

Benutzungshandbuch

Objekt-Theater

Version 1.0 (08.09.2009)

Dietrich Boles

Universität Oldenburg

Inhaltsverzeichnis

1	Einleitung	7
1.1	Motivation	7
1.2	Voraussetzungen.....	7
1.3	Entwicklungshinweise.....	7
1.4	Aufbau des Benutzerhandbuch	8
2	Installation.....	9
2.1	Voraussetzungen.....	9
2.2	Download und Installation	9
2.3	Erste Schritte	9
3	Grundlagen	13
3.1	Analogien	13
3.2	Theaterstücke.....	13
3.3	Bühnen	13
3.4	Akteure	14
3.5	Requisiten	15
3.6	Aufführungen.....	15
3.7	Hilfsklassen	16
3.8	Run-Methode.....	16
4	Objekt-Theater-Simulator.....	17
4.1	Menüs.....	17
4.2	Toolbar	20

4.3	Klassenbereich	21
4.4	Bühnenbereich	22
4.5	Meldungsbereich	23
4.6	Editor	23
5	Theater-API.....	24
5.1	Stage	24
5.1.1	Gestaltungsmethoden.....	25
5.1.2	Getter-Methoden.....	25
5.1.3	Kollisionserkennungsmethoden	25
5.1.4	Event-Methoden.....	26
5.2	Component, Actor und Prop	26
5.2.1	Manipulationsmethoden.....	26
5.2.2	Getter-Methoden.....	27
5.2.3	Kollisionserkennungsmethoden	27
5.2.4	Event-Methoden.....	28
5.2.5	Akteure	28
5.3	Performance	28
5.4	Maus- und Tastatur-Events	29
5.4.1	KeyInfo.....	30
5.4.2	MouseInfo	30
5.5	TheaterImage und TheaterIcon	31
5.6	Kollisionserkennung	31
6	Referenzhandbuch.....	33
6.1	Stage	33

6.2	Component, Actor, Prop	42
6.2.1	Component	42
6.2.2	Actor	50
6.2.3	Prop	51
6.3	Performance	52
6.4	TheaterImage	55
6.5	TheaterIcon	60
6.6	Ereignisse	61
6.6.1	KeyInfo	61
6.6.2	MouseInfo	62
6.7	Kollisionserkennung	65
6.7.1	PixelArea	65
6.7.2	Rectangle	66
6.7.3	Point	68
6.7.4	Cell	70
6.7.5	CellArea	71
7	Literatur zur objektorientierten Programmierung	74
8	Beispiel-Theaterstücke	75
8.1	Beispiel-Theaterstück Hamster	75
8.2	Beispiel-Theaterstück Asteroids	75
8.3	Beispiel-Theaterstück Balloons	76
8.4	Beispiel-Theaterstück Piano	77
8.5	Beispiel-Theaterstück Mandelbrot	77
8.6	Beispiel-Theaterstück Curve	78

8.7	Beispiel-Theaterstück Tetris	78
8.8	Beispiel-Theaterstück Smess	79

1 Einleitung

Objekt-Theater ist ein Werkzeug zum Erlernen der objektorientierten Programmierung mit Java. 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 objektorientierte Programmierung hat sich heutzutage durchgesetzt. Programme bestehen aus Klassen, die die Struktur und das Verhalten bestimmter gleichartiger Objekte definieren. Diese interagieren durch Aufrufe von Methoden miteinander. Hauptvorteile der objektorientierten Programmierung sind eine hohe Wiederverwendbarkeit und eine einfache flexible Erweiterbarkeit existierender Programmteile.

Bei der Ausbildung von Programmierern, bspw. in der Schule oder in der Hochschule, stellen die Schüler bzw. Studierenden häufig die Frage: Was sind denn eigentlich Objekte, was ist der Unterschied zu Klassen? Und genau hier setzt das Werkzeug Objekt-Theater an: Objekte werden durch Icons auf dem Bildschirm repräsentiert. Methodenaufrufe für die Objekte führen zu Manipulationen der Icons. Dadurch wird das Ablaufverhalten von Programmen visualisiert.

1.2 Voraussetzungen

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

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

1.3 Entwicklungshinweise

Objekt-Theater ist ein Werkzeug des Programmier-Theaters. Andere Werkzeuge dieser Werkzeugfamilie zum Programmieren lernen sind Solist, eine Entwicklungsumgebung für Miniprogrammierwelt-Simulatoren, und Threadnocchio, ein Werkzeug zum Erlernen der parallelen Programmierung mit Java-Threads. Objekt-Theater ähnelt dabei deutlich Threadnocchio. Während im Objekt-Theater

Objekte durch Icons repräsentiert werden, werden in Threadnocchio Threads durch Icons visualisiert.

Die grundlegende Idee für das Programmier-Theater entstammt einem anderen Werkzeug, nämlich Greenfoot (www.greenfoot.org). In Greenfoot werden ebenfalls Objekte durch Icons repräsentiert. Der Unterschied zwischen Greenfoot und Objekt-Theater besteht dabei im Wesentlichen in einer anderen Programmier-API. Die zugrundeliegende Arbeitsweise der beiden Werkzeuge ist jedoch identisch.

1.4 Aufbau des Benutzerhandbuch

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

2 Installation

2.1 Voraussetzungen

Voraussetzung zum Starten von Objekt-Theater 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

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

Beim ersten Start sucht Objekt-Theater 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 Objekt-Theater-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 Objekt-Theater-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 Objekt-Theater zu verstehen. Spielen Sie danach am besten zunächst einmal mit ein paar Beispielszenarien herum, die sich im Unterordner *plays* im Objekt-Theater-Ordner befinden. Gehen Sie dabei folgendermaßen vor:

Doppelklicken Sie die Datei *objekttheater.jar* im Objekt-Theater-Ordner. Damit wird der Objekt-Theater-Simulator gestartet. Zunächst erscheint ein Startbild (siehe Abbildung 2.1), dann das Hauptfenster des Objekt-Theater-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 *hamster* (siehe Abbildung 2.3). Es öffnet sich ein neues Hauptfenster mit dem Hamster-Theaterstück. Das andere Hauptfenster können Sie schließen.

Öffnen Sie nun im Dateibaum des Hamster-Fensters den Ordner *Actor*. Aktivieren Sie oberhalb vom Eintrag *Hamster* das Popup-Menü und wählen Sie im Popup-Menü den Eintrag *new Hamster()*. Klicken Sie dann irgendwo in das Hamster-Territorium, das Bild mit den Kacheln (die Bühne). Dort wird dann ein Hamster platziert. Wiederholen Sie dies ein paar Mal.

Führen Sie dieselbe Aktion im Dateibaumordner *Prop* für den Eintrag *Mauer* oder *Korn* aus und platzieren Sie Mauern und Körner im Hamster-Territorium (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 Hamster laufen los. Mauern weichen sie aus, auf Kacheln mit Körnern fressen sie jeweils ein Korn.

Sie können auch während einer Simulation weitere Hamster, Mauern und Körner im Territorium platzieren.

Durch Doppelklick auf Hamster, Mauer oder Korn im Dateibaum öffnen Sie jeweils einen Editor und können sich die Implementierung der beiden Klassen auf der Basis der Theater-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 Hamster, Mauern und Körner erzeugen und eine neue Simulation starten.



Abb. 2.1: Startbild des Objekt-Theater-Simulators

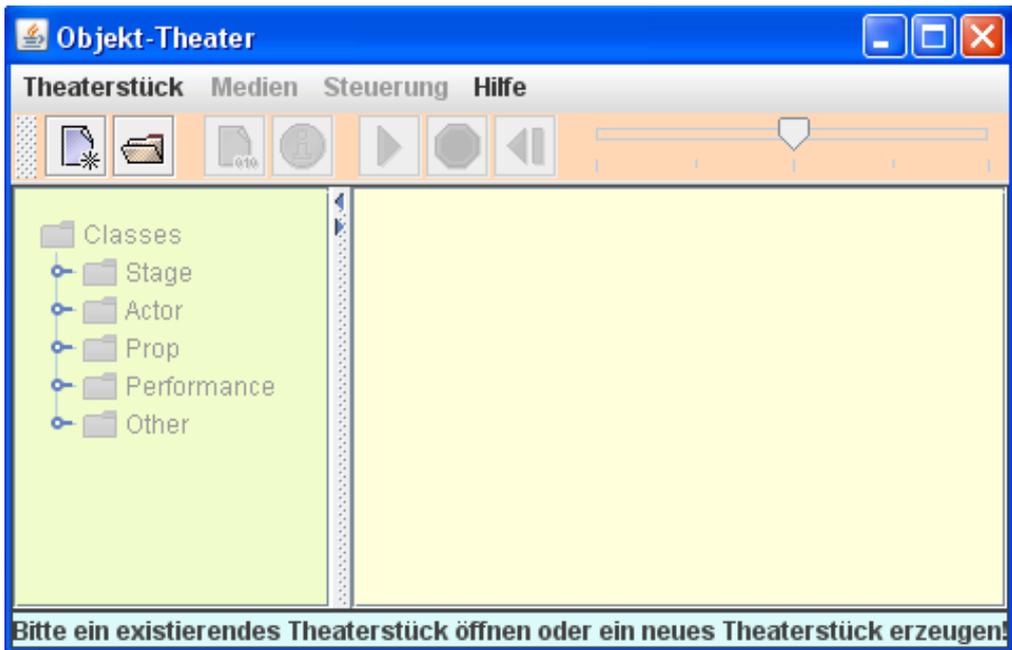


Abb. 2.2: Hauptfenster des Objekt-Theater-Simulators

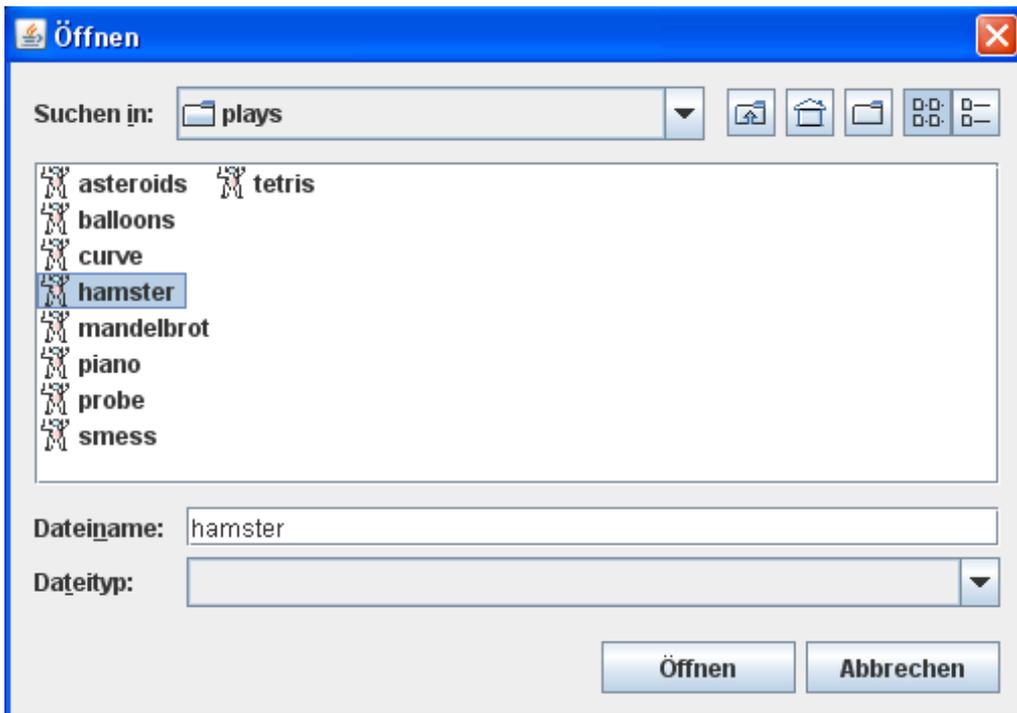


Abb. 2.3: Auswahl eines existierenden Theaterstückes

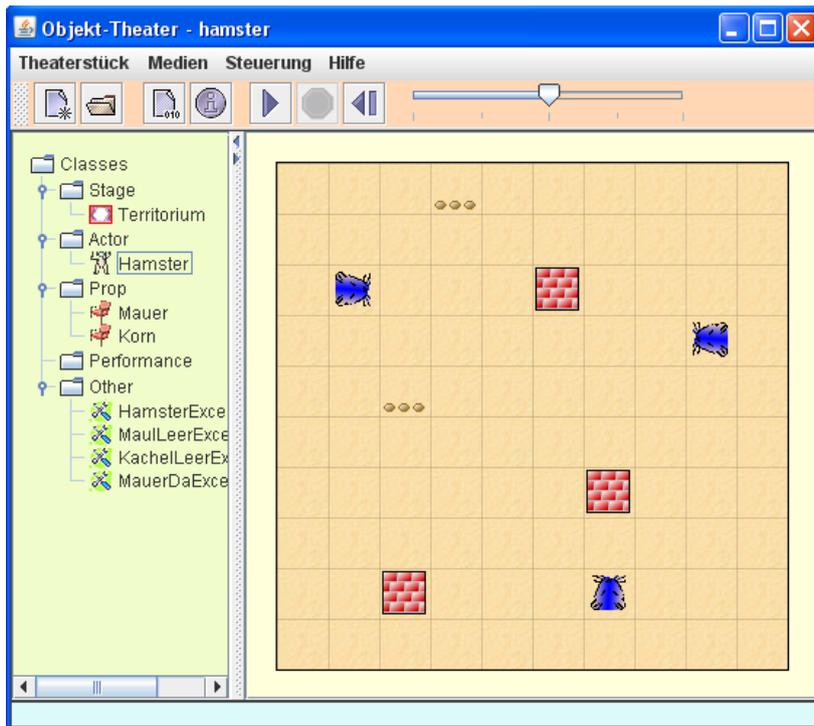


Abb. 2.4: Hauptfenster mit Hamster-Theaterstück

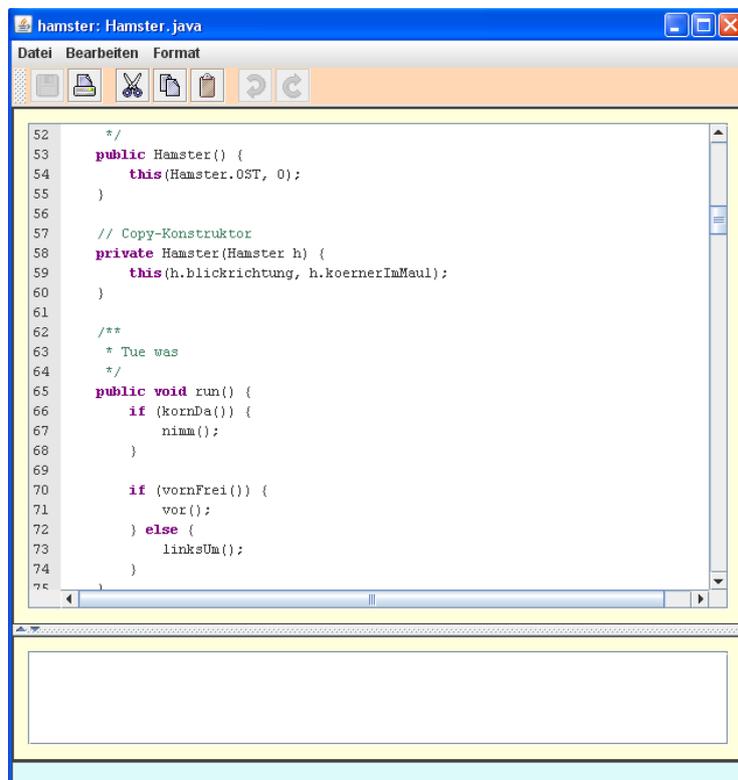


Abb. 2.5: Editor mit der Actor-Klasse Hamster

3 Grundlagen

Objekt-Theater führt in die objektorientierte Programmierung mit Java ein. In Klassen werden die Struktur und das Verhalten gleichartiger Gebilde, so genannter Objekte definiert.

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“.

3.2 Theaterstücke

Ein objektorientiertes Programm wird in Objekt-Theater als *Theaterstück* (Play) bezeichnet. Ein Theaterstück ist dabei gleichbedeutend mit dem Begriff „Projekt“, der in vielen Programmierungsumgebungen benutzt wird. Ein Theaterstück setzt sich aus einer Menge von Klassen zusammen, die im Allgemeinen von durch Objekt-Theater zur Verfügung gestellten Klassen – die so genannten Theater-Klassen – abgeleitet werden.

3.3 Bühnen

Handlungsumfeld in Objekt-Theater 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).

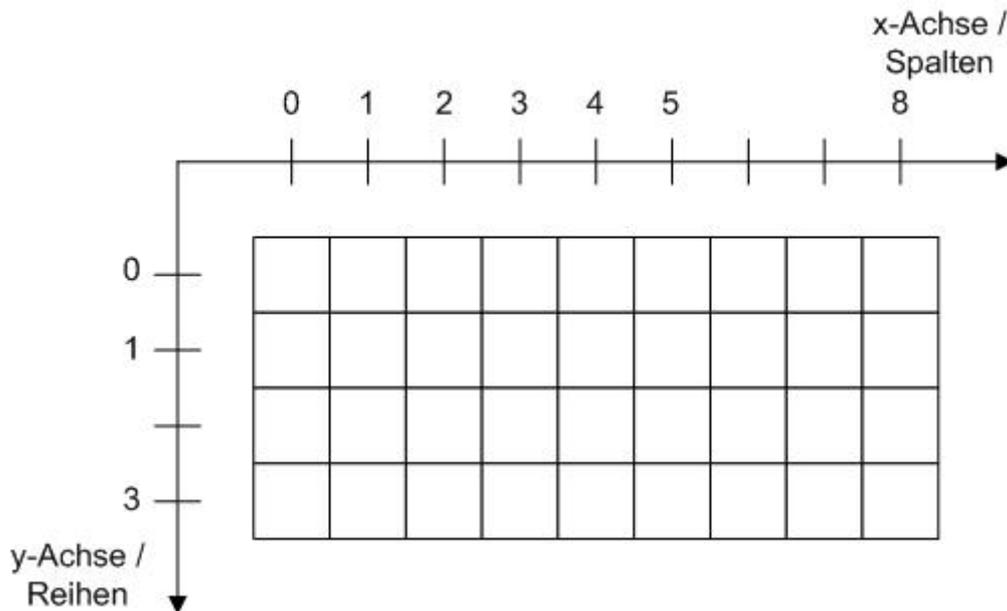


Abb. 3.1: Bühnenaufbau

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 Objekt-Theater repräsentiert als Klassen, die von der Theater-Klasse `Stage` abgeleitet werden. Ein 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 im Dateibaum 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

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

Akteuren kann ein Icon 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 Icons dabei auf den Mittelpunkt der Zelle gesetzt.

Programmiertechnisch handelt es sich bei den Akteuren um Objekte. Die Klasse `Actor` ist dazu von der Java-Klasse `Object` abgeleitet. Das Eigenleben eines Akteurs wird 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-Drop-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) Icons 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 Icon - auf der Bühne befinden. Im Unterschied zu Akteuren sind Requisiten jedoch eher passive Objekte, d.h. sie besitzen keine `run`-Methode, in der Aktivitäten definiert werden können. Und das ist auch schon der einzige Unterschied zwischen Akteuren und Requisiten. Programmiertechnisch sind Requisiten nämlich Objekte von Klassen, die von der Theater-Klasse `Prop` abgeleitet sind. Sowohl die Klasse `Actor` als auch die Klasse `Prop` sind aber von der Theater-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 Icon 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 Objekt-Theater mit der *Aufführung* des Theaterstücks (*Performance*) assoziiert. Normalerweise wird dabei eine Standard-

Aufführung genutzt. Objekt-Theater-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 Theater-Klasse *Performance*. Welche Aufführung dann tatsächlich genutzt wird, kann vor dem Start der Aufführung durch einen einfachen Mausklick auf die entsprechende Klasse im Dateibaum 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*. Die Methoden werden unmittelbar nach der entsprechenden Steuerungsaktion ausgeführt, auch wenn diese durch die Steuerungselemente im Objekt-Theater-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 Theater-Klassen abgeleiteten Klassen lassen sich weitere Hilfsklassen definieren und in die Programme einbinden. Diese Klassen werden als *Hilfsklassen* bezeichnet.

3.8 Run-Methode

In Klassen, die von der Theater-Klasse *Actor* abgeleitet sind, kann eine Methode *public void run* definiert werden, die das Eigenleben eines entsprechenden Akteurs definiert. Wird eine Aufführung durch Anklicken des *run*-Buttons in der Toolbar gestartet, passiert folgendes: Für jeden Akteur, der sich aktuell auf der Bühne befindet, wird nacheinander (in einer bestimmten Reihenfolge), seine *run*-Methode aufgerufen.

4 Objekt-Theater-Simulator

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

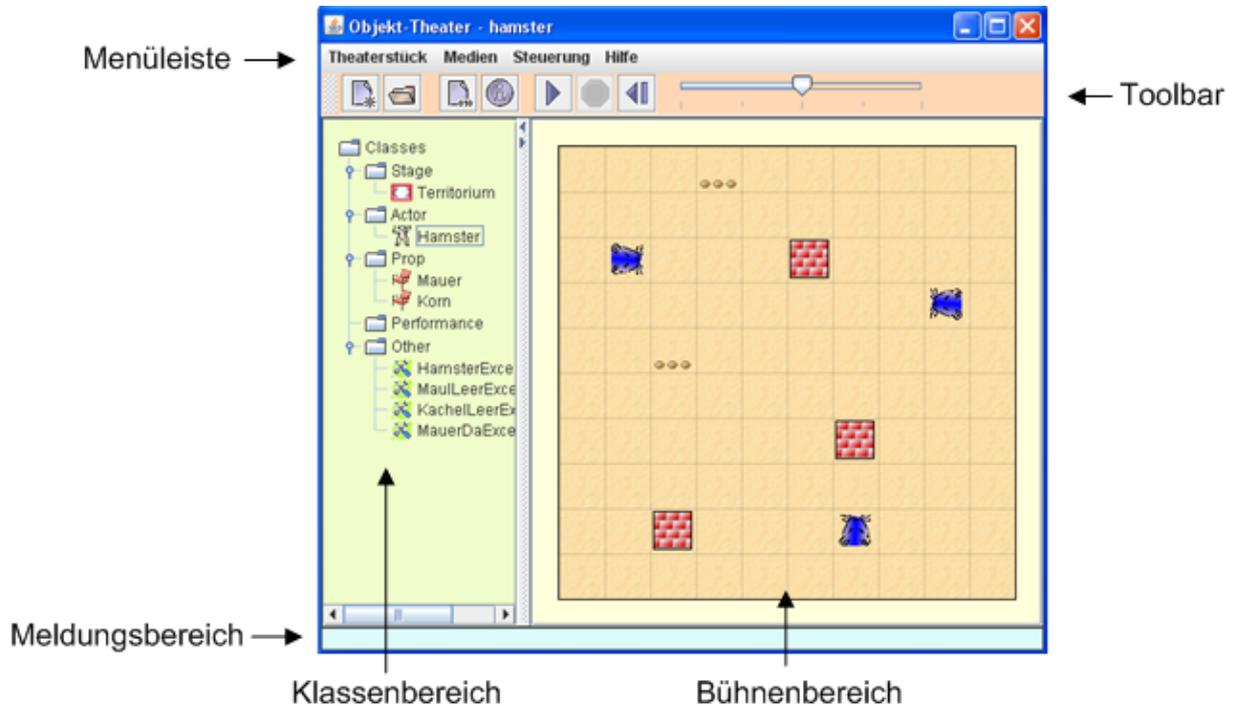


Abb. 4.1: Aufbau des Objekt-Theater-Simulators

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 Objekt-Theater-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 Menü *Theaterstück* können Sie neue Theaterstücke erzeugen, existierende Theaterstücke öffnen sowie die Bearbeitung des angezeigten Objekt-Theater-Theaterstücks beenden (siehe Abbildung 4.2).



Abb. 4.2: Theaterstück-Menü

Weiterhin ist es über den Menüeintrag *Exportieren* möglich, Theaterstücke als Standalone-Programme zu generieren. Nach Auswahl eines entsprechenden Verzeichnisses und eines optionalen Startbildes werden in dem angegebenen Verzeichnis zwei Dateien *performance.jar* und *performance.html* erzeugt. (Hinweis: Die Namen der beiden Dateien dürfen nicht verändert werden!) Durch Doppelklick auf die Datei *performance.jar* bzw. Aufruf via `java -jar performance.jar` wird die Standalone-Anwendung gestartet. Die Datei *performance.html* lädt die Datei *performance.jar* als Applet. Der Objekt-Theater-Ordner enthält ein Unterverzeichnis *exported_plays*, in dem für die beigefügten Beispiel-Theaterstücke entsprechende Standalone-Programme erzeugt wurden. Abbildung 4.3 demonstriert dies für das Beispiel-Theaterstück *smess*. Über die Toolbar können Aufführungen des Theaterstücks gesteuert werden.

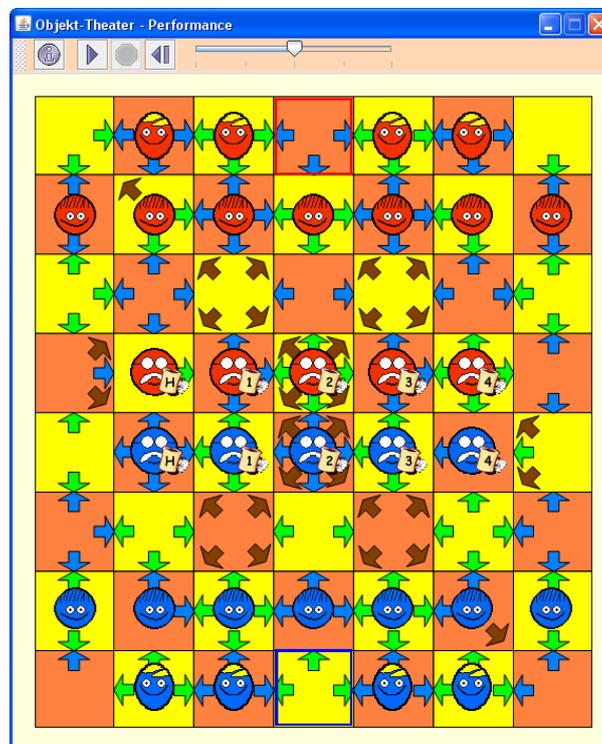


Abb. 4.3: Generierte Standalone-Anwendung

Das Menü *Medien* unterstützt den Import von Bild- und Ton-Dateien in die Unterordner *images* bzw. *sounds* des entsprechenden Theaterstücks (siehe Abbildung 4.4). Nach der Auswahl des entsprechenden Eintrags erscheint eine Dateiauswahlbox, über die Sie die zu importierende(n) Datei(en) auswählen können. Alternativ können Sie natürlich auch auf der Ebene des Betriebssystems die Dateien in die Unterordner kopieren.



Abb. 4.4: Medien-Menü

Das Menü *Steuerung* stellt Funktionen zur Steuerung der Aufführung eines Theaterstücks zur Verfügung (siehe Abbildung 4.5). Hier werden Sie aber höchstwahrscheinlich eher die entsprechenden Buttons in der Toolbar benutzen.



Abb. 4.5: Steuerung-Menü

Über das Menü *Hilfe* haben Sie insbesondere die Möglichkeit, dieses Benutzerhandbuch zu öffnen. Weiterhin haben Sie Zugriff auf die detaillierte Referenz der Theater-API im javadoc-Format (siehe auch Abbildung 4.6).

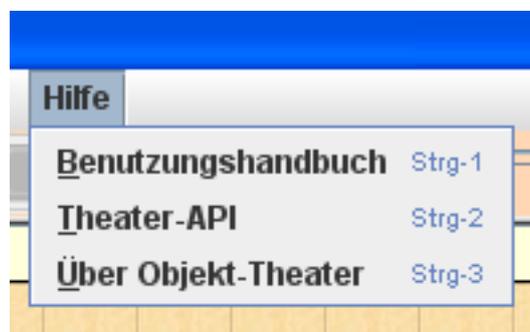


Abb. 4.6: Hilfe-Menü

4.2 Toolbar

Über die Toolbar sind die wichtigsten Funktionalitäten des Simulators durch einen einfachen Mausklick ausführbar. Abbildung 4.7 skizziert den Aufbau der Toolbar.

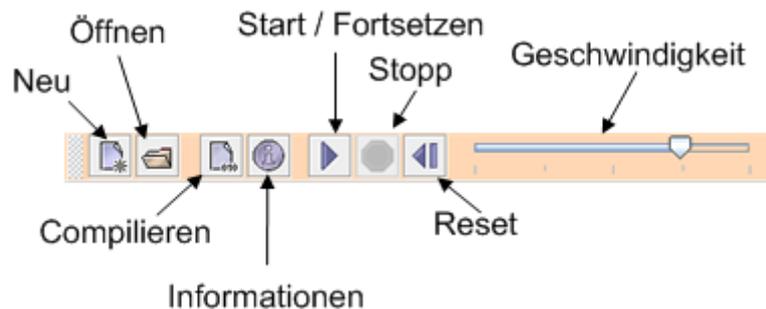


Abb. 4.7: Aufbau der Toolbar

- *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 existierende Theaterstück 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 Compilerbutton erscheint mit rotem Hintergrund, wenn Klassen geändert und gespeichert aber noch nicht neu kompiliert wurden oder wenn noch nicht beseitigte Compilerfehler 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. Was passiert ist, das fortlaufend hintereinander die run-Methode aller Akteure, die sich auf der Bühne befinden, ausgeführt wird.
- *Stopp*: Steuerungsbutton zum Anhalten einer Aufführung.
- *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-Menü (siehe Abbildung 4.8). 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.

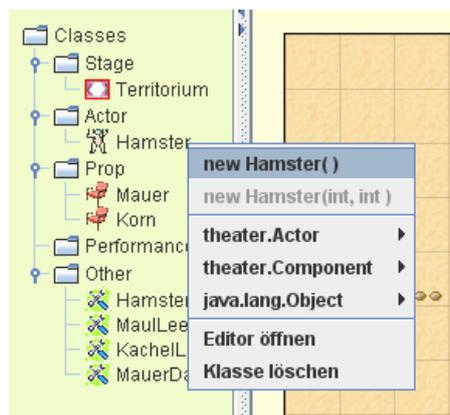


Abb. 4.8: Klassen-Popup-Menü

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 Objekt-Theater 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-Menü zu aktivieren. In diesem Popup-Menü erscheinen alle public-Methoden der entsprechenden Klasse und können interaktiv für das entsprechende Objekt ausgeführt werden (siehe Abbildung 4.9).



Abb. 4.9: Komponenten-Popup-Menü

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 4.10 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 Compilerfehler im Compilerfehlerbereich. Durch Anklicken der Fehlermeldung wird im Programmbereich an die entsprechende Stelle gesprungen. Im Meldungsbereich ganz unten im Editorfenster werden Hinweise und Benutzungsfehler angezeigt.

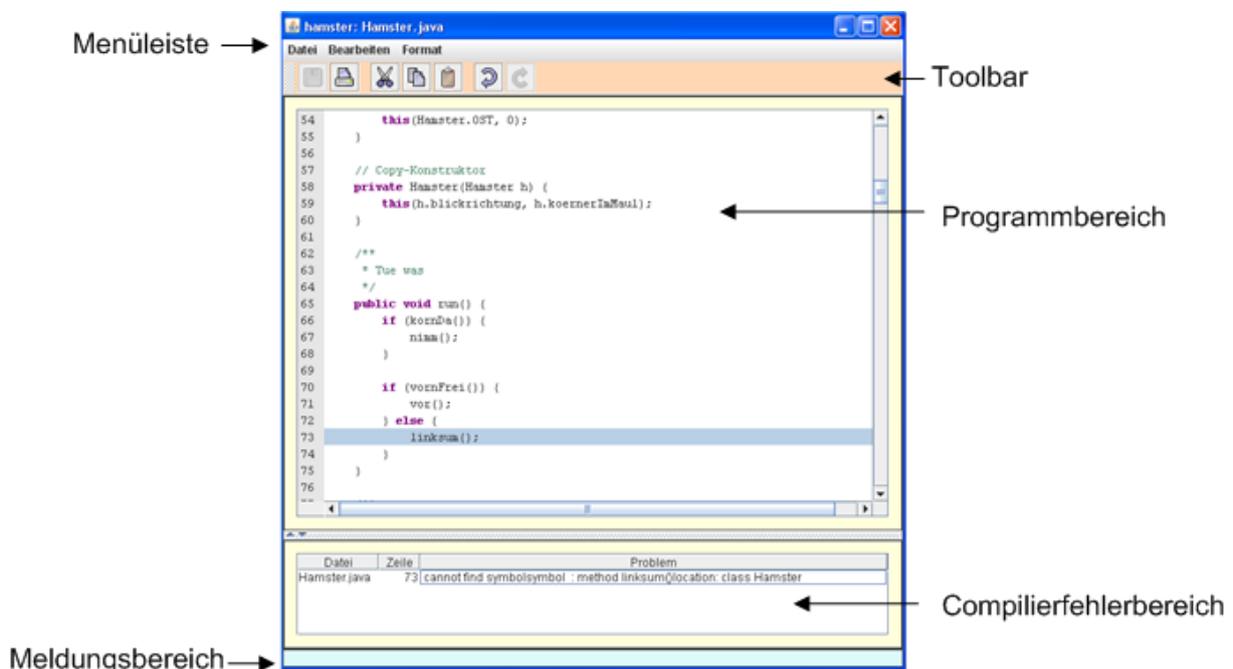


Abb. 4.10: Aufbau des Editors

5 Theater-API

Die Theater-API oder –Klassenbibliothek besteht aus einer Menge von Klassen und Interfaces, auf deren Grundlage Objekt-Theater-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
- **TheaterIcon**: Klasse zum Erzeugen und Manipulieren von animierten GIF-Bildern
- **MouseInfo**: Klasse, die Informationen zu Maus-Ereignissen verwaltet
- **KeyInfo**: Klasse, die Informationen zu Tastatur-Ereignissen verwaltet
- **PixelArea**: Interface zur Umsetzung der Kollisionserkennung
- **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 Objekt-Theater-internen Änderung der Bühne während einer Aufführung wird unmittelbar automatisch die geänderte Bühne im Objekt-Theater-Simulator neu gezeichnet. Für Änderungen, auf die Objekt-Theater 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`, `keyReleased`, `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 `Object` abgeleitet. 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 Methode `run` überschreiben, also ein Eigenleben 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 Umplatzen 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 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 Ermittlung 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`, `keyReleased`, `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 einer Aufführung werden die `run`-Methoden aller Akteure, die sich auf der Bühne befinden, nacheinander wiederholt aufgerufen (daher sollten Schleifen innerhalb der `run`-Methode vermieden werden). Befinden sich mehrere Akteure auf der Bühne, ist die Reihenfolge des Aufrufs ihrer `run`-Methoden undefiniert. Mittels der Methode `setPriority` kann jedoch Akteuren eine Priorität zugeordnet werden, für die gilt: `run`-Methoden von Akteuren mit einer höheren Priorität werden vor den `run`-Methoden von Akteuren mit einer niedrigeren Priorität aufgerufen.

5.3 Performance

Die Klasse `Performance` definiert Methoden zur Steuerung und Verwaltung der Ausführung von Objekt-Theater-Programmen:

- `stop`: Stoppt 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`
- `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 Theaterstücks 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. 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`, `keyReleased`, `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`. 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 Icon 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 und TheaterIcon

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

Initialisiert wird ein `TheaterImage` über entsprechende Konstruktoren entweder mittels einer Bilddatei, eines bereits existierenden `TheaterImage`s 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), `setTransparency` (legt die Transparenz 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.

`TheaterIcon` ist eine von der Klasse `TheaterImage` abgeleitete Klasse für die Verwendung von Animated-GIF-Bildern. `TheaterIcon`-Objekte können jedoch nicht manipuliert werden. Die entsprechenden Methoden sind jeweils leer überschrieben.

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.

Objekt-Theater 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 Icon 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 (08.09.2009)
 */
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)

    /**
     * Muss überschrieben werden und die Bühnen-spezifischen Aktionen enthalten

```

```

*/
public void run()

/**
 * 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()

/**
 * Ändert die Größe der Bühne
 *
 * @param noOfCols
 *         Anzahl an Spalten
 * @param noOfRows
 *         Anzahl an Reihen
 */
public void setSize(int noOfCols, int noOfRows)

/**
 * Ändert die Größe der Bühne
 *
 * @param noOfCols
 *         Anzahl an Spalten
 * @param noOfRows
 *         Anzahl an Reihen
 * @param cellSize
 *         Größe jeder Zelle
 */
public void setSize(int noOfCols, int noOfRows, int cellSize)

/**
 * 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>
 * Seiteneffekt: Nach erfolgreicher Platzierung der Komponente auf der Bühne
 * wird die Methode addToStage der Komponente mit dem entsprechenden
 * Stage-Objekt als Parameter aufgerufen.
 * </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 (darf

```

```

*          nicht null sein)
* @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.
*
* @param comp
*          die Komponente, die von der Bühne entfernt werden soll
*/
public void remove(Component comp)

/**
* Entfernt alle Komponenten der Liste von der Bühne.
*
* @param components
*          enthält die Komponenten, die von der Bühne entfernt werden
*          sollen
*/
public void remove(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
*          image des Theaterstück-Verzeichnisses befinden
* @throws IllegalArgumentException
*          wird geworfen, wenn die angegebene Datei keine gültige
*          lesbare Bilddatei ist
*/
public final 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 final void setBackground(TheaterImage image)

/**
* Liefert das Hintergrundbild der Bühne als TheaterImage-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 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 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 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 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 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 Methode run überschreiben 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
 * umplatzen 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 (08.09.2009)
 */
public class Component extends Object 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, falls es sich um einen Akteur handelt; false, falls es
     *         sich um eine Requisite handelt
     */
    protected Component(boolean isActor)

    /**

```

```

    * Muss überschrieben werden und die Actor-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 von der
 * Bühne diese Methode addToStage 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 addToStage(Stage stage)

/**
 * Mit Hilfe der add-Methode der Klasse Stage können Komponente in einer
 * bestimmten Spalte und Reihe auf der 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 einer 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 der
*         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 der
*         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 der
 *         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>
 * Die Methode wird nur aufgerufen, wenn der HandlingKeyEvents-Status
 * gesetzt ist (was standardmäßig der Fall ist).
 * </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 alle Methoden
* ihrer Oberklasse Component.
* <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 Klasse Component genutzt werden können.
*
* @author Dietrich Boles, Universität Oldenburg, Germany
* @version 1.0 (08.09.2009)
*
*/
public class Actor extends Component {

    /**
     * Default-Priorität eines Akteurs
     */
    public final static int DEF_PRIORITY = 0;

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

    /**
     * Muss überschrieben werden und die Actor-spezifischen Aktionen enthalten
     *
     * @see theater.Component#run()
     */
    public void run()

    /**
     * Ordnet dem Akteur eine neue Priorität zu. Diese spielt bei der Ausführung
     * der run-Methoden eine Rolle. Je höher die Priorität eines Akteurs ist,
     * desto eher wird seine run-Methode aufgerufen. Bei Akteuren gleicher
     * Priorität ist die Reihenfolge der Ausführung unbestimmt.
     *
     * @param priority
     *         die Priorität
     */
    final public void setPriority(int priority)

    /**
     * Liefert die Priorität des Akteurs
     *
     * @return die Priorität des Akteurs
     */
    final public int getPriority()
}

```

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 (08.09.2009)
 *
*/
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, dass 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 Objekt-Theater-Programmen(stop, 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,
 * 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 Theaterstücks 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 (08.09.2009)
 */
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()

/**
 * Diese Methode wird unmittelbar nach dem Starten bzw. Fortsetzen 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 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()

/**
 * Wechselt die Bühne. Der Bühnenwechsel wird nicht unmittelbar beim Aufruf
 * der Methode sondern erst nach Abschluss der run-Runde durchgeführt.
 *
 * @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. 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.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (08.09.2009)
 */
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 Icons 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, jpg 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) {
        this.image = new BufferedImage(width, height,
            BufferedImage.TYPE_INT_ARGB);
        this.init();
    }
}
```

```

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

/**
 * interner Default-Konstruktor
 */
protected TheaterImage()

/**
 * 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()

/**
 * Setzt die Transparenz; Standardmaessig ist der Transparenzwert 255 (gar
 * nicht durchsichtig)
 *
 * @param t
 *         Wert zwischen 0 (ganz durchsichtig) und 255 (gar nicht
 *         durchsichtig)
 */
public void setTransparency(int t)

/**
 * Liefert die Transparenz
 *
 * @return die eingestellte Transparenz; Wert zwischen 0 (ganz durchsichtig)
 *         und 255 (gar nicht durchsichtig)
 */
public int getTransparency()

/**
 * 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 eine bestimmte 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 getAwtImage()
}

```

6.5 TheaterIcon

```

package theater;

/**
 * TheaterIcon ist eine von der Theater-Klasse TheaterImage abgeleitete Klasse,
 * die die Verwendung von Animated-GIF-Ikons ermöglicht.
 * <p>
 * </p>
 * Achtung: Die meisten Methoden dieser Klasse überschreiben die von der Klasse
 * TheaterImage geerbten Methoden als leere Methoden, da Animated-GIFs nicht
 * manipuliert werden können!
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (08.09.2009)
 */
public class TheaterIcon extends TheaterImage {

    /**
     * Konstruktor zum Initialisieren eines TheaterIcons mit einem Bild aus
     * einer Datei. Erlaubte Bildformate sind gif, jpg 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 TheaterIcon(String filename) throws IllegalArgumentException

    /**
     * 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()

    /**
     * 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 Image-Objekt
     */
    public java.awt.Image getAwtImage()
}

```

6.6 Ereignisse

6.6.1 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 keyReleased. 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 (08.09.2009)
 */
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
}

```

6.6.2 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 (08.09.2009)
*/
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 Ikon.
 *
 * @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 Ikon.
 *
 * @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 Ikon.
 *
 * @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)
}

```

6.7 Kollisionserkennung

6.7.1 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 (08.09.2009)
 */
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);
}

```

6.7.2 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 (08.09.2009)
 */
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()
}

```

6.7.3 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 (08.09.2009)
 */
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 Klasse 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()

}

```

6.7.4 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 (08.09.2009)
 */
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()
}

```

6.7.5 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 (08.09.2009)
 */
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 objektorientierten Programmierung

Objekt-Theater ist ein Werkzeug, das durch den Einsatz von Visualisierungstechniken Programmierern beim Erlernen der objektorientierten Programmierung hilft. Objekt-Theater selbst ist kein Lehrbuch für die objektorientierte Programmierung. Es gibt jedoch eine Menge von Lehrbüchern für die objektorientierte Programmierung, von denen die meines Erachtens geeignetsten im Folgenden aufgelistet werden.

- Boles, D. und Boles, C.: „Objektorientierte Programmierung spielend gelernt mit dem Java-Hamster-Modell“, Vieweg+Teubner.
- Ratz, D., Scheffler, J., Seese, D. und Wiesenberger, J.: „Grundkurs Programmieren in Java, Band 1“, Hanser-Verlag.
- Heinisch, C., Müller, F. und Goll, J.: „Java als erste Programmiersprache“, Vieweg+Teubner.

8 Beispiel-Theaterstücke

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

8.1 Beispiel-Theaterstück Hamster

Das Theaterstück hamster bildet das Java-Hamster-Modell (siehe www.java-hamster-modell.de) für Objekt-Theater nach. Das Java-Hamster-Modell ist eine so genannte Miniprogrammierwelt. In einem Territorium leben Hamster, die mit Hilfe von Befehlen wie `vor` und `linksUm` durch das Territorium gesteuert werden können und dabei Körner aufnehmen und abgeben können.

Die `run`-Methode der Actor-Klasse Hamster ist nur default-mäßig implementiert. Sie muss an entsprechende Hamster-Aufgaben angepasst werden.

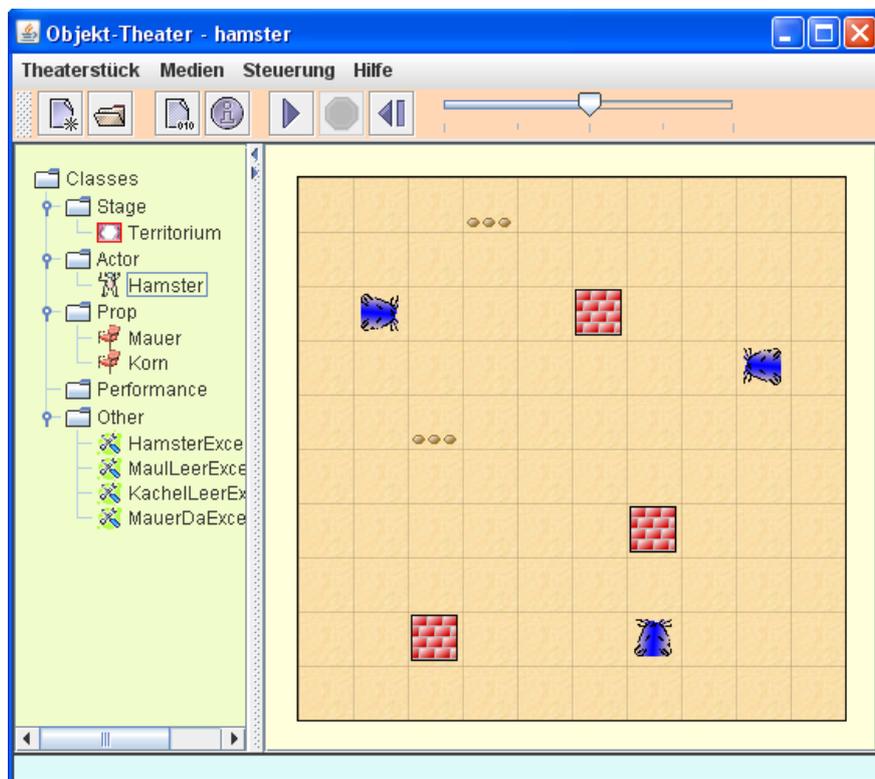


Abb. 8.1: Objekt-Theater-Simulator mit Hamster-Theaterstück

8.2 Beispiel-Theaterstück Asteroids

Asteroids ist ein Standard-Szenario von Greenfoot (www.greenfoot.org). Die Klassen wurden weitgehend von Greenfoot übernommen und lediglich an die Theater-API angepasst. Der Benutzer kann mittels der Tastatur ein Raumschiff steuern, Schüsse abgeben und muss versuchen, Felsbrocken auszuweichen.

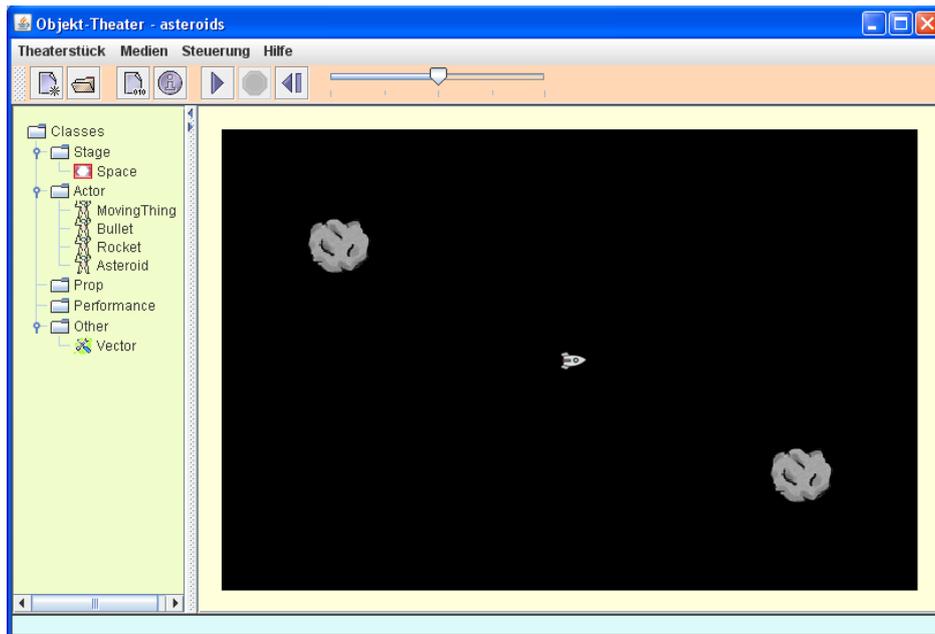


Abb. 8.2: Objekt-Theater-Simulator mit Asteroids-Theaterstück

8.3 Beispiel-Theaterstück *Balloons*

Balloons ist ein Standard-Szenario von Greenfoot (www.greenfoot.org). Die Klassen wurden weitgehend von Greenfoot übernommen und lediglich an die Theater-API angepasst. Der Benutzer kann mittels der Maus Ballons zum Platzen bringen.

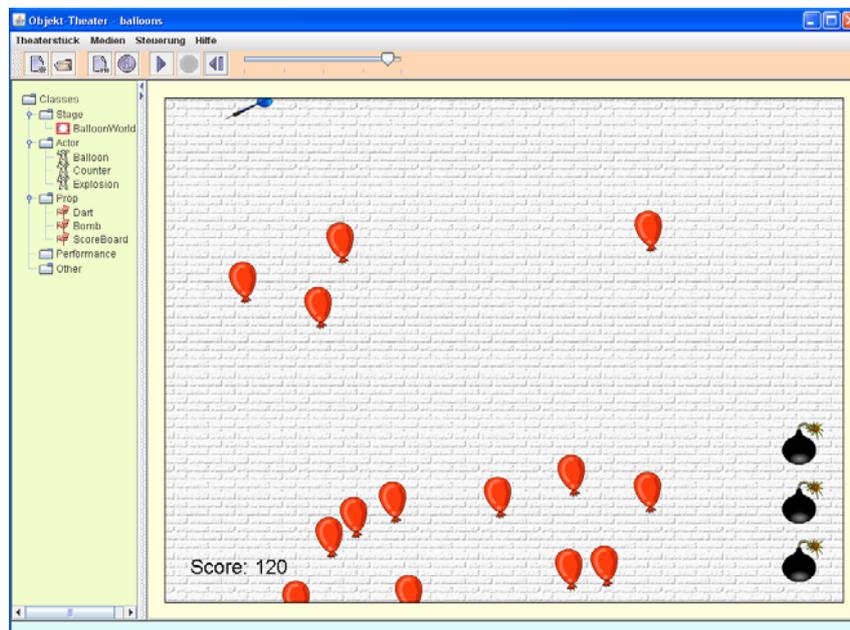


Abb. 8.3: Objekt-Theater-Simulator mit Balloons-Theaterstück

8.4 Beispiel-Theaterstück Piano

Piano ist ein Standard-Szenario von Greenfoot (www.greenfoot.org). Die Klassen wurden weitgehend von Greenfoot übernommen und lediglich an die Theater-API angepasst. Der Benutzer kann mittels der Tastatur Klavier spielen.

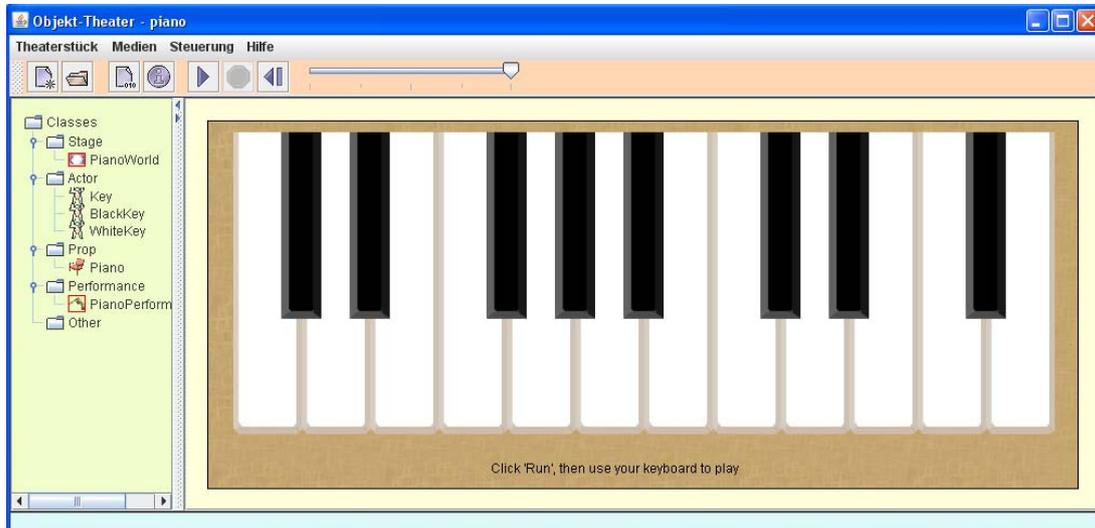


Abb. 8.4: Objekt-Theater-Simulator mit Piano-Theaterstück

8.5 Beispiel-Theaterstück Mandelbrot

Das Theaterstück Mandelbrot stellt die Mandelbrot-Menge grafisch dar. Mit Hilfe der Maus kann in die Menge rein und aus der Menge raus gezoomt werden.

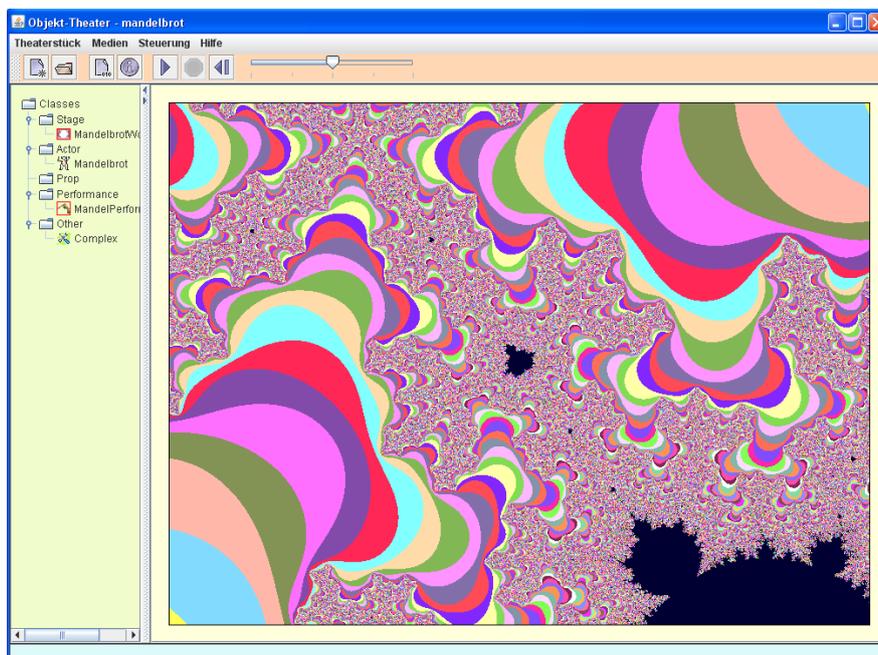


Abb. 8.5: Objekt-Theater-Simulator mit Mandelbrot-Theaterstück

8.6 Beispiel-Theaterstück Curve

Das Theaterstück Curve ist ein Multi-Player-Game, bei dem die Spieler versuchen müssen, Kurven zu zeichnen und die anderen Spieler beim Kurveichnen zu stören.

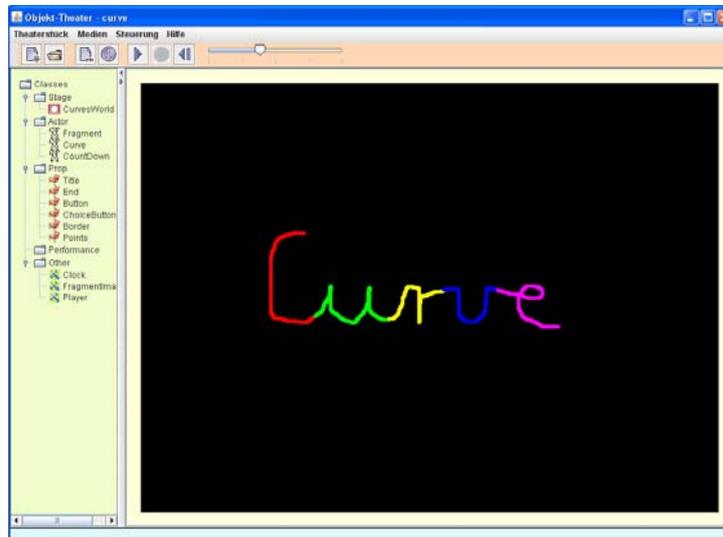


Abb. 8.6: Objekt-Theater-Simulator mit Curve-Theaterstück

8.7 Beispiel-Theaterstück Tetris

Das Theaterstück Tetris bildet das bekannte Spiel Tetris nach.

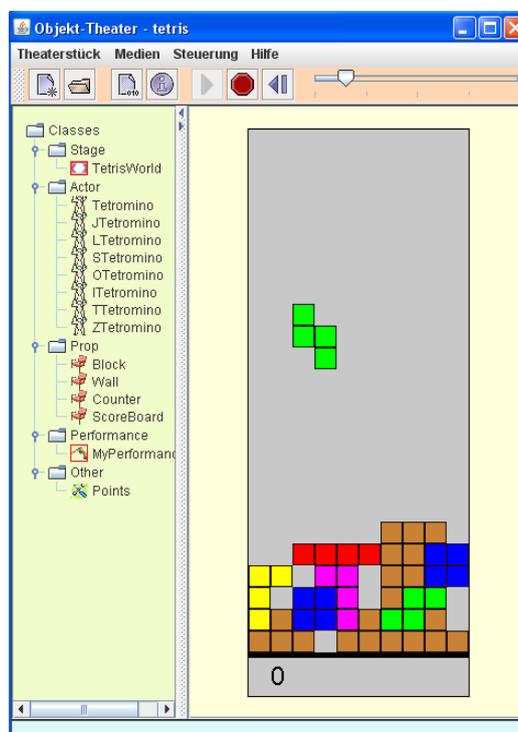


Abb. 8.7: Objekt-Theater-Simulator mit Tetris-Theaterstück

8.8 Beispiel-Theaterstück Smess

Smess ist ein Schach-ähnliches Spiel für Kinder. Mit Hilfe des Theaterstücks Smess ist es möglich, gegen andere Spieler oder den Computer Smess zu spielen.

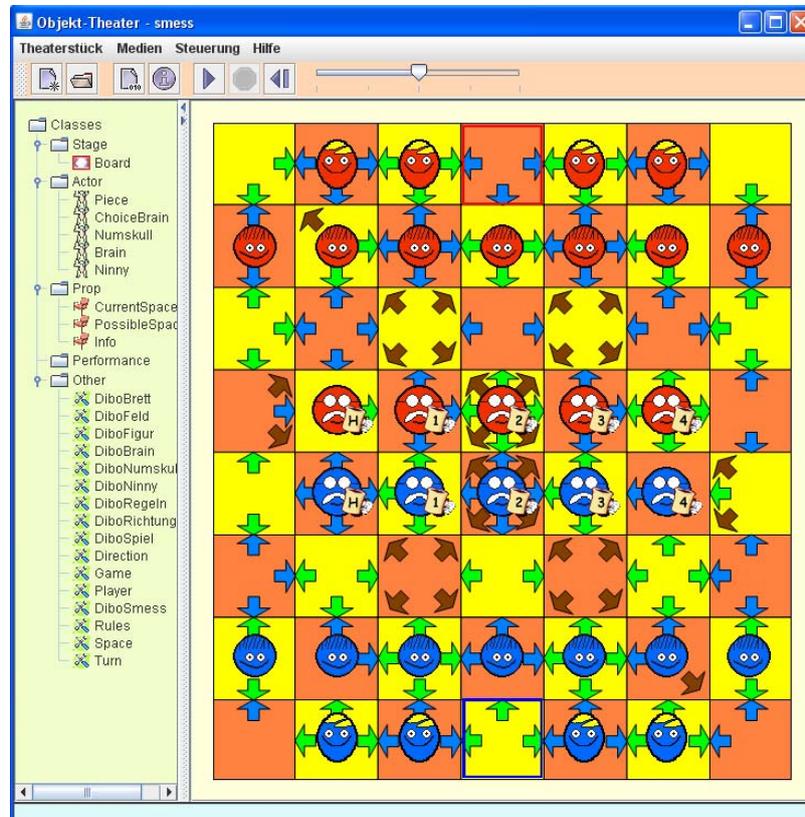


Abb. 8.8: Objekt-Theater-Simulator mit Smess-Theaterstück