Point implements PixelArea {

   /**
    * x-Koordinate des Punktes
    */
   protected int x;

   /**
    * y-Koordinate des Punktes
    */
   protected int y;

   /**
    * Konstruktor zum Initialisieren eines Punktes mit seiner x- und
    * y-Koordinate.
    *
    * @param x
    *            x-Koordinate des Punktes
    * @param y
    *            y-Koordinate des Punktes
    */
   public Point(int x, int y)

   /**
    * Copy-Konstruktor zum Initialisieren eines Punktes mit einem
    * java.awt.Point-Objekt
    *
    * @param p
    *            ein Objekt der Klase java.awt.Point (darf nicht null sein)
    */
   public Point(java.awt.Point p)

   /**
    * Überprüft, ob der Punkt mit den als Parameter übergebenen Koordinaten
    * gleich dem aufgerufenen Punkt ist.
    *
    * @param x
    *            x-Koordinate des Punktes
    * @param y
    *            y-Koordinate des Punktes
    * @return genau dann true, wenn die beiden Punkte gleich sind
    *
    * @see theater.PixelArea#contains(int, int)
    */
   public boolean contains(int x, int y)

   /**
    * Überprüft, ob der aufgerufene Punkt innerhalb der als Parameter
    * übergebenen PixelArea liegt.
    *
```

```
     * @param area
     *            die zu vergleichende PixelArea
     * @return genau dann true, wenn der aufgerufene Punkt innerhalb der als
     *         Parameter übergebenen PixelArea liegt
     *
     * @see theater.PixelArea#isInside(theater.PixelArea)
     */
    public boolean isInside(PixelArea area)

    /**
     * Überprüft, ob der aufgerufene Punkt die als Parameter übergebene
     * PixelArea schneidet, d.h. innerhalb der PixelArea liegt.
     *
     * @param area
     *            die zu vergleichende PixelArea
     * @return genau dann true, wenn der aufgerufene Punkt innerhalb der als
     *         Parameter übergebenen PixelArea liegt
     *
     * @see theater.PixelArea#intersects(theater.PixelArea)
     */
    public boolean intersects(PixelArea area)

    /**
     * Liefert die x-Koordinate des Punktes auf der Bühne.
     *
     * @return die x-Koordinate des Punktes auf der Bühne
     */
    public int getX()

    /**
     * Liefert die y-Koordinate des Punktes auf der Bühne.
     *
     * @return die y-Koordinate des Punktes auf der Bühne
     */
    public int getY()

    /**
     * Wandelt den Punkt in ein Objekt der Klasse java.awt.Point um.
     *
     * @return der Punkt als java.awt.Point-Objekt
     */
    public java.awt.Point toAWTPoint()

}
```

6.6.4 Cell

```
package theater;

/**
 * Die Klasse Cell repräsentiert eine Zelle der Bühne. Sie implementiert das
 * Interface PixelArea, so dass mit dieser Klasse Kollisionen von Zellen mit
 * anderen Gebieten der Bühne überprüft werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class Cell implements PixelArea {

    /**
     * Reihe der Zelle
     */
    protected int row;

    /**
     * Spalte der Zelle
     */
```

```
    protected int col;

    /**
     * Konstruktor zum Initialisieren einer Zelle mit seiner Spalte und Reihe.
     *
     * @param col
     *            die Spalte der Zelle
     * @param row
     *            die Reihe der Zelle
     */
    public Cell(int col, int row)

    /**
     * Überprüft, ob der Punkt mit den als Parameter übergebenen Koordinaten
     * innerhalb der aufgerufenen Zelle liegt.
     *
     * @param x
     *            x-Koordinate des Punktes
     * @param y
     *            y-Koordinate des Punktes
     * @return genau dann true, wenn der Punkt mit den als Parameter übergebenen
     *         Koordinaten innerhalb der aufgerufenen Zelle liegt
     *
     * @see theater.PixelArea#contains(int, int)
     */
    public boolean contains(int x, int y)

    /**
     * Überprüft, ob die aufgerufene Zelle innerhalb der als Parameter
     * übergebenen PixelArea liegt.
     *
     * @param area
     *            die zu vergleichende PixelArea
     * @return genau dann true, wenn die aufgerufene Zelle innerhalb der als
     *         Parameter übergebenen PixelArea liegt
     *
     * @see theater.PixelArea#isInside(theater.PixelArea)
     */
    public boolean isInside(PixelArea area)

    /**
     * Überprüft, ob die aufgerufene Zelle die als Parameter übergebene
     * PixelArea schneidet.
     *
     * @param area
     *            die zu vergleichende PixelArea
     * @return genau dann true, wenn die aufgerufene Zelle die als Parameter
     *         übergebene PixelArea schneidet
     * @see theater.PixelArea#intersects(theater.PixelArea)
     */
    public boolean intersects(PixelArea area)

    /**
     * Liefert die Spalte der Zelle.
     *
     * @return die Spalte der Zelle
     */
    public int getCol()

    /**
     * Liefert die Reihe der Zelle.
     *
     * @return die Reihe der Zelle
     */
    public int getRow()
}
```

6.6.5 CellArea

```
package theater;

/**
 * Die Klasse CellArea repräsentiert ein Menge von Zellen (genauer ein
 * rechteckiges Gebiet von Zellen) der Bühne. Sie implementiert das Interface
 * PixelArea, so dass mit dieser Klasse Kollisionen von Zellen mit anderen
 * Gebieten der Bühne überprüft werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (12.11.2008)
 *
 */
public class CellArea implements PixelArea {

   /**
    * Spalte der linken oberen Ecke des CellArea-Gebietes
    */
   protected int fromCol;

   /**
    * Reihe der linken oberen Ecke des CellArea-Gebietes
    */
   protected int fromRow;

   /**
    * Breite, d.h. Anzahl an Spalten des CellArea-Gebietes
    */
   protected int numberOfCols;

   /**
    * Höhe, d.h. Anzahl an Spalten des CellArea-Gebietes
    */
   protected int numberOfRows;

   /**
    * Konstruktor zum Initialisieren der CellArea.
    *
    * @param fromCol
    *            Spalte der linken oberen Ecke des CellArea-Gebietes
    * @param fromRow
    *            Reihe der linken oberen Ecke des CellArea-Gebietes
    * @param numberOfCols
    *            Breite, d.h. Anzahl an Spalten des CellArea-Gebietes
    * @param numberOfRows
    *            Höhe, d.h. Anzahl an Spalten des CellArea-Gebietes
    */
   public CellArea(int fromCol, int fromRow, int numberOfCols, int numberOfRows)

   /**
    * Überprüft, ob der Punkt mit den als Parameter übergebenen Koordinaten
    * innerhalb der aufgerufenen CellArea liegt.
    *
    * @param x
    *            x-Koordinate des Punktes
    * @param y
    *            y-Koordinate des Punktes
    * @return genau dann true, wenn der Punkt mit den als Parameter übergebenen
    *         Koordinaten innerhalb der aufgerufenen CellArea liegt
    *
    * @see theater.PixelArea#contains(int, int)
    */
   public boolean contains(int x, int y)

   /**
    * Überprüft, ob die aufgerufene CellArea komplett innerhalb der als
    * Parameter übergebenen PixelArea liegt.
    *
```

```
     * @param area
     *            die zu vergleichende PixelArea
     * @return genau dann true, wenn die aufgerufene CellArea komplett innerhalb
     *         der als Parameter übergebenen PixelArea liegt
     *
     * @see theater.PixelArea#isInside(theater.PixelArea)
     */
    public boolean isInside(PixelArea area)

    /**
     * Überprüft, ob die aufgerufene CellArea die als Parameter übergebene
     * PixelArea schneidet.
     *
     * @param area
     *            die zu vergleichende PixelArea
     * @return genau dann true, wenn die aufgerufene CellArea die als Parameter
     *         übergebene PixelArea schneidet
     * @see theater.PixelArea#intersects(theater.PixelArea)
     */
    public boolean intersects(PixelArea area)

    /**
     * Liefert die Spalte der linken oberen Ecke der CellArea.
     *
     * @return die Spalte der linken oberen Ecke der CellArea
     */
    public int getFromCol()

    /**
     * Liefert die Reihe der linken oberen Ecke der CellArea.
     *
     * @return die Reihe der linken oberen Ecke der CellArea
     */
    public int getFromRow()

    /**
     * Liefert die Breite, d.h. die Anzahl an Spalten der CellArea.
     *
     * @return die Anzahl an Spalten der CellArea
     */
    public int getNumberOfCols()

    /**
     * Liefert die Höhe, d.h. die Anzahl an Reihen der CellArea.
     *
     * @return die Anzahl an Reihen der CellArea
     */
    public int getNumberOfRows()

}
```

7 Literatur zur parallelen Programmierung

Threadnocchio ist ein Werkzeug, das durch den Einsatz von Visualisierungstechniken Programmierern beim Erlernen der parallelen Programmierung hilft. Threadnocchio selbst ist kein Lehrbuch für die parallele Programmierung. Es gibt jedoch eine Menge von Lehrbücher für die parallele Programmierung, von denen die meines Erachtens geeignetsten im Folgenden aufgelistet werden.

7.1 Allgemeine Lehrbücher für parallele Programmierung

Herrtwich, R. G. und Hommel, G.: Nebenläufige Programme. Springer, 1994.

Tanenbaum, A. und van Stehen, M.: Verteilte Systeme. Prinzipien und Paradigmen, Pearson Studium, 2007.

7.2 Lehrbücher für die parallele Programmierung mit Java-Threads

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

Oechsle, R.: Parallele und verteilte Anwendungen in Java. Carl Hanser Verlag, 2007.

8 Beispiel-Theaterstücke

Die folgenden Theaterstücke dienen als Beispiele für den Einsatz von Threadnocchio. Sie finden die Theaterstücke im Unterverzeichnis *plays* des Threadnocchio-Ordners (siehe auch Kapitel 2).

8.1 Beispiel-Theaterstück Startrek

Startrek ist ein sehr kleines Threadnocchio-Theaterstück, das aber bereits die wichtigsten Konzepte von und Möglichkeiten mit Threadnocchio aufzeigt. 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.

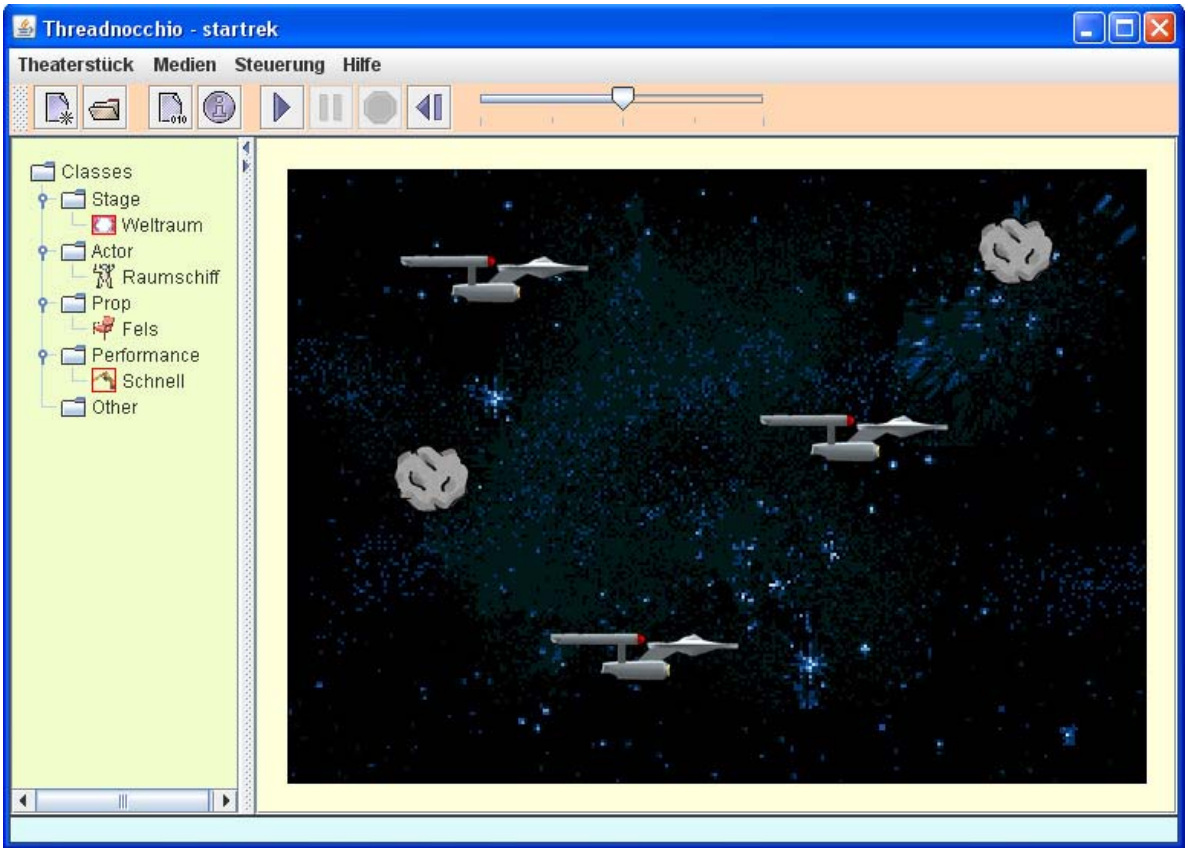

Abb. 2: Threadnocchio-Simulator mit Startrek-Theaterstück

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

```
import theater.*;

import java.util.List;
```

```
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 das Popup-Menü der Klasse `Raumschiff` 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 theater.*;

import java.awt.event.MouseEvent;

import java.util.List;


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);
    }
}
```

8.2 Beispiel-Theaterstück Philosophen

Das Philosophen-Theaterstück demonstriert eine Lösung des Philosophenproblems mit Threadnocchio. Bei dem Philosophenproblem (engl. Dining Philosophers Problem) handelt es sich um ein Fallbeispiel aus dem Bereich der Theoretischen Informatik. Dabei soll erklärt werden, wie ein Deadlock bei Prozessen entstehen kann. Das Problem wurde von E. W. Dijkstra formuliert.

Es sitzen fünf Philosophen an einem runden Tisch, und jeder hat einen Teller mit Spaghetti vor sich. Zum Essen von Spaghetti benötigt jeder Philosoph zwei Gabeln.

Allerdings sind nur fünf Gabeln vorhanden, die zwischen den Tellern liegen. Die Philosophen können also nicht alle fünf gleichzeitig speisen.

Die Philosophen sitzen am Tisch und denken über philosophische Probleme nach. Wenn einer hungrig wird, greift er zuerst die Gabel links von seinem Teller, dann die auf der rechten Seite und beginnt zu essen. Wenn er satt ist, legt er die Gabeln wieder zurück und beginnt wieder zu denken. Sollte eine Gabel nicht an ihrem Platz liegen, wenn der Philosoph sie aufnehmen möchte, so wartet er, bis die Gabel wieder verfügbar ist.

Solange nur einzelne Philosophen hungrig sind, funktioniert dieses Verfahren wunderbar. Es kann aber passieren, dass sich alle fünf Philosophen gleichzeitig entschließen zu essen. Sie ergreifen also alle gleichzeitig ihre linke Gabel und nehmen damit dem jeweils links von ihnen sitzenden Kollegen seine rechte Gabel weg. Nun warten alle fünf darauf, dass die rechte Gabel wieder auftaucht. Dies passiert aber nicht, da keiner der fünf seine linke Gabel zurücklegt. Ein kjlassischer Deadlock mit der Folge, dass die Philosophen verhungern. (Quelle: Wikipedia - http://de.wikipedia.org/wiki/Philosophenproblem).

Im folgenden Threadnocchio-Theaterstück wird zur Lösung des Deadlock-Problems eine goldene Gabel eingeführt. Stellt ein Philosoph fest, dass seine linke Gabel die vergoldete Gabel ist, nimmt er zunächst die rechte.

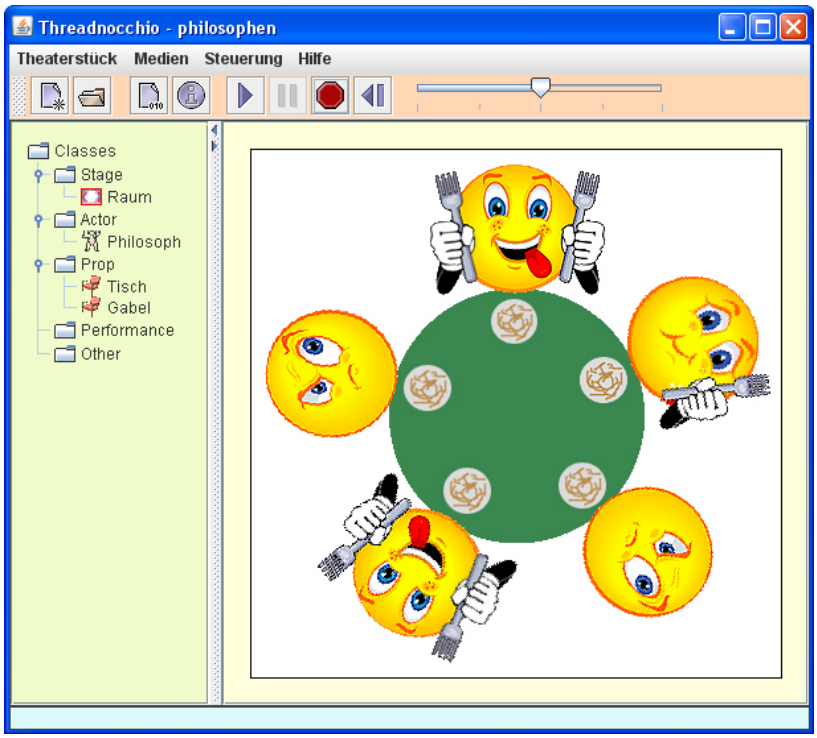

Abb. 2: Threadnocchio-Simulator mit Philosophen-Theaterstück

Die Bühne im Philosophen-Theaterstück wird über die folgende Klasse `Raum` realisiert. Im Konstruktor der Klasse wird die Ausgangsposition aufgebaut, bei der 5 Philosphen um einen Tisch sitzen, auf dem Teller und Gabeln liegen.

```
import theater.*;


public class Raum extends Stage {
    public Raum() {
        super(400, 400, 1);

        Tisch tisch = new Tisch();
        this.add(tisch, 200, 200);

        Gabel g1 = new Gabel();
        g1.setRotation(155);
        this.add(g1, 160, 145);

        Gabel g2 = new Gabel();
        g2.setRotation(80);
        this.add(g2, 135, 215);

        Gabel g3 = new Gabel();
        g3.setRotation(10);
        this.add(g3, 200, 270);

        Gabel g4 = new Gabel();
        g4.setRotation(290);
        this.add(g4, 265, 215);

        Gabel g5 = new Gabel(true);
        g5.setRotation(220);
        this.add(g5, 240, 145);

        Philosoph p1 = new Philosoph(g5, g1);
        p1.setName("p1");
        p1.setRotation(0);
        this.add(p1, 200, 60);

        Philosoph p2 = new Philosoph(g1, g2);
        p2.setName("p2");
        p2.setRotation(280);
        this.add(p2, 60, 170);

        Philosoph p3 = new Philosoph(g2, g3);
        p3.setName("p3");
        p3.setRotation(216);
        this.add(p3, 125, 320);

        Philosoph p4 = new Philosoph(g3, g4);
        p4.setName("p4");
        p4.setRotation(140);
        this.add(p4, 300, 305);

        Philosoph p5 = new Philosoph(g4, g5);
        p5.setName("p5");
        p5.setRotation(70);
        this.add(p5, 335, 150);
    }
}
```

Es gibt eine Actor-Klasse namens `Philosoph`. In dieser Klasse werden die Aktionen der Philosophen festgelegt. Der Einsatz eines Reentrant-Locks ist aus

Darstellungsgründen notwendig, weil es sonst passieren kann, dass optisch mehr als 5 Gabeln auf dem Bildschirm erscheinen

```
import theater.Actor;
import theater.Performance;

import java.util.concurrent.locks.ReentrantLock;


public class Philosoph extends Actor {
    private static ReentrantLock lock = new ReentrantLock();
    private Gabel linkeGabel;
    private Gabel rechteGabel;

    Philosoph(Gabel linkeGabel, Gabel rechteGabel) {
        this.setImage("Denker.gif");
        this.linkeGabel = linkeGabel;
        this.rechteGabel = rechteGabel;
    }

    public void run() {
        while (true) {
            this.denken();
            this.nimmGabeln();
            this.essen();
            this.gibGabeln();
        }
    }

    private void nimmGabeln() {
        if (this.linkeGabel.istVergoldet()) {
            this.nimmRechteGabel();
            changeImages(this, "PhiloMitRechterGabel.gif", this.rechteGabel,
                "KeineGabel.gif");
            this.nimmLinkeGabel();
            changeImages(this, "Esser.gif", this.linkeGabel, "KeineGabel.gif");
        } else {
            this.nimmLinkeGabel();
            changeImages(this, "PhiloMitLinkerGabel.gif", this.linkeGabel,
                "KeineGabel.gif");
            this.nimmRechteGabel();
            changeImages(this, "Esser.gif", this.rechteGabel, "KeineGabel.gif");
        }
    }

    private void nimmLinkeGabel() {
        this.linkeGabel.aufnehmen(this);
    }

    private void nimmRechteGabel() {
        this.rechteGabel.aufnehmen(this);
    }

    private void gibGabeln() {
        lock.lock();
        this.gibLinkeGabel();
        changeImages(this, "PhiloMitRechterGabel.gif", this.linkeGabel,
            this.linkeGabel.istVergoldet() ? "GoldGabel.gif" : "Gabel.gif");
        lock.unlock();
        lock.lock();
        this.gibRechteGabel();
        changeImages(this, "Denker.gif", this.rechteGabel,
            this.rechteGabel.istVergoldet() ? "GoldGabel.gif" : "Gabel.gif");

        lock.unlock();
    }

    private void gibLinkeGabel() {
```

```
        this.linkeGabel.ablegen();
    }

    private void gibRechteGabel() {
        this.rechteGabel.ablegen();
    }

    private void denken() {
        int bedenkZeit = (int) (Math.random() * 5000);

        try {
            Thread.sleep(bedenkZeit);
        } catch (Exception exc) {
        }
    }

    private void essen() {
        Performance.getPerformance().playSound("Ruelpser.wav");
        int kauZeit = (int) (Math.random() * 3000);

        try {
            Thread.sleep(kauZeit);
        } catch (Exception exc) {
        }
    }

    private void changeImages(Philosoph p, String pImg, Gabel g, String gImg) {
        lock.lock();
        p.setImage(pImg);
        g.setImage(gImg);
        lock.unlock();
    }
}
```

Als Requisiten werden ein Tisch und 5 Gabeln benötigt. Die Teller sind im Hintergrundbild der Klasse Tisch vorhanden.

```
import theater.*;

public class Tisch extends Prop {
    public Tisch() {
        this.setImage("Tisch.gif");
    }
}
```

Die Synchronisation der Philosophen erfolgt über die Gabeln, die entsprechende Synchronisationskonstrukte von Java einsetzen (synchronized, wait, notify).

```
import theater.*;


public class Gabel extends Prop {
    private boolean verfuegbar;
    private boolean istVergoldet;

    Gabel() {
        this(false);
    }

    Gabel(boolean istVergoldet) {
        if (istVergoldet) {
            this.setImage("GoldGabel.gif");
        } else {
            this.setImage("Gabel.gif");
```

```
        }

        this.istVergoldet = istVergoldet;
        this.verfuegbar = true;
    }

    synchronized void aufnehmen(Philosoph p) {
        while (!this.verfuegbar) {
            try {
                this.wait();
            } catch (Exception e) {
            }
        }

        this.verfuegbar = false;
    }

    synchronized void ablegen() {
        this.verfuegbar = true;
        this.notify(); // Nachbarn informieren
    }

    boolean istVergoldet() {
        return this.istVergoldet;
    }
}
```

8.3 Beispiel-Theaterstück Hamster

In dem Buch „Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell“ wird zum Erlernen der parallelen Programmierung mit Java-Threads das so genannte *Java-Hamster-Modell* eingesetzt (siehe auch Kapitel 7). Die Visualisierung von Threads erfolgt hierin mit virtuellen Hamstern, die der Programmierer durch ein virtuelles Territorium steuert und Aufgaben lösen lässt. Das Theaterstück Hamster überträgt das Java-Hamster-Modell in Threadnocchio.

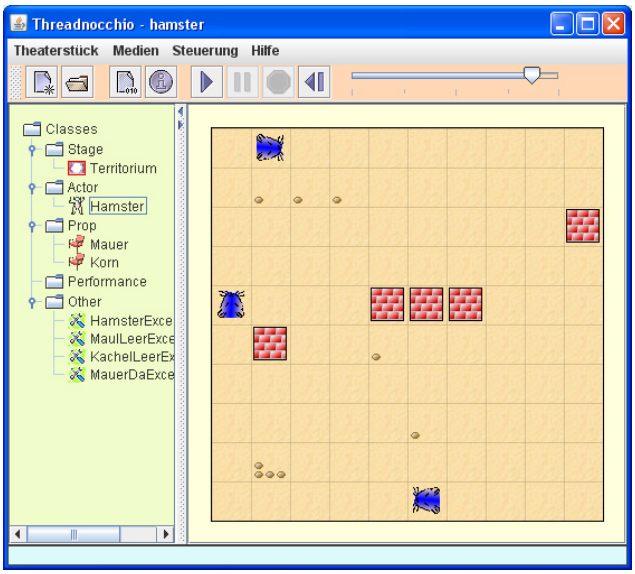

Abb. 2: Threadnocchio-Simulator mit Hamster-Theaterstück

Die Klasse Territorium wird als Bühne genutzt. Hamster sind Akteure, die Aufgaben selbstständig zu lösen haben. Mauern und Körner stellen Requisiten dar.

8.3.1 Klasse Territorium

```
/**
 * Die Klasse stellt eine Repraesentation des Hamster-Territoriums dar. Die
 * Methoden dienen zum Abfragen bestimmter Zustandswerte des aktuellen
 * Territoriums.
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class Territorium extends Stage {
    static Object[][] kacheln = null;

    /**
     * Erzeugt ein neues Territorium mit 10x10 Kacheln
     */
    public Territorium() {
        this(10, 10);
    }

    /**
     * Erzeugt ein neues Territorium in der angegebenen Groesse
     *
     * @param reihen
     *            Anzahl an Reihen
     * @param spalten
     *            Anzahl an Spalten
     */
    public Territorium(int reihen, int spalten) {
        super((spalten < 1) ? 10 : spalten, (reihen < 1) ? 10 : reihen, 35);
        this.setBackground("kachel.jpg");
    }

    /**
     * liefert die Anzahl an Reihen im Territorium
     *
     * @return die Anzahl an Reihen im Territorium
     */
    public int getAnzahlReihen() {
        return this.getNumberOfRows();
    }

    /**
     * liefert die Anzahl an Spalten im Territorium
     *
     * @return die Anzahl an Spalten im Territorium
     */
    public int getAnzahlSpalten() {
        return this.getNumberOfColumns();
    }

    /**
     * ueberprueft, ob sich auf der Kachel (reihe/spalte) eine Mauer befindet;
     * es wird genau dann true geliefert, wenn sich auf der angegebenen Kachel
     * eine Mauer befindet oder wenn sich die angegebenen Werte ausserhalb des
     * Territoriums befinden
     *
     * @param reihe
     *            Reihe der Kachel
     * @param spalte
     *            Spalte der Kachel
     * @return true geliefert, wenn sich auf der angegebenen Kachel eine Mauer
     *         befindet oder wenn sich die angegebenen Werte ausserhalb des
```

```
     *         Territoriums befinden; sonst false
     */
    public boolean mauerDa(int reihe, int spalte) {
        return this.getComponentsAt(spalte, reihe, Mauer.class).size() > 0;
    }

    /**
     * liefert die Gesamtzahl an Koernern, die im Territorium auf Kacheln
     * herumliegen
     *
     * @return die Gesamtzahl an Koernern, die im Territorium auf Kacheln
     *         herumliegen
     */
    public int getAnzahlKoerner() {
        int anzahl = 0;

        for (int r = 0; r < this.getAnzahlReihen(); r++) {
            for (int s = 0; s < this.getAnzahlSpalten(); s++) {
                anzahl += this.getAnzahlKoerner(r, s);
            }
        }

        return anzahl;
    }

    /**
     * liefert die Anzahl an Koernern auf der Kachel (reihe/spalte) oder 0,
     * falls die Kachel nicht existiert oder durch eine Mauer blockiert ist
     *
     * @param reihe
     *            Reihe der Kachel
     * @param spalte
     *            Spalte der Kachel
     * @return die Anzahl an Koernern auf der Kachel (reihe/spalte) oder 0,
     *         falls die Kachel nicht existiert oder durch eine Mauer blockiert
     *         ist
     */
    public int getAnzahlKoerner(int reihe, int spalte) {
        List<Component> comps = this.getComponentsAt(spalte, reihe, Korn.class);

        if ((comps == null) || (comps.size() == 0)) {
            return 0;
        }

        return ((Korn) comps.get(0)).getAnzahl();
    }

    /**
     * liefert die Gesamtzahl an existierenden Hamstern im Territorium
     *
     * @return die Gesamtzahl an existierenden Hamstern im Territorium
     */
    public int getAnzahlHamster() {
        return this.getComponents(Hamster.class).size();
    }

    /**
     * liefert alle existierenden Hamster im Territorium
     *
     * @return alle existierenden Hamster im Territorium
     */
    public Hamster[] getHamster() {
        return this.getComponents(Hamster.class).toArray(new Hamster[0]);
    }

    /**
     * liefert die Anzahl an Hamstern auf der Kachel (reihe/spalte) oder 0,
     * falls die Kachel nicht existiert oder durch eine Mauer blockiert ist
     *
     * @param reihe
```

```
     *            Reihe der Kachel
     * @param spalte
     *            Spalte der Kachel
     * @return die Anzahl an Hamstern auf der Kachel (reihe/spalte) oder 0,
     *         falls die Kachel nicht existiert oder durch eine Mauer blockiert
     *         ist
     */
    public int getAnzahlHamster(int reihe, int spalte) {
        return this.getComponentsAt(spalte, reihe, Hamster.class).size();
    }

    /**
     * liefert alle Hamster, die aktuell auf der Kachel (reihe/spalte) stehen
     * (inkl. dem Standard-Hamster)
     *
     * @param reihe
     *            Reihe der Kachel
     * @param spalte
     *            Spalte der Kachel
     * @return alle Hamster, die aktuell auf der Kachel (reihe/spalte) stehen
     */
    public Hamster[] getHamster(int reihe, int spalte) {
        return this.getComponentsAt(spalte, reihe, Hamster.class)
                .toArray(new Hamster[0]);
    }

    /**
     * Besetzt das Territorium mit einer festgelegten Population
     */
    public void populationGenerieren() {
        Hamster h1 = new Hamster();
        h1.linksUm();
        h1.linksUm();
        h1.linksUm();
        this.add(h1, 0, 0);

        Hamster h2 = new Hamster();
        h2.linksUm();
        this.add(h2, this.getNumberOfColumns() - 1, this.getNumberOfRows() - 1);

        Hamster h3 = new Hamster();
        this.add(h3, this.getNumberOfColumns() / 2, this.getNumberOfRows() / 2);

        this.koernerGenerieren((this.getNumberOfColumns() * this.getNumberOfRows()) /
2);
    }

    /**
     * Platziert zufaellig eine angegebene Anzahl von Koernern im Territorium
     *
     * @param wieviele
     *            Anzahl der zu platzierenden Koerner
     */
    public void koernerGenerieren(int wieviele) {
        Random random = new Random();

        for (int i = 0; i < wieviele; i++) {
            Korn korn = new Korn();
            int col = random.nextInt(this.getNumberOfColumns());
            int row = random.nextInt(this.getNumberOfRows());
            this.add(korn, col, row);
        }
    }

    /**
     * Liefert ein Objekt, das eine Kachel repräsentiert
     *
     * @param reihe
     *            Reihe
     * @param Spalte
```

```
     *            Spalte
     * @return Kachelobjekt
     */
    public synchronized Object getKachel(int reihe, int spalte) {
        if (Territorium.kacheln == null) {
            Territorium.kacheln = new
Object[this.getAnzahlReihen()][this.getAnzahlSpalten()];

            for (int r = 0; r < Territorium.kacheln.length; r++) {
                for (int s = 0; s < Territorium.kacheln[r].length; s++) {
                    Territorium.kacheln[r][s] = new Object();
                }
            }
        }

        return Territorium.kacheln[reihe][spalte];
    }
}
```

8.3.2 Klasse Hamster

```
import theater.*;


/**
 * Repraesentation von objektorientierten Hamstern im Java-Hamster-Modell
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class Hamster extends Actor {
    /**
     * Blickrichtung Nord
     */
    public final static int NORD = 0;

    /**
     * Blickrichtung Ost
     */
    public final static int OST = 1;

    /**
     * Blickrichtung Sued
     */
    public final static int SUED = 2;

    /**
     * Blickrichtung West
     */
    public final static int WEST = 3;
    private int blickrichtung;
    private int koernerImMaul;

    /**
     * Konstruktor zum Erzeugen eines neuen Hamsters mit der angegebenen
     * Blickrichtung und Anzahl an Koernern im Maul
     *
     * @param blickrichtung
     *            Blickrichtung des Hamsters
     * @param anzahlKoerner
     *            Anzahl an Koernern im Maul
     */
    public Hamster(int blickrichtung, int anzahlKoerner) {
        this.setImage("hamster.png");
        this.setBlickrichtung(blickrichtung);
        this.koernerImMaul = anzahlKoerner;
    }
```

```
/**
 * Konstruktor zum Erzeugen eines neuen Hamsters mit Blickrichtung OST und
 * keinem Korn im Maul
 */
public Hamster() {
    this(Hamster.OST, 0);
}

// Copy-Konstruktor
private Hamster(Hamster h) {
    this(h.blickrichtung, h.koernerImMaul);
}

/**
 * Dummy-Aktionen
 */
public void run() {
    while (true) {
        while (this.vornFrei()) {
            this.vor();

            if (this.kornDa()) {
                this.nimm();
            }
        }

        this.linksUm();
    }
}

/**
 * liefert genau dann true, wenn sich in Blickrichtung vor dem aufgerufenen
 * Hamster keine Mauer befindet (wenn sich der Hamster in Blickrichtung am
 * Rand des Territoriums befindet, wird false geliefert)
 *
 * @return true, wenn sich in Blickrichtung vor dem aufgerufenen Hamster
 *         keine Mauer befindet; sonst false
 */
public boolean vornFrei() {
    int col = this.getColumn();
    int row = this.getRow();

    switch (this.blickrichtung) {
    case SUED:
        row++;

        break;

    case OST:
        col++;

        break;

    case NORD:
        row--;

        break;

    case WEST:
        col--;

        break;
    }

    if ((col >= this.getStage().getNumberOfColumns()) ||
            (row >= this.getStage().getNumberOfRows()) || (col < 0) ||
            (row < 0)) {
        return false;
    }
```

```
    return this.getStage().getComponentsAt(col, row, Mauer.class).size() == 0;
}

/**
 * liefert genau dann true, wenn auf der Kachel, auf der sich der
 * aufgerufene Hamster gerade befindet, mindestens ein Korn liegt
 *
 * @return true, wenn auf der Kachel, auf der sich der aufgerufene Hamster
 *         gerade befindet, mindestens ein Korn liegt; sonst false
 */
public boolean kornDa() {
    return this.getStage()
                .getComponentsAt(this.getColumn(), this.getRow(), Korn.class)
                .size() > 0;
}

/**
 * liefert genau dann true, wenn der aufgerufene Hamster keine Koerner im
 * Maul hat
 *
 * @return true, wenn der aufgerufene Hamster keine Koerner im Maul hat;
 *         sonst false
 */
public boolean maulLeer() {
    return this.koernerImMaul == 0;
}

/**
 * Der aufgerufene Hamster springt auf die in Blickrichtung vor ihm liegende
 * Kachel.
 *
 * @throws MauerDaException
 *             wird geworfen, wenn die Kachel in Blickrichtung vor dem
 *             Hamster durch eine Mauer blockiert ist oder der Hamster in
 *             Blickrichtung am Rand des Territoriums steht
 */
public void vor() throws MauerDaException {
    if (!this.vornFrei()) {
        throw new MauerDaException(this, this.getRow(), this.getColumn());
    }

    switch (this.blickrichtung) {
    case SUED:
        this.setLocation(this.getColumn(), this.getRow() + 1);

        break;

    case OST:
        this.setLocation(this.getColumn() + 1, this.getRow());

        break;

    case NORD:
        this.setLocation(this.getColumn(), this.getRow() - 1);

        break;

    case WEST:
        this.setLocation(this.getColumn() - 1, this.getRow());

        break;
    }
}

/**
 * Der aufgerufene Hamster dreht sich linksum.
 */
public void linksUm() {
    switch (this.blickrichtung) {
```

```
    case SUED:
        this.setBlickrichtung(Hamster.OST);

        break;

    case OST:
        this.setBlickrichtung(Hamster.NORD);

        break;

    case NORD:
        this.setBlickrichtung(Hamster.WEST);

        break;

    case WEST:
        this.setBlickrichtung(Hamster.SUED);

        break;
    }
}

/**
 * Der aufgerufene Hamster frisst ein Korn auf der Kachel, auf der er sich
 * gerade befindet.
 *
 * @throws KachelLeerException
 *             wird geworfen, wenn auf der Kachel, auf der sich der Hamster
 *             gerade befindet, kein Korn liegt
 */
public void nimm() throws KachelLeerException {
    if (!this.kornDa()) {
        throw new KachelLeerException(this, this.getRow(), this.getColumn());
    }

    this.koernerImMaul++;

    Korn korn = (Korn) this.getStage()
                            .getComponentsAt(this.getColumn(),
            this.getRow(), Korn.class).get(0);
    korn.inkAnzahl(-1);
}

/**
 * Der aufgerufene Hamster legt ein Korn auf der Kachel ab, auf der er sich
 * gerade befindet.
 *
 * @throws MaulLeerException
 *             wird geworfen, wenn der Hamster keine Koerner im Maul hat
 */
public void gib() throws MaulLeerException {
    if (this.maulLeer()) {
        throw new MaulLeerException(this);
    }

    this.koernerImMaul--;
    this.getStage().add(new Korn(), this.getColumn(), this.getRow());
}

/**
 * liefert die Reihe der Kachel des Territoriums, auf der sich der
 * aufgerufene Hamster gerade befindet
 *
 * @return die Reihe der Kachel des Territoriums, auf der sich der
 *         aufgerufene Hamster gerade befindet
 */
public int getReihe() {
    return this.getRow();
}
```

```
/**
 * liefert die Spalte der Kachel des Territoriums, auf der sich der
 * aufgerufene Hamster gerade befindet
 *
 * @return die Spalte der Kachel des Territoriums, auf der sich der
 *         aufgerufene Hamster gerade befindet
 */
public int getSpalte() {
    return this.getColumn();
}

/**
 * liefert die Blickrichtung, in die der aufgerufene Hamster gerade schaut
 * (die gelieferten Werte entsprechen den obigen Konstanten)
 *
 * @return die Blickrichtung, in die der aufgerufene Hamster gerade schaut
 */
public int getBlickrichtung() {
    return this.blickrichtung;
}

/**
 * liefert die Anzahl der Koerner, die der aufgerufene Hamster gerade im
 * Maul hat
 *
 * @return die Anzahl der Koerner, die der aufgerufene Hamster gerade im
 *         Maul hat
 */
public int getAnzahlKoerner() {
    return this.koernerImMaul;
}

/**
 * liefert die Gesamtzahl an existierenden Hamstern im Territorium
 *
 * @return die Gesamtzahl an existierenden Hamstern im Territorium
 */
public int getAnzahlHamster() {
    return this.getStage().getComponents(Hamster.class).size();
}

// Blickrichtung setzen
private void setBlickrichtung(int richtung) {
    this.blickrichtung = richtung;

    switch (this.blickrichtung) {
    case SUED:
        this.setRotation(90);

        break;

    case OST:
        this.setRotation(0);

        break;

    case NORD:
        this.setRotation(270);

        break;

    case WEST:
        this.setRotation(180);

        break;

    default:
        break;
    }
}
```

```
    // wird aufgerufen, wenn der Hamster in das Territorium platziert wird
    public void addedToStage(Stage stage) {
        // Hamster kann nicht auf Mauer platziert werden
        if (this.getStage()
                    .getComponentsAt(this.getColumn(), this.getRow(),
                    Mauer.class).size() > 0) {
            this.getStage().remove(this);

            return;
        }

        this.setZCoordinate(1);
    }

    // nur wenn auf der Kachel keine Mauer steht
    public void setLocation(int col, int row) {
        if (this.getStage().getComponentsAt(col, row, Mauer.class).size() == 0) {
            super.setLocation(col, row);
        }
    }
}
```

8.3.3 Klasse Mauer

```
import theater.*;

import java.util.List;


/**
 * Repraesentation von Mauern im Java-Hamster-Modell
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class Mauer extends Prop {
    public Mauer() {
        this.setImage("mauer.png");
    }

    public void addedToStage(Stage stage) {
        this.removeComponentsOnCell(this.getColumn(), this.getRow());
    }

    public void setLocation(int col, int row) {
        if ((this.getColumn() == col) && (this.getRow() == row)) {
            return;
        }

        this.removeComponentsOnCell(col, row);
        super.setLocation(col, row);
    }

    private void removeComponentsOnCell(int x, int y) {
        List<Component> comps = this.getStage().getComponentsAt(x, y);

        for (Component comp : comps) {
            if (comp != this) {
                this.getStage().remove(comp);
            }
        }
    }
}
```

8.3.4 Klasse Korn

```
import theater.*;

import java.util.List;


/**
 * Repraesentation von Koernern im Java-Hamster-Modell
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class Korn extends Prop {
    private int anzahl;

    public Korn() {
        this(1);
    }

    public Korn(int anzahl) {
        this.anzahl = anzahl;
        setImage("korn.png");
    }

    // wird aufgerufen, wenn das Korn in das Territorium platziert wird
    public void addedToStage(Stage stage) {
        // Wenn auf der Kachel schon eine Mauer ist, wird das Korn wieder
        // entfernt
        List<Component> mauer = this.getStage()
                                    .getComponentsAt(this.getColumn(),
                this.getRow(), Mauer.class);

        if (mauer.size() > 0) {
            this.getStage().remove(this);

            return;
        }

        // bereits Korn auf Kachel?
        List<Component> koerner = this.getStage()
                                      .getComponentsAt(this.getColumn(),
                this.getRow(), Korn.class);

        if (koerner.size() > 1) {
            Korn korn = (Korn) koerner.get(0);

            if (korn == this) {
                korn = (Korn) koerner.get(1);
            }

            this.getStage().remove(korn);
            this.anzahl += korn.anzahl;
        }

        this.setImage("korn" + Math.min(this.anzahl, 12) + ".png");

        // es werden maximal 12 Koerner angezeigt
    }

    // liefert die Information, das wie vielte Korn dieses Korn auf der Kachel
    // ist
    protected int getAnzahl() {
        return this.anzahl;
    }

    public void setLocation(int col, int row) {
        if ((this.getColumn() == col) && (this.getRow() == row)) {
```

```
            return;
        }

        // Mauer auf Kachel?
        if (this.getStage().getComponentsAt(col, row, Mauer.class).size() > 0) {
            return;
        }

        // bereits Korn auf Kachel?
        List<Component> koerner = this.getStage()
                                    .getComponentsAt(col, row, Korn.class);

        if (koerner.size() > 0) {
            Korn korn = (Korn) koerner.get(0);
            this.getStage().remove(korn);
            this.anzahl += korn.anzahl;
            this.setImage("korn" + Math.min(this.anzahl, 12) + ".png");
        }

        super.setLocation(col, row);
    }

    void inkAnzahl(int n) {
        this.anzahl += n;

        if (this.anzahl <= 0) {
            this.getStage().remove(this);
        } else {
            this.setImage("korn" + Math.min(this.anzahl, 12) + ".png");
        }
    }
}
```

8.3.5 Exception-Klassen

```
/**
 * Oberklasse aller Exception-Klassen des Java-Hamster-Modells. Bei allen
 * Exceptions des Java-Hamster-Modells handelt es sich um Unchecked-Exception,
 * die nicht unbedingt abgefangen bzw. deklariert werden muessen.
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class HamsterException extends RuntimeException {
    /**
     * Hamster, der die Exception verschuldet hat
     */
    private Hamster hamster;

    /**
     * Konstruktor, der die Exception mit dem Hamster initialisiert, der die
     * Exception verschuldet hat.
     *
     * @param hamster
     *            der Hamster, der die Exception verschuldet hat
     */
    public HamsterException(Hamster hamster) {
        super("");
        this.hamster = hamster;
    }

    /**
     * liefert den Hamster, der die Exception verschuldet hat
     *
     * @return der Hamster, der die Exception verschuldet hat
     */
    public Hamster getHamster() {
```

```
        return this.hamster;
    }
}

/**
 * Hamster-Exception die den Fehler repraesentiert, dass fuer einen Hamster ohne
 * Koerner im Maul die Methode gib aufgerufen wird.
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class MaulLeerException extends HamsterException {
    /**
     * Konstruktor, der die Exception mit dem Hamster initialisiert, der die
     * Exception verschuldet hat.
     *
     * @param hamster
     *            der Hamster, der die Exception verschuldet hat
     */
    public MaulLeerException(Hamster hamster) {
        super(hamster);
    }

    /**
     * liefert eine der Exception entsprechende Fehlermeldung
     *
     * @return Fehlermeldung
     * @see java.lang.Throwable#getMessage()
     */
    public String getMessage() {
        return "Der Hamster hat keine Koerner im Maul!";
    }
}

/**
 * Hamster-Exception die den Fehler repraesentiert, dass fuer einen Hamster auf
 * einer Kachel ohne Koerner die Methode nimm aufgerufen wird.
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class KachelLeerException extends HamsterException {
    private int reihe;
    private int spalte;

    /**
     * Konstruktor, der die Exception mit dem die Exception verschuldenden
     * Hamster und den Koordinaten der koernerlosen Kachel initialisiert.
     *
     * @param hamster
     *            der Hamster, der die Exception verschuldet hat
     * @param reihe
     *            Reihe der koernerlosen Kachel
     * @param spalte
     *            Spalte der koernerlosen Kachel
     */
    public KachelLeerException(Hamster hamster, int reihe, int spalte) {
        super(hamster);
        this.reihe = reihe;
        this.spalte = spalte;
    }

    /**
     * liefert die Reihe der koernerlosen Kachel
     *
     * @return die Reihe der koernerlosen Kachel
     */
    public int getReihe() {
```

```
        return this.reihe;
    }

    /**
     * liefert die Spalte der koernerlosen Kachel
     *
     * @return die Spalte der koernerlosen Kachel
     */
    public int getSpalte() {
        return this.spalte;
    }

    /**
     * liefert eine der Exception entsprechende Fehlermeldung
     *
     * @return Fehlermeldung
     * @see java.lang.Throwable#getMessage()
     */
    public String getMessage() {
        return "Auf der Kachel (" + reihe + "," + spalte +
        ") liegen keine Koerner!";
    }
}

/**
 * Hamster-Exception die den Fehler repraesentiert, dass fuer einen Hamster, der
 * vor einer Mauer steht, die Methode vor aufgerufen wird auf.
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class MauerDaException extends HamsterException {
    private int reihe;
    private int spalte;

    /**
     * Konstruktor, der die Exception mit dem die Exception verschuldenden
     * Hamster und den Koordinaten der durch eine Mauer belegten Kachel
     * initialisiert.
     *
     * @param hamster
     *            der Hamster, der die Exception verschuldet hat
     * @param reihe
     *            Reihe der Mauer-Kachel
     * @param spalte
     *            Spalte der Mauer-Kachel
     */
    public MauerDaException(Hamster hamster, int reihe, int spalte) {
        super(hamster);
        this.reihe = reihe;
        this.spalte = spalte;
    }

    /**
     * liefert die Reihe, in der die Mauer steht
     *
     * @return die Reihe, in der die Mauer steht
     */
    public int getReihe() {
        return this.reihe;
    }

    /**
     * liefert die Spalte, in der die Mauer steht
     *
     * @return die Spalte, in der die Mauer steht
     */
    public int getSpalte() {
        return this.spalte;
```

```
    }

    /**
     * liefert eine der Exception entsprechende Fehlermeldung
     *
     * @return Fehlermeldung
     * @see java.lang.Throwable#getMessage()
     */
    public String getMessage() {
        return "Die Kachel (" + reihe + "," + spalte +
        ") ist durch eine Mauer blockiert!";
    }
}
```

8.4 Beispiel-Theaterstück Leser-Schreiber-Hamster

In dem Buch „Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell“ wird in Kapitel 11 das Leser-Schreiber-Problem, wie das Philosophenproblem ein klassisches Problem der parallelen Programmierung, behandelt und mit Hilfe der Hamster visualisiert. Im Folgenden wird das dortige Programm in ein Threadnocchio-Theaterstück übertragen, wobei die Klassen aus Abschnitt 8.3 genutzt werden.

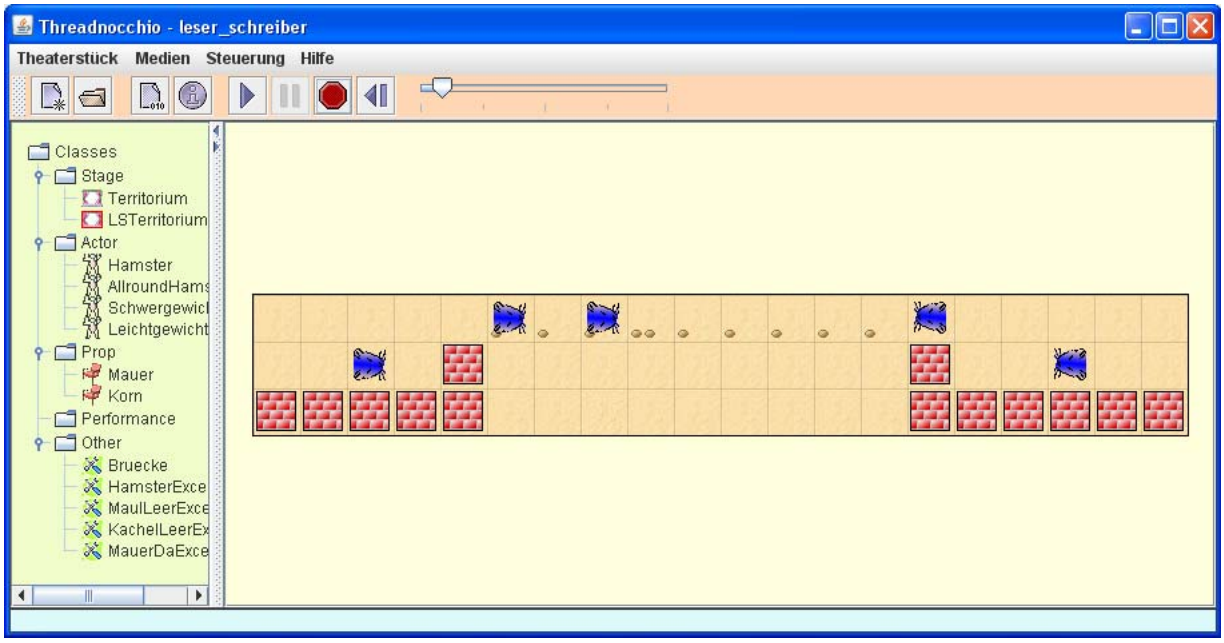

Abb. 2: Threadnocchio-Simulator mit Leser-Schreiber-Hamster-Theaterstück

Die Klasse `LSTerritorium` repräsentiert die Ausgangsbühne.

```
/**
 * Die Klasse stellt eine Repraesentation des Leser-Schreiber-Territoriums dar.
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
```

```
public class LSTerritorium extends Territorium {
    public LSTerritorium() {
        super(3, 20);

        // Mauern platzieren
        for (int i = 0; i < 5; i++) {
            this.add(new Mauer(), i, 2);
        }

        for (int i = 14; i < 20; i++) {
            this.add(new Mauer(), i, 2);
        }

        this.add(new Mauer(), 4, 1);
        this.add(new Mauer(), 14, 1);

        // Koerner platzieren
        for (int i = 5; i < 14; i++) {
            this.add(new Korn(), i, 0);
        }

        this.add(new Korn(), 8, 0);

        // Schwergewichte starten
        int ANZAHL_SCHWERGEWICHTE = 2;

        for (int i = 0; i < ANZAHL_SCHWERGEWICHTE; i++) {
            Hamster ham = new Schwergewicht();
            ham.setPriority(Math.min((i + 1) * 2, Thread.MAX_PRIORITY));
            this.add(ham, this.getAnzahlSpalten() - 1, 1);
        }

        // Leichtgewichte starten
        int ANZAHL_LEICHTGEWICHTE = 3;

        for (int i = 0; i < ANZAHL_LEICHTGEWICHTE; i++) {
            Hamster ham = new Leichtgewicht();
            ham.setPriority(Math.min((i + 1) * 2, Thread.MAX_PRIORITY));
            this.add(ham, 0, 1);
        }
    }
}
```

Die Klasse `AllroundHamster` stellt nützliche Funktionen zur Verfügung, die von den Actor-Klassen `Schwergewicht` und `Leichtgewicht` genutzt werden.

```
/**
 * die Klasse erweitert den Befehlssatz eines normalen Hamsters um viele
 * nuetzliche Befehle
 *
 * @author Dietrich Boles (Universitaet Oldenburg)
 * @version 1.0 (12.12.2008)
 *
 */
public class AllroundHamster extends Hamster {
    /**
     * initialisiert einen neuen AllroundHamster mit den uebergebenen Werten
     *
     * @param r
     *            Reihe
     * @param s
     *            Spalte
     * @param b
     *            Blickrichtung
     * @param k
     *            Anzahl Koerner im Maul
```

```
 */
public AllroundHamster(int b, int k) {
    super(b, k);
}

/**
 * der Hamster dreht sich "anzahlDrehungen" mal um 90 Grad nach links
 *
 * @param anzahlDrehungen
 *            Anzahl der linksum-Drehungen
 */
public synchronized void linksUm(int anzahlDrehungen) {
    for (int i = 0; i < anzahlDrehungen; i++) {
        this.linksUm();
    }
}

/**
 * der Hamster dreht sich um 180 Grad
 */
public synchronized void kehrt() {
    this.linksUm(2);
}

/**
 * der Hamster dreht sich um 90 Grad nach rechts
 */
public synchronized void rechtsUm() {
    this.linksUm(3);
}

/**
 * der Hamster laeuft "anzahl" Schritte, maximal jedoch bis zur naechsten
 * Mauer; geliefert wird die tatsaechliche Anzahl gelaufener Schritte
 *
 * @param anzahl
 *            maximal zu laufende Schritte
 * @return tatsaechliche Anzahl gelaufener Schritte
 */
public synchronized int vor(int anzahl) {
    int schritte = 0;

    while (this.vornFrei() && (anzahl > 0)) {
        this.vor();
        schritte = schritte + 1;
        anzahl = anzahl - 1;
    }

    return schritte;
}

/**
 * der Hamster legt "anzahl" Koerner ab, maximal jedoch so viele, wie er im
 * Maul hat; geliefert wird die tatsaechliche Anzahl abgelegter Koerner
 *
 * @param anzahl
 *            maximal abzulegende Kerner
 * @return tatsaechliche Anzahl ablegter Koerner
 */
public synchronized int gib(int anzahl) {
    int abgelegteKoerner = 0;

    while (!this.maulLeer() && (anzahl > 0)) {
        this.gib();
        abgelegteKoerner = abgelegteKoerner + 1;
        anzahl = anzahl - 1;
    }

    return abgelegteKoerner;
}
```

```
/**
 * der Hamster frisst "anzahl" Koerner, maximal jedoch so viele, wie auf der
 * aktuellen Kachel liegen
 *
 * @param anzahl
 *            maximal aufzunehmende Koerner
 * @return tatsaechlich Anzahl aufgenommener Koerner
 */
public synchronized int nimm(int anzahl) {
    int gefresseneKoerner = 0;

    while (this.kornDa() && (anzahl > 0)) {
        this.nimm();
        gefresseneKoerner = gefresseneKoerner + 1;
        anzahl = anzahl - 1;
    }

    return gefresseneKoerner;
}

/**
 * der Hamster legt alle Koerner, die er im Maul hat, auf der aktuellen
 * Kachel ab; geliefert wird die Anzahl abgelegter Koerner
 *
 * @return Anzahl abgelegter Koerner
 */
public synchronized int gibAlle() {
    int abgelegteKoerner = 0;

    while (!this.maulLeer()) {
        this.gib();
        abgelegteKoerner = abgelegteKoerner + 1;
    }

    return abgelegteKoerner;
}

/**
 * der Hamster frisst alle Koerner auf der aktuellen Kachel; geliefert wird
 * die Anzahl gefressener Koerner
 *
 * @return Anzahl aufgenommener Koerner
 */
public synchronized int nimmAlle() {
    int gefresseneKoerner = 0;

    while (this.kornDa()) {
        this.nimm();
        gefresseneKoerner = gefresseneKoerner + 1;
    }

    return gefresseneKoerner;
}

/**
 * der Hamster laeuft bis zur naechsten Mauer; geliefert wird die Anzahl
 * ausgefuehrter Schritte
 *
 * @return Anzahl ausgefuehrter Schritte
 */
public synchronized int laufeZurWand() {
    int schritte = 0;

    while (this.vornFrei()) {
        this.vor();
        schritte = schritte + 1;
    }

    return schritte;
```

```
}

/**
 * der Hamster testet, ob links von ihm die Kachel frei ist
 *
 * @return true, falls die Kachel links vom Hamster frei ist, false sonst
 */
public synchronized boolean linksFrei() {
    this.linksUm();

    boolean frei = this.vornFrei();
    this.rechtsUm();

    return frei;
}

/**
 * der Hamster testet, ob rechts von ihm die Kachel frei ist
 *
 * @return true, falls die Kachel rechts vom Hamster frei ist, false sonst
 */
public synchronized boolean rechtsFrei() {
    this.rechtsUm();

    boolean frei = this.vornFrei();
    this.linksUm();

    return frei;
}

/**
 * der Hamster testet, ob hinter ihm die Kachel frei ist
 *
 * @return true, falls die Kachel hinter dem Hamster frei ist, false sonst
 */
public synchronized boolean hintenFrei() {
    this.kehrt();

    boolean frei = this.vornFrei();
    this.kehrt();

    return frei;
}

/**
 * der Hamster dreht sich so lange um, bis er in die uebergebene
 * Blickrichtung schaut
 *
 * @param richtung
 *            die Richtung, in die der Hamster schauen soll
 */
public synchronized void setzeBlickrichtung(int richtung) {
    while (this.getBlickrichtung() != richtung) {
        this.linksUm();
    }
}

/**
 * der Hamster laeuft in der Spalte, in der er gerade steht, zur angegebenen
 * Reihe; Voraussetzung: die Reihe existiert und es befinden sich keine
 * Mauern auf dem gewaehlten Weg
 *
 * @param reihe
 *            Reihe, in die der Hamster laufen soll
 */
public synchronized void laufeZuReihe(int reihe) {
    if (reihe == this.getReihe()) {
        return;
    }
```

```
    if (reihe > this.getReihe()) {
        this.setzeBlickrichtung(Hamster.SUED);
    } else {
        this.setzeBlickrichtung(Hamster.NORD);
    }

    while (reihe != this.getReihe()) {
        this.vor();
    }
}

/**
 * der Hamster laeuft in der Reihe, in der er gerade steht, zur angegebenen
 * Spalte; Voraussetzung: die Spalte existiert und es befinden sich keine
 * Mauern auf dem gewaehlten Weg
 *
 * @param spalte
 *            Spalte, in die der Hamster laufen soll
 */
public synchronized void laufeZuSpalte(int spalte) {
    if (spalte == this.getSpalte()) {
        return;
    }

    if (spalte > this.getSpalte()) {
        this.setzeBlickrichtung(Hamster.OST);
    } else {
        this.setzeBlickrichtung(Hamster.WEST);
    }

    while (spalte != this.getSpalte()) {
        this.vor();
    }
}

/**
 * der Hamster laeuft zur Kachel (reihe/spalte); Voraussetzung: die Kachel
 * existiert und es befinden sich keine Mauern im Territorium bzw. auf dem
 * gewaehlten Weg
 *
 * @param reihe
 *            Reihe der Zielkachel
 * @param spalte
 *            Spalte der Zielkachel
 */
public synchronized void laufeZuKachel(int reihe, int spalte) {
    this.laufeZuReihe(reihe);
    this.laufeZuSpalte(spalte);
}

/**
 * ueberprueft, ob auf der Kachel, auf der der Hamster aktuell steht,
 * mindestens eine bestimmte Anzahl an Koernern liegt
 *
 * @param anzahl
 *            Anzahl der geforderten Koerner
 * @return true, falls auf der aktuellen Kachel mindestens "anzahl"-Koerner
 *         liegen
 */
public synchronized boolean koernerDa(int anzahl) {
    return ((Territorium) this.getStage()).getAnzahlKoerner(this.getReihe(),
        this.getSpalte()) >= anzahl;
}

/**
 * liefert die Kachel, auf der der Hamster gerade steht; das Objekt kann als
 * Sperr-Objekt fuer Aktionen auf der entsprechenden Kachel genutzt werden
 * kann
 *
 * @return die Kachel, auf der der Hamster gerade steht
```

```
     */
    public synchronized Object getKachel() {
        return ((Territorium) this.getStage()).getKachel(this.getReihe(),
            this.getSpalte());
    }

    /**
     * Hamster schlaeft die uebergebene Zeit (in Millisekunden)
     *
     * @param millisekunden
     *            die zu schlafende Zeit
     */
    public synchronized void schlafen(int millisekunden) {
        try {
            Thread.sleep(millisekunden);
        } catch (InterruptedException exc) {
        }
    }
}

class Schwergewicht extends AllroundHamster {
    Schwergewicht() {
        super(Hamster.WEST, 0);
    }

    public void run() {
        while (true) {
            this.laufeZurBruecke();
            Bruecke.getBruecke().betretenSchwergewicht();
            this.laufeZumDuftendenKorn();

            // riechen
            this.verlasseDieBruecke();
            Bruecke.getBruecke().verlassenSchwergewicht();
            this.laufeZurueck();
        }
    }

    void laufeZurBruecke() {
        while (this.vornFrei()) {
            this.vor();
        }

        this.rechtsUm();
        this.vor();
        this.linksUm();
        this.vor();
    }

    void laufeZumDuftendenKorn() {
        do {
            this.vor();
        } while (!this.koernerDa(2));

        // duftendes Korn gefunden
    }

    void verlasseDieBruecke() {
        this.kehrt();

        while (this.kornDa()) {
            this.vor();
        }
    }

    void laufeZurueck() {
        this.vor();
        this.rechtsUm();
        this.vor();
        this.linksUm();
```

```
        while (this.vornFrei()) {
            this.vor();
        }

        this.kehrt();
    }
}

class Leichtgewicht extends AllroundHamster {
    Leichtgewicht() {
        super(Hamster.OST, 0);
    }

    public void run() {
        while (true) {
            this.laufeZurBruecke();
            Bruecke.getBruecke().betretenLeichtgewicht();
            this.laufeZumDuftendenKorn();

            // riechen
            this.verlasseDieBruecke();
            Bruecke.getBruecke().verlassenLeichtgewicht();
            this.laufeZurueck();
        }
    }

    void laufeZurBruecke() {
        while (this.vornFrei()) {
            this.vor();
        }

        this.linksUm();
        this.vor();
        this.rechtsUm();
        this.vor();
    }

    void laufeZumDuftendenKorn() {
        do {
            this.vor();
        } while (!this.koernerDa(2));

        // duftendes Korn gefunden
    }

    void verlasseDieBruecke() {
        this.kehrt();

        while (this.kornDa()) {
            this.vor();
        }
    }

    void laufeZurueck() {
        this.vor();
        this.linksUm();
        this.vor();
        this.rechtsUm();

        while (this.vornFrei()) {
            this.vor();
        }

        this.kehrt();
    }
}
```

Synchronisiert wird über ein Objekt der Klasse `Bruecke`.

```
class Bruecke {
    private static Bruecke bruecke = new Bruecke();
    private int anzahlSchwergewichte;
    private int anzahlLeichtgewichte;

    private Bruecke() {
        this.anzahlSchwergewichte = 0;
        this.anzahlLeichtgewichte = 0;
    }

    static Bruecke getBruecke() {
        return Bruecke.bruecke;
    }

    // Ein schwergewichtiger Hamster darf nur dann die Brücke
    // betreten, wenn die Anzahl der Schwergewichte
    // und die Anzahl der Leichtgewichte, die sich gerade
    // auf der Brücke befinden, gleich Null ist.
    synchronized void betretenSchwergewicht() {
        while (!((this.anzahlSchwergewichte == 0) &&
                 (this.anzahlLeichtgewichte == 0))) {
            try {
                this.wait();
            } catch (InterruptedException exc) {
            }
        }

        this.anzahlSchwergewichte++;
    }

    synchronized void verlassenSchwergewicht() {
        this.anzahlSchwergewichte--;
        this.notifyAll();
    }

    // Ein leichtgewichtiger Hamster darf die Brücke nicht
    // betreten, wenn die Anzahl der
    // Schwergewichte, die sich gerade auf der Brücke befinden,
    // größer als Null ist.
    synchronized void betretenLeichtgewicht() {
        while (this.anzahlSchwergewichte > 0) {
            try {
                this.wait();
            } catch (InterruptedException exc) {
            }
        }

        this.anzahlLeichtgewichte++;
    }

    synchronized void verlassenLeichtgewicht() {
        this.anzahlLeichtgewichte--;
        this.notifyAll();
    }
}
```

Auf der Basis der Klassen aus Abschnitt 8.3 können auf analoge Art und Weise auch andere Beispiele aus dem Buch „Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell“ schnell und einfach in Threadnocchio-Theaterstücke umgewandelt werden.