# Benutzungshandbuch

# Solist

Version 2.0 (26.08.2010)

Dietrich Boles

Universität Oldenburg

# Inhaltsverzeichnis


1 Einleitung ..... 10

1.1 Motivation ..... 10

1.1.1 Programmier-Miniwelten ..... 10

1.1.2 Begriffliches ..... 11

1.1.3 Entwicklung von Programmierlernumgebungen ..... 12

1.1.4 Einsatz und Nutzen von Solist ..... 12

1.2 Aufbau und Komponenten ..... 13

1.3 Voraussetzungen ..... 13

1.4 Änderungen von Version 2.0 gegenüber Version 1.0 ..... 13

1.5 Aufbau des Benutzungshandbuchs ..... 14

2 Installation ..... 15

2.1 Voraussetzungen ..... 15

2.2 Download, Installation und Start ..... 15

2.3 Materialien ..... 15

3 Erste Schritte ..... 17

3.1 Öffnen einer existierenden Miniwelt ..... 17

3.2 Schreiben und Ausführen eines Programms ..... 19

3.3 Gestalten des Territoriums ..... 19

3.4 Ändern des Modells der Mini-Programmierwelt ..... 20

3.5 Generieren einer Programmierlernumgebung ..... 21

4 Grundlagen ..... 24
4.1 Theaterstücke bzw. Miniwelten ..... 24
4.2 Bühnen ..... 24
4.3 Akteure ..... 25
4.4 Requisiten ..... 25
4.5 Aufführungen ..... 26
4.6 Hilfsklassen ..... 26
5 Solist-Simulator ..... 27
5.1 Menüs ..... 27
5.2 Toolbar ..... 30
5.3 Klassenbereich ..... 32
5.4 Bühnenbereich ..... 33
5.5 Meldungsbereich ..... 33
5.6 Editorbereich ..... 33
5.7 Editor-Fenster ..... 33
6 Theater-API ..... 36
6.1 Stage ..... 36
6.1.1 Gestaltungsmethoden ..... 37
6.1.2 Getter-Methoden ..... 37
6.1.3 Kollisionserkennungsmethoden ..... 37
6.1.4 Event-Methoden ..... 38
6.2 Component, Actor und Prop ..... 38
6.2.1 Manipulationsmethoden ..... 38
6.2.2 Getter-Methoden ..... 39

6.2.3 Kollisionserkennungsmethoden ..... 39
6.2.4 Event-Methoden ..... 40
6.2.5 Solist ..... 40
6.3 Performance ..... 40
6.4 Maus- und Tastatur-Events ..... 41
6.4.1 KeyInfo ..... 42
6.4.2 MouseInfo ..... 42
6.5 TheaterImage und TheaterIcon ..... 43
6.6 Kollisionserkennung ..... 44
7 Entwicklungsfunktionalitäten im Detail ..... 45
7.1 Anlegen und Verwalten von Klassen ..... 45
7.1.1 Definition einer neuen Stage-Klasse ..... 46
7.1.2 Definition einer neuen Actor-Klasse ..... 48
7.1.3 Definition einer neuen Prop-Klasse ..... 51
7.1.4 Definition einer neuen Performance-Klasse ..... 52
7.1.5 Definition einer neuen Hilfs-Klasse ..... 53
7.1.6 Erweitern der Klasse Hamster ..... 53
7.1.7 Klassen löschen ..... 54
7.2 Verwalten und Editieren von Solist-Programmen ..... 55
7.2.1 Schreiben eines neuen Solist-Programms ..... 56
7.2.2 Ändern des aktuellen Solist-Programms ..... 56
7.2.3 Löschen des aktuellen Solist-Programm ..... 56
7.2.4 Abspeichern des aktuellen Solist-Programms ..... 56
7.2.5 Öffnen eines einmal abgespeicherten Solist-Programms ..... 57

7.2.6 Drucken eines Solist-Programms ..... 57
7.2.7 Editier-Funktionen ..... 57
7.3 Compilieren ..... 58
7.3.1 Compilieren ..... 58
7.3.2 Beseitigen von Fehlern ..... 58
7.4 Gestalten und Verwalten der Bühne ..... 59
7.4.1 Abspeichern der Bühne ..... 60
7.4.2 Wiederherstellen einer abgespeicherten Bühne ..... 60
7.5 Interaktives Ausführen von Solist-Befehlen ..... 60
7.5.1 Befehlsfenster ..... 61
7.5.2 Parameter ..... 61
7.5.3 Rückgabewerte von Funktionen ..... 61
7.5.4 Befehls-Popup-Menü ..... 62
7.5.5 Klassen-Popup-Menü ..... 62
7.6 Ausführen von Solist-Programmen ..... 63
7.6.1 Starten eines Solist-Programms ..... 63
7.6.2 Stoppen eines Solist-Programms ..... 63
7.6.3 Pausieren eines Hamster-Programms ..... 63
7.6.4 Während der Ausführung eines Solist-Programms ..... 63
7.6.5 Einstellen der Geschwindigkeit ..... 64
7.6.6 Wiederherstellen der Bühne ..... 64
7.7 Debuggen von Solist-Programmen ..... 64
7.7.1 Beobachten der Programmausführung ..... 65
7.7.2 Schrittweise Programmausführung ..... 66

7.7.3 Breakpoints ..... 66
7.7.4 Debugger-Fenster ..... 67
7.8 Anlegen und Verwalten von Aktionsbuttons ..... 67
7.8.1 NewPropHandler ..... 68
7.8.2 ClickHandler ..... 69
7.8.3 ActionHandler ..... 69
7.9 Generieren einer PLU ..... 73
7.10 Scratch-PLUs ..... 75
7.11 Konsole ..... 78
7.12 PLU-Typen ..... 80
7.12.1 Imperative PLUs ..... 80
7.12.2 Objektbasierte PLUs ..... 81
8 Tipps und Tricks ..... 83
8.1 addedToStage ..... 83
8.2 setLocation ..... 83
8.3 Synchronisation ..... 84
8.4 Einfrieren ..... 85
8.5 Zustandsabfrage ..... 86
9 Referenzhandbuch ..... 88
9.1 Stage ..... 88
9.2 Component, Actor, Prop ..... 97
9.2.1 Component ..... 97
9.2.2 Actor ..... 106
9.2.3 Prop ..... 107

9.3 Performance ..... 107
9.4 Bilder ..... 111
9.4.1 TheaterImage ..... 111
9.4.2 TheaterIcon ..... 116
9.5 Ereignisse ..... 117
9.5.1 KeyInfo ..... 117
9.5.2 MouseInfo ..... 118
9.6 Kollisionserkennung ..... 122
9.6.1 PixelArea ..... 122
9.6.2 Rectangle ..... 122
9.6.3 Point ..... 124
9.6.4 Cell ..... 126
9.6.5 CellArea ..... 127
9.7 Aktionsbuttons ..... 129
9.7.1 ActionHandler ..... 129
9.7.2 ClickHandler ..... 130
9.7.3 NewPropHandler ..... 130
9.8 Annotationen ..... 131
9.8.1 Description ..... 131
9.8.2 Invisible ..... 131
10 Beispiel-Theaterstücke ..... 132
10.1 Theaterstück demo und oodemo ..... 132
10.2 Theaterstück hamster ..... 132
10.3 Theaterstück kara ..... 133

10.4 Theaterstück frosch ..... 133
10.5 Theaterstück turtle ..... 134
10.6 Theaterstück geisterstunde ..... 134
10.7 Theaterstück gameoflife ..... 135
10.8 Theaterstück sudoku ..... 135
10.9 Theaterstück damen ..... 136
10.10 Theaterstück springer ..... 136
10.11 Theaterstück hanoi ..... 137
10.12 Theaterstück tictactoe ..... 137
10.13 Theaterstück terminal ..... 138
10.14 Theaterstück fruechte ..... 138
10.15 Weitere Theaterstücke ..... 139

# 1 Einleitung

Solist ist eine spezielle Entwicklungsumgebung (IDE) zum einfachen Entwickeln von Programmierlernumgebungen. Mit Hilfe graphisch-interaktiver Hilfsmittel können erfahrenere Programmierer auf der Grundlage einer übersichtlichen Programmier-API so genannte Mini-Programmierwelten definieren, daraus Programmierlern-umgebungen generieren und diese dann Programmieranfängern zur Verfügung stellen

## *1.1 Motivation*

Heftig diskutiert wird immer wieder die Frage, ob die Programmierung noch zur Informatik-Grundausbildung gehören soll. Für fast alles gibt es heutzutage bereits fertige Computer-Anwendungen. Reicht es nicht aus, den Umgang mit diesen zu erlernen, als zu wissen, wie sie intern funktionieren? Probleme einer solchen anwendungsorientierten Einführung in die Informatik sind die Kurzlebigkeit heutiger Anwendungen und die entstehende Abhängigkeit von den Produkten bzw. Anbietern. Gefragt ist daher eher Konzeptwissen (wie funktionieren Softwaresysteme, wie werden sie erstellt), das letztendlich den Transfer auf konkrete Anwendungen erlaubt.

### 1.1.1 Programmier-Miniwelten

Das Erlernen von Programmiersprachen und die Entwicklung von Programmen sind allerdings nicht trivial. Sicher gibt es Schüler, die sehr schnell damit zurechtkommen. Viele haben aber Probleme mit der Komplexität der Thematik und sind schnell frustriert. Für einen einfachen Einstieg in die komplexe Welt der Programmierung wurden daher bereits in den 70er Jahren des vergangenen Jahrhunderts so genannte Mini-Programmierwelten (micro worlds) entwickelt. Sie reduzieren den zugrunde liegenden Befehlssatz auf ein Minimum, enthalten spielerische Elemente und verbinden mit den entsprechenden Befehlen visuelle Aktionen von virtuellen Akteuren auf dem Bildschirm.

Ur-Vater dieser Miniwelten ist „Karel the Robot“. Karel ist ein virtueller Roboter, der mit Befehlen, wie *move* oder *turnLeft* durch eine virtuelle Landschaft gesteuert werden kann und dabei verschiedene ihm gestellte Aufgaben zu lösen hat.

Eine andere weit verbreitete Mini-Programmierwelt ist das „Hamster-Modell“. Programmieranfänger erlernen hiermit die grundlegenden Konzepte der Programmierung sowie den Programmentwicklungsprozess, indem sie virtuelle Hamster durch eine virtuelle Körner-Landschaft steuern. Der Befehlsatz besteht dabei aus vier Grundbefehlen (*vor*, *linksUm*, *gib*, *nimm*) und drei Testbefehlen (*vornFrei*, *kornDa*, *maulLeer*). Programmieranfängern werden bestimmte Aufgaben gestellt (der Hamster soll alle Körner in seinem Territorium finden und fressen, der Hamster soll ein Labyrinth durchlaufen, …), zu denen sie dann Hamster-Programme entwickeln müssen, die die Aufgaben lösen. Mit dem so genannten Hamster-Simulator wird dabei eine Programmierlernumgebung zur Verfügung gestellt, mit

dem Hamster-Programme eingegeben, compiliert, getestet und ausgeführt werden können (siehe http://www-java-hamster-modell.de)

Weitere bekannte Mini-Programmierwelten sind „Kara der Marienkäfer“, bei dem ein virtueller Marienkäfer Kleeblätter suchen und dabei Pilze verschieben und Baumstümpfen ausweichen muss oder die „Schildkrötengraphik“, bei der eine virtuelle Schildkröte mit Hilfe eines Stiftes Zeichnungen (Fraktale) malen kann.

Derartige Mini-Programmierwelten weisen im Allgemeinen folgende Eigenschaften auf:

- Es gibt einen (virtuellen) Akteur, der mit Hilfe bestimmter Befehle durch eine (virtuelle) Landschaft gesteuert werden kann.
- Die Komplexität eines Computers wird auf einen minimalen Satz von Befehlen eingeschränkt, die dem Akteur erteilt werden können.
- Die Programmierkonzepte werden schrittweise und aufeinander aufbauend eingeführt und jeweils durch zahlreiche Beispiele demonstriert.
- Anhand vieler Aufgaben mit einfach zu verstehenden Aufgabenstellungen können die Programmieranfänger selbst überprüfen, ob sie den Stoff nicht nur verstanden haben, sondern auch praktisch umsetzen können.
- Für die Bearbeitung der Aufgaben existiert eine einfach zu bedienende Programmierlernumgebung, die die Erstellung und den Test von Programmen unterstützt und Programmausführungen, d.h. die Aktionen der Akteure, visuell auf dem Bildschirm darstellt.

Mit diesen Eigenschaften tragen Mini-Programmierwelten Empfehlungen von Didaktikern, wie der Einstieg in die Programmierung erfolgen könnte, voll und ganz Rechnung [Br97, BOM99].

Prinzipiell sind die meisten der existierenden Mini-Programmierwelten zunächst einmal programmiersprachenunabhängig. Für viele existieren mehrere Implementierungen, die unterschiedliche Programmiersprachen unterstützen. Als Grundlage von Solist wurde jedoch bewusst die Programmiersprache Java gewählt und ab Version 2.0 zusätzlich die visuelle Programmiersprache Scratch. Java ist eine moderne Programmiersprache, die sich in den letzten Jahren sowohl im Ausbildungsbereich als auch im industriellen Umfeld immer mehr durchgesetzt hat. Scratch wird aktuell in vielen Schulen für die ersten Schritte beim Programmieren lernen eingesetzt.

### 1.1.2 Begriffliches

Im Folgenden soll zwischen den beiden Begriffen Mini-Programmierwelt und Programmierlernumgebung unterschieden werden.

Bei einer Mini-Programmierwelt (**MPW**) handelt es sich um ein (didaktisches) Modell, in dem die Akteure und anderen Objekte einer virtuellen Landschaft, ihre Beziehungen untereinander und ihre möglichen Aktionen und die Befehle zum Steuern der Akteure festgelegt werden. Beim Hamster-Modell gibt es bspw. einen

virtuellen Hamster als Akteur sowie Mauern und Körner als weitere Objekte. Diese befinden sich in einer Landschaft, die aus einzelnen Kacheln besteht. Der Hamster kann Körner in sein Maul aufnehmen, er darf nicht gegen Mauern laufen. Als Befehle gibt es die Grundbefehle *vor*, *linksUm*, *nimm* und *gib* und die Testbefehle *vornFrei*, *kornDa* und *maulLeer*.

Um mit einer MPW tatsächlich programmieren lernen zu können, bedarf es einer Programmierlernumgebung (**PLU**), oft auch als Simulator bezeichnet. Hierbei handelt es sich um ein Computerprogramm, das eine konkrete MPW abbildet und das das Erstellen und Testen von Programmen der MPW unterstützt. Beim Hamster-Modell bildet der Hamster-Simulator eine solche PLU.

### 1.1.3 Entwicklung von Programmierlernumgebungen

Das (theoretische) Entwickeln einer für Programmiernanfänger geeigneten und interessanten Mini-Programmierwelt sollte (für kreative Didaktiker) eigentlich ein geringeres Problem darstellen, als das (software-technische) Entwickeln einer dazu gehörigen für Programmieranfänger einfach zu bedienenden trotzdem aber ausreichend mächtigen Programmierlernumgebung. Eine PLU muss als Bestandteile zumindest Komponenten zum Editieren und Compilieren von Programmen, zum visuellen Gestalten einer virtuellen Landschaft und zum Ausführen entsprechender Programme mit visuellem Feedback der einzelnen Befehle in der Landschaft zur Verfügung stellen. Die Entwicklung einer solchen PLU ist – selbst für erfahrene Programmierer – sicher eine Aufgabe, die mehrere Wochen oder Monate in Anspruch nehmen kann.

### 1.1.4 Einsatz und Nutzen von Solist

Genau an dieser Stelle setzt nun Solist an. Solist ist eine spezielle Entwicklungsumgebung für PLUs. Mit Hilfe von Solist können einigermaßen erfahrene Programmierer innerhalb kurzer Zeit PLUs entwickeln. Bspw. dauerte die Nachimplementierung des Hamster-Simulators in Solist lediglich ca. 3-4 Stunden! Etwa dieselbe Zeit hat die Entwicklung weitere PLUs für die Kara-Modellwelt und die Schildkrötengraphik (siehe die Beispiele in Kapitel 10) gekostet.

Vielleicht fragen Sie sich nun: Es gibt doch bereits eine Reihe von Mini-Programmierwelten mit dazugehörigen Programmierlernumgebungen. Wofür werden denn überhaupt noch weitere benötigt? Antwort:

- Als Entwickler des Hamster-Modell und des Hamster-Simulators bekomme ich oft Emails von Lehrern, die fragen, ob es nicht möglich ist, bestimmte Eigenschaften des Modells bzw. Simulators ändern zu können. Einige wollen den Befehl *rechtsUm* als weiteren Grundbefehl, andere wollen englischsprachige Befehle, andere wollen die Icons austauschen, andere wollen den Hamster gegen einen anderen Akteur umtauschen, usw. Durch Einsatz von Solist können derartige Änderungswünsche an existierenden PLUs innerhalb kurzer Zeit problemlos selber durchgeführt werden.

- Nach geraumer Zeit wird es (ungeduldigen) Programmieranfängern oft langweilig, immer nur Programme derselben MPW schreiben zu müssen. Natürlich wäre es für Lehrer möglich, bspw. nach einiger Zeit vom Hamster-Modell zu Kara zu wechseln. Allerdings bedarf das doch einer Umgewöhnung der Programmieranfänger, was die Handhabung der dazu gehörigen PLU betrifft. Mit Solist erstellte PLUs sind jedoch in ihrer Bedienung gleichartig, so dass ein solcher Wechsel problemlos möglich ist.
- Solist erlaubt nicht nur die (Nach-)Implementierung von PLUs für MPWs, die einen allgemeinen Einstieg in die Programmierung erleichtern. Mit Hilfe von Solist können auch wesentlich speziellere MPWs umgesetzt werden. In Kapitel 10 wird dies an mehreren Beispielen demonstriert. Es wird bspw. eine PLU vorgestellt, die das Verständnis von Programmieranfängern für die Rekursion anhand des Problems „Türme von Hanoi“ fördern soll.

## *1.2 Aufbau und Komponenten*

Solist ist ein Werkzeug aus der Familie des so genannten Programmiertheaters. Neben Solist besteht dieses aktuell aus zwei weiteren Werkzeugen:

- Objekt-Theater: Ein Werkzeug zum Erlernen der objektorientierten Programmierung durch das Entwicklen kleiner Spiele und Simulationen.
- Threadnocchio: Ein Werkzeug zum Erlernen der parallelen Programmierung mit Java-Threads.

Solist besteht dabei aus zwei Komponenten:

- einer graphisch-interaktiven Entwicklungsumgebung, dem so genannten Solist-Simulator,
- und einer vordefinierten Klassenbibliothek, der Theater-Klassenbibliothek oder Theater-API.

Die API ist bei allen Werkzeugen des Programmiertheaters nahezu identisch.

## *1.3 Voraussetzungen*

Zielgruppe von Solist sind bspw. Informatik-Lehrer, die Programmierlernumgebungen für selbst ausgedachte Mini-Programmierwelten implementieren und ihren Schülern zur Verfügung stellen wollen. Voraussetzung zur Benutzung von Solist ist die Beherrschung der Basissprachkonzepte von Java (Klassen, Objekte, Attribute, Methoden, Anweisungen, Variablen, Schleifen, …). Nicht notwendig sind Kenntnisse der JDK-Klassenbibliothek bspw. zur Entwicklung von graphischen Oberflächen (GUIs) mit Java-AWT oder Java-Swing.

## *1.4 Änderungen von Version 2.0 gegenüber Version 1.0*

- Beseitigung einiger interner Fehler

- Generierte Simulatoren besitzen nun „Neu“-, „Laden“- und „Speichern“-Buttons in der Toolbar
- Generierung von Scratch-Simulatoren (siehe Kapitel 7.10)

## *1.5 Aufbau des Benutzungshandbuchs*

Dieses Benutzungshandbuch ist so aufgebaut, dass nach dieser einleitenden Motivation in Kapitel 2 die Installation und Struktur von Solist beschrieben wird. In Kapitel 3 werden Ihnen erste Schritte beim Umgang mit Solist dadurch vorgestellt, dass der mit Solist erstellte Hamster-Simulator-Light um einen Befehl rechtsUm erweitert wird. Kapitel 4 geht auf die Grundlagen von Solist, insbesondere die zugrunde liegende Theater-Metapher ein. Kapitel 5 widmet sich dem Solist-Simulator und seinen Komponenten, Kapitel 6 anschließend der Theater-API, die überblicksartig vorgestellt wird. In Kapitel 7 geht es dann ins Detail. In diesem Kapitel werden die einzelnen Schritte beim Arbeiten mit Solist dadurch erläutert, dass inkrementell eine kleine Demo-MPW umgesetzt und daraus letztendlich ein Demo-Simulator generiert wird. Kapitel 8 enthält einige nützliche Tipps und Tricks für den Umgang mit Solist. Kapitel 9 ist ein Referenzhandbuch, in dem alle Klassen der Theater-API und deren Methoden ausführlich beschrieben werden. Letztendlich werden in Kapitel 10 die Theaterstücke kurz vorgestellt, die standardmäßig Solist als Beispiele und Anregungen für eigene MPWs beigefügt sind.

# 2 Installation

## *2.1 Voraussetzungen*

Voraussetzung zum Starten von Solist 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, Installation und Start*

Solist hat eine eigene Website: http://www.programmierkurs-java.de/solist. Von dieser Website muss eine Datei *solist-1.0.zip* herunter geladen und entpackt werden. Es entsteht ein Ordner namens *solist-1.0* (der so genannte *Solist-Ordner*), in dem sich eine Datei *solist.jar* befindet. Durch Doppelklick auf diese Datei wird die Solist-Entwicklungsumgebung (im Folgenden auch als Solist-Simulator oder Solist-IDE bezeichnet) gestartet (alternativ: Eingabe des Befehls „`java –jar solist.jar`“ in einer Konsole).

Beim ersten Start sucht Solist 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 *solist.properties* im Solist-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 die Solist-IDE arbeitet. In der Version 1.0 wird allerdings nur deutsch unterstützt.

## *2.3 Materialien*

Der Solist-Ordner enthält folgende Unterordner und Dateien

- *handbuch*: Dieser Unterordner enthält das Benutzungshandbuch von Solist im HTML und PDF-Format.
- *api*: Dieser Unterordner enthält die Theater-API im javadoc-HTML-Format.
- *plays*: Dieser Unterordner enthält die Solist beigefügten Beispiel-Theaterstücke (siehe auch Kapitel 10). Den Beispielen beigefügt sind auch die zugehörigen Handbücher im MSWORD-2003-Format. Wenn Sie eigene MPWs entwickeln und dazu Benutzungshandbücher erstellen wollen, können Sie gerne auf die bereits fertig gestellten Benutzungshandbücher zurückgreifen.
- *simulatoren*: Dieser Unterordner enthält die generierten Simulatoren für die Beispiel-Theaterstücke.
- *solist.properties*: Diese Datei enthält Properties, wie Sprache und JDK.
- *jalopy-license.txt*: Intern wird das Sourcecode-Formatierungstool JALOPY verwendet. Diese Datei enthält die Lizenzbedingungen für JALOPY.

- *solist-license.txt*: Die Lizenzbedingungen für Solist.
- *solist.jar*: Starten Sie den Solist-Simulator durch Doppelklick auf diese Datei bzw. Eingabe des Befehls „`java –jar solist.jar`“

Nehmen Sie bitte keine Umbenennungen an den Dateien und Ordner vor! Insbesondere darf die Datei `solist.jar` nicht umbenannt werden.

# 3 Erste Schritte

Lesen oder überfliegen Sie zumindest zunächst dieses Handbuch, um die Grundlagen von Solist zu verstehen. Spielen Sie danach am besten zunächst einmal mit ein paar Beispielszenarien herum, die sich im Unterordner *plays* im Solist-Ordner befinden (siehe auch Kapitel 10). Gehen Sie bspw. folgendermaßen vor:

## *3.1 Öffnen einer existierenden Miniwelt*

Doppelklicken Sie die Datei *solist.jar* im Solist-Ordner. Damit wird die Solist-IDE gestartet. Zunächst erscheint ein Startbild (siehe Abbildung 3.1), dann nach ein paar Sekunden das Hauptfenster der Solist-IDE (siehe Abbildung 3.2).

Wählen Sie im Menü *Miniwelt* den Eintrag *Öffnen…* Begeben Sie sich über die Dateiauswahl in den Unterordner *plays* (engl. Theaterstücke) und wählen Sie ein Theaterstück aus, bspw. das Theaterstück *hamster* (siehe Abbildung 3.3). Es öffnet sich ein neues Hauptfenster mit dem Theaterstück *Hamster* (siehe Abbildung 3.4). Das andere Hauptfenster können Sie schließen.


Abb. 3.1: Startbild

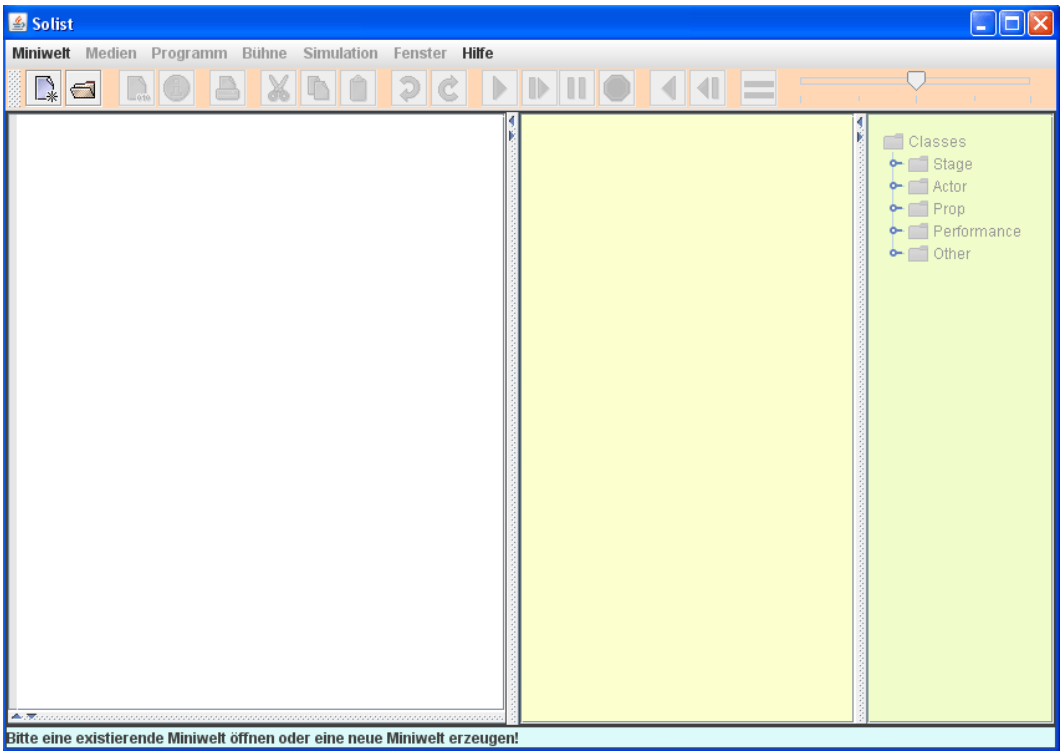



Abb. 3.2: Hauptfenster

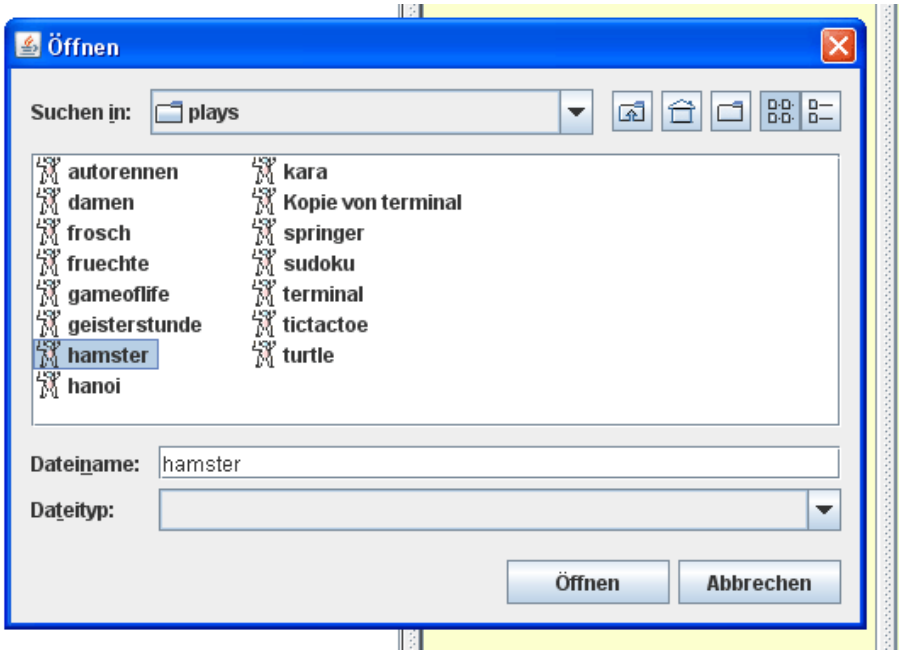



Abb. 3.3: Auswahl des Theaterstückes *hamster*

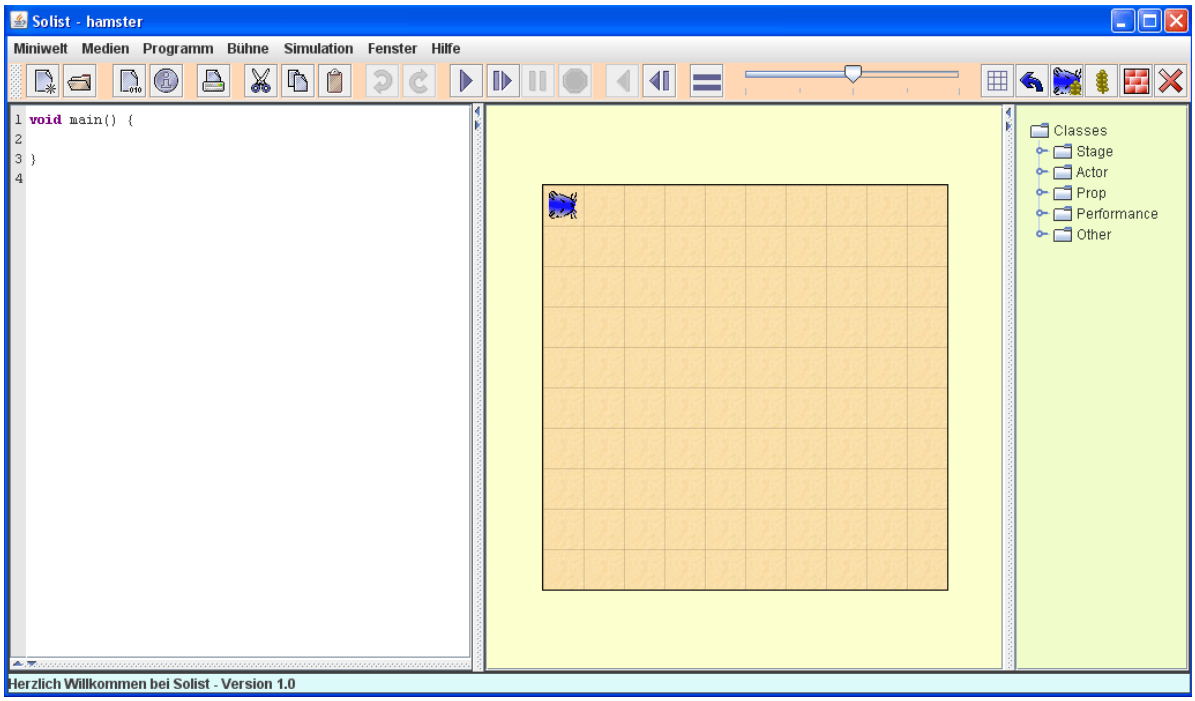



Abb. 3.4: Solist-Hauptfenster mit dem Hamster-Theaterstück nach dem Öffnen

## 3.2 Schreiben und Ausführen eines Programms

Geben Sie im Editor des Hauptfensters ein kleines Hamster-Beispielprogramm ein, zum Besipiel:

```
void main() {
    while (vornFrei()) {
        vor();
    }
}
```

Drücken Sie zum Kompilieren des Programms in der Toolbar den Compile-Button (3. Button der Toolbar von links). Wenn das Kompilieren erfolgreich war (es erscheint eine Dialogbox mit einer entsprechenden Meludng), drücken Sie zum Ausführen des Programms in der Toolbar den Start-Button (11. Button der Tollbar von links): Der Hamster läuft durch das Territorium.

## 3.3 Gestalten des Territoriums

Mit Hilfe der Aktionsbuttons rechts in der Toolbar können Sie das Territorium gestalten, bspw. Mauern und Körner setzen. Drücken Sie auf den jeweiligen Button und dann auf die entsprechende Kachel des Territoriums. Sie können die Komponenten auch nachträglich noch durch Anklicken und Verschieben mit gedrückter Maustaste im Territorium umplatzieren. Weiterhin können Sie über den Komponenten ein Popup-Menü aktivieren, in dem die Befehle, die die Komponente kennt, erscheinen und bei Auswahl ausgeführt werden (siehe Abbildung 3.5).

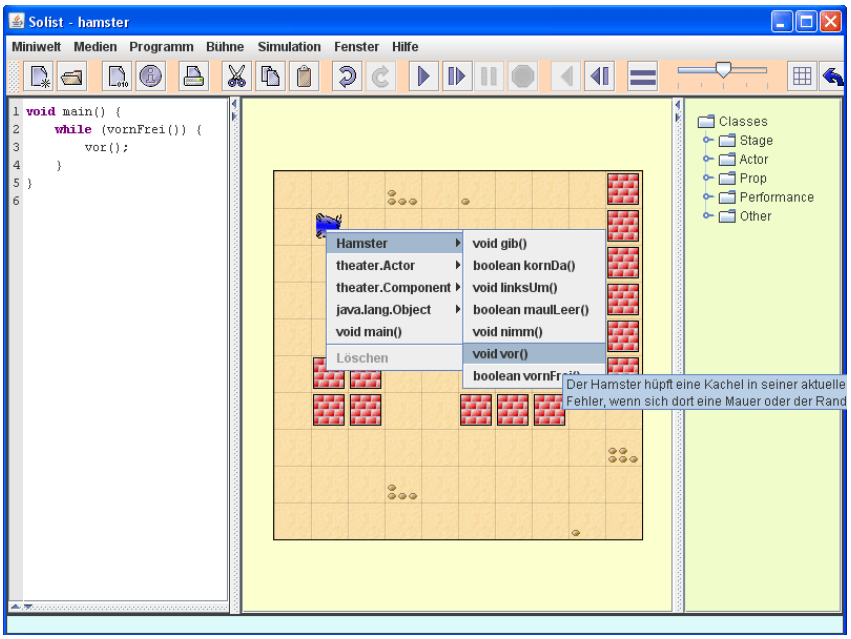



Abb. 3.5: Popup-Menü des Hamsters

## *3.4 Ändern des Modells der Mini-Programmierwelt*

Um den Grundbefehlssatz des Hamsters durch den Befehl *rechtsUm* zu ergänzen, öffnen Sie den Ordner *Actor* durch Doppelklick auf das entsprechende Ordersymbol. In dem Ordner befindet sich eine Klasse namens *Hamster*. Doppelklicken Sie den Namen „Hamster“. Es öffnet sich ein Editor, in dem die Klasse *Hamster* implementiert ist. Scrollen Sie nach unten zur Methode *linksUm* (siehe Abbildung 3.6) und fügen Sie dort die folgende Methode *rechtsUm* ein:

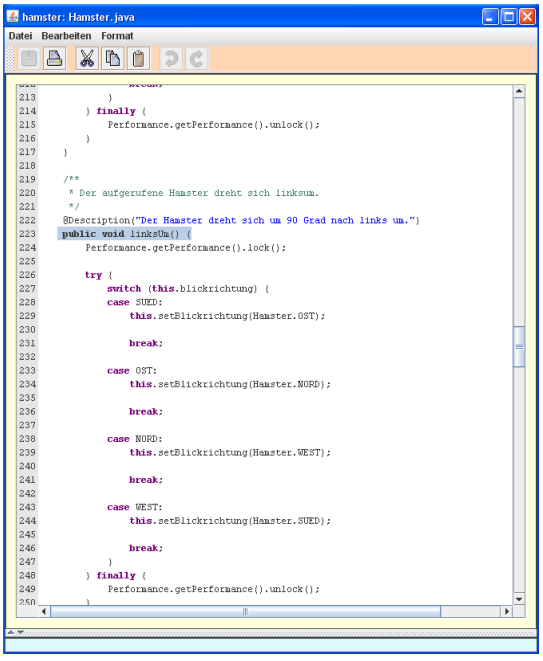



Abb. 3.6: Methode *linksUm* der Klasse *Hamster*

```
/**
 * Der aufgerufene Hamster dreht sich rechtsum.
 */
@Description("Der Hamster dreht sich um 90 Grad nach rechts um.")
public void rechtsUm() {
    Performance.getPerformance().lock();

    try {
        switch (this.blickrichtung) {
        case SUED:
            this.setBlickrichtung(Hamster.WEST);

            break;

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

            break;

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

            break;

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

            break;
        }
    } finally {
        Performance.getPerformance().unlock();
    }
}
```

Speichern Sie anschließend die Änderungen ab (im Editor-Fenster Menü „Datei“, Eintrag „speichern“) und kompilieren Sie durch Drücken des (inzwischen rot-erscheinenden) Compilier-Buttons im Hauptfenster (3. Button der Toolbar von links). Wenn Sie (syntaktische) Fehler gemacht haben, erscheint eine Dialogbox mit entsprechenden Hinweisen. Wenn alles geklappt hat, erscheint eine Dialogbox mit der Meldung OK.

Hinweis: Nach dem Compilieren wird in Solist (nicht in den später generierten PLUs) das Territorium neu initialisiert.

## *3.5 Generieren einer Programmierlernumgebung*

Um eine PLU für das leicht erweiterte Hamster-Modell zu generieren, wählen Sie im Menü „Miniwelt“ den Eintrag „Generieren…“. Es öffnet sich ein kleines Fenster (siehe Abbildung 3.7). Geben Sie einen selbst gewählten Namen für Ihre PLU und ein Verzeichnis ein, in dem die PLU gespeichert werden soll. Drücken Sie dann auf den OK-Button.

Begeben Sie sich dann in das entsprechend ausgewählte Verzeichnis. Hier finden Sie eine Datei „simulator.jar“, die Sie durch Doppelklick starten können: Es öffnet sich Ihre erste selbst erstellte Programmierlernumgebung (siehe Abbildung 3.8)! Sie unterscheidet sich im Wesentlichen dadurch von der Solist-IDE, dass der Dateibaum

rechts im Hauptfenster fehlt. Dieser Dateibaum ist für die Implementierung der Modellwelt zuständig, aber damit sollen die Programmieranfänger ja auch gar nichts mehr zu tun haben.

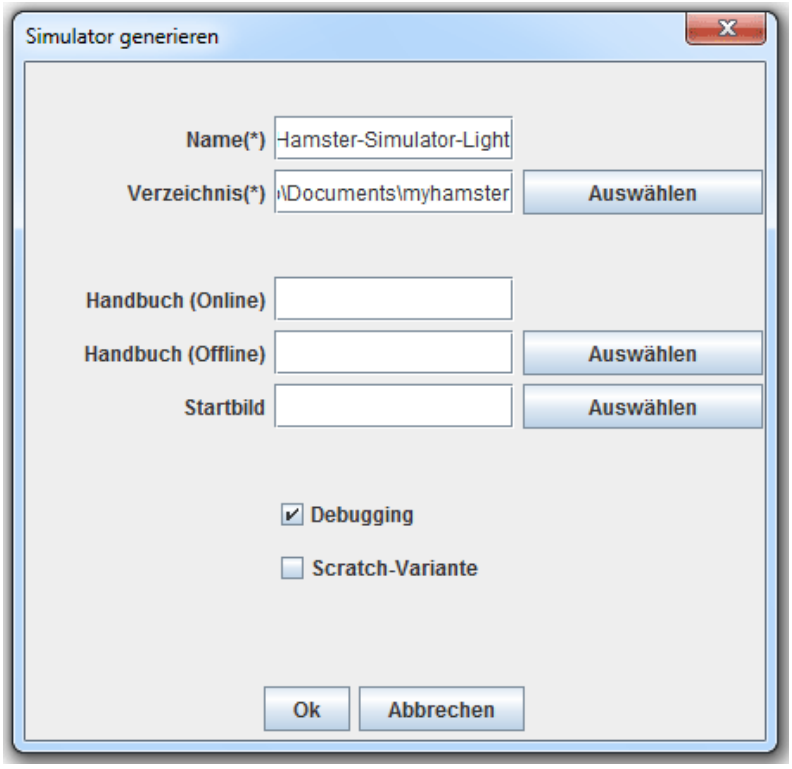



Abb. 3.7: Generieren-Dialogbox

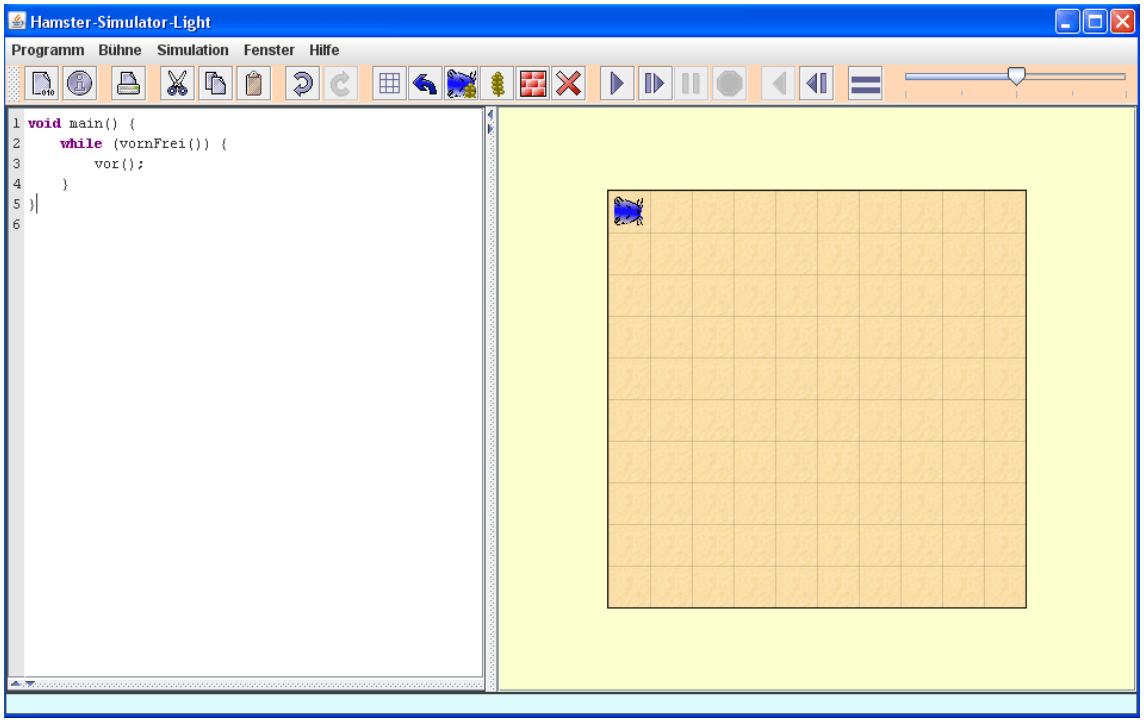



Abb. 3.8: Generierte Hamster-PLU

Geben Sie dann im Editor der generierten PLU das folgende Programm ein:

```
void main() {
    while (vornFrei()) {
        vor();
    }

    rechtsUm();

    while (vornFrei()) {
        vor();
    }
    rechtsUm();
}
```

Der Compilieren-Button (das ist in der generierten PLU der linke Button der Toolbar) erscheint rot. Klicken Sie ihn zum Kompilieren an. Wenn Sie sich nicht vertippt haben, sollte eine Dialogbox mit der Meldung OK erscheinen. Danach können Sie das Programm durch Anklicken des Start-Buttons (15. Button der Toolbar) ausführen. Sie sehen, in dem von Ihnen generierten Simulator gehört nun auch der Befehl rechtsUm zu den Standard-Befehlen des Hamsters!

# 4 Grundlagen

## *4.1 Theaterstücke bzw. Miniwelten*

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

## *4.2 Bühnen*

Handlungsumfeld in Solist 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).

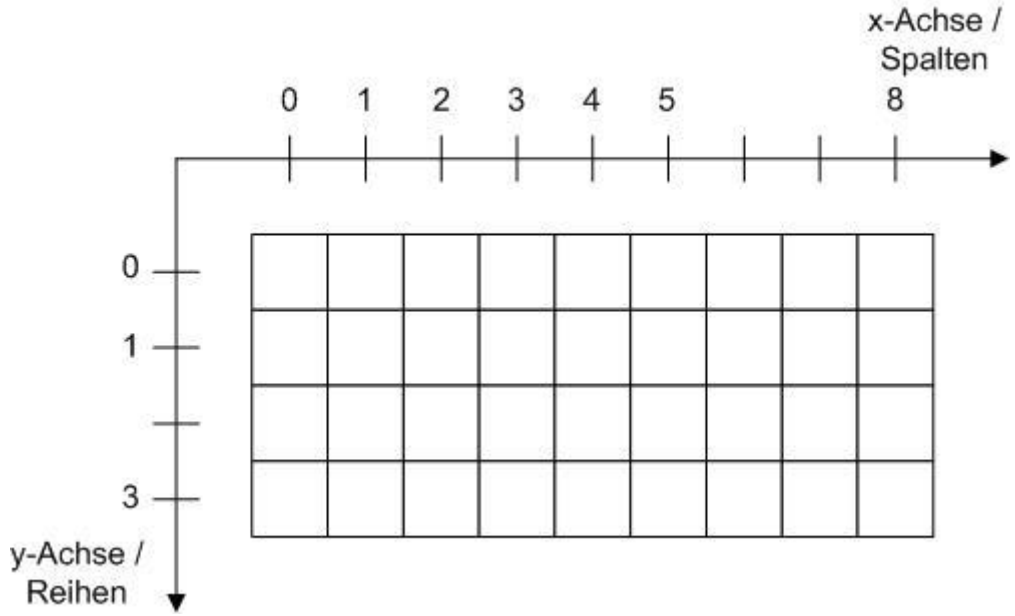



Abb. 3.1: Bühnenaufbau

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 Solist repräsentiert als Klassen, die von der durch Solist vordefinierten Klasse `Stage` abgeleitet werden. Ein Solist-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 bzw. vor der Generierung einer PLU durch einen einfachen Mausklick auf die entsprechende Klasse ausgewählt werden. Die ausgewählte Bühne wird als aktive Bühne bezeichnet. Auch während einer Aufführung ist ein Wechsel der Bühne durch Aufruf einer entsprechenden Methode möglich.

## *4.3 Akteure*

Solist ist ein Werkzeug des so genannten Programmiertheaters. Schauspieler, die auf der Bühne agieren, werden hier als *Akteure* (Actor) bezeichnet. Akteure sind dabei Objekte von Klassen, die von der vordefinierten Klasse `Actor` abgeleitet sind. In Solist gibt es dabei immer genau einen Akteur, der so genannte *Solist* (daher hat das Werkzeug auch seinen Namen). Es kann jedoch durchaus mehrere von der Klasse Actor abgeleitete Actor-Klassen geben. Wie bei den Bühnen auch, wird die aktive Actor-Klasse vor der Programmausführung bzw. vor der PLU-Generierung durch einen einfachen Mausklick auf die entsprechende Klasse ausgewählt werden.

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

Der Solist muss durch entsprechende Anweisungen im Konstruktor der aktiven Stage-Klasse erzeugt und initial auf der Bühne platziert werden.

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

Im Hamster-Theaterstück ist der Hamster der Solist.

## *4.4 Requisiten*

Neben dem Solist können sich noch *Requisiten* (Prop) – genauer gesagt ein ihnen jeweils zugeordnetes Ikon - auf der Bühne befinden. Im Unterschied zum Solist sind Requisiten jedoch passive Objekte. Und das ist auch schon der einzige Unterschied zwischen dem Solist und Requisiten. Programmiertechnisch sind Requisiten nämlich Objekte von Klassen, die von der vordefinierten Klasse `Prop` abgeleitet sind. Sowohl

die Klasse `Actor` als auch die Klasse `Prop` sind aber von der 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.

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

Im Hamster-Theaterstück sind Mauern und Körner Requisiten.

## *4.5 Aufführungen*

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

Die Variation einer Ausführung ergibt sich aus dem Überschreiben bestimmter Methoden der Klasse Performance. Dies sind insbesondere die Methoden `started`, `stopped`, `suspended`, `resumed` und `speedChanged`. Die Methoden werden unmittelbar nach der entsprechenden Steuerungsaktion ausgeführt, auch wenn diese durch die Steuerungselemente im Solist-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.

## *4.6 Hilfsklassen*

Neben den von den Klassen *Stage*, *Actor*, *Prop* und *Performance* abgeleiteten Klassen lassen sich weitere Hilfsklassen definieren und in die Programme einbinden. Diese Klassen werden als *Hilfsklassen* bezeichnet.

# 5 Solist-Simulator

Der Solist-Simulator (auch als Solist-IDE bezeichnet) ist ein graphisch-interaktives Werkzeug zum Entwickeln und Implementieren von Mini-Programmierwelten bzw. Theaterstücken und Generieren entsprechender Programmierlernumgebungen. Abbildung 5.1 skizziert seinen Aufbau am Beispiel des geöffneten Hamster-Theaterstückes.

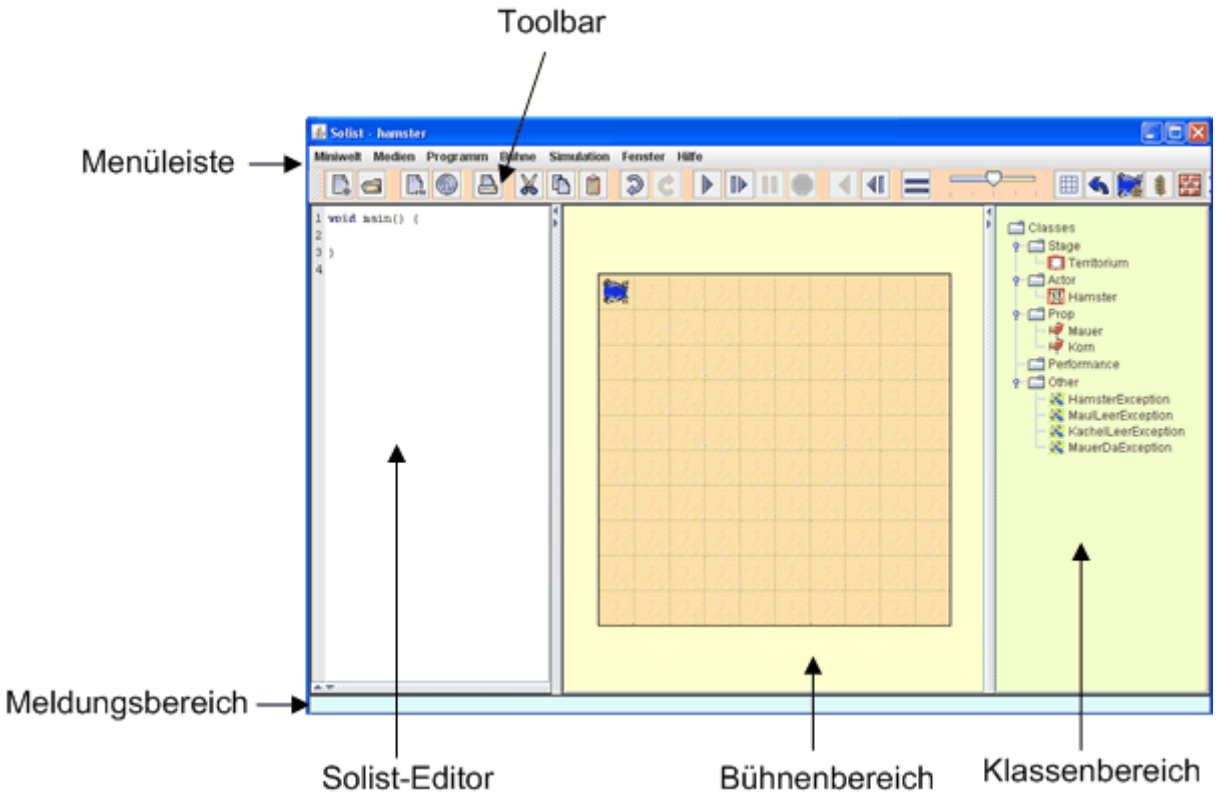



Abb. 5.1: Aufbau des Solist-Simulators

Ganz oben befindet sich die Menüleiste mit den sieben Menüs *Miniwelt*, *Medien*, *Programm, Bühne, Simulation, Fenster* und *Hilfe*. In der darunter liegenden Toolbar sind die wichtigsten Funktionalitäten des Simulators durch einen einzelnen Mausklick ausführbar. Im mittleren Bereich sieht man links einen Editor (der so genannte *Solist-Editor*), in dem Programme der entsprechende MPW eingegeben werden können. Rechts angeordnet ist der Klassenbereich, in dem die existierenden Klassen des bearbeiteten Theaterstücks angezeigt werden. In der Mitte des Bühnenbereiches wird die aktuelle Bühne angezeigt. Ganz unten im Meldungsbereich werden Hinweise und Fehlermeldungen ausgegeben.

## *5.1 Menüs*

Die sieben Menüs des Threadnocchio-Hauptfensters stellen zahlreiche Funktionen zur Bearbeitung und Ausführung von Theaterstücken Verfügung. Es sei jedoch

anzumerken, dass Sie die am häufigsten benutzten Funktionen auch schnell und direkt über entsprechende Buttons der Toolbar auslösen können (siehe Abschnitt 5.2).

Über das Menü *Miniwelt* können Sie neue Miniwelten bzw. Theaterstücke erzeugen, existierende Theaterstücke öffnen sowie die Bearbeitung des angezeigten Solist-Theaterstücks beenden (siehe Abbildung 4.2).


Abb. 5.2: Miniwelt-Menü

Das Menu *Medien* unterstützt den Import von Bild- und Ton-Dateien in die Unterordner *images* bzw. *sounds* des entsprechenden Theaterstücks (siehe Abbildung 5.3). Nach der Auswahl des entsprechenden Eintrags erscheint eine Dateiauswahlbox, über die Sie die zu importierende Datei auswählen können. Alternativ können Sie natürlich auch auf der Ebene des Betriebssystems die Dateien in die entsprechenden Unterordner des Theaterstückes kopieren.


Abb. 5.3: Medien-Menü

Programme lassen sich über das Menü *Programm* handhaben (siehe Abbildung 5.4). Hierüber ist es möglich, alle Klassen und das aktuelle Programm des Solist-Editors zu compilieren oder das im Solist-Editor befindliche Programm in einer Datei zu speichern bzw. später wieder zu laden. Auch das Drucken des Solist-Programms wird unterstützt. Weiterhin finden sich in dem Menü Funktionalitäten, die den Solist-Editor betreffen, wie Rückgängig machen von Aktionen, Wiederherstellen von Aktionen, Ausschneiden, Kopieren und Einfügen. Wollen Sie, dass beim Editieren automatisch eingerückt wird, wenn die Enter-Taste gedrückt wird, so können Sie entsprechend die Einrückung aktivieren. Auch die Schriftgröße des Solist-Editors lässt sich anpassen.

Generell lassen sich zwei Typen von PLU unterscheiden, die vom Solist-Simulator generiert werden können: eher imperative PLUs und objektbasierte PLUs. Welchen

Typ die von Ihnen erstellte PLU unterstützen soll, können Sie im untersten Menü-Item auswählen (siehe hierzu auch Abschnitt 7.11).


Abb. 5.4: Programm-Menü

Das Menü *Bühne* unterstützt die Handhabung der Bühne (siehe Abbildung 5.5). Über dieses Menü können Sie die aktuell dargestellte Bühne abspeichern und später gegebenenfalls wieder laden. Außerdem können Sie die aktuell dargestellte Bühne auch im PNG- oder GIF-Format abspeichern.

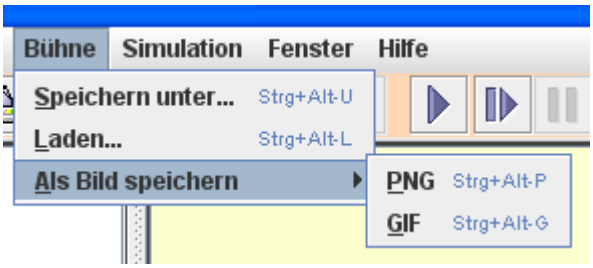



Abb. 5.5: Bühne-Menü

Das Menu *Simulation* stellt Funktionen zur Steuerung der Aufführung eines Theaterstücks zur Verfügung (siehe Abbildung 5.6). Hier werden Sie aber höchstwahrscheinlich eher die entsprechenden Buttons in der Toolbar benutzen.


Abb. 5.6: Simulation-Menü

Über das Menü *Fenster* lassen sich weitere Fenster öffnen bzw. schließen. Ausgaben und Eingaben mittels System.out, System.err und System.in werden an die Konsole weitergeleitet. Über das Befehlsfenster lassen sich Solist-Befehle interaktiv ausführen. Das Debuggerfenster zeigt den Stacktrace und die Variablenbelegungen bei der Ausführung eines Solist-Programms an, wenn die Ablaufverfolgung eingeschaltet ist (siehe auch Abbildung 5.8).


Abb. 5.8: Fenster-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 5.9).

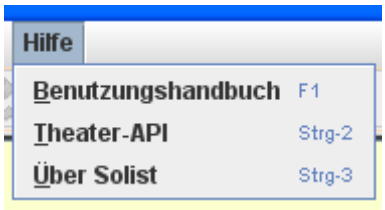



Abb. 5.9: Hilfe-Menü

## *5.2 Toolbar*

Über die Toolbar sind die wichtigsten Funktionalitäten des Simulators durch einen einfachen Mausklick ausführbar. Abbildung 5.10 skizziert den Aufbau der Toolbar. Rechts von der Toolbar hinaus können dazu weitere so genannte Aktionsbuttons angelegt werden (siehe Abschnitt 7.8).

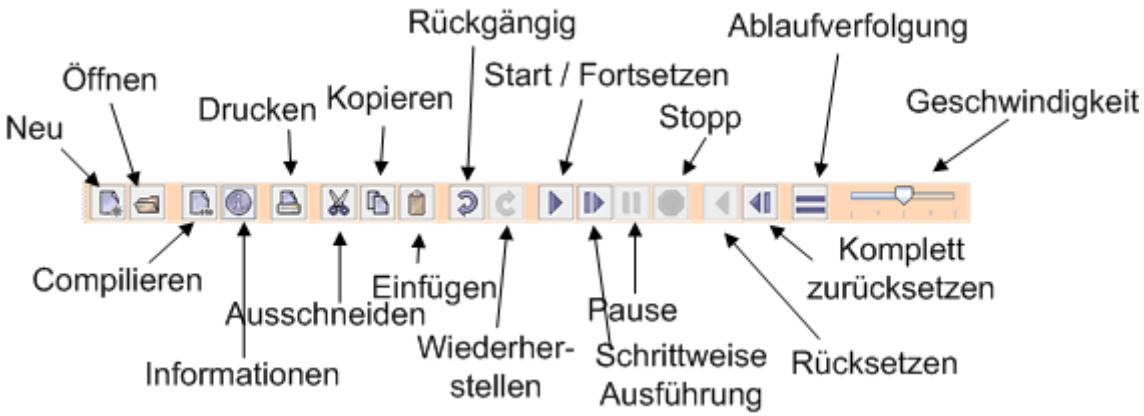



Abb. 5.10: Aufbau der Toolbar

- *Neu*: Button zum Erzeugen und Öffnen eines neuen Theaterstücks. Nach dem Anklicken öffnet sich eine Dialogbox, in der der Name des neuen Theaterstücks angegeben werden muss. Nach Klick auf „OK“ in der Dialogbox öffnet sich ein weiteres Simulatorfenster für das (noch leere) neue Theaterstück. Anzumerken ist, dass mehrere Theaterstücke gleichzeitig geöffnet und bearbeitet werden können. Einzelne Aktionen (Speichern, Kompilieren, …) beziehen sich aber nur jeweils auf das entsprechende Theaterstück.
- *Öffnen*: Button zum Öffnen eines bereits existierenden Theaterstücks. Nach dem Anklicken öffnet sich eine Dialogbox, in der das existierendeTheaterstück ausgewählt werden kann. Solist-Theaterstücke werden dabei durch ein spezielles Symbol (eine Theaterpuppe) symbolisiert.
- *Compilieren*: Button zum Compilieren aller Klassen des Theaterstücks. Enthalten Klassen syntaktische Fehler wird pro Klasse ein Editorfenster geöffnet und die Fehler werden dort angezeigt. Der Compilierbutton erscheint mit rotem Hintergrund, wenn Klassen geändert und gespeichert aber noch nicht neu compiliert wurden oder wenn noch nicht beseitigte Compilierfehler bestehen.
- *Informationen*: Button zum Editieren bzw. Abrufen genereller Informationen zum Theaterstück. Beim Anklicken des Buttons öffnet sich ein Fenster, in dem Informationen zum Theaterstück (Entwickler, Version, Anleitung, …) eingegeben bzw. abgerufen werden können.
- *Drucken* : Button zum Drucken des sich aktuell im Solist-Editor befindlichen Programms
- *Ausschneiden, Kopieren, Einfügen, Rückgängig, Wiederherstellen*: Gängige Editor-Funktionen für den Solist-Editor.
- *Start / Fortsetzen*: Steuerungsbutton zum Starten einer Aufführung des Theaterstücks. Beim Anklicken des Buttons wird eine Aufführung gestartet oder fortgesetzt, falls sie pausiert war.
- *Schrittweise Ausführung*: Ausführung der nächsten Anweisung.
- *Pause*: Steuerungsbutton zum Pausieren einer Aufführung. Beim Anklicken des Buttons wird eine laufende Aufführung pausiert bzw. angehalten. Durch anschließendes Klicken auf den Start/Fortsetzen- oder den Schrittweise-Ausführung-Button kann sie anschließend wieder fortgesetzt werden.
- *Stopp*: Steuerungsbutton zum Stoppen einer Aufführung. Beim Anklicken des Buttons wird eine Aufführung endgültig gestoppt.

- *Zurücksetzen*: Setzt eine Aufführung zurück auf den Zustand, den sie beim letztmaligen Starten innehatte.
- *Komplett zurücksetzen*: Steuerungsbutton zum Zurücksetzen einer Aufführung. Beim Anklicken dieses Buttons wird das Theaterstück in seinen Ausgangszustand zurückgesetzt, also komplett reinitialisiert.
- *Geschwindigkeit*: Steuerungsregler zum Regeln der Geschwindigkeit einer Aufführung. Je weiter der Regler rechts steht, desto schneller ist die Ausführungsgeschwindigkeit.
- *Ablaufverfolgung*: Aktiviert die Ablaufverfolgung (siehe Debugger-Funktionen in Abschnitt 7.7).

## *5.3 Klassenbereich*

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

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

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

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

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

Nach der Ausführung eines new-Items für eine Prop-Klasse wird ein Objekt der entsprechenden Klasse erzeugt (Aufruf des Default-Konstruktors der Klasse). Durch Klicken mit der Maus auf die Bühne wird die Komponente an der entsprechenden Stelle auf die Bühne platziert.

In Solist ist es möglich, für ein Theaterstück mehrere Bühnen, mehrere Actor-Klassen und mehrere Aufführungen zu definieren. Die aktive Bühne, die aktive Actor-

Klasse (auch als *Solist-Klasse* bezeichnet) bzw. die aktive 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.

## *5.4 Bühnenbereich*

Im Bühnenbereich wird die aktuell aktive Bühne angezeigt. Hier agiert während einer Aufführung der Solist.

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

## *5.5 Meldungsbereich*

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

## *5.6 Editorbereich*

Im Editorbereich des Hauptfensters – der so genannte *Solist-Editor* – können auf der Grundlage der in der aktuellen Solist-Klasse definierten Befehle Testprogramme entwickelt werden.

## *5.7 Editor-Fenster*

Nach dem Doppelklick auf den Namen einer Klasse im Klassenbereich öffnet sich ein Editor–Fenster mit der entsprechenden Klassendefinition. Im Editor kann die entsprechende Klasse editiert werden. Abbildung 5.11 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, Bearbeiten und Format (siehe Abbildungen 5.12, 5.13 und 5.14). 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 Hauptfenster) erscheinen Compilierfehler im Compilierfehlerbereich. Im Meldungsbereich ganz unten im Editorfenster werden Hinweise und Benutzungsfehler angezeigt.

Interessant ist die Funktion „Formatieren“ im Menü „Format“. Hiermit lässt sich der Sourcecode automatisch gemäß der Java-Codekonventionen formatieren. Eingesetzt wird hierbei das Tool JALOPY.

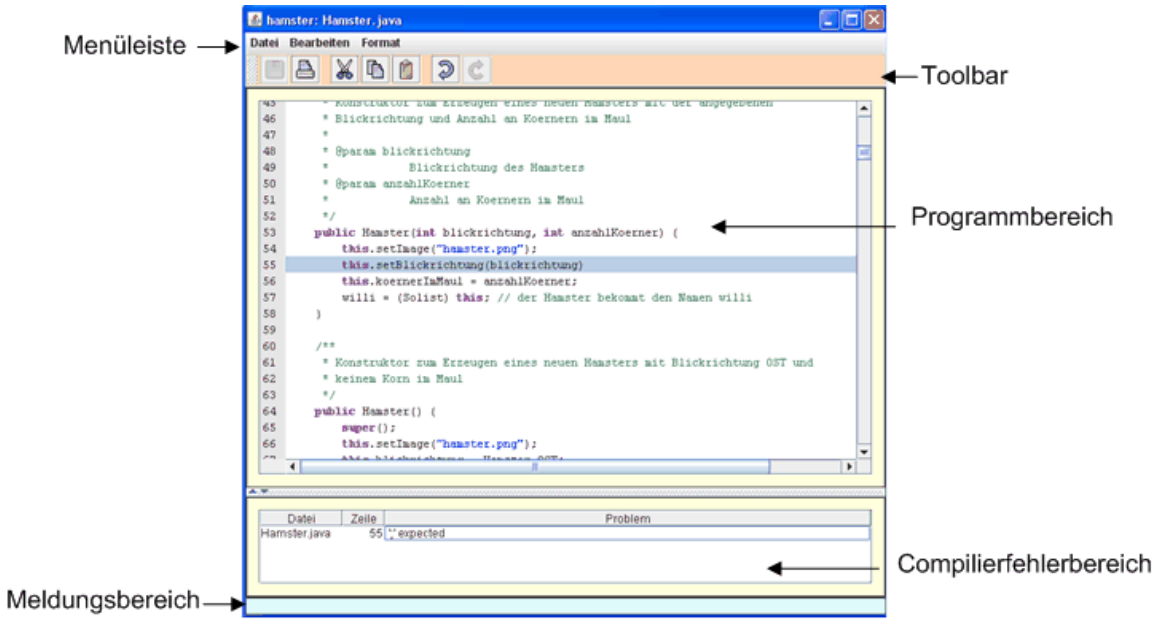



Abb. 5.11: Aufbau des Editors


Abb. 5.12: Datei-Menü

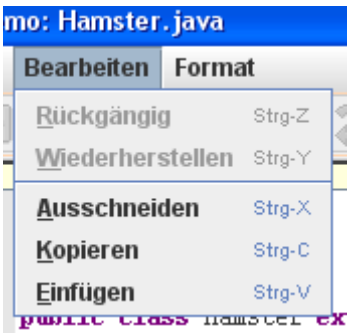



Abb. 5.13: Bearbeiten-Menü

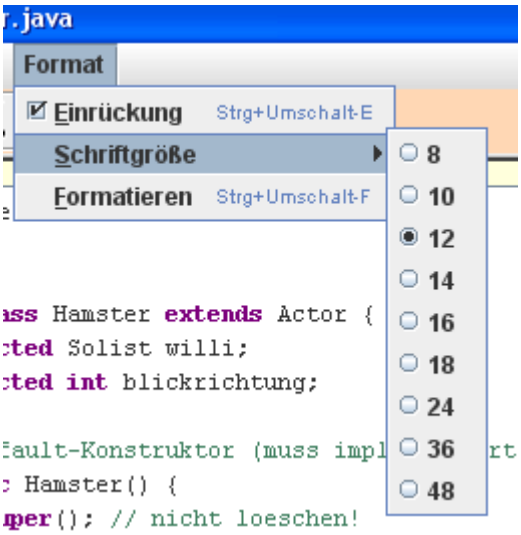



Abb. 5.14: Format-Menü

# 6 Theater-API

Die Theater-API oder –Klassenbibliothek besteht aus einer Menge von Klassen und Interfaces, auf deren Grundlage Solist-Theaterstücke 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** und **TheaterIcon**: Klassen zum Erzeugen und Manipulieren von Bildern und Icons
- **MouseInfo**: Klasse, die Informationen zu Maus-Ereignissen verwaltet
- **KeyInfo**: Klasse, die Informationen zu Tastatur-Ereignissen verwaltet
- **PixelArea**: Interface zur Umsetzung der Kollisionsentdeckung
- **Rectangle**: Klasse, die das Interface PixelArea für rechteckige Gebiete implementiert
- **Point**: Klasse, die das Interface PixelArea für einzelne Pixel implementiert
- **Cell**: Klasse, die das Interface PixelArea für einzelne Zellen einer Bühne implementiert
- **CellArea**: Klasse, die das Interface PixelArea für rechteckige Gebiete mit mehreren Zellen einer Bühne implementiert

## *6.1 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 auch noch 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)

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

### 6.1.2 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.

### 6.1.3 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.

### 6.1.4 Event-Methoden

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

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

## *6.2 Component, Actor und Prop*

Die Klasse `Component` 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 eine Methode namens `main` überschreiben können, die die Aktionen des Solisten definiert.

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)

### 6.2.1 Manipulationsmethoden

Mit Hilfe zweier `setImage`-Methoden ist es möglich, einer Komponente ein Bild oder Icon zuzuordnen, über das sie auf der Bühne repräsentiert wird. Zulässig sind Bilder

im gif-, jpg und png-Format. Bilddateien müssen im Unterverzeichnis „images“ des entsprechenden Theaterstück-Verzeichnisses gespeichert werden.

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

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

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

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

### 6.2.2 Getter-Methoden

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

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

### 6.2.3 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.

### 6.2.4 Event-Methoden

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

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

### 6.2.5 Solist

In Solist-Theaterstücken existiert immer genau ein Objekt einer Actor-Klasse. Das ist der so genannte Solist. Der Solist muss im Default-Konstruktor jeder Stage-Klasse erzeugt und auf der Bühne plaziert werden. Das geschieht durch Aufruf der folgenden Anweisung:

```
// erzeugt den Solist und platziert in oben links auf der Bühne
setSolist(Actor.createSolist(), 0, 0);
```

Die statische Methode `createSolist` der Klasse *Actor* erzeugt einen Solist als Instanz der gerade aktiven Actor-Klasse (wenn es nur eine Actor-Klasse gibt, wird automatisch diese genommen). Die Methode `setSolist` plaziert den erzeugten Solist auf die entsprechende Zelle der Bühne. Es ist nicht erlaubt und möglich, dass mehrere Solisten gleichzeitig existieren, es ist auch nicht möglich, den Solist explizit von der Bühne zu entfernen (man kann ihn jedoch unsichtbar machen) und es nicht möglich, den Solist zur Laufzeit eines Programms auszuwechseln.

## *6.3 Performance*

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

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

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

- `started`
- `stopped`
- `suspended`
- `resumed`
- `speedChanged`

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

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

Über die Methode `setActiveStage` ist es möglich, die aktuell aktive Bühne gegen eine andere Bühne auszutauschen, d.h. das Bühnenbild zu wechseln (die Methode ist aktuell allerdings nicht implementiert). 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.

## *6.4 Maus- und Tastatur-Events*

Sowohl die Klasse `Stage` als auch die Klasse `Component` definieren die von der Java-GUI-Programmierung bekannten Methoden zur Verarbeitung von Maus- und Tastatur-Events: `keyTyped`, `keyPressed`, `keyRealeased`, `mousePressed`, `mouseReleased`, `mouseClicked`, `mouseDragged`, `mouseMoved`, `mouseEntered`, `mouseExited`. Die Events entsprechen dabei den Events des Java-AWT in den Klassen `java.awt.event.KeyListener`, `java.awt.event.MouseListener` bzw. `java.awt.event.MouseMotion-Listener`. Zur Dokumentation sei auf die Java-AWT-Dokumentation verwiesen. Den Methoden werden Objekte vom Typ `KeyInfo` bzw. `MouseInfo` übergeben,

über die genauere Informationen über das entsprechende Event abgefragt werden können.

Soll der Solist, 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 in Solist nicht 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.

### 6.4.1 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.

### 6.4.2 MouseInfo

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

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

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

## *6.5 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 TheaterImages oder leer in einer gewünschten Größe. Im ersteren Fall sind Bilder im gif-, jpg und png-Format zulässig. Die Bilddateien müssen im Unterverzeichnis „images“ des entsprechenden Theaterstück-Verzeichnisses gespeichert werden.

Die wichtigsten Bild-Manipulations-Methoden der Klasse `TheaterImage` sind: `setColor`(legt die Zeichenfarbe fest), `setFont` (legt den Zeichenfont fest), `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.

## 6.6 Kollisionserkennung

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

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

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

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

Durch das Konzept der PixelArea ist eine Kollisionserkennung sehr flexibel ohne eine größere Menge an Methoden realisierbar. So kann sehr einfach abgefragt werden, ob sich bspw. zwei Akteure überlagern oder ob sich ein Akteur komplett im sichtbaren Bereich der Bühne befindet. Bei Bedarf kann ein Programmierer weitere Klassen definieren (bspw. Kreise oder Polygone), die das Interface `PixelArea` implementieren, womit sich dann ohne weitere Änderungen überprüfen ließe, ob sich bspw. der Solist in einem bestimmten durch ein Polygon definierten Bereich der Bühne aufhält.

# 7 Entwicklungsfunktionalitäten im Detail

Als Hauptfunktionsbereiche des Solist-Simulators lassen sich identifizieren:

- Anlegen und Verwalten von Klassen
- Verwalten und Editieren von Solist-Programmen
- Compilieren
- Gestalten und Verwalten der Bühne
- Interaktives Ausführen von Solist-Befehlen
- Ausführen von Solist-Programmen
- Debuggen von Solist-Programmen
- Anlegen und Verwalten von Aktionsbuttons
- Generieren einer PLU

Wir werden uns die einzelnen Funktionsbereiche nun im Detail anschauen und dabei unsere erste kleine MPW realisieren: ein ganz einfacher Hamster-Simulator, der einen Teil des Java-Hamster-Modells umsetzt. Starten Sie dazu Solist und erzeugen Sie durch Anklicken des ersten Buttons in der Toolbar ein neues Theaterstück. Wählen Sie den Namen „demo“.

## *7.1 Anlegen und Verwalten von Klassen*

Das Anlegen und Verwalten von Klassen erfolgt im Klassenbereich (siehe Abbildung 7.1 rechts). Hier befindet sich ein Elementbaum mit den Ordnern Stage, Actor, Prop, Performance und Other.

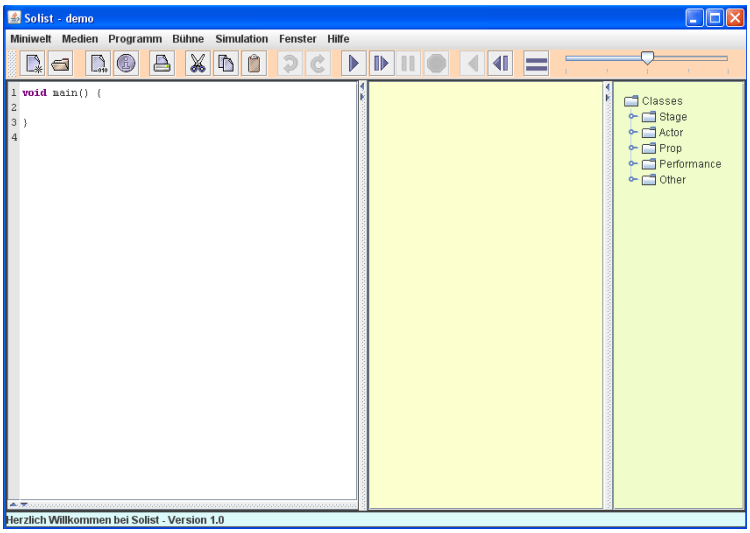



Abb. 7.1: Klassenbereich

Nach dem Erzeugen eines neuen Theaterstückes sind die einzelnen Ordner leer, d.h. es gibt noch keine Klassen.

### 7.1.1 Definition einer neuen Stage-Klasse

Als erstes werden wir nun eine Stage-Klasse erzeugen. Aktivieren Sie dazu oberhalb des Ordners *Stage* ein Popup-Menü (Klick mit der rechten Maustaste). Das Menü enthält ein einzelnes Item „Neue Unterklasse…“. Aktivieren Sie es und geben Sie in der erscheinenden Dialogbox für die Klasse einen Namen ein. Wir wählen den Namen „Territorium“. Nach dem Drücken des OK-Buttons erscheint das in Abbildung 7.2 skizzierte Bild. Wir haben eine neue Stage-Klasse (mit Dummy-Werten) erzeugt.

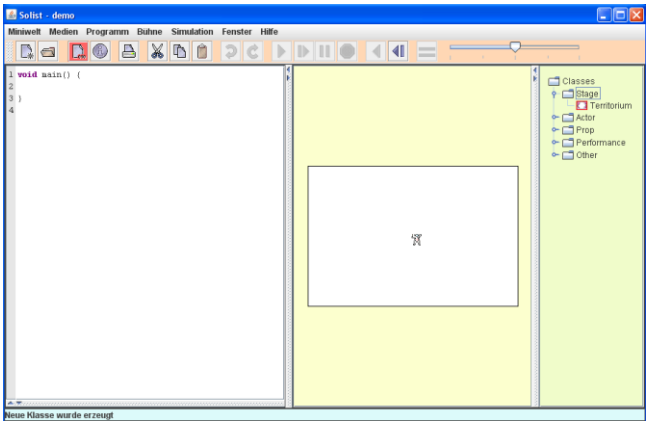



Abb. 7.2: Neue Stage-Klasse

Aktivieren Sie nun im Klassenbereich oberhalb des Namens „Territorium“ ein Popup-Menü. Wählen Sie darin das Item „Editor öffnen“ aus (alternativ können Sie auch einen Doppelklick auf den Namen „Territorium“ ausführen). Es öffnet sich ein Editor-Fenster, in dem die mit Dummy-Werten erzeugte Klasse „Territorium“ angezeigt wird (siehe Abbildung 7.3). Standardmäßig besteht das Territorium aus 30*20 Zellen, die jeweils 10 Pixel hoch und breit sind, und es wird ein Default-Solist in der Mitte des Territoriums erzeugt.

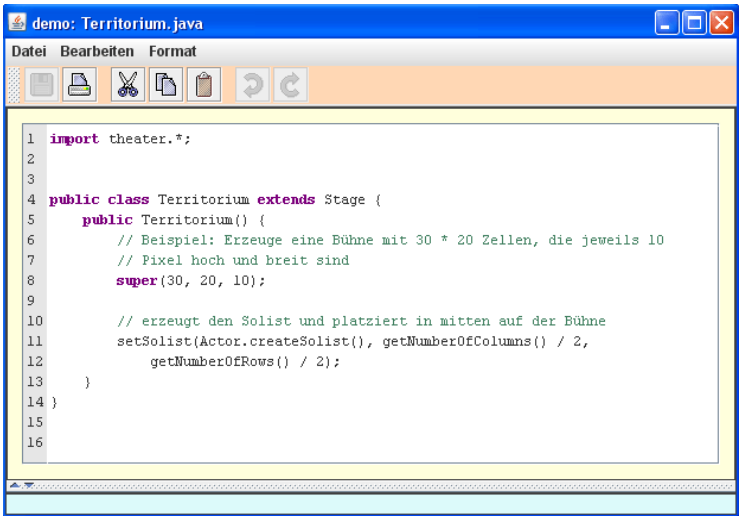



Abb. 7.3: Editor mit Stage-Klasse „Territorium“

Ändern und speichern Sie nun den Inhalt des Klasse Territorium wie folgt ab:

```
import theater.*;


public class Territorium extends Stage {
    public Territorium() {
        // Beispiel: Erzeuge eine Bühne mit 12 * 8 Zellen, die jeweils 35
        // Pixel hoch und breit sind
        super(12, 8, 35);

        setBackground("kachel.jpg");

        // erzeugt den Solist und platziert in oben links auf der Bühne
        setSolist(Actor.createSolist(), 0, 0);
    }
}
```

Insbesondere wollen wir der Bühne einen Hintergrund zuordnen, und zwar mit einem Bild. Um Bilder verfügbar zu haben, müssen wir Sie zunächst importieren. Aktivieren Sie dazu im Menü „Medien“ das Item „Bilder importieren“. Es erscheint eine spezielle Dateiauswahl-Dialogbox. Navigieren Sie in dieser Box in das Unterverzeichnis „images“ des Beispiel-Theaterstücks „hamster“, selektieren Sie alle Dateien und Klicken Sie auf „Selektieren“: Damit werden alle Bilder in das Unterverzeichnis „images“ unseres neuen Theaterstücks „demo“ kopiert (siehe auch Abbildung 7.4)

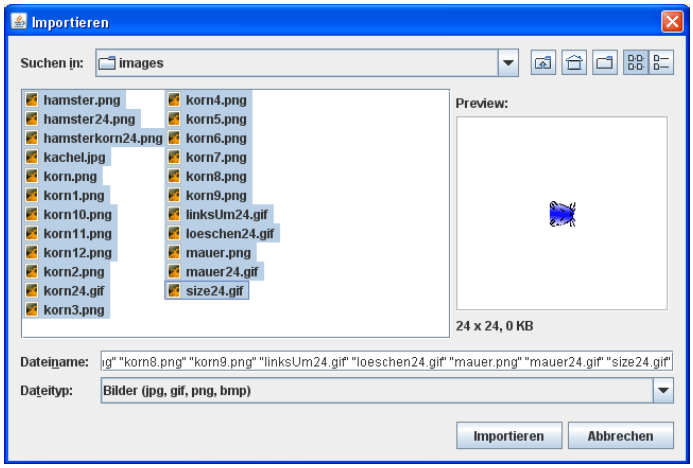



Abb. 7.4: Importieren von Bildern

Nach dem Abspeichern der geänderten Klasse *Territorium* erscheint der Compilieren-Button rot. Das deutet daraufhin, dass sich eine Klasse geändert hat und compiliert werden muss. Klicken Sie den Button an. Wenn Sie sich nicht vertippt haben und die Klasse fehlerfrei ist, erscheint nun das in Abbildung 7.5 skizzierte Territorium.

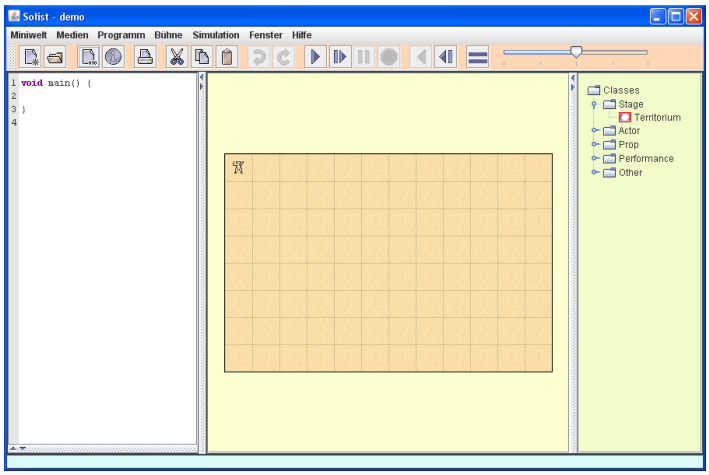



Abb. 7.5: Geändertes Territorium

### 7.1.2 Definition einer neuen Actor-Klasse

Nun wollen wir eine neue Actor-Klasse definieren, in der unser Solist, ein Hamster, implementiert wird. Dazu aktivieren wir im Klassenbereich auf dem Ordner *Actor* ein Popup-Menü, wählen das Item „Neue Unterklasse…“ und geben in der erscheinenden Dialogbox den Namen „Hamster“ ein. Nach dem Drücken des OK-Buttons erscheint im Ordner „Actor“ die Klasse „Hamster“. Durch Doppelklick öffnen wir den Editor der Klasse Hamster. Er hat die in Abbildung 7.6 skizzierte Gestalt.

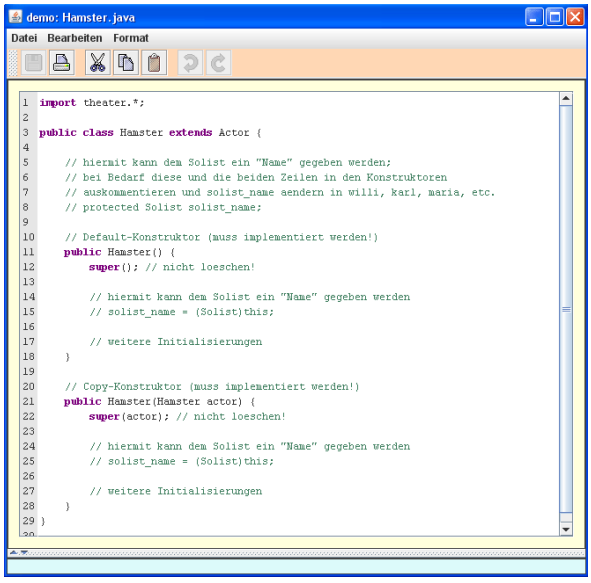



Abb. 7.6: Editor der Actor-Klasse Hamster

Ändern und speichern Sie nun den Inhalt der Klasse Hamster wie folgt ab:

```
import theater.*;

public class Hamster extends Actor {
    protected Solist willi;
    protected int blickrichtung;
```

```
    // Default-Konstruktor (muss implementiert werden!)
    public Hamster() {
        super(); // nicht loeschen!
        setImage("hamster.png");

        // hiermit wird dem Solist der Name willi gegeben
        willi = (Solist) this;

        // weitere Initialisierungen
        blickrichtung = 0; // Default = Osten
    }

    // Copy-Konstruktor (muss implementiert werden!)
    public Hamster(Hamster actor) {
        super(actor); // nicht loeschen!

        // hiermit wird dem Solist der Name willi gegeben
        willi = (Solist) this;

        // weitere Initialisierungen
        blickrichtung = actor.blickrichtung;
    }

    @Description("Der Hamster dreht sich um 90 Grad nach links um.")
    public void linksUm() {
        this.blickrichtung = (this.blickrichtung + 1) % 4;
        this.setRotation(this.getRotation() - 90);
    }
}
```

Drei Dinge sind hierbei zu erwähnen. Zum einen ordnen wir dem Hamster ein Bild zu („hamster.png“), und zum anderen geben wir ihm den Namen „willi“, über den wir ihn hinterher in Solist-Programmen ansprechen können. Weiterhin definieren wir den zunächst einzigen Hamster-Befehl *linksUm* in der entsprechenden Methode. Achten Sie bitte darauf, dass in Actor-Klassen immer ein Default- und ein Copy-Konstruktor in der oben skizzierten Art und Weise implementiert werden müssen. Nach dem Compilieren erscheint das Hauptfenster in der in Abbildung 7.7 skizzierten Form.

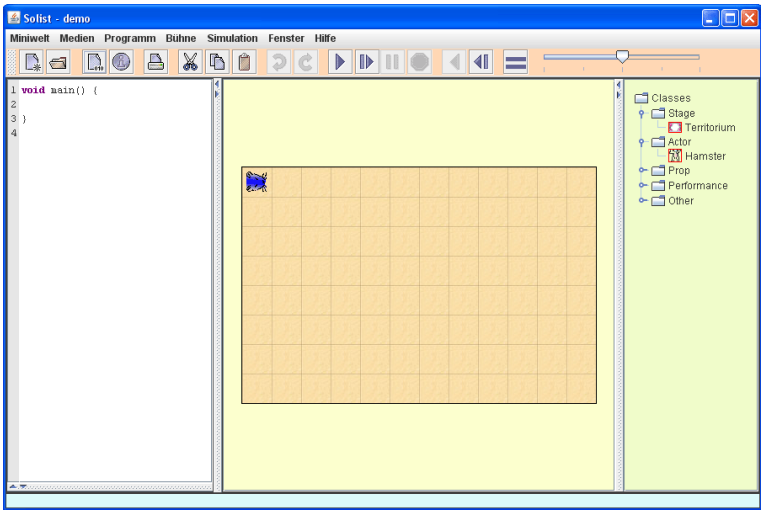



Abb. 7.7: Geänderte Hamster-Klasse

Aktivieren Sie nun oberhalb des Hamster-Ikons auf der Bühne ein Popup-Menü und wählen Sie im Untermenü „Hamster“ den Eintrag „linksUm“ aus (siehe Abbildung 7.8). Der Hamster dreht sich um 90 Grad nach links.

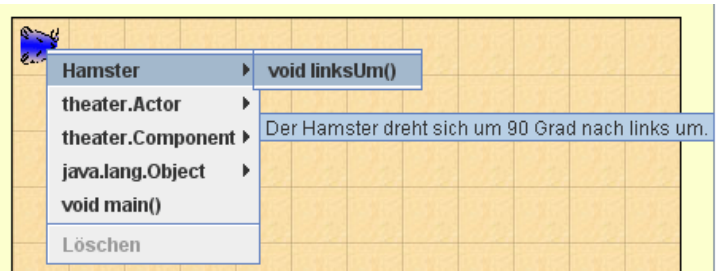



Abb. 7.8: Hamster-Popup-Menü

Alternativ können Sie auch über das Menü „Fenster“ das Befehlsfenster öffnen. Hierin erscheint ebenfalls der Befehl „linksUm“ (siehe Abb. 7.9). Bei jedem Anklicken dreht sich der Hamster um 90 Grad nach links. Achten Sie darauf, dass wenn Sie mit der Maus etwas länger über dem Item „linksUm“ verweilen, ein Tooltip erscheint. Dieser enthält die Beschreibung, die wir bei der Definition der Methode *linksUm* in der Klasse *Hamster* in der Annotation *Description* angegeben haben.


Abb. 7.9: Das Befehlsfenster

Per Drag-and-Drop über das Hamster-Ikon können Sie den Hamster auch auf andere Kacheln verschieben und Sie können auch andere (geerbte) Befehle über das Popup-Menü aufrufen.

Übrigens können wir nun auch bereits unser erstes Hamster-Programm schreiben und ausführen. Geben Sie dazu im Solist-Editor des Hauptfensters folgendes Solist-Programm ein:

```
void main() {
    for (int i=0; i < 10; i++) {
        linksUm();
    }
}
```

Compilieren Sie und führen Sie dann das Programm durch Anklicken des Start-Buttons in der Toolbar aus: Der Hamster dreht sich 10 Mal linksum.

Da Sie dem Hamster den Namen *willi* gegeben haben, können Sie Befehle übrigens auch in einer eher objektorientierten Notation über den Namen *willi* aufrufen. Das gilt auch für neu definierte Prozeduren und Funktionen:

```
void main() {
```

```
    for (int i=0; i < 10; i++) {
        willi.rechtsUm();
    }
}

void rechtsUm() {
    willi.linksUm();
    willi.linksUm();
    willi.linksUm();
}
```

### 7.1.3 Definition einer neuen Prop-Klasse

Nun wollen wir eine neue Prop-Klasse definieren, über die sich das Platzieren von Mauern im Territorium realisieren lässt. Dazu aktivieren wir im Klassenbereich auf dem Ordner *Prop* ein Popup-Menü, wählen das Item „Neue Unterklasse…“ und geben in der erscheinenden Dialogbox den Namen „Mauer“ ein. Nach dem Drücken des OK-Buttons erscheint im Ordner „Prop“ die Klasse „Mauer“. Durch Doppelklick öffnen wir den Editor der Klasse *Mauer*. Er hat die in Abbildung 7.10 skizzierte Gestalt.

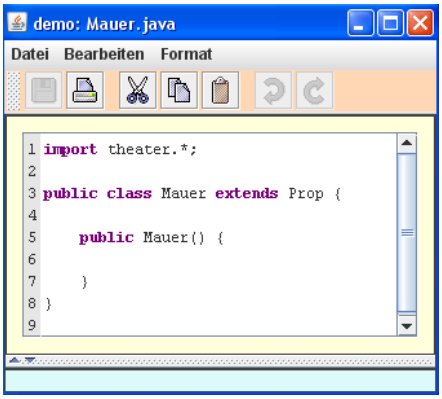



Abb. 7.10: Editor der Klasse Mauer

Ändern Sie die Klasse nun wie folgt ab:

```
import theater.*;

public class Mauer extends Prop {

    public Mauer() {
        setImage("mauer.png");
    }
}
```

Compilieren Sie anschließend. Aktivieren Sie dann im Klassenbereich oberhalb von „Mauer“ ein Popup-Menü und wählen Sie darin das Item „new Mauer()“ aus. Es heftet sich ein Mauer-Ikon an die Maus. Wenn Sie dann auf der Bühne eine Kachel anklicken, wird die Mauer dort abgelegt. Wierholen Sie dies, bis sich bspw. das in

Abbildung 7.11 skizzierte Territorium ergibt. Auch Mauern können Sie im Territorium per Drag-and-Drop umplatzieren und auch für Mauern können Sie ein Popup-Menü aktivieren und darüber geerbte Befehle aufrufen. Über das Item „Löschen“ im Popup-Menü ist es auch möglich, Mauern wieder zu löschen.

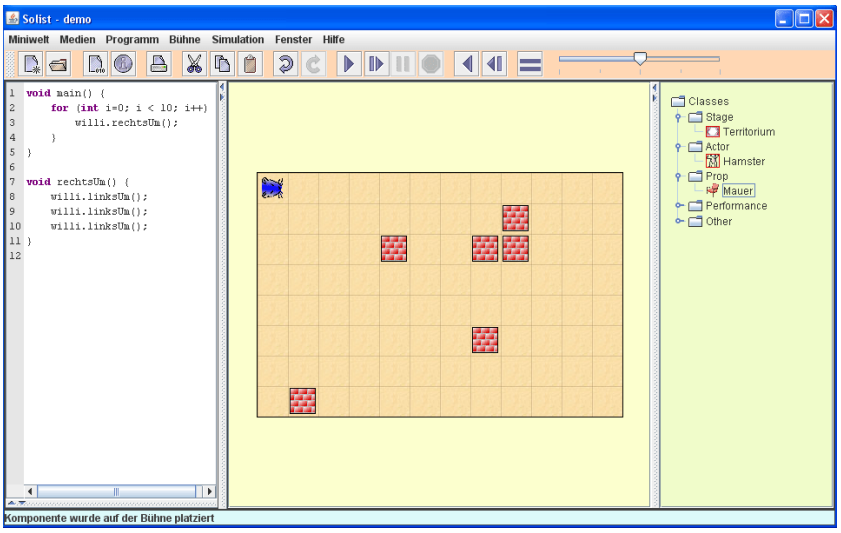



Abb. 7.11: Bühne mit Mauern

### 7.1.4 Definition einer neuen Performance-Klasse

Nun wollen wir eine neue Performance-Klasse definieren, über die beim Starten eines Solist-Programms die Geschwindigkeit immer automatisch auf die Mitte eingestellt wird. Dazu aktivieren wir im Klassenbereich auf dem Ordner Performance ein Popup-Menü, wählen das Item „Neue Unterklasse…“ und geben in der erscheinenden Dialogbox den Namen „Perf1“ ein. Nach dem Drücken des OK-Buttons erscheint im Ordner „Performance“ die Klasse „Perf1“. Durch Doppelklick öffnen wir den Editor der Klasse Perf1 und geben dort folgendes ein:

```
import theater.*;

public class Perf1 extends Performance {

    public void started() {
        setSpeed((Performance.MAX_SPEED + Performance.MIN_SPEED) / 2);
    }
}
```

Anschließend compilieren wir und klicken dann mit der linken Maus auf den Namen „Perf1“ im Klassenbereich. Dadurch sollte das Ikon vor dem Namen rot umrandet werden. Das zeigt an, dass diese Klasse die nun aktive Performance-Klasse ist. Wenn Sie nun ein Solist-Programm ausführen, wird beim Start immer automatisch die Geschwindigkeit auf die Mitte eingestellt, was sie am Geschwindigkeitsregler des Hauptfensters sehen können.

Theoretisch können Sie auch mehrere Performance-Klassen definieren und müssen dann vor dem Ausführen von Programmen bzw. dem Generieren einer PLU die zu berücksichtigende Performance-Klasse anklicken und damit aktivieren.

### 7.1.5 Definition einer neuen Hilfs-Klasse

Nun wollen wir eine neue Hilfs-Klasse definieren, in unserem Fall eine Exception-Klasse. Dazu aktivieren wir im Klassenbereich auf dem Ordner *Other* ein Popup-Menü, wählen das Item „Neue Unterklasse…“ und geben in der erscheinenden Dialogbox den Namen „MauerDaException“ ein. Nach dem Drücken des OK-Buttons erscheint im Ordner „Other“ die Klasse „MauerDaException“. Durch Doppelklick öffnen wir den Editor der Klasse MauerDaException und geben dort folgendes ein:

```
import theater.*;

public class MauerDaException extends RuntimeException {

    public MauerDaException() {
    }

    public String getMessage() {
        return "Vor dem Hamster befindet sich eine Mauer!";
    }
}
```

### 7.1.6 Erweitern der Klasse Hamster

Mit den beiden gerade definierten Klassen *Mauer* und *MauerDaException* wollen wir nun unsere MPW um zwei weitere Befehle *vor* und *vornFrei* erweitern. Dazu öffnen wir wieder den Editor der Klasse Hamster (Doppelklick auf „Hamster“ im Klassenbereich) und erweitern die Klasse *Hamster* durch die beiden folgenden Methoden:

```
    @Description("Liefert genau dann true, wenn sich vor dem Hamster keine Mauer " +
    "und auch nicht der Rand des Territoriums befindet.")
    public boolean vornFrei() {
        // ermittle aktuelle Position
        int col = this.getColumn();
        int row = this.getRow();

        // ermittle vorne-Position in Abhaengigkeit von Blickrichtung
        switch (this.blickrichtung) {
        case 0: // Ost
            col++;
            break;
        case 1: // Nord
            row--;
            break;
        case 2: // West
            col--;
            break;
        case 3: // Sued
            row++;
            break;
        }

        // ueberpruefe vorne-Position
        if ((col >= this.getStage().getNumberOfColumns()) ||
                (row >= this.getStage().getNumberOfRows()) || (col < 0) ||
                (row < 0)) {
            // Hamster befindet sich am Rand
            return false;
        }
```

```
    // Mauer vor dem Hamster?
    return this.getStage().getComponentsAt(col, row, Mauer.class).size() == 0;
}

@Description("<html>Der Hamster hüpft eine Kachel in seiner aktuellen " +
             "Blickrichtung nach " +
             "vorn.<br>Fehler, wenn sich dort eine Mauer oder der Rand " +
             "des Territoriums befindet.</html>")
public void vor() throws MauerDaException {
    if (!this.vornFrei()) {
        throw new MauerDaException();
    }

    switch (this.blickrichtung) {
    case 0: // Ost
        this.setLocation(this.getColumn() + 1, this.getRow());
        break;
    case 1: // Nord
        this.setLocation(this.getColumn(), this.getRow() - 1);
        break;
    case 2: // West
        this.setLocation(this.getColumn() - 1, this.getRow());
        break;
    case 3: // Sued
        this.setLocation(this.getColumn(), this.getRow() + 1);
        break;
    }
}
```

Überprüfen Sie die Korrektheit dieser beiden Methoden, indem Sie nach dem Compilieren das Befehlsfenster öffnen und die entsprechenden Items anklicken (siehe Abbildung 7.12)

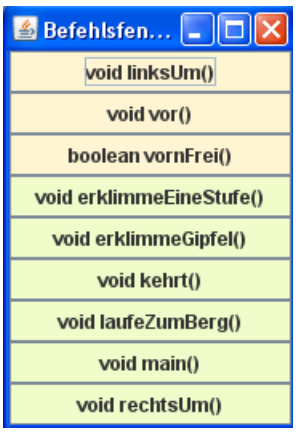



Abb. 7.12: Erweitertes Befehlsfenster

### 7.1.7 Klassen löschen

Wenn Sie oberhalb eines Klassennamens im Klassenbereich ein Popup-Menü aktivieren, erscheint dort auch ein Item „Klasse löschen“. Hierüber können Sie die Klasse löschen. Das Umbenennen oder Verschieben von Klassen in einen anderen Ordner ist in der aktuellen Version des Solist-Simulators leider nicht möglich.

## *7.2 Verwalten und Editieren von Solist-Programmen*

Nun sind wir auch soweit, dass wir komplette Solist-Programme schreiben bzw. verwalten können. Geben Sie bspw. das folgende Programm in den Editorbereich des Hauptfensters (der so genannte Solist-Editor) ein.

```
/* Aufgabe:
Der Hamster steht mit Blickrichtung Osten
vor einem Berg mit regelmäßigen
jeweils eine Kachel hohen Stufen.
Er bekommt die Aufgabe, den Berg zu erklimmen
und auf dem Gipfel stehen zu bleiben.
*/

void main() {
    laufeZumBerg();
    erklimmeGipfel();
}

void laufeZumBerg() {
    while (vornFrei()) {
        vor();
    }
}

void erklimmeGipfel() {
    do {
        erklimmeEineStufe();
    } while (!vornFrei());
}

void erklimmeEineStufe() {
    linksUm();
    vor();
    rechtsUm();
    vor();
}

void rechtsUm() {
    kehrt();
    linksUm();
}

void kehrt() {
    linksUm();
    linksUm();
}
```

Unmittelbar nach der Eingabe des ersten Zeichens erscheint der Compilierbutton in der Toolbar rot. Das ist ein Zeichen dafür, dass kompiliert werden muss, um die Änderungen wirksam werden zu lassen. Nach einem Speichern-Button suchen Sie umsonst. Den gibt es im Solist-Simulator nicht. Stattdessen wird Ihr Programm, wenn Sie auf den Compilier-Button drücken automatisch in einer Datei namens *Solist.java* abgespeichert.

Compilieren Sie nun. Wenn Sie keine Fehler eingetippt haben, erscheint eine Dialogbox mit der Meldung „Compilierung erfolgreich“ und die rote Frabe des Compilier-Buttons verschwindet. Öffnen Sie nun einmal über das Menü „Fenster“ das Befehlsfenster. Neben den in der Klasse Hamster definierten Befehlen erscheinen

dort auch alle in Ihrem Solist-Programm definierten Prozeduren und Funktionen (siehe auch Abbildung 7.12).

Für das Verwalten und Editieren von Programmen ist das Menü „Programm“ wichtig. Unterhalb der Menüleiste ist eine spezielle Toolbar zu sehen, über die Sie die wichtigsten Funktionen der Menüs auch schneller erreichen und ausführen können. Schieben Sie einfach mal die Maus über die Buttons. Dann erscheint jeweils ein Tooltipp, der die Funktionalität des Buttons anzeigt. Die für das Editieren von Programmen wichtigen Buttons der Toolbar werden in Abbildung 7.13 skizziert.

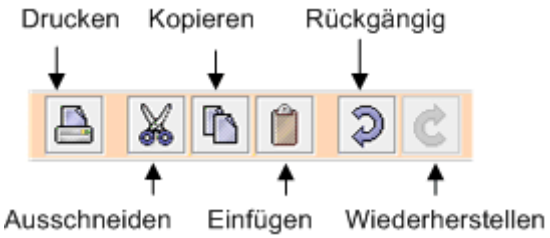



Abb. 7.13: Editor-Buttons der Toolbar

### 7.2.1 Schreiben eines neuen Solist-Programms

Das Schreiben eines neuen Solist-Programms ist durch das entsprechende Eintippen des Sourcecodes im Editor-Bereich möglich.

### 7.2.2 Ändern des aktuellen Solist-Programms

Möchten Sie Teile des aktuellen Solist-Programms ändern, klicken Sie im Editor-Bereich einfach an die entsprechende Stelle und fügen dort die entsprechenden Wörter ein oder löschen sie.

### 7.2.3 Löschen des aktuellen Solist-Programm

Komplett löschen können Sie das aktuelle Solist-Programm des Editor-Bereichs, indem Sie den kompletten Sourcecode mit der Maus selektieren und dann in der Toolbar den „Ausschneiden“-Button (6. Button von links) drücken. Alternativ kann im Programm-Menü das Menü-Item „Neu“ angeklickt werden.

### 7.2.4 Abspeichern des aktuellen Solist-Programms

Normalerweise müssen Sie sich nicht um das Speichern des aktuellen Programms kümmern. Es wird automatisch vor dem Compilieren in einer internen Datei abgespeichert. Wenn Sie jedoch ein Programm explizit dauerhaft abspeichern möchten, können Sie im „Programm“-Menü das „Speichern unter…“-Menü-Item anklicken. Es öffnet sich eine Dateiauswahl-Dialogbox, über die Sie die gewünschte Datei auswählen können.

### 7.2.5 Öffnen eines einmal abgespeicherten Solist-Programms

Möchten Sie ein einmal abgespeichertes Solist-Programm wieder in den Editorbereich laden, können Sie dies über das Menü-Item „Laden…“ des „Programm“-Menüs tun. Nach dem Anklicken des Items erscheint eine Dateiauswahl-Dialogbox, über die Sie die gewünschte Datei auswählen können.

Achtung: Beim Laden einer abgespeicherten Datei geht der aktuelle Inhalt des Editor-Bereichs verloren! Sie müssen ihn also gegebenenfalls vorher in einer Datei abspeichern.

### 7.2.6 Drucken eines Solist-Programms

Über den „Drucken“-Button (5. Toolbar-Button von links) können Sie das aktuelle Programm des Editor-Bereichs drucken. Es öffnet sich eine Dialogbox, in der Sie die entsprechenden Druckeinstellungen vornehmen und den Druck starten können.

### 7.2.7 Editier-Funktionen

Im Editor-Bereich können Sie – wie bei anderen Editoren auch – über die Tastatur Zeichen eingeben bzw. wieder löschen. Darüber hinaus stellt der Editor ein paar weitere Funktionalitäten zur Verfügung, die über das „Programm“-Menü bzw. die entsprechenden Editor-Buttons in der Toolbar aktiviert werden können.

- „Ausschneiden“-Button (6. Toolbar-Button von links): Hiermit können Sie komplette Passagen des Editor-Bereichs in einem Schritt löschen. Markieren Sie die zu löschende Passage mit der Maus und klicken Sie dann den Button an. Der markierte Text verschwindet.
- „Kopieren“-Button (7. Toolbar-Button von links): Hiermit können Sie komplette Passagen des Editor-Bereichs in einen Zwischenpuffer kopieren. Markieren Sie die zu kopierende Passage mit der Maus und klicken Sie dann den Button an.
- „Einfügen“-Button (8. Toolbar-Button von links): Hiermit können Sie den Inhalt des Zwischenpuffers an die aktuelle Cursorposition einfügen. Wählen Sie zunächst die entsprechende Position aus und klicken Sie dann den Button an. Der Text des Zwischenpuffers wird eingefügt.
- „Rückgängig“-Button (9. Toolbar-Button von links): Wenn Sie durchgeführte Änderungen des Sourcecode – aus welchem Grund auch immer – wieder rückgängig machen wollen, können Sie dies durch Anklicken des Buttons bewirken.
- „Wiederherstellen“-Button (10. Toolbar-Button von links): Rückgängig gemachte Änderungen können Sie mit Hilfe dieses Buttons wieder herstellen.

Die Funktionalitäten „Kopieren“ und „Einfügen“ funktionieren übrigens auch über einzelne Programme hinaus. Es ist sogar möglich, mit Hilfe der Betriebssystem-Kopieren-Funktion Text aus anderen Programmen (bspw. Microsoft Word) zu kopieren und hier einzufügen.

Die gerade aufgelisteten Funktionen finden Sie auch im „Programm“-Menü. Als zusätzliche Funktionalitäten werden dort angeboten:

- Einrückung: Durch Anklicken des Menü-Items wird der Einrück-Modus aktiviert bzw. deaktiviert. Ist der Einrück-Modus aktiviert, werden beim Eingeben von Text im Editor-Bereich automatisch Einrückungen an die entsprechende Spalte vorgenommen, wenn Sie die Enter-Taste drücken.
- Schriftgröße: Hierüber können Sie die Schriftgröße des Editor-Bereichs anpassen.

## *7.3 Compilieren*

Beim Compilieren werden Programme – genauer gesagt der Sourcecode – auf ihre (syntaktische) Korrektheit überprüft und im Erfolgsfall ausführbare Programme erzeugt. Zum Compilieren von Programmen dient im Hamster-Simulator der „Compilieren“-Button (dritter Button der Toolbar von links) bzw. das Menü-Item „Compilieren“ im „Programm“-Menü.

An der Farbe des Compilier-Buttons können Sie erkennen, ob Compilieren aktuell notwendig ist oder nicht. Immer wenn Sie Änderungen im Editor-Bereich vorgenommen haben, erscheint der Button rot, und die Änderungen werden erst dann berücksichtigt, wenn Sie (erneut) compiliert haben. Erscheint der Button in einer neutralen Farbe, ist kein Compilieren notwendig.

### 7.3.1 Compilieren

Wenn Sie den „Compilieren“-Button anklicken, wird das Programm, das gerade im Editor-Bereich sichtbar ist, in einer internen Datei (mit dem Namen „Solist.java“) abgespeichert und kompiliert. Außerdem werden auch alle anderen Klassen, die Sie im Klassenbereich angelegt haben, kompiliert,

Wenn Ihr Programm syntaktisch korrekt ist, erscheint nach ein paar Sekunden eine Dialogbox mit einer der Meldung „Compilierung erfolgreich“. Es wurde ein (neues) ausführbares Programm erzeugt.

### 7.3.2 Beseitigen von Fehlern

Wenn das Programm des Editor-Bereichs oder eine Klasse des Klassenbereichs Fehler enthält, erscheint eine Dialogbox, in der alle fehlerhaften Programme bzw. Klassen aufgelistet werden. Alle Editore von fehlerhaften Klassen werden automatisch geöffnet. In jedem dieser Editore öffnet sich unterhalb des Editor-Bereichs in einer Scroll-Pane ein neuer Bereich, der die Fehlermeldungen des Compilers anzeigt (siehe Abbildung 7.14). Jede Fehlermeldung erscheint in einer eigenen Zeile. Jede Zeile enthält eine Beschreibung des (wahrscheinlichen) Fehlers sowie die entsprechende Zeile der Anweisung im Programm. Wenn Sie eine Fehlermeldung anklicken, wird die entsprechende Anweisung im Eingabebereich blau markiert und der Maus-Cursor an die entsprechende Stelle gesetzt. Sie müssen nun die einzelnen Fehler beseitigen und dann erneut speichern und compilieren, bis

Ihr Programm und Ihre Klassen keine Fehler mehr enthalten. Der Fehlermeldungsbereich schließt sich dann automatisch wieder.

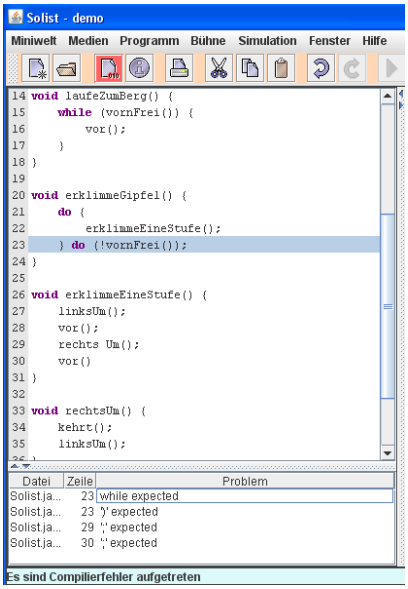



Abb. 7.14: Fehlermeldungen des Compilers

Achtung: Die Interpretation von Fehlermeldungen, die der Compiler ausgibt, ist nicht trivial. Die Meldungen sind nicht immer besonders präzise und oft auch irreführend. Häufig gibt der Compiler mehrere Fehlermeldungen aus, obwohl es sich nur um einen einzelnen Fehler handelt. Deshalb beherzigen Sie gerade am Anfang folgende Hinweise: Arbeiten Sie die Fehlermeldungen immer von oben nach unten ab. Wenn der Compiler eine große Menge von Fehlermeldungen liefert, korrigieren Sie zunächst nur eine Teilmenge und kompilieren Sie danach erneut.

## *7.4 Gestalten und Verwalten der Bühne*

Bevor wir uns die nächsten Funktionalitäten anschauen, gestalten Sie zunächst einmal das Territorium, so dass es wie in Abbildung 7.15 skizziert ausschaut

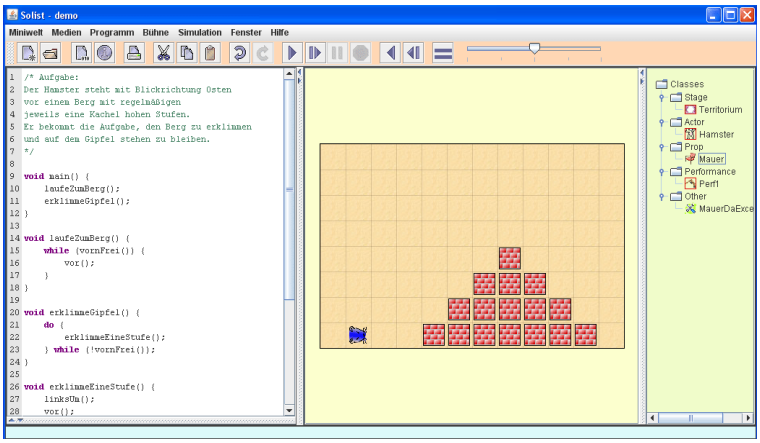



Abb. 7.15: Bühne gestalten

Der Hamster ist der Solist. In einer Solist-MPW existiert immer genau ein Solist. Es ist nicht möglich, mehrere Solisten zu erzeugen und es ist auch nicht möglich, den einen existierenden Solist zu löschen.

Dahingegen können von den von Ihnen definierten Prop-Klassen neue Objekte erzeugt und auf der Bühne platziert werden. Aktivieren Sie bspw. im Klassenbereich oberhalb von „Mauer“ ein Popup-Menü und wählen Sie darin das Item „new Mauer()“ aus. Es heftet sich ein Mauer-Ikon an die Maus. Wenn Sie dann auf der Bühne eine Kachel anklicken, wird die Mauer dort abgelegt.

Alle Komponenten der Bühne können Sie im Territorium per Drag-and-Drop auf dem entsprechenden Ikon umplatzieren und für alle Komponenten der Bühne können Sie über deren Ikon ein Popup-Menü aktivieren und darüber (geerbte) Befehle aufrufen. Über das Item „Löschen“ im Popup-Menü ist es auch möglich, Prop-Komponenten wieder zu löschen.

### 7.4.1 Abspeichern der Bühne

Sie können einmal gestaltete Bühnen in einer Datei abspeichern und später wieder laden. Zum Abspeichern der aktuellen Bühne aktivieren Sie im Menü „Bühne“ das Menü-Item „Speichern unter…“. Es öffnet sich eine Dateiauswahl-Dialogbox. Hierin können Sie den Ordner auswählen und den Namen einer Datei eingeben, in die das aktuelle Territorium gespeichert werden soll.

Weiterhin ist es möglich, die aktuelle Bühne als Bild (gif- oder png-Datei) abzuspeichern. Eine entsprechende Funktion findet sich im „Bühne“-Menü.

### 7.4.2 Wiederherstellen einer abgespeicherten Bühne

Abgespeicherte Bühnen können mit dem „Laden“-Menü-Item des „Bühne“-Menüs wieder geladen werden. Es erscheint eine Dateiauswahl-Dialogbox, in der Sie die zu ladende Datei auswählen können. Nach dem Anklicken des OK-Buttons schließt sich die Dialogbox und die entsprechende ist wiederhergestellt.

Achtung: Der Zustand der Bühne, der vor dem Ausführen der Bühne-Laden-Funktion Gültigkeit hatte, geht dabei verloren. Speichern Sie ihn daher gegebenenfalls vorher ab.

Hinweis: Das Wiederherstellen von abgespeicherten Bühnen ist nur möglich, wenn in der Zwischenzeit die Klassen des Klassenbereichs nicht verändert worden sind.

## *7.5 Interaktives Ausführen von Solist-Befehlen*

Sie können einzelne Solist-Befehle – so werden die von Ihnen in Ihrer Actor-Klasse definierten Befehle genannt - oder selbst definierte Funktionen und Prozeduren bspw. zu Testzwecken auch interaktiv ausführen. Aktivieren Sie dazu im Menü „Fenster“ den Eintrag „Befehlsfenster“. Es öffnet sich das so genannte Befehlsfenster (siehe Abbildung 7.15).

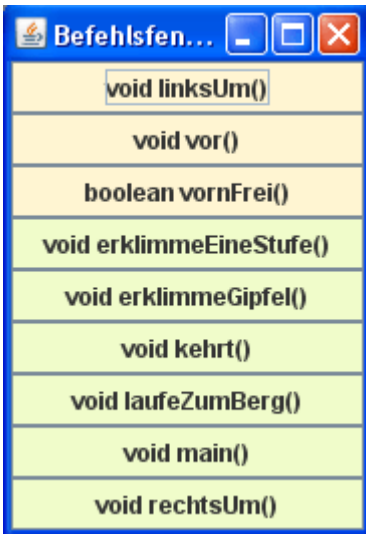



Abb. 7.15: Befehlsfenster

### 7.5.1 Befehlsfenster

Im Befehlsfenster werden die von Ihnen definierten Solist-Befehle (mit orangenem Hintergrund) sowie die Prozeduren und Funktionen dargestellt, die im aktuellen Soliost-Programm beim letztmaligen erfolgreichen Compilieren definiert waren (mit grünlichem Hintergrund).

Beim Anklicken mit der Maus werden die entsprechenden Befehle jeweils ausgeführt. Die Ausführung erfolgt dabei ohne Anzeige von Zwischenzuständen. D.h. wird bspw. eine Prozedur `rechtsUm` dadurch definiert, dass dreimal der Befehl `linksUm` aufgerufen wird, und wird diese Prozedur `rechtsUm` im Befehlsfenster aktiviert, dreht sich der Hamster tatsächlich nur einmal um 90 Grad nach rechts.

Dauert die Ausführung eines Befehls mehr als 5 Sekunden (vermutlich enthält dann die Funktion eine Endlosschleife), wird der Befehl abgebrochen und der Solist-Simulator wird komplett auf seinen Startzustand zurückgesetzt.

### 7.5.2 Parameter

Besitzt eine Prozedur oder Funktion Parameter, erscheint nach ihrer Aktivierung im Befehlsfenster eine Dialogbox, in der die entsprechenden aktuellen Parameterwerte eingegeben werden müssen. Hinweis: Aktuell wird nur die Eingabe von Werten der Java-Standard-Datentypen (`int`, `boolean`, `float`, ...) sowie die Eingabe von Zeichenketten (Strings) unterstützt.

### 7.5.3 Rückgabewerte von Funktionen

Bei der Ausführung von Funktionen wird der jeweils gelieferte Wert in einem Dialogfenster dargestellt.

### 7.5.4 Befehls-Popup-Menü

Alternativ zur Benutzung des Befehlsfensters ist es auch möglich, über ein Popup-Menü die Solist-Befehle interaktiv auszuführen. Sie können dieses Popup-Menü durch Drücken der rechten Maustaste oberhalb des Solist-Ikons im Territorium aktivieren (siehe Abbildung 7.16).

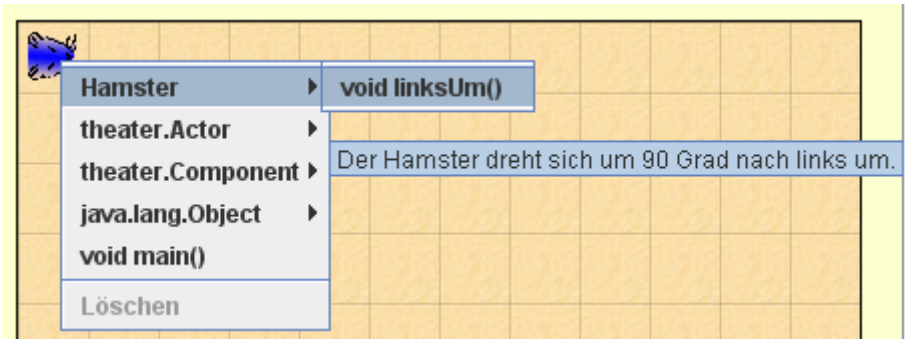



Abb. 7.16: Befehls-Popup-Menü

Auch für Prop-Komponenten ist es auf diese Art und Weise möglich, deren Methoden auszuführen.

In das Popup-Menü bzw. Befehlsfenster aufgenommen werden dabei alle Methoden der entsprechenden Klasse, selbst als private deklarierte Methoden. Möchten Sie, dass eine bestimmte Methode nicht im Popup-Menü oder im Befehlsfenster erscheint, müssen Sie sie mit der Annotation Invisible aus dem Paket theater markieren. Probieren Sie dies einmal dadurch aus, dass Sie im aktuellen Solist-Programm die Methode kehrt entsprechend markieren:

```
@theater.Invisible
void kehrt() {
    linksUm();
    linksUm();
}
```

Nach dem Kompilieren erscheint diese Methode nun nicht mehr im Befehlsfenster und Popup-Menü des Hamsters.

### 7.5.5 Klassen-Popup-Menü

Wenn Sie oberhalb einer Klasse im Klassenbereich ein Popup-Menü aktivieren, werden dort die Konstruktoren der Klasse sowie alle static-Methoden der Klasse aufgelistet. Wenn Sie das entsprechende Item anklicken, wird die Methode ausgeführt. Auch hier können Sie die Annotation Invisible einsetzen, um eine Methode bzw. einen Konstruktor nicht erscheinen zu lassen.

## *7.6 Ausführen von Solist-Programmen*

Ausgeführt werden können (erfolgreich compilierte) Solist-Programme mit Hilfe der in Abbildung 7.17 skizzierten Buttons der Toolbar. Alle Funktionen sind darüber hinaus auch über das Menü „Simulation“ aufrufbar.

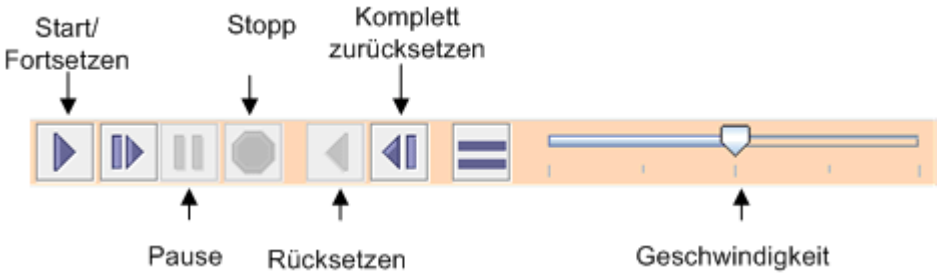



Abb. 6.17: Simulationsbuttons der Toolbar

### 7.6.1 Starten eines Solist-Programms

Bevor ein Solist-Programm ausgeführt werden kann, muss es erfolgreich compiliert worden sein. Gestartet werden kann das aktuelle Hamster-Programm dann durch Anklicken des „Start/Fortsetzen“-Buttons (11. Toolbar-Button von links).

Nach dem Starten eines Solist-Programms wird der Solist auf der Bühne aktiv und tut das, was das Programm ihm vorgibt. Während des Ausführens eines Solist-Programms wird der Editor-Bereich ausgegraut, d.h. es können während der Ausführung eines Programms keine Änderungen am Sourcecode durchgeführt werden.

### 7.6.2 Stoppen eines Solist-Programms

Die Ausführung eines Solist-Programms kann durch Anklicken des „Stopp“-Buttons (14. Button der Toolbar von links) jederzeit abgebrochen werden.

### 7.6.3 Pausieren eines Hamster-Programms

Möchten Sie ein in Ausführung befindliches Programm (kurzfristig) anhalten, können Sie dies durch Anklicken des „Pause“-Buttons (13. Button der Toolbar von links) tun. Wenn Sie anschließend auf den „Start/Fortsetzen“-Button klicken, wird die Programmausführung fortgesetzt.

### 7.6.4 Während der Ausführung eines Solist-Programms

Treten bei der Ausführung eines Programms Laufzeitfehler auf, z.B. wenn Ihr Hamster gegen eine Mauer donnert, wird eine Dialogbox geöffnet, die eine entsprechende Fehlermeldung enthält. Nach dem Anklicken des OK-Buttons in der Dialogbox wird das Solist-Programm beendet. Weiterhin öffnet sich das Konsolen-Fenster, in dem ebenfalls die Fehlermeldung ausgegeben wird.

Während der Ausführung eines Solist-Programms ist es durchaus noch möglich, die Bühne umzugestalten, also bspw. dort neue Mauern zu platzieren.

### 7.6.5 Einstellen der Geschwindigkeit

Mit dem Schieberegler in der Toolbar können Sie die Geschwindigkeit der Programmausführung beeinflussen. Je weiter links der Regler steht, desto langsamer wird das Programm ausgeführt. Je weiter Sie den Regler nach rechts verschieben, umso schneller flitzt der Solist über die Bühne.

### 7.6.6 Wiederherstellen der Bühne

Beim Testen eines Programms recht hilfreich ist der „Rücksetzen“-Button (15. Button der Toolbar von links). Sein Anklicken bewirkt, dass die Bühne in den Zustand zurückversetzt wird, den sie vor dem letztmaligen Start eines Programms inne hatte. Dieses Rücksetzen funktioniert jedoch nicht mehr, wenn Sie in der Zwischenzeit eine Klasse des Klassenbereiches verändert haben.

Über den „Komplett Zurücksetzen“-Button (16. Button der Toolbar von links) ist eine komplette Reinitialisierung des Solist-Simulators möglich. Sollte es irgendwann einmal bei der Benutzung des Simulators zu unerklärlichen Fehlern können, klicken Sie einfach diesen Button an.

## *7.7 Debuggen von Solist-Programmen*

*Debugger* sind Hilfsmittel zum Testen von Programmen. Sie erlauben es, während der Programmausführung den Zustand des Programms zu beobachten und gegebenenfalls sogar interaktiv zu ändern. Damit sind Debugger sehr hilfreich, wenn es um das Entdecken von Laufzeitfehlern und logischen Programmfehlern geht.

Der Debugger des Solist-Simulators ermöglicht während der Ausführung eines Solist-Programms das Beobachten des Programmzustands. Sie können sich während der Ausführung eines Solist-Programms anzeigen lassen, welche Anweisung des Sourcecodes gerade ausgeführt wird und welche Werte die Variablen aktuell speichern. Die interaktive Änderung von Variablenwerten wird aktuell nicht unterstützt.

Die Funktionen des Debuggers sind eng mit den Funktionen zur Programmausführung verknüpft. Sie finden die Funktionen im Menü „Simulation“. Es bietet sich jedoch an, die entsprechenden Buttons der Toolbar zu verwenden. Neben dem „Start/Fortsetzen“-, dem „Pause“- und dem „Stopp“-Button gehören die zwei Buttons „Schrittweise Ausführung“ und „Ablaufverfolgung“ zu den Debugger-Funktionen (siehe auch Abbildung 7.18).

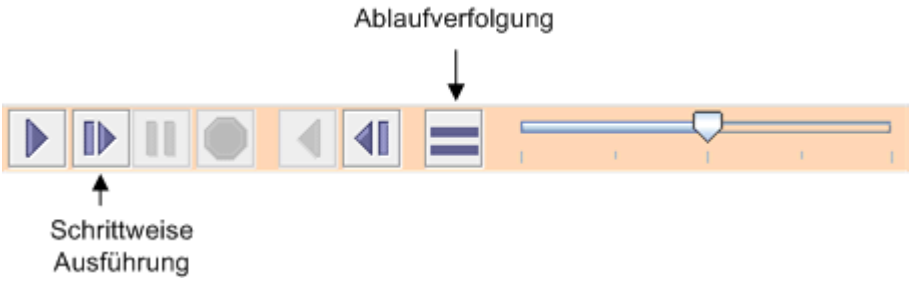



Abb. 7.18: Debugging-Buttons der Toolbar

### 7.7.1 Beobachten der Programmausführung

Durch Anklicken des Buttons „Ablaufverfolgung“ (17. Button der Toolbar von links) aktivieren bzw. (bei erneuten Anklicken) deaktivieren Sie die Ablaufverfolgung des Debuggers. Bei der Aktivierung öffnet sich dazu das Debugger-Fenster (siehe Abbildung 7.19). Dieses können Sie auch über das Menü „Fenster“ sichtbar bzw. unsichtbar machen.

Ist die Ablaufverfolgung aktiviert, wird bei der Ausführung des Programms im Editor-Bereich der Befehl (bzw. die entsprechende Zeile), der als nächstes ausgeführt wird, blau markiert. Bei einem Prozedur- bzw. Funktionsaufruf wird in die entsprechende Funktion gesprungen. Weiterhin werden im Debugger-Fenster der aktuelle Stack der Funktionsaufrufe (Name der Funktion und aktuelle Position der Ausführung der Funktion) sowie die aktuelle Belegung der Variablen dargestellt.

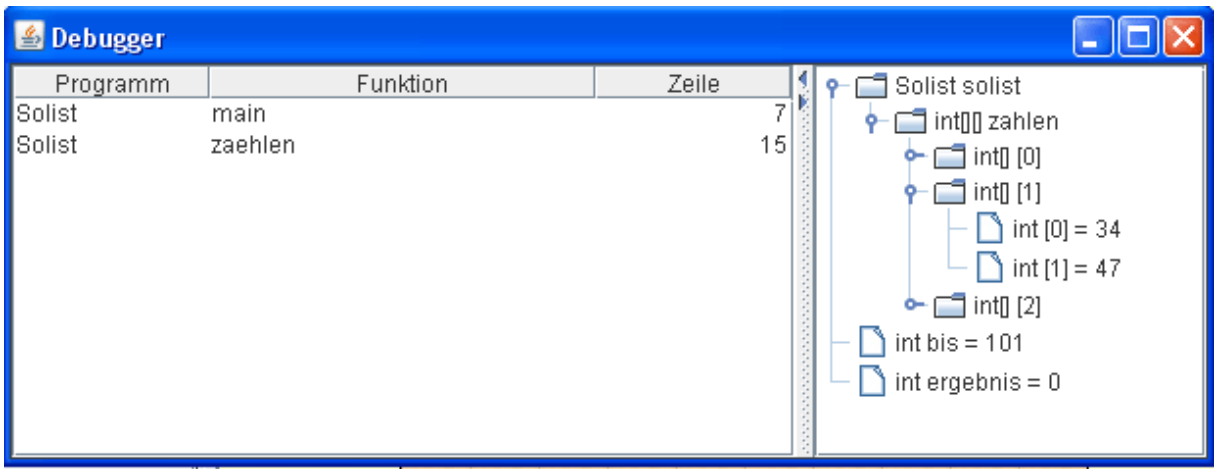



Abb. 7.19: Debugger-Fenster

Im linken Bereich des Debugger-Fensters werden Informationen zu den aktiven Funktionen angezeigt, und zwar jeweils der Name der Funktion und die aktuelle Position der Ausführung der Funktion. Ganz unten erscheint die aktuell aktive Funktion, darüber gegebenenfalls die Funktion, die diese Funktion aufgerufen hat, usw. Ganz oben steht also immer die `main`-Funktion.

Im rechten Bereich des Debugger-Fensters werden die aktiven Variablen und ihre aktuellen Werte angezeigt. Die Darstellung erfolgt dabei in einem Elementbaum, d.h. bei komplexen Variablen, wie Arrays, können Sie durch Anklicken des Symbols vor dem Variablennamen die Komponenten einsehen.

Auch während die Ablaufverfolgung aktiviert ist, können Sie die Programmausführung durch Anklicken des „Pause“-Buttons anhalten und durch anschließendes Anklicken des „Start/Fortsetzen“-Buttons wieder fortfahren lassen. Auch die Geschwindigkeit der Programmausführung lässt sich mit dem Schieberegler anpassen.

### 7.7.2 Schrittweise Programmausführung

Mit dem Toolbar-Button „Schrittweise Ausführung“ (12. Button der Toolbar von links) ist es möglich, ein Programm schrittweise, d.h. Anweisung für Anweisung auszuführen. Immer, wenn Sie den Button anklicken, wird die nächste Anweisung (bzw. Zeile) ausgeführt.

Sie können das Programm mit der schrittweisen Ausführung starten. Sie können jedoch auch zunächst das Programm normal starten, dann pausieren und ab der aktuellen Anweisung schrittweise ausführen. Eine normale Programmweiterführung ist jederzeit durch Klicken des „Start“-Buttons möglich.

### 7.7.3 Breakpoints

Wenn Sie das Programm an einer bestimmten Stelle anhalten möchten, können Sie in der entsprechenden Zeile einen so genannten Breakpoint setzen. Führen Sie dazu vor oder während der Programmausführung im Editor-Bereich mit der Maus einen Doppelklick auf die entsprechende Zeilennummer aus. Breakpoints werden durch violett hinterlegte Zeilennummern dargestellt (siehe Abbildung 7.20). Ein Doppelklick auf einen Breakpoint löscht den Breakpoint wieder. Über der Spalte mit den Zeilennummern lässt sich auch durch Klicken der rechten Maustaste ein Popup-Menü aktivieren, das als Funktionen das Setzen bzw. Entfernen von Breakpoints bzw. das gleichzeitige Löschen aller Breakpoints bietet.

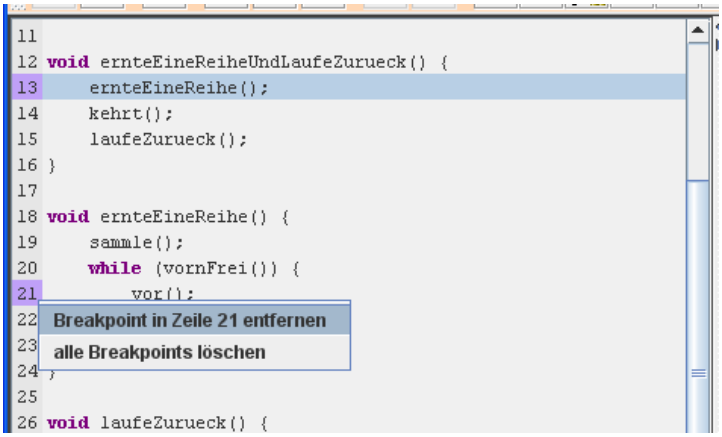



Abb. 7.20: Breakpoints

Wenn Sie ein Programm mit Breakpoints ausführen, wird die Ausführung jedesmal pausiert, wenn eine Zeile mit einem Breakpoint erreicht wird (unabhängig davon, ob die Ablaufverfolgung eingeschaltet ist). Durch Drücken des „Start/Fortsetzen“-Buttons kann die Ausführung des Programms fortgesetzt werden.

### 7.7.4 Debugger-Fenster

Das Debugger-Fenster wird automatisch bei Aktivierung der Ablaufverfolgung sichtbar gemacht. Über das Menü „Fenster“ lässt es sich jedoch auch explizit sichtbar bzw. unsichtbar machen. Das Fenster besteht aus einem rechten und einem linken Teil innerhalb einer Split-Pane. Der Inhalt des Debugger-Fensters wird nur dann aktualisiert, wenn die Ablaufverfolgung aktiviert ist.

Im linken Bereich des Fensters werden die aktuell aufgerufenen Funktionen und die jeweilige Zeilennummer dargestellt, in der sich die Programmausführung innerhalb der Funktion gerade befindet. Ganz oben steht dabei immer die main-Funktion, ganz unten die zuletzt aufgerufene Funktion.

Im rechten Bereich werden Variablen und deren aktuelle Werte dargestellt. Im Normalfall sind das genau die Variablen, die in der zuletzt aufgerufenen Funktion (also der im linken Bereich ganz unten stehenden Funktion) gültig sind. Die Darstellung erfolgt dabei in einem Elementbaum. Bei strukturierten Variablen kann durch Mausklick auf das Symbol vor dem Variablennamen die Struktur weiter geöffnet (bzw. wieder geschlossen) werden.

Im pausierten Zustand ist es möglich, sich auch die Werte lokaler Variablen anderer Funktionsinkarnationen anzuschauen. Das ist möglich, indem man im linken Bereich auf den entsprechenden Funktionsnamen klickt.

## *7.8 Anlegen und Verwalten von Aktionsbuttons*

Wenn Sie aus Ihrer bisher definierten MPW eine PLU generieren lassen würde, gäbe es ein Problem. In der generierten PLU fehlt nämlich der Klassenbereich, so dass es in der PLU nicht möglich wäre, bspw. über das Popup-Menü der Klasse Mauer Mauern zu erzeugen und diese auf der Bühne zu platzieren.

Im Folgenden werden Sie sehen, wie es möglich ist, so genannte Aktionsbuttons anzulegen, die in der Toolbar erscheinen und mit deren Aktivierung bestimmte Funktionalitäten einhergehen.

Um einen neuen Aktionsbutton anzulegen, aktivieren Sie im Menü „Miniwelt“ den Item „Neuer Aktionsbutton“. Es erscheint eine Dialogbox, in der Sie dem Aktionsbutton einen Namen zuordnen und seinen Typ angeben müssen. Der Name muss ein gültiger Java-Bezeichner sein. Unterschieden werden 3 Typen: ActionHandler, NewPropHandler, ClickHandler. Ein NewPropHandler ist für das Erzeugen neuer Prop-Komponenten geeignet, ein ClickHandler für Klick-Aktionen auf Komponenten der Bühne und ein ActionHandler für beliebige andere Aktionen.

### 7.8.1 NewPropHandler

Wir wollen nun zunächst einen Button anlegen, mit dem es möglich ist, Mauern zu erzeugen und auf der Bühne zu platzieren. Aktivieren Sie dazu bitte im Menü „Miniwelt“ das Item „Neuer Aktionsbutton“ und geben Sie in der erscheinenden Dialogbox den Namen „NeueMauer“ an und wählen Sie den Typ „NewPropHandler“ aus. Klicken Sie dann auf den OK-Button. In der Toolbar erscheint ganz rechts ein neuer Button.

Dem neuen Button wollen wir nun zunächst ein sinnvolles Ikon zuordnen. Aktivieren Sie auf dem Button ein Popup-Menü und klicken Sie das Item „Icon…“ an. Es öffnet sich eine Dateiauswahl-Dialogbox, die die Bilder des Verzeichnisses „images“ anzeigt. Wählen Sie ein Bild aus (in unserem Beispiel das Bild „mauer24.gif“) und klicken Sie den OK-Button. Wenn das Verzeichnis „images“ noch kein sinnvolles Bild enthält, müssen Sie zunächst eines importieren (Menü „Medien“).

Nun wollen wir dem Button einen aussagekräftigen Tooltip zuordnen. Aktivieren Sie wieder oberhalb des Buttons das Popup-Menü, wählen Sie das Item „Tooltip…“ aus und geben Sie in der erscheinenden Dialogbox den Text „Neue Mauer erzeugen“ ein.

Als letztes müssen wir nun noch die Aktion implementieren, die ausgelöst werden soll, wenn ein Benutzer den Button später anklickt. Dazu wählen wir diesmal im Popup-Menü das Item „Editor…“. Es öffnet sich ein Editor mit folgendem Inhalt:

```
import theater.*;

public class NeueMauer extends NewPropHandler {

    public Prop newProp() {
        return null;
    }
}
```

Von Interesse ist die Methode *newProp*, über die ein neues Prop-Objekt erzeugt werden soll. In unserem Beispiel soll ja eine neue Mauer erzeugt werden. Ersetzen Sie deshalb die Anweisung `return null;` durch die Anweisung `return new Mauer();`

Speichern Sie anschließend die Datei ab und kompilieren Sie. Wenn das erfolgreich war, klicken Sie den neuen Button an. Analog zum Erzeugen neuer Prop-Objekte über das Popup-Menü der entsprechenden Klasse im Klassenbereich können Sie nun neue Prop-Komponenten (in unserem Fall Mauern) auf der Bühne platzieren.

Um nicht für jedes Platzieren einer Mauer erneut den „Neue Mauer erzeugen“-Aktionsbutton anklicken zu müssen, können Sie auch folgendes tun: Drücken Sie die Shift-Taste der Tastatur und Klicken bei gedrückter Shift-Taste den Aktionsbutton an. Solange Sie nun die Shift-Taste gedrückt halten, können Sie durch Anklicken einer Kachel des Hamster-Territoriums dort eine Mauer platzieren. Aktionsbutton vom Typ NewPropHandler besitzen immer automatisch diese sehr sinnvolle Funktionalität.

### 7.8.2 ClickHandler

Als nächstes legen wir einen neuen Aktionsbutton vom Typ ClickHandler an. Mit diesem wollen wir erreichen, dass der Benutzer Mauern im Territorium löschen kann. Aktivieren Sie dazu bitte im Menü „Miniwelt“ das Item „Neuer Aktionsbutton“ und geben Sie in der erscheinenden Dialogbox den Namen „MauernLoeschen“ an und wählen Sie den Typ „ClickHandler“ aus. Klicken Sie dann auf den OK-Button. In der Toolbar erscheint ganz rechts ein neuer Button. Ordnen Sie diesem zunächst das Ikon „loeschen24.gif“ und den Tooltip-Text „Mauern löschen“ zu. Öffnen Sie dann den Editor. Er enthält folgenden Text:

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

public class MauernLoeschen extends ClickHandler {

    public void handleClick(Stage stage, Actor solist, int col,
     int row, List<Component> clickedComponents) {

    }
}
```

Die Methode handleClick ist noch leer. Fügen Sie dort folgende Anweisung ein:

```
stage.remove(clickedComponents);
```

Speichern Sie dann ab, kompilieren Sie und platzieren Sie zunächst einige Mauern im Territorium. Klicken Sie den neuen Aktionsbutton dann an. Sie aktivieren damit seine Funktionalität, was durch einen dunkelgrauen Hintergrund angezeigt wird. Solange der Aktionsbutton aktiviert ist, können Sie nun einzelne Kacheln des Territoriums anklicken. Befinden sich auf einer angeklickten Kachel ein oder mehrere Mauern, werden diese gelöscht.

Deaktivieren können Sie den Button dadurch, dass Sie ihn erneut anklicken. Er wird automatisch auch dann aktiviert, wenn ein anderer Aktionsbutton angeklickt wird.

Was letztendlich passiert, ist folgendes: Solange ein Akionsbutton vom Typ ClickHandler aktiviert ist, wird jedesmal, wenn der Benutzer auf die Bühne klickt, dessen Methode handleClick aufgerufen. Übergeben werden dabei die aktuelle Bühne (stage), der aktuelle Solist (solist), die Spalte (col) und Reihe (row) der angeklickten Zelle sowie eine Liste mit allen Komponenten k, die auf der Zelle platziert sind, für die also gilt: k.getRow() == row && k.getColumn() == column. In unserem Beispiel werden alle diese Komponenten von der Bühne entfernt.

### 7.8.3 ActionHandler

Als nächstes legen wir einen neuen Aktionsbutton vom Typ ActionHandler an. Mit diesem wollen wir erreichen, dass der Benutzer die Größe des Territoriums ändern kann. Aktivieren Sie dazu bitte im Menü „Miniwelt“ das Item „Neuer Aktionsbutton“ und geben Sie in der erscheinenden Dialogbox den Namen „GroesseAendern“ an und wählen Sie den Typ „ActionHandler“ aus. Klicken Sie dann auf den OK-Button.

In der Toolbar erscheint ganz rechts ein neuer Button. Ordnen Sie diesem zunächst das Ikon „size24.gif“ und den Tooltip-Text „Größe des Territoriums ändern“ zu. Öffnen Sie dann den Editor. Er enthält folgenden Text:

```
import theater.*;

public class GroesseAendern extends ActionHandler {

    public void handleAction(Stage stage, Actor solist) {

    }
}
```

Ändern Sie den Text wie folgt ab:

```
import theater.*;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.util.List;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;


public class GroesseAendern extends ActionHandler {
    public void handleAction(Stage stage, Actor solist) {
        SizeDialog dialog = new SizeDialog(stage);
        dialog.setVisible(true);
    }
}


class SizeDialog extends JDialog {
    JTextField reihen;
    JTextField spalten;
    Stage stage;

    public SizeDialog(Stage stage) {
        super((Frame) null, "Territorium-Größe", true);
        this.stage = stage;

        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        this.setLayout(new BorderLayout());

        JPanel rPanel = new JPanel();
        rPanel.setLayout(new FlowLayout());

        JLabel nR = new JLabel("Reihen: ");
        rPanel.add(nR);
        this.reihen = new JTextField(15);
```

```
        this.reihen.setText("" + stage.getNumberOfRows());
        this.reihen.addActionListener(new OKButtonAction(this));
        rPanel.add(this.reihen);
        this.add(rPanel, BorderLayout.NORTH);

        JPanel sPanel = new JPanel();
        sPanel.setLayout(new FlowLayout());

        JLabel nS = new JLabel("Spalten: ");
        sPanel.add(nS);
        this.spalten = new JTextField(15);
        this.spalten.setText("" + stage.getNumberOfColumns());
        this.spalten.addActionListener(new OKButtonAction(this));
        sPanel.add(this.spalten);
        this.add(sPanel, BorderLayout.CENTER);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER));

        JButton ok = new JButton("Ok");
        ok.addActionListener(new OKButtonAction(this));
        buttonPanel.add(ok);

        JButton cancel = new JButton("Abbrechen");
        cancel.addActionListener(new CancelButtonAction(this));
        buttonPanel.add(cancel);
        this.add(buttonPanel, BorderLayout.SOUTH);

        this.pack();
        this.setResizable(false);

        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        this.setLocation((d.width / 2) - (this.getWidth() / 2),
            (d.height / 2) - (this.getHeight() / 2));
    }
}


class OKButtonAction implements ActionListener {
    SizeDialog dialog;

    OKButtonAction(SizeDialog d) {
        this.dialog = d;
    }

    public void actionPerformed(ActionEvent e) {
        try {
            int spalten = new Integer(dialog.spalten.getText());
            int reihen = new Integer(dialog.reihen.getText());

            // mindestens 1 Reihe und 1 Spalte
            if (spalten < 1) {
                spalten = 1;
            }

            if (reihen < 1) {
                reihen = 1;
            }

            Actor solist = dialog.stage.getSolist();

            if ((solist.getRow() >= reihen) || (solist.getColumn() >= spalten)) {
                // Solist ist außerhalb der neuen Bühne; er wird auf
                // die Kachel (0/0) umplatziert, wobei dort unter Umständen
                // noch eine Mauer entfernt werden muss
                List<Component> mauern = dialog.stage.getComponentsAt(0, 0,
                        Mauer.class);
```

```
                dialog.stage.remove(mauern);
                dialog.stage.getSolist().setLocation(0, 0);
            }

            // neue Bühnengröße setzen
            dialog.stage.setSize(spalten, reihen);
            this.dialog.dispose();
        } catch (NumberFormatException exc) {
        }
    }
}


class CancelButtonAction implements ActionListener {
    JDialog dialog;

    CancelButtonAction(JDialog d) {
        this.dialog = d;
    }

    public void actionPerformed(ActionEvent e) {
        this.dialog.dispose();
    }
}
```

Speichern Sie ab, kompilieren Sie und probieren Sie den neuen Button aus. Bei dessen Anklicken erscheint eine Dialogbox, in der Sie die Anzahl an Reihen und Spalten eingeben können. Wenn Sie dies tun und den OK-Button klicken, wird das Territorium in der angegebenen Größe neu aufgebaut.

Was bei einem Aktionsbutton des Typs ActionHandler passiert, ist letztendlich folgendes: Wird ein solcher Button angeklickt, wird die Methode handleAction aufgerufen, der die aktuelle Bühne und der aktuelle Solist als Parameter übergeben werden. Durch die Methode muss dann die gewünschte Funktionalität implementiert werden.

Definieren Sie nun (zum Selbsttest) einen weiteren Aktionsbutton mit dem Namen „LinksUm“ vom Typ ActionHandler, ordnen ihm das Bild „linksUm24.gif“ und den Tooltip „drehe den Hamster um 90 Grad nach links“ zu und ändern Sie im Editor die Klasse wie folgt ab:

```
import theater.*;

public class LinksUm extends ActionHandler {

    public void handleAction(Stage stage, Actor solist) {
        ((Hamster)solist).linksUm();
    }
}
```

Wenn der Benutzer diesen Button anklickt, dreht sich der Hamster um 90 Grad nach links.

Die Reihenfolge Ihrer Aktionsbuttons in der Toolbar können Sie übrigens verändern. Aktivieren Sie dazu das Popup-Menü eines Aktionsbuttons und wählen über das Item „Verschieben“ aus, ob der Button nach links oder rechts verschoben werden soll. Ordnen Sie die Aktionsbutton nun so an, wie in Abbildung 7.21 skizziert wird.

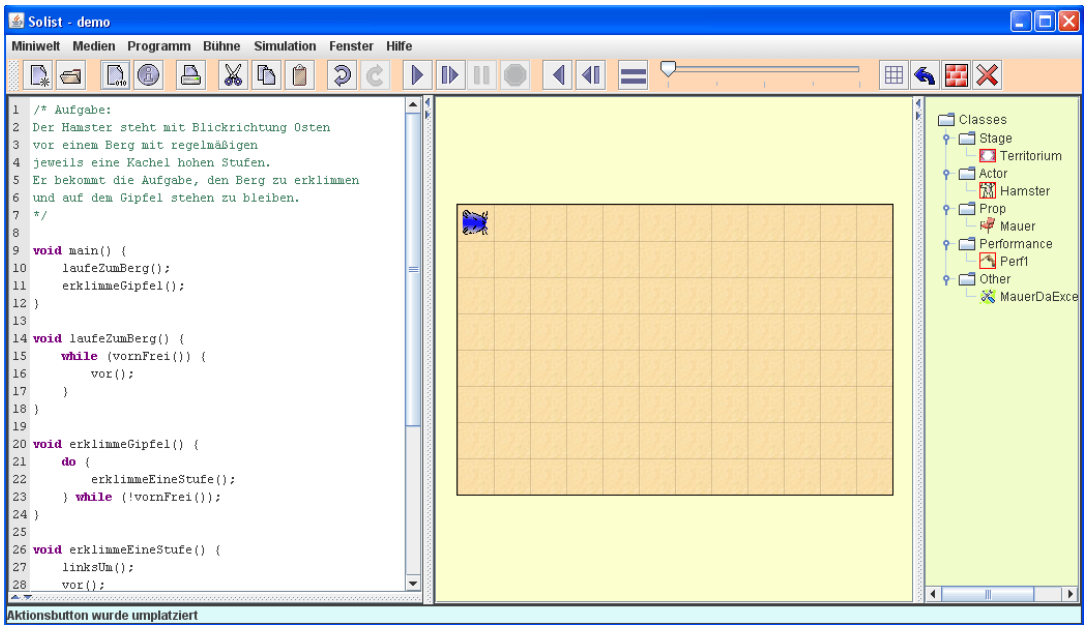



Abb. 7.21: Solist mit neu definierten Aktionsbuttons

## *7.9 Generieren einer PLU*

Nun sind wir soweit, dass wir aus unserer entwickelten MPW eine PLU generieren können. Dazu klicken wir im Menü „Miniwelt“ das Item „Generieren…“ an. Es öffnet sich die in Abbildung 7.22 dargestellte Dialogbox. Hierin müssen bzw. können wir folgende Einträge vornehmen:

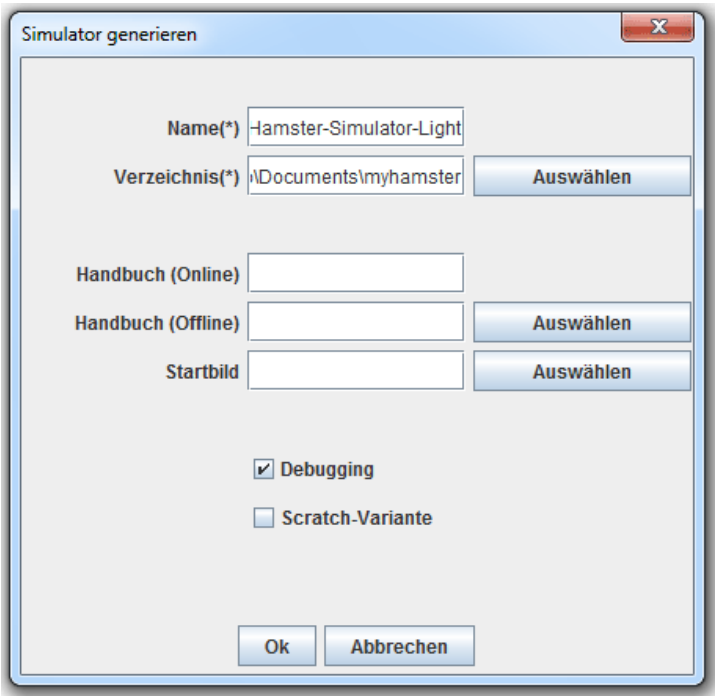



Abb. 7.22: Generieren-Dialogbox

- Name (obligatorisch): Einen Namen für die generierte PLU. Dieser wird später in der Titelzeile des entsprechenden Fensters angezeigt.
- Verzeichnis (obligatorisch): Ein Verzeichnis, in dem die generierten Dateien abgespeichert werden.
- Handbuch (online): Eine URL zu einem möglichen Online-Handbuch für die generierte PLU.
- Handbuch (offline): Ein Handbuch als PDF-Datei bzw. alternativ eine zip-Datei mit HTML-Seiten sowie eine Datei namens index.html (Hinweis: Aufgrund eines Fehler in der JVM dürfen die Dateinamen in der zip-Datei keine Umlaute enthalten).
- Startbild: Ein so genannter Spashscreen, das ist eine gif-, png- oder jpg-Datei, die beim Starten der generierten PLU angezeigt werden soll, während notwendige Ressourcen geladen werden.
- Debugging: Ist das Häckchen gesetzt, enthält die generierte PLU auch die in Abschnitt 7.7 vorgestellten Debugging-Funktionalitäten. Bei nicht gesetztem Häkchen werden diese in der PLU entfernt.
- Scratch-Variante: Ist das Häckchen gesetzt, wird ein Scratch-Simulator generiert (siehe Kapitel 7.10)

Wir geben nun bspw. den Namen „Demo-Hamster-Simulator“ ein, wählen das Verzeichnis „demosimulator“ aus und geben als URL-Handbuch die URL http://www.java-hamster-modell.de ein. Dann klicken wir auf den OK-Button. Innerhalb weniger Sekunden wird dann die PLU generiert.

Um uns einen Einblick von unserer PLU zu verschaffen, begeben wir uns in das angegebene Verzeichnis „demosimulator“. Dort gibt es eine Datei „simulator.jar“, eine Datei „solist.properties“ und einen Ordner „data“. Starten lässt sich die generierte PLU durch Doppelklick auf die Datei „simulator.jar“. Beim ersten Start sucht Solist nach der JDK-Installation auf Ihrem Computer. Sollte diese nicht gefunden werden, werden Sie aufgefordert, den entsprechenden Pfad einzugeben. Der Pfad wird anschließend in der Datei *solist.properties* gespeichert, wo er später wieder geändert werden kann, wenn Sie bspw. eine aktuellere JDK-Version auf Ihrem Rechner installieren sollten. In der Properties-Datei können weitere Einstellungen vorgenommen werden. Im Ordner „data“ liegen die notwendigen Klassen und Ressourcen. Außerdem gibt es einen Unterordner „programs“, der standardmäßig beim Abspeichern von Programmen und Bühnen genutzt wird.

Wenn Sie die Datei „simulator.jar“ doppelklicken, öffent sich die generierte PLU (siehe Abbildung 7.23).

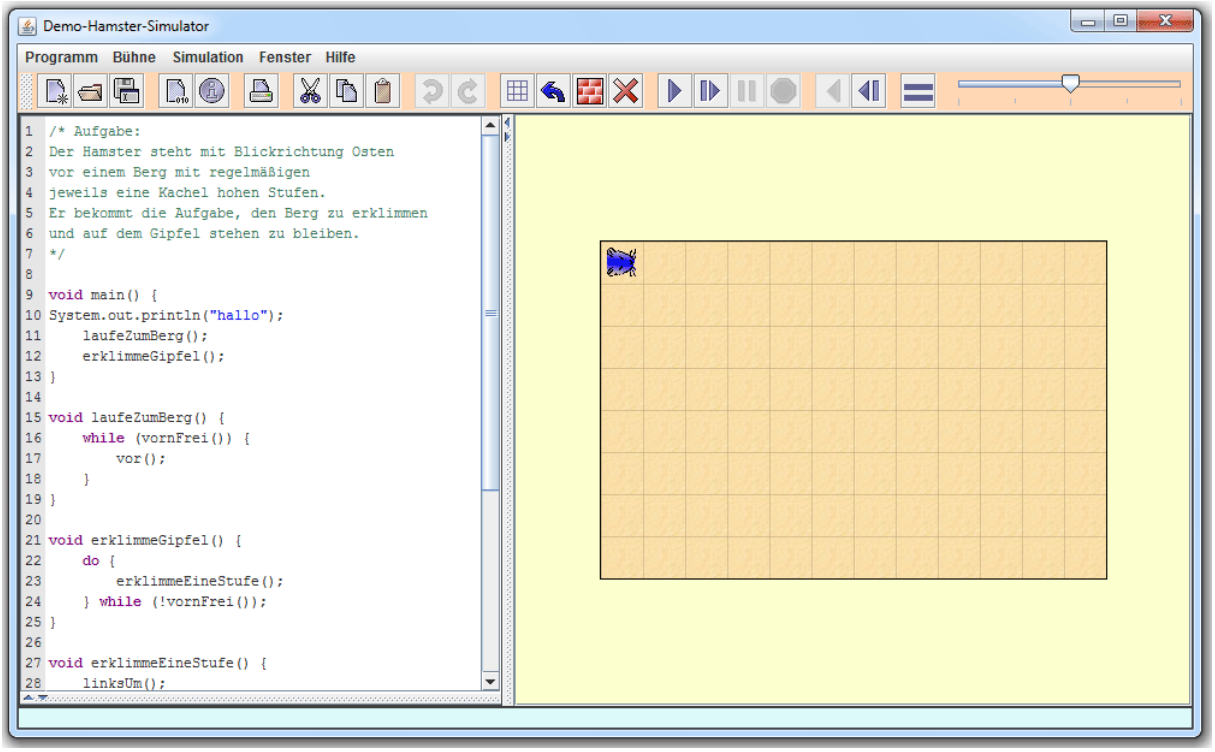



Abb. 7.23: Generierte PLU

Die generierte PLU unterscheidet sich in insbesondere in folgenden Funktionalitäten vom Solist-Simulator selbst:

- Im Hauptfenster fehlt der Klassenbereich.
- In der Toolbar fehlen die beiden ersten Buttons (d.h. zum Anlegen bzw. Öffnen von Theaterstücken).
- Es fehlen die Menüs Miniwelt und Medien.
- Die Aktionsbuttons (hier Gestaltungsbuttons genannt) stehen links von den Steuerungsbuttons in der Toolbar.
- Nach dem Compilieren wird die Bühne nicht jedesmal neu initialisiert.

Die meisten anderen Funktionalitäten stehen voll zur Verfügung. Ich empfehle Ihnen, einmal einen Blick in das Benutzungshandbuch des „Hamster-Simulator-Light“ zu werfen, um sich einen Überblick über alle Funktionalitäten zu verschaffen (siehe Abschnitt 10.2).

## *7.10 Scratch-PLUs*

Normalerweise werden durch Solist PLUs bzw. Simulatoren generiert, die die Programmiersprache Java unterstützen, d.h. die Nutzer der generierten PLUs schreiben ihre Programme in Java. Java ist eine textuelle Programmiersprache, bei der Programme vor ihrer Ausführung zunächst von einem Compiler auf syntaktische Korrektheit überprüft werden müssen. Gerade Programmieranfänger haben aber häufig Probleme mit der beim Eingeben des Programmcodes erforderlichen

Genauigkeit und den Fehlermeldungen des Compilers. Diesbezüglich versprechen visuelle, interpretierte Programmiersprachen Abhilfe.

Eine davon ist die in den vergangenen Jahren sehr populär gewordene Programmiersprache bzw. Programmierumgebung *Scratch*. Anders als bei textuellen Programmiersprachen müssen hier die Programmierer keinen textuellen Sourcecode schreiben. Vielmehr setzen sich Scratch-Programme aus graphischen Bausteinen zusammen (siehe Abbildung 7.24). Die Programmierumgebung unterstützt dabei das Erstellen von Programmen durch einfache Drag-and-Drop-Aktionen mit der Maus. In Scratch können dem Programmierer keine syntaktischen Fehler mehr unterlaufen und auf einen Compiler kann verzichtet werden. Mehr Informationen zu Scratch findet man im Internet unter http://scratch.mit.edu/.

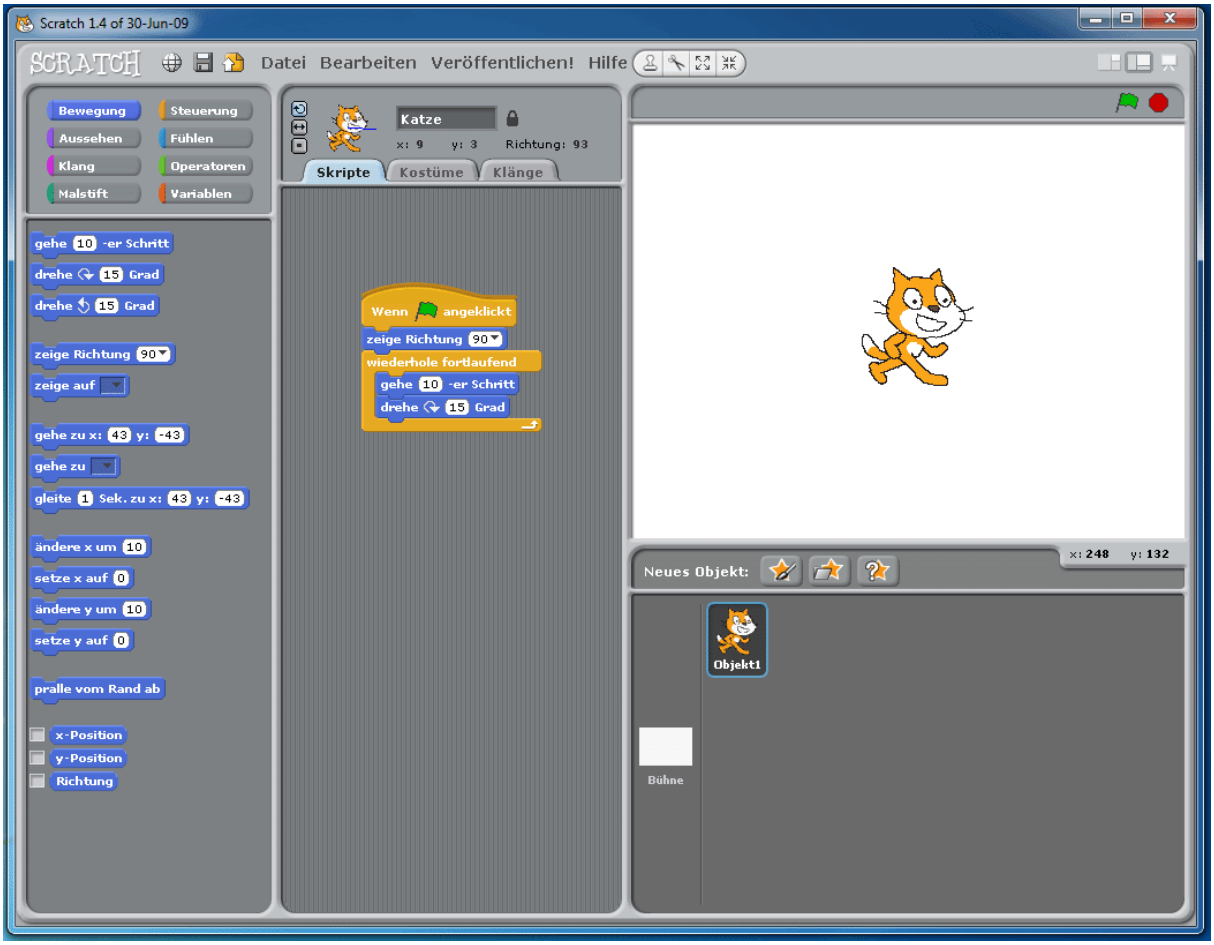



Abb. 7.24: Scratch

Um die Vorteile, die Scratch für Programmieranfänger verspricht, auch denjenigen zugänglich zu machen, die die mit Solist generierten PLUs nutzen möchten, haben wir in Version 2.0 von Solist eine Scratch-Variante integriert. Davon bekommen Sie in Solist selbst nichts mit, außer dass Sie in der Generieren-Dialogbox (siehe Abbildung 7.22) bei „Scratch-Variante“ ein Häkchen setzen müssen. In diesem Fall können Benutzer in den generierten PLUs nicht in Java programmieren, sondern in der visuellen Programmiersprache Scratch.

Aktuell wurden von der Basis-Theaterstücken „hamster“ und „kara“ (siehe Kapitel 10.2 und 10.3) Scratch-PLUs generiert. Werfen Sie am besten mal einen Blick in die Benutzungshandbücher des Scratch-Hamster- und Scratch-Kara-Simulators, um einen Eindruck von Scratch-Simulatoren zu bekommen.

Lassen Sie uns nun mal aus unserer entwickelten Demo-Hamster-MPW eine Scratch-PLU generieren. Setzen Sie dazu in der Generieren-Dialogbox (siehe Abbildung 7.22) das Häkchen bei „Scratch-Variante“ und klicken dann auf den OK-Button. Wenn Sie dann im erstellten Verzeichnis die Datei „simulator.jar“ doppelklicken, öffnet sich die in Abbildung 7.25 dargellte Scratch-PLU.

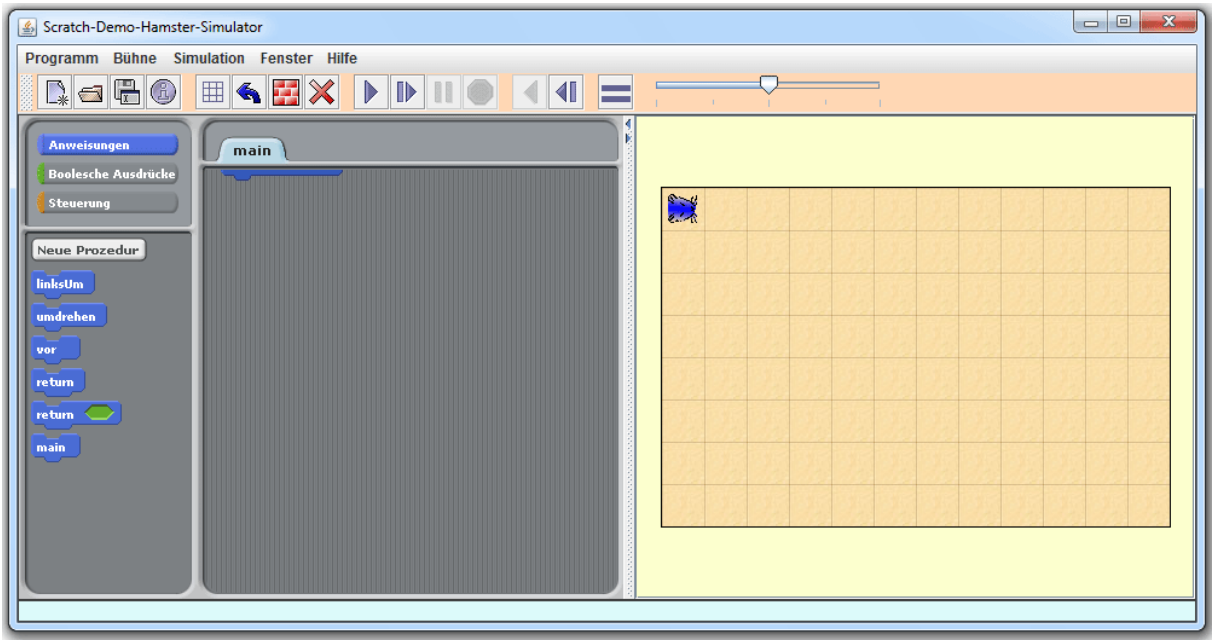



Abb. 7.25: Generierte Scratch-PLU

Scratch-Programme setzen sich aus Blöcken zusammen, die Befehle bzw. Anweisungen repräsentieren. Die Blöcke lassen sich per Drag-and-Drop mit der Maus zusammenstellen. Blaue Blöcke (Anweisungsblöcke) repräsentieren die in der Solist-Klasse definierten void-Methoden, grüne Blöcke repräsentieren die in der Solist-Klasse definierten boolean-Methoden (Boolescher-Ausdruck-Blöcke). Weiterhin existieren orangene Blöcke (Steuerungsblöcke) für die Kontrollstrukuren if-Anweisung, if-else-Anweisung, while-Schleife und do-while-Schleife. Beim Ausführen des in Abbildung 7.26 dargestellten Scratch-Programms unserer Demo-MPW würde der Hamster als Solist bis zur nächsten Mauer laufen und sich dann umdrehen.

Die generierten Scratch-PLUs unterstützen weiterhin die Definition neuer Prozeduren und boolescher Funktionen. Variablen werden allerdings nicht unterstützt.

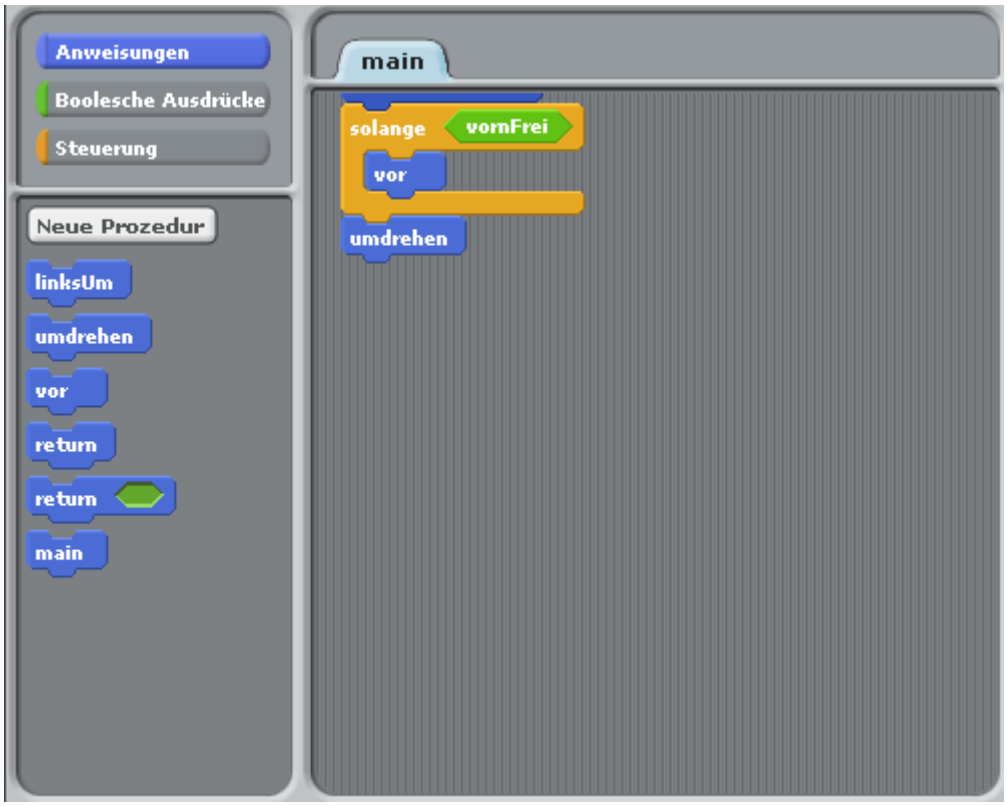



Abb. 7.26: Scratch-Programm

Bei der Generierung von Scratch-PLUs sind folgende Einschränkungen zu beachten:

- Von jeder in der Solist-Klasse definierten Methode mit dem Funktionstyp `void` (also eine Prozedur) wird ein Anweisungsblock generiert. Die Methode muss als `public` deklariert sein, darf nicht als `abstract` deklariert sein und darf keine Parameter besitzen. Außerdem darf sie nicht als Invisible annotiert sein.
- Von jeder in der Solist-Klasse definierten Methode mit dem Funktionstyp `boolean` (also eine boolesche Funktion) wird ein Boolescher-Ausdruck-Block generiert. Die Methode muss als `public` deklariert sein, darf nicht als `abstract` deklariert sein und darf keine Parameter besitzen. Außerdem darf sie nicht als Invisible annotiert sein.
- Alle anderen Methoden der Solist-Klasse werden ignoriert.

Bezüglich unserer Demo-MPW bedeutet das, dass es in der generierten Scratch-PLU die Anweisungsblöcke *linksUm*, *umdrehen* und *vor* sowie den Boolescher-Ausdrucks-Block *vornFrei* gibt.

## *7.11 Konsole*

Die Konsole ist ein zusätzliches Fenster, das bei bestimmten Aktionen automatisch geöffnet wird, sich aber über das Menü „Fenster“ auch explizit öffnen bzw. schließen lässt (siehe Abbildung 7.27).

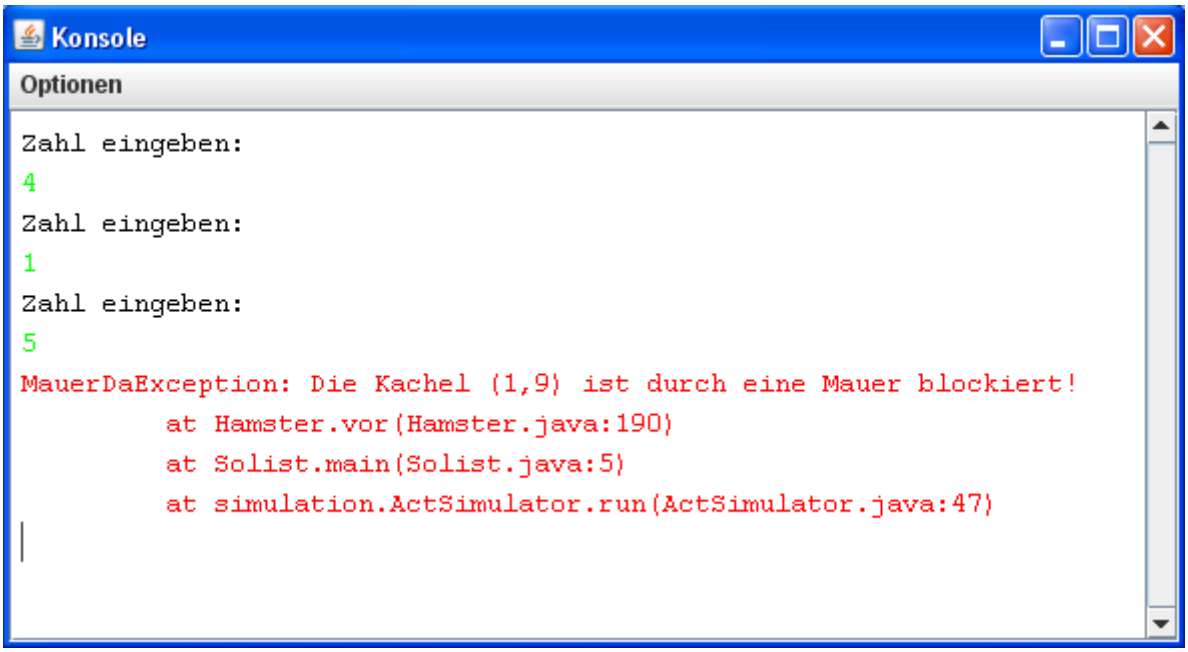



Abb. 7.27: Konsole

Die Konsole ist für die Java-Eingabe mittels `System.in` und die Ausgabe mittels `System.out` und `System.err` zuständig. Wird in Ihrem Programm bspw. der Befehl `System.out.println(„hallo“);` ausgeführt, wird die Zeichenkette `hallo` in der Konsole ausgegeben. `System.out` und `System.err` unterscheiden sich durch eine schwarze (out) bzw. eine rote (err) Ausgabe des entsprechenden Textes. Auch Fehlermeldungen des Hamster-Simulators erscheinen in der Konsole (bspw. Laufzeitfehler, wenn der Hamster gegen eine Mauer rennt).

Ein Eingabebefehl via `System.in` blockiert das Hamster-Programm so lange, bis der Nutzer eine entsprechende Eingabe in der Konsole getätigt und im Allgemeinen durch Drücken der Enter-Taste abgeschlossen hat. Die folgende Funktion `readInt` erwartet bspw. vom Nutzer die Eingabe einer Zahl in der Konsole und liefert den entsprechenden Wert an das Hamster-Programm. Wenn der Nutzer in der Konsole den Wert 4 eingibt, hüpft der Hamster vier Kacheln nach vorne (insofern er nicht gegen eine Mauer rennt).

```
void main() {
    System.out.println("Zahl eingeben: ");
    int zahl = readInt();
    while (zahl > 0) {
        vor();
        zahl = zahl - 1;
    }
}

// Einlesen eines int-Wertes
int readInt() {
    try {
        java.io.BufferedReader input =
            new java.io.BufferedReader(
```

```
                new java.io.InputStreamReader(System.in));
            String eingabe = input.readLine();
          return new Integer(eingabe);
        } catch (Throwable exc) {
          return 0;
        }
}
```

Die Konsole enthält in der Menüleiste ein Menü namens „Optionen“. Hierin finden sich drei Menü-Items, über die es möglich ist, den aktuellen Inhalt der Konsole zu löschen, den aktuellen Inhalt der Konsole in einer Datei abzuspeichern bzw. die Konsole zu schließen.

## *7.12 PLU-Typen*

Die Solist bzw. den generierten PLUs zugrunde liegende Programmiersprache ist Java. Um den Programmieranfänger, die die PLUs benutzen sollen, das Leben etwas leichter zu machen, gibt es jedoch leichte Unterschiede. Unterschieden werden in Solist generell zwei verschiedene Typen von PLUs: Imperative PLUs und objektbasierte PLUs. Die Typauswahl erfolgt über das Menü „Programm“ im untersten Item. Standardmäßig ist der Typ „imperative Solist-Programme“ eingestellt (Häkchen ist gesetzt). Sie sollten sich beim Entwickeln eines neuen Theaterstückes ganz zu Anfang für einen Typ entscheiden, weil nachträgliche Änderungen zwar möglich sind, diese dann jedoch auch Änderungen an evtl. bereits fertig gestellten Beispielprogrammen bedingen.

### 7.12.1 Imperative PLUs

Normalerweise bestehen Java-Programme immer aus einer Klasse und einer Methode mit der Signatur `public static void main(String[] args)`. Der Programmstart erfolgt bei der Übergabe der Klasse an die JVM durch Aufruf dieser main-Prozedur.

In Solist kann in imperativen PLUs die Klassendeklaration weggelassen werden. Die main-Prozedur hat darüber hinaus die verkürzte Signatur `void main()`. Im Hintergrund existiert ein Precompiler, der die entsprechenden Umsetzungen automatisch erledigt.

Die wichtigsten Unterschiede zwischen purem Java und imperativen Solist-Programmen sind hier nochmal vollständig aufgelistet:

- Anders als in Java bedarf es in imperativen Solist-Programmen keiner Klassen-Definition.
- In imperativen Solist-Programmen gibt es keine Methode mit der Signatur `public static void main(String[] args)`. Stattdessen gibt es die so genannte main-Prozedur mit der Signatur `void main()`.
- Solist-Programme sind imperative, keine objektorientierten Programme. Es brauchen also keine Objekte erzeugt zu werden. Trotzdem müssen

Prozeduren und Funktionen und globale Variablen nicht als `static` deklariert werden.

- Ein imperatives Solist-Programm setzt sich aus der main-Prozedur sowie weiteren Prozeduren bzw. Funktionen zusammen, die vor oder nach der main-Prozedur definiert werden dürfen. Variablen können global, d.h. außerhalb von Prozeduren, oder lokal, d.h. innerhalb von Prozeduren, definiert werden. Der Gültigkeitsbereich von globalen Variablen erstreckt sich über das gesamte Programm, der Gültigkeitsbereich von lokalen Variablen ist auf den Prozedurrumpf beschränkt.
- Der Aufruf bzw. Start eines imperativen Solist-Programms bewirkt die automatische Ausführung der main-Prozedur.

Eine eher objektbasierte Notation beim Aufruf von Solist-Befehlen in impertaiven Solist-Programmen ist trotzdem möglich, indem dem Solist ein Name zugeordnet wird. Wie das genau funktioniert, können Sie in Abschnitt 7.1.2 nachlesen.

### 7.12.2 Objektbasierte PLUs

Manche Lehrenden bevorzugen bei der Benutzung von PLUs eine eher objektbasierte Notation, bei der Objekte über Namen angesprochen werden und in denen Programme innerhalb einer Klassendefinition eingeschlossen werden. Bei dem Beispiel-Theaterstück kara als Nachbildung des Original-JavaKara-Simulators (siehe Kapitel 10) ist das beispielsweise der Fall.

Bei der Wahl einer eher objektbasierten PLU haben Solist-Programme die folgende Gestalt:

```
public class Solist extends <Solist-Klasse> {

    void main() {

    }

}
```

`<Solist-Klasse>` muss dabei durch den Namen der Solist-Klasse ersetzt werden.

Auch Programme objektbasierte PLUs werden durch Aufruf der main-Prozedur bzw. main-Methode gestartet. Zusätzlich kann dem Solist wie in Abschnitt 7.1.2 beschrieben ein Name zugeordnet und Befehlen dieser Name vorangestellt werden.

Im Beispiel-Theaterstück oodemo wird die Entwicklung objektbasierter MPWs demonstriert. Die Klassen dieses Theaterstücks unterscheiden sich nicht von den Klassen des oben entwickelten Theaterstücks demo. Einziger Unterschied ist, dass in dem Menü „Programm“ das Häkchen beim Item „imperative Solist-Programme“ entfernt wurde. Die Solist-Programme haben im Theaterstück oodemo allerdings die folgende Gestalt:

```
/* Aufgabe:
Der Hamster steht mit Blickrichtung Osten
```

```
vor einem Berg mit regelmäßigen
jeweils eine Kachel hohen Stufen.
Er bekommt die Aufgabe, den Berg zu erklimmen
und auf dem Gipfel stehen zu bleiben.
*/
public class Solist extends Hamster {

    void main() {
        willi.laufeZumBerg();
        willi.erklimmeGipfel();
    }

    void laufeZumBerg() {
        while (willi.vornFrei()) {
            willi.vor();
        }
    }

    void erklimmeGipfel() {
        do {
            willi.erklimmeEineStufe();
        } while (!willi.vornFrei());
    }

    void erklimmeEineStufe() {
        willi.linksUm();
        willi.vor();
        willi.rechtsUm();
        willi.vor();
    }

    void rechtsUm() {
        kehrt();
        linksUm();
    }

    void kehrt() {
        willi.linksUm();
        willi.linksUm();
    }

}
```

# 8 Tipps und Tricks

In diesem Abschnitt werden einige Tipps und Tricks vorgestellt. Weitere Tipps und Tricks entnehmen Sie bitte den Beispielen in Kapitel 10.

## *8.1 addedToStage*

Wenn Sie eine Komponente (Actor oder Prop) erzeugen, wird sein Konstruktor aufgerufen. Normalerweise erfolgt in einem Konstruktor eine Initialisierung eines Objektes. Bestimmte geerbte Methoden einer Komponente können jedoch nicht bereits im Konstruktor sondern erst nach der Platzierung der Komponente auf der Bühne aufgerufen werden (getRow, getColumn, setLocation, …). Um derartige Initialisierungen trotzdem durchführen zu können, wird nach der Platzierung einer Komponente auf der Bühne (via einer der add-Methoden der Klasse Stage) die Methode `public void addedToStage(Stage stage)` aufgerufen und ihr als Parameter eine Referenz auf die entsprechende Bühne übergeben. Überschreiben Sie also gegebenenfalls diese Methode und führen Sie entsprechende Initialisierungen innerhalb dieser Methode durch.

Möchte bspw. ein Programmierer erreichen, dass eine Komponente unabhängig davon, wohin sie der Benutzer platzieren will, zu Anfang immer in die linke obere Ecke der Bühne platziert wird, kann er dies auf folgende Art und Weise erreichen:

```
import theater.*;

public class Dummy extends Prop {
    public Dummy() {
        // setLocation(0, 0); hier nicht erlaubt
    }

    @Override
    public void addedToStage(Stage stage) {
        setLocation(0, 0);
    }
}
```

## *8.2 setLocation*

Der in Kapitel 7 entwickelte Demo-Hamster-Simulator enthält noch einige unschöne Aspekte. Beispielweise ist es möglich, den Hamster und Mauern per Drag-and-Drop außerhalb des sichtbaren Bereichs der Bühne zu platzieren. Auch ist es möglich, eine Mauer über den Hamster zu platzieren oder auf einer Kachel zwei Mauern zu platzieren. Dies lässt sich durch das Überschreiben der Component-Methode *setLocation* korrigieren. Die Methode wird nämlich aufgerufen, wenn eine neue Komponente auf der Bühne platziert wird und auch wenn sie dort bspw. per Drag-and-Drop umplatziert wird. Um die Probleme zu beseitigen, definieren wir folgende Methode *setLocation* in der Klasse *Hamster*

```
    @Invisible
    public void setLocation(int col, int row) {
        if ((col >= 0) && (row >= 0) &&
```

```
            (col < this.getStage().getNumberOfColumns()) &&
            (row < this.getStage().getNumberOfRows()) &&
             this.getStage().getComponentsAt(col, row, Mauer.class).size() == 0)) {
            // nur innerhalb der sichtbaren Bühne und nur, falls sich auf der
            // angegebenen Kachel keine Mauer befindet
            super.setLocation(col, row);
        }
    }
```

und folgende Methode *setLocation* in der Klasse *Mauer*

```
    @Invisible
    public void setLocation(int col, int row) {
        if ((this.getColumn() == col) && (this.getRow() == row)) {
            // die Mauer befindet sich bereits auf dieser Zelle
            return;
        }

        if ((col < 0) || (row < 0) ||
                (col >= this.getStage().getNumberOfColumns()) ||
                (row >= this.getStage().getNumberOfRows())) {
            // nur innerhalb der sichtbaren Bühne
            return;
        }

        Actor solist = getStage().getSolist();

        if ((col == solist.getColumn()) && (row == solist.getRow())) {
            // Mauer kann nicht auf die Zelle umplatziert werden, auf der
            // sich der Solist befindet
            return;
        }

        // Mauer auf Kachel?
        if (this.getStage().getComponentsAt(col, row, Mauer.class).size() > 0) {
            // eine Mauer kann nicht auf eine Kachel platziert werden, auf der sich
            // bereits eine Mauer befindet
            return;
        }

        // ruft die geerbte Methode auf, die die tatsächliche Umplatzierung
        // vornimmt
        super.setLocation(col, row);
    }
```

## *8.3 Synchronisation*

Die Ausführung von Solist-Programmen, die interaktive Ausführung von Solist-Befehlen und die Ausführung von Aktionen der Aktionsbuttons erfolgen in unterschiedlichen Threads. Das kann Probleme mit sich bringen, wenn bspw. Nutzer während der Ausführung bzw. Pausierung eines Programms über das Befehlsfenster Befehle ausführen oder Aktionsbuttons anklicken. Die Aktionen gilt es also zu synchronisieren. Zur Synchronisation stehen in der Klasse *Performance* zwei Methoden *lock* bzw. *unlock* zur Verfügung. Nach Aufruf der Methode *lock* kann kein weiterer Thread irgendwelche Methoden der Theater-API aufrufen, und zwar bis wieder die Methode *unlock* aufgerufen worden ist. Der verwendete Lock ist dabei ein ReentrantLock, d.h. wird der Befehl *lock* n mal aufgerufen, muss auch der Befehl *unlock* n mal aufgerufen werden, um die Sperre zu beseitigen.

Um ganz sicher zu sein, dass keine Synchronisationsfehler auftreten können, sollten Sie also jede von außen aufrufbare Methode Ihrer Klassen in entsprechende lock-unlock-Aufrufe einschließen, und zwar auf folgende Art und Weise:

```
public void methode() {
    Performance.getPerformance().lock();

    try {
        ...
    } finally {
        Performance.getPerformance().unlock();
    }
}
```

Schauen wir uns das mal konkret am Fall der Methode *linksUm* der Klasse Hamster an:

```
@Description("Der Hamster dreht sich um 90 Grad nach links um.")
public void linksUm() {
    Performance.getPerformance().lock();

    try {
        this.blickrichtung = (this.blickrichtung + 1) % 4;
        this.setRotation(this.getRotation() - 90);
    } finally {
        Performance.getPerformance().unlock();
    }
}
```

Durch Einsatz des Locks ist es hier sichergestellt, dass die Blickrichtung und Rotation des Hamsters sich immer in einem konsistenten Zustand befinden. Die Verwendung eines try-finally-Konstruktes sorgt dafür, dass für jeden Aufruf der lock-Methode auch auf jeden Fall die unlock-Methode aufgerufen wird, auch wenn irgendwelche Laufzeitfehler auftreten sollten. Ansonsten könnte es zu unfreiwilligen Blockaden kommen.

Hinweis: Die Aufrufe der Methoden der Aktionsbutton-Klassen werden automatisch intern gelockt. Diese brauchen also nicht explizit von Ihnen gelockt zu werden.

## *8.4 Einfrieren*

Stellen Sie sich vor, Sie definieren in der Klasse *Hamster* folgende Methode *umdrehen*:

```
@Description("Der Hamster dreht sich um 180 Grad um.")
public void umdrehen() {
    Performance.getPerformance().lock();

    try {
        linksUm();
        linksUm();
    } finally {
        Performance.getPerformance().unlock();
    }
}
```

Wenn diese Methode in einem Solist-Programm aufgerufen wird und die Geschwindigkeit auf langsam steht, können Sie sehen, dass sich der Hamster zweimal linksum dreht und nach jedem linksUm-Befehl macht er entsprechend der eingestellten Geschwindigkeit eine kleine Pause. Das kann durch Verwendung der Methoden *freeze* und *unfreeze* der Klasse *Performance* geändert werden. Der Aufruf der Methode *freeze* bewirkt ein Einfrieren des aktuellen Zustands der Bühne. Erst wenn die Methode *unfreeze* aufgerufen wird, wird die Bühne wieder aktualisiert. Im eingefrorenen Zustand entfallen darüberhinaus die Pausen bei eingestellter niedriger Geschwindigkeit.

```
@Description("Der Hamster dreht sich um 180 Grad um.")
    public void umdrehen() {
        Performance.getPerformance().lock();
        Performance.getPerformance().freeze();

        try {
            linksUm();
            linksUm();
        } finally {
            Performance.getPerformance().unfreeze();
            Performance.getPerformance().unlock();
        }
    }
```

Wird die Methode *freeze* mehrmals hintereinander aufgerufen, bedarf es übrigens genauso vieler *unfreeze*-Aufrufe, um den Eingefroren-Zustand wieder zu verlassen.

## *8.5 Zustandsabfrage*

Manchmal ist es notwendig zu wissen, ob eine bestimmte Aktion während einer laufenden Simulation stattfindet oder nicht. Bspw. möchte der Programmierer verhindern, dass die Aktion eines Aktionsbutton ausgeführt wird, während eine Simulation läuft. Dies kann durch eine entsprechende Performance-Klasse umgesetzt werden, die die beiden Methoden *started* und *stopped* überschreibt.

Beispiel: Bei unserem Demo-Beispiel soll die Größe des Territoriums nicht während einer laufenden Simulation verändert werden können. Unsere *Perf1*-Klasse muss dann folgendermaßen verändert werden:

```
import theater.*;


public class Perf1 extends Performance {
    private boolean running = false;

    @Override
    public void started() {
        setSpeed((Performance.MAX_SPEED + Performance.MIN_SPEED) / 2);
        running = true;
    }

    @Override
    public void stopped() {
```

```
        running = false;
    }

    public boolean isRunning() {
        return running;
    }
}
```

Und die Klasse des Aktionsbuttons *GroesseAendern* muss folgendermaßen angepasst werden:

```
public class GroesseAendern extends ActionHandler {
    public void handleAction(Stage stage, Actor solist) {
        if (((Perf1)Performance.getPerformance()).isRunning()) {
            return;
        }
        SizeDialog dialog = new SizeDialog(stage);
        dialog.setVisible(true);
    }
}
```

# 9 Referenzhandbuch

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

## *9.1 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>
 * Achtung: In Solist kann maximal 1 Akteur auf der Bühne platziert werden: der
 * Solist!
 * <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 (03.11.2009)
 *
 */
public class Stage implements PixelArea, Serializable {


   /**
    * Über den Konstruktor wird die Größe der Bühne festgelegt. Eine Bühne
    * besteht dabei aus einem rechteckigen Gebiet, das sich aus gleichförmigen
    * quadratischen Zellen zusammensetzt.
    *
    * @param noOfCols
    *            Anzahl der Spalten der Bühne
    * @param noOfRows
    *            Anzahl der Reihen der Bühne
    * @param cellSize
    *            Größe einer Zelle in Pixeln
    */
```

```
public Stage(int noOfCols, int noOfRows, int cellSize)

/**
 * Liefert die Anzahl an Spalten der Bühne.
 *
 * @return die Anzahl an Spalten der Bühne
 */
public int getNumberOfColumns()

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

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

/**
 * 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 den Solist auf der Bühne. Befindet sich bereits ein anderer
 * Solist auf der Bühne, wird dieser entfernt.
 * <p>
 * </p>
 * Angegeben werden die Spalte und Reihe der Zelle, auf der der Solist
 * platziert werden soll. Unter Platzierung ist dabei zu verstehen, dass der
 * Mittelpunkt des Icons, das den Solist repräsentiert, auf den Mittelpunkt
 * der Zelle gesetzt wird. Da das Koordinatensystem der Bühne nicht begrenzt
 * ist, kann der Solist auch außerhalb der Bühne platziert werden. Wenn der
 * Solist bereits auf der Bühne platziert ist, passiert nichts.
 * <p>
 * </p>
 * Seiteneffekt: Nach erfolgreicher Platzierung des Solist auf der Bühne
 * wird dessen Methode addedToStage mit dem entsprechenden Stage-Objekt als
 * Parameter aufgerufen.
 * <p>
 * </p Achtung: Ein Solist darf maximal einer Bühne zugeordnet sein.
 *
 * @param solist
 *            der Solist
 * @param col
```

```
     *            Spalte, in der der Solist platziert werden soll
     * @param row
     *            Reihe, in der der Solist platziert werden soll
     */
    public void setSolist(Actor solist, int col, int row)

    /**
     * liefert den aktuellen Solist
     *
     * @return der aktuelle Solist oder null
     */
    public Actor getSolist()

    /**
     * Platziert eine neue Komponente auf der Bühne. Angegeben werden die Spalte
     * und Reihe der Zelle, auf der die Komponente platziert werden soll. Unter
     * Platzierung ist dabei zu verstehen, dass der Mittelpunkt des Icons, das
     * die Komponente repräsentiert, auf den Mittelpunkt der Zelle gesetzt wird.
     * Da das Koordinatensystem der Bühne nicht begrenzt ist, kann eine
     * Komponente auch außerhalb der Bühne platziert werden. Wenn die Komponente
     * bereits auf der Bühne platziert ist, passiert nichts.
     * <p>
     * </p>
     * Seiteneffekt: Nach erfolgreicher Platzierung der Komponente auf der Bühne
     * wird die Methode addedToStage der Komponente mit dem entsprechenden
     * Stage-Objekt als Parameter aufgerufen.
     * <p>
     * </p 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(Prop comp, int col, int row)

    /**
     * Entfernt eine Komponente von der Bühne. Wenn die Komponente nicht auf der
     * Bühne platziert ist, passiert nichts. Achtung: Der Solist kann nicht von
     * der Bühne entfernt werden!
     *
     * @param comp
     *            die Komponente, die von der Bühne entfernt werden soll
     */
    public void remove(Component comp)

    /**
     * Entfernt alle Komponenten der Liste von der Bühne. Achtung: Der Solist
     * kann nicht von der Bühne entfernt werden!
     *
     * @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 in Solist nicht 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 in Solist nicht 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 nicht 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 nicht 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 nicht 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 in Solist nicht 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
 * in Solist nicht 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 nicht 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 nicht 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 nicht 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 nicht 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 nicht 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 nicht 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 nicht 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)

}
```

## 9.2 Component, Actor, Prop

### 9.2.1 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.
 * <p>
 * </p>
 * Die wichtigsten Methoden der Klasse Component sind Methoden, um Akteuren und
 * Requisiten ein Icon zuzuordnen und sie auf der Bühne bewegen, also
 * umplatzieren oder bspw. rotieren zu können. Weiterhin ist eine Menge von
```

```
 * Getter-Methoden definiert, um zum einen ihren Zustand abfragen und zum
 * anderen das Bühnen-Objekt ermitteln zu können, dem sie u.U. zugeordnet sind.
 * <p>
 * </p>
 * Wie die Klasse Stage enthält die Klasse Component darüber hinaus
 * Kollisionserkennungsmethoden zum Entdecken von Kollisionen der entsprechenden
 * Komponente mit anderen Komponenten sowie Methoden zur Verwaltung von Maus-
 * und Tastatur-Events. Die Klasse Component implementiert das Interface
 * PixelArea, so dass Akteure und Requisiten unmittelbar selbst in die
 * Kollisionserkennung mit einbezogen werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
public class Component extends Object implements PixelArea, Serializable {

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


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

   /**
    * Copy-Konstruktor
    *
    * @param comp
    *             die zu klonende Komponente
    */
   public Component(Component comp)

   /**
    * 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 addedToStage für die Komponente aufgerufen. Als
 * Parameter wird dabei das Bühnenobjekt übergeben. Sollen für eine
 * Komponente bestimmte Aktionen ausgeführt werden, sobald sie einer Bühne
 * zugeordnet wird, muss die Methode entsprechend überschrieben werden.
 *
 * @param stage
 *            das Objekt, das die Bühne repräsentiert, auf die die
 *            Komponente platziert wurde
 */
public void addedToStage(Stage stage)

/**
 * Mit Hilfe der remove-Methode der Klasse Stage kann eine Komponente von
 * einer Bühne entfernt werden. Nach der erfolgreichen Entfernung wird von
 * der Bühne diese Methode removedFromStage für die Komponente aufgerufen.
 * Als Parameter wird dabei das Bühnenobjekt übergeben. Sollen für eine
 * Komponente bestimmte Aktionen ausgeführt werden, sobald sie von einer
 * Bühne entfernt wird, muss die Methode entsprechend überschrieben werden.
 *
 * @param stage
 *            das Objekt, das die Bühne repräsentiert, von der die
 *            Komponente entfernt wurde
 */
public void removedFromStage(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 in Solist nicht 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 in Sloist nicht der Fall.
 *
 * @return true, falls die Komponente Tastatur-Ereignisse behandelt
 */
public boolean isHandlingKeyEvents()

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

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

}
```

### 9.2.2 Actor

```
package theater;

/**
 * Die Klasse Actor ist die Basisklasse aller Akteure. Sie erbt alle Methoden
 * ihrer Oberklasse Component.
 * <p>
 * </p>
 * Im Normalfall gibt es in Solist nur eine Actor-Klasse. Dies ist die
 * Oberklasse der Solist-Klasse, von der der Solist erzuegt wird. Sollte es
 * mehrere Actor-Klassen geben, muss der Programmierer die Solist-Oberklasse
 * explizit auswählen.
 * <p>
 * </p>
 * Im Solist-Programm muss die Methode "void main()" überschrieben
 * werden. In dieser wird das Eigenleben des Solisten festgelegt, wobei u. a.
 * die geerbten Methoden der Klasse Component genutzt werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
public class Actor extends Component implements Serializable {

   /**
    * Default-Konstruktor der Klasse Actor
    */
```

```
   public Actor()

   /**
    * Copy-Konstruktor der Klasse Actor
    *
    * @param actor
    *             der zu klonende Akteur
    */
   public Actor(Actor actor)

   /**
    * die von der Solist-Klasse zu überschreibende Methode
    */
   public void main()

   /**
    * erzeugt eine Instanz der Klasse Solist
    *
    * @return der Solist
    */
   public static Actor createSolist()

   /**
    * erzeugt einen Clon des uebergebenen Solist
    *
    * @param solist
    *             der zu klonende Solist
    * @return der geklonte Solist
    */
   public static Actor cloneSolist(Actor solist)

}
```

### 9.2.3 Prop

```
package theater;

import java.io.Serializable;

/**
 * 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 (03.11.2009)
 *
 */
public class Prop extends Component implements Serializable {

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

}
```

## *9.3 Performance*

```
package theater;
```

```
/**
 * Die Klasse Performance definiert Methoden zur Steuerung und Verwaltung der
 * Ausführung von Theater-Programmen(stop, suspend, setSpeed, freeze, unfreeze).
 * <p>
 * </p>
 * Weiterhin werden Methoden definiert, die unmittelbar nach entsprechenden
 * Steuerungsaktionen aufgerufen werden, und zwar auch, wenn die Aktionen durch
 * die Steuerungsbuttons des Simulators ausgelöst wurden(started, stopped,
 * speedChanged). Möchte ein Programmierer zusätzliche Aktionen mit den
 * entsprechenden Steuerungsaktionen einhergehen lassen, kann er eine
 * Unterklasse der Klasse Performance definieren und hierin die entsprechende
 * Methode überschreiben.
 * <p>
 * </p>
 * Zusätzlich stellt die Klasse Performance eine Methode playSound zur
 * Verfügung, mit der eine Audio-Datei abgespielt werden kann. Die Datei muss
 * sich im Unterverzeichnis "sounds" des entsprechenden Theraterstücks befinden.
 * Unterstützt werden die Formate wav, au und aiff.
 * <p>
 * </p>
 * Über die Methode setActiveStage ist es möglich, die aktuell aktive Bühne
 * gegen eine andere Bühne auszutauschen, d.h. das Bühnenbild zu wechseln. Unter
 * Umständen aktive Akteure der alten Bühne werden dabei nicht automatisch
 * gestoppt. Die Methode getActiveStage liefert die gerade aktive Bühne.
 * <p>
 * </p>
 * Das während einer Aufführung aktuelle Performance-Objekt kann mit Hilfe der
 * statischen Methode getPerformance ermittelt werden. Ein Wechsel des
 * Performance-Objektes ist während einer Aufführung nicht möglich.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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 in
 * Solist NICHT aufgerufen, wenn der Nutzer im Theater-Simulator den
 * Stopp-Button anklickt.
 */
public final void stop()

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

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

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

/**
 * Diese Methode wird aufgerufen, wenn sich die Geschwindigkeit des
```

```
 * Theaterstücks ändert.
 * <p>
 * </p>
 * Sollen mit einer Geschwindigkeitsänderung weitere Aktionen einhergehen,
 * muss der Programmierer eine Klasse von der Klasse Performance ableiten
 * und diese Methode entsprechend überschreiben.
 *
 * @param newSpeed
 *            die neue Geschwindigkeit
 */
public void speedChanged(int newSpeed)

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

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

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

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

/**
 * Bei Aufruf dieser Methode wird der Eingefroren-Zustand wieder verlassen.
 * Dabei gilt: Zu jedem Aufruf von freeze muss es einen Aufruf von unfreeze
 * geben. Zusätzliche Aufrufe von unfreeze sind wirkungslos.
 */
public void unfreeze()


/**
```

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

   /**
    * Setzt eine systeminterne Sperre. Während die Sperre gesetzt ist, sind
    * keine Benutzeraktionen (auch kein pause, stop) wirksam. Diese werden erst
    * nach einem anschließenden Aufruf der Methode unlock ausgeführt. Die
    * Methode sollte genutzt werden, um das Ändern von internen Attributen und
    * entsprechende Auswirkungen auf die Bühne als atomare Aktionen
    * durchzuführen.
    * <p>
    * </p>
    * Bei der Sperre handelt es sich um einen Reentrant-Lock! D.h. es können
    * durchaus mehrere lock-Aufrufe verschachtelt werden, für jeden lock-Aufruf
    * ist jedoch für eine Entsperrung ein entsprechender unlock-Aufruf
    * notwendig!
    */
   public void lock()

   /**
    * Gibt die mittels der Methode lock gesetzte Sperre wieder frei.
    */
   public void unlock()

}
```

## *9.4 Bilder*

### 9.4.1 TheaterImage

```
package theater;

/**
 * TheaterImage ist eine Theater-Klasse mit vielfältigen Methoden zum Erzeugen
 * und Manipulieren von Bildern bzw. Ikons, die dann Akteuren, Requisiten oder
 * der Bühne zugeordnet werden können. Die Bilder lassen sich dabei auch noch
 * zur Laufzeit verändern, so dass mit Hilfe der Klasse TheaterImage bspw.
 * Punktezähler für kleinere Spiele implementiert werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
public class TheaterImage implements Serializable {

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

```
 *            Actor.class, Prop.class, Performance.class)
 */
public TheaterImage(Class<?> father)

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

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

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

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

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

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

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

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

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

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

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

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

/**
 * Zeichnet im TheaterImage ein Polygon in der aktuellen Zeichenfarbe. Es
 * wird automatisch ein Linie hinzugefügt, die das Polygon schließt.
 *
 * @param xPoints
 *            x-Koordinaten der Linien
 * @param yPoints
 *            y-Koordinaten der Linien
 * @param nPoints
```

```
 *            Anzahl der Liniensegmente
 */
public void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)

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

/**
 * Zeichnet ein existierendes TheaterImage an einer bestimmten Stelle in das
 * aufgerufene TheaterImage
 *
 * @param image
 *            das TheaterImage, das gezeichnet werden soll (darf nicht null
 *            sein)
 * @param x
 *            x-Koordinate, an der das Image gezeichnet werden soll
 * @param y
 *            y-Koordinate, an der das Image gezeichnet werden soll
 */
public void drawImage(TheaterImage image, int x, int y)

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

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

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

/**
 * Zeichnet im TheaterImage ein gefülltes Polygon in der aktuellen
```

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

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

/**
 * Setzt ein bestimmtes Pixel des TheaterImages auf eine bestimmte Farbe.
 *
 * @param x
 *            x-Koordinate des Pixels
 * @param y
 *            y-Koordinate des Pixels
 * @param color
 *            neue Farbe des Pixels
 */
public void setColorAt(int x, int y, java.awt.Color color)

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

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

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

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

/**
 * Skaliert das TheaterImage auf einebestimmte Größe.
 *
 * @param width
 *            die neue Breite des TheaterImages
 * @param height
```

```
     *            die neue Höhe des TheaterImages
     */
    public void scale(int width, int height)

    /**
     * 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()

    /**
     * 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()

}
```

### 9.4.2 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 (03.11.2009)
 *
 */
public class TheaterIcon extends TheaterImage {

    /**
     * Konstruktor zum Initialisieren eines TheaterIcons mit einem Bild aus
     * einer Datei. Erlaubte Bildformate sind gif, jpq und png.
     *
     * @param filename
     *            Name der Bilddatei; die Datei muss sich im Unterverzeichnis
     *            "images" des Theaterstücks befinden
     * @throws IllegalArgumentException
     *             wird geworfen, wenn die Datei nicht existiert, keine gültige
     *             Bilddatei ist oder nicht lesbar ist
     */
    public 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 TheaterIcon durch ein
    * Image-Objekt realisiert. Diese Methode liefert das
    * entsprechende Objekt. Achtung:
    *
    * @return das aktuelle interne Image-Objekt
    */
   public java.awt.Image getAwtImage()

}
```

## *9.5 Ereignisse*

### 9.5.1 KeyInfo

```
package theater;

/**
 * Sowohl die Klasse Stage als auch die Klasse Component definieren die von der
 * Java-GUI-Programmierung bekannten Methoden zur Verarbeitung von
 * Tastatur-Events: keyTyped, keyPressed und keyRealeased. Die Events
 * entsprechen dabei den Events des Java-AWT in der Klasse
 * java.awt.event.KeyListener. Den Methoden werden Objekte vom Typ KeyInfo
 * übergeben, über die genauere Informationen über das entsprechende Event
 * abgefragt werden können.
 * <p>
 * </p>
 * Die Klasse KeyInfo ist von der Klasse java.awt.event.KeyEvent abgeleitet, so
 * dass auch alle deren Methoden benutzt werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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

}
```

## 9.5.2 MouseInfo

```
package theater;

/**
```

```
 * Sowohl die Klasse Stage als auch die Klasse Component definieren die von der
 * Java-GUI-Programmierung bekannten Methoden zur Verarbeitung von Maus-Events:
 * mousePressed, mouseReleased, mouseClicked, mouseDragged, mouseMoved,
 * mouseEntered und mouseExited. Die Events entsprechen dabei den Events des
 * Java-AWT in den Klassen java.awt.event.MouseListener bzw.
 * java.awt.event.MouseMotionListener. Den Methoden werden Objekte vom Typ
 * MouseInfo übergeben, über die genauere Informationen über das entsprechende
 * Event abgefragt werden können.
 * <p>
 * </p>
 * Die Klasse MouseInfo ist von der Klasse java.awt.event.MouseEvent abgeleitet,
 * so dass auch alle deren Methoden benutzt werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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 Ikons.
    *
    * @return die x-Koordinate des Maus-Events relativ gesehen zum
    *         Source-Objekt
    *
    * @see java.awt.event.MouseEvent#getX()
    */
   public int getX()

   /**
    * Überschreibt die geerbte Methode. In dem Fall, dass die Bühne über das
    * Maus-Event informiert wird, also das aktuelle Bühnenobjekt das
    * Source-Objekt ist, liefert die Methode die y-Koordinate des Mauszeigers
    * bezüglich des Hintergrundes. In dem Fall, dass eine Komponente über das
    * Maus-Event informiert wird, also die Komponente das Source-Objekt ist,
    * liefert die Methode die y-Koordinate des Mauszeigers bezüglich des der
    * Komponente zugeordneten Ikons.
    *
```

```
 * @return die y-Koordinate des Maus-Events relativ gesehen zum
 *         Source-Objekt
 *
 * @see java.awt.event.MouseEvent#getY()
 */
public int getY()

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

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

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

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

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

```
     * benachrichtigte Objekte über die Methode getUserObject abfragen können.
     *
     * @return das dem MouseInfo-Objekt mittels der Methode setUserObject
     *         zugeordnete Objekt oder null, falls kein Objekt zugeordnet wurde.
     */
    public Object getUserObject()

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

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

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

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

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

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

}
```

## 9.6 Kollisionserkennung

### 9.6.1 PixelArea

```
package theater;

/**
 * PixelArea ist ein Interface, das die Grundlage der
 * Kollisionserkennungsmethoden darstellt. Eine PixelArea kann man sich dabei
 * als ein beliebiges Gebiet auf der Bühne vorstellen. Neben einigen zur
 * Verfügung gestellten Standardklassen (Point, Rectangle, Cell, CellArea)
 * implementieren auch die Klassen Stage und Component das Interface. Dadurch
 * sind nur sehr wenige Methoden zur Kollisionserkennung notwendig, die jedoch
 * sehr flexibel und umfassend eingesetzt werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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);
}
```

### 9.6.2 Rectangle

```
package theater;

/**
 * Die Klasse Rectangle repräsentiert ein rechteckiges Gebiet auf der Bühne. Sie
 * implementiert das Interface PixelArea, so dass mit dieser Klasse Kollisionen
 * von rechteckigen Gebieten mit anderen Gebieten der Bühne überprüft werden
```

```
 * können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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()
}
```

### 9.6.3 Point

```
package theater;
```

```
/**
 * Die Klasse Point repräsentiert ein Pixel auf der Bühne. Sie implementiert das
 * Interface PixelArea, so dass mit dieser Klasse Kollisionen von Pixeln mit
 * anderen Gebieten der Bühne überprüft werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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 Klase java.awt.Point (darf nicht null sein)
    */
   public Point(java.awt.Point p)

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

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

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

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

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

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

}
```

### 9.6.4 Cell

```
package theater;

/**
 * Die Klasse Cell repräsentiert eine Zelle der Bühne. Sie implementiert das
 * Interface PixelArea, so dass mit dieser Klasse Kollisionen von Zellen mit
 * anderen Gebieten der Bühne überprüft werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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()
}
```

### 9.6.5 CellArea

```
package theater;

/**
```

```
 * Die Klasse CellArea repräsentiert ein Menge von Zellen (genauer ein
 * rechteckiges Gebiet von Zellen) der Bühne. Sie implementiert das Interface
 * PixelArea, so dass mit dieser Klasse Kollisionen von Zellen mit anderen
 * Gebieten der Bühne überprüft werden können.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
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()

}
```

## *9.7 Aktionsbuttons*

### 9.7.1 ActionHandler

```
package theater;

/**
 * Klasse für die Realisierung von Aktionsbuttons.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
public abstract class ActionHandler {

    /**
     * Die überschriebene Methode wird aufgerufen, wenn der Benutzer den
```

```
    * Aktionsbutton anklickt.
    *
    * @param stage
    *            die aktuelle Bühne
    * @param solist
    *            der aktuelle Solist
    */
   public abstract void handleAction(Stage stage, Actor solist);

}
```

### 9.7.2 ClickHandler

```
package theater;

import java.util.List;

/**
 * Klasse für die Realisierung von Aktionsbuttons, bei denen Mausklicks auf der
 * Bühne zu bestimmten Aktionen führen sollen.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
public abstract class ClickHandler {

   /**
    * Die überschriebene Methode wird aufgerufen, wenn der Aktionsbutton
    * aktiviert ist und der Benutzer auf die Bühne klickt.
    *
    * @param stage
    *            die aktuelle Bühne
    * @param solist
    *            der Solist
    * @param col
    *            die Spalte der Bühne, auf der der Mausklick erfolgt ist
    * @param row
    *            die Reihe der Bühne, auf der der Mausklick erfolgt ist
    * @param clickedComponents
    *            u.U. angeklickte Komponenten auf der Bühne (in der Reihenfolge
    *            ihrer Z-Koordinaten)
    */
   public abstract void handleClick(Stage stage, Actor solist, int col,
      int row, List<Component> clickedComponents);

}
```

### 9.7.3 NewPropHandler

```
package theater;

/**
 * Klasse für die Realisierung von Aktionsbuttons, über die ein neues
 * Prop-Objekt erzeugt und auf der Bühne platziert werden soll.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 *
 */
public abstract class NewPropHandler {

   /**
    * Liefert das neu erzeugt Prop-Objekt. Die überschriebene Methode wird
```

```
    * aufgerufen, wenn der Benutzer den Aktionsbutton anklickt.
    *
    * @return
    */
   public abstract Prop newProp();

}
```

## *9.8 Annotationen*

### 9.8.1 Description

```
package theater;

/**
 * Beschreibung für eine Methode oder ein Attribut, die als Tooltip eines
 * Befehls im Befehlsfenster erscheinen sollen.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 */
@Documented
@Target( { ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Description {
   String value();
}
```

### 9.8.2 Invisible

```
package theater;

/**
 * Eine Actor- oder Prop-Methode, die als Invisible deklariert ist, wird nicht
 * im Popup-Menu der Komponente und auch nicht im Befehlsfenster angezeigt.
 *
 * @author Dietrich Boles, Universität Oldenburg, Germany
 * @version 1.0 (03.11.2009)
 */
@Documented
@Target( { ElementType.METHOD, ElementType.CONSTRUCTOR })
@Retention(RetentionPolicy.RUNTIME)
public @interface Invisible {

}
```

# 10 Beispiel-Theaterstücke

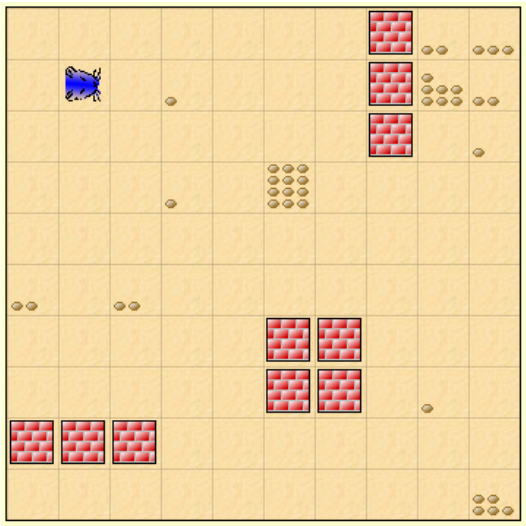
Die folgenden Theaterstücke dienen als Beispiele für den Einsatz von Solist. Sie finden die Theaterstücke im Unter-Ordner *plays* des Solist-Ordners (siehe auch Kapitel 2). Im Unter-Ordner *simulatoren* finden Sie die aus den Theaterstücken generierten Simulatoren.

## *10.1 Theaterstück demo und oodemo*

Die Theaterstücke *demo* und *oodemo* sind vollständige Versionen der in diesem Handbuch exemplarisch entwickelten Demo-Simulatoren.

## *10.2 Theaterstück hamster*

Das Theaterstück *hamster* spiegelt eine Abbildung des Java-Hamster-Modells und des Hamster-Simulators wider (siehe http://www.java-hamster-modell.de) Programmieranfänger müssen Programme schreiben, in denen sie einen Hamster durch ein virtuelles Territorium steuern und Aufgaben wie das Einsammeln von Körnern oder das Umlaufen von Mauern lösen lassen.



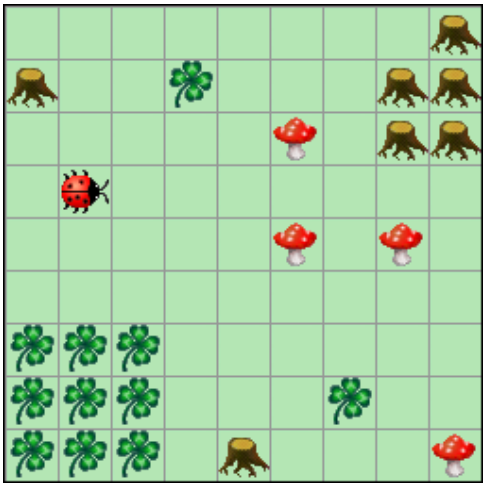
Hamster-Simulator-Light

Das Theaterstück hamster demonstriert viele Aspekte von Solist. Denjenigen von Ihnen, die zum ersten Mal mit Solist arbeiten, empfehle ich, sich als erstes einmal die einzelnen Klassen des hamster-Theaterstücks anzuschauen. Dann werden viele Aspekte und Funktionalitäten von Solist unmittelbar klar. Das hamster-Theaterstück kann auch als Ausgangspunkt für eigene Ideen ähnlicher MPWs genutzt werden.

## *10.3 Theaterstück kara*

Das Theaterstück *kara* spiegelt eine Abbildung des Java-Kara-Modells wider (siehe http://www.swisseduc.ch/informatik/karatojava/javakara/). Programmieranfänger müssen Programme schreiben, in denen sie Kara, den Marienkäfer durch ein Territorium steuern und dabei Kleeblätter fressen und Pilze verschieben lassen.

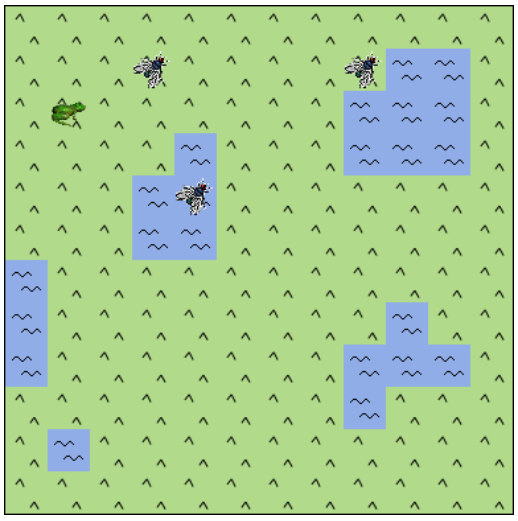
Durch dieses Theaterstück wird die Umsetzung eher objektbasierter Solist-Programme (siehe Abschnitt 7.11) demonstriert.



Solist-Kara-Simulator

## *10.4 Theaterstück frosch*

Ein dem Hamster-Modell ähnliches Modell, bei dem ein Frosch übers Gras hüpfen und durchs Wasser schwimmen muss.

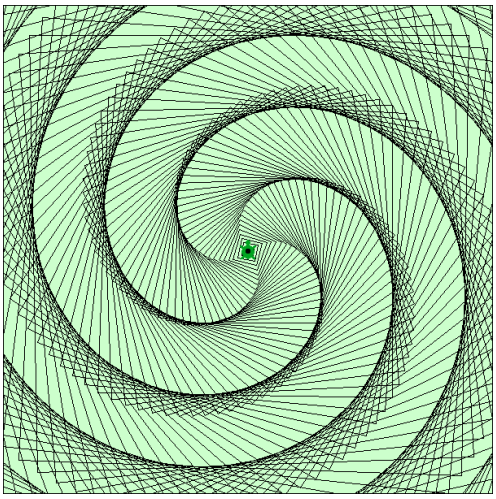


Frosch-Simulator

Das frosch-Theaterstück zeigt schön, wie einfach es ist, das ursprüngliche hamster-Theaterstück an geringfügig geänderte Anforderungen anzupassen.

## *10.5 Theaterstück turtle*

Der MPW-Klassiker: Steuern Sie eine Schildkröte über ein virtuelles Zeichenblatt und lassen sie schöne Zeichnungen anfertigen. Turtle-Programme dienen insbesondere dazu, beim Zeichnen so genannter Fraktale den Einsatz von Rekursion zu verdeutlichen.



Turtle-Simulator

Das turtle-Theaterstück demonstriert unter anderem den Einsatz der Klasse TheaterImage.

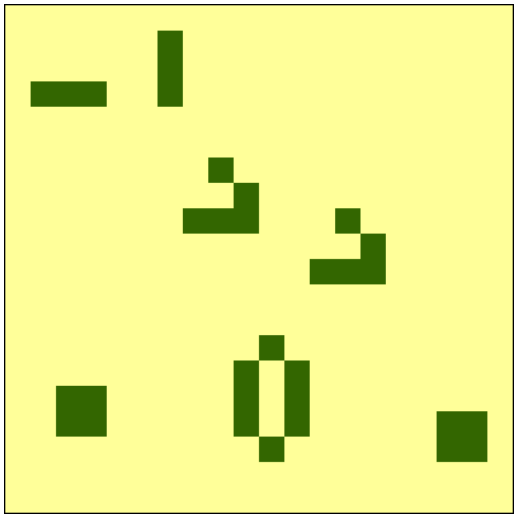
## *10.6 Theaterstück geisterstunde*

Eine PLU für wirkliche Programmieranfänger: Programmieranfänger müssen Programme schreiben, in denen sie ein Mädchen zu einem Kürbis steuern. Doch Vorsicht: Es ist Halloween und unsichtbare Geister sind unterwegs!

Geisterstunde-Simulator

## *10.7 Theaterstück gameoflife*

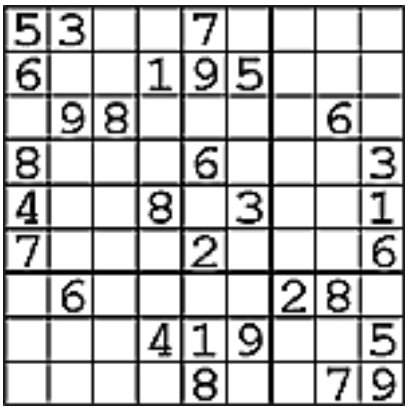
Das Theaterstück gameoflife setzt das bekannte Game-of-Life-System um. Programmieranfänger müssen ein Programm entwickeln, das das Game-of-Life simuliert. Dabei müssen sie Arrays einsetzen.



Game-of-Life-Simulator

## *10.8 Theaterstück sudoku*

Das Theaterstück sudoku ermöglicht, Programme zu entwickeln, die beliebige Sudoku-Rätsel lösen. Sehr schön veranschaulichen lässt sich hierdurch das Prinzip der Rekursion und des Backtrackings.

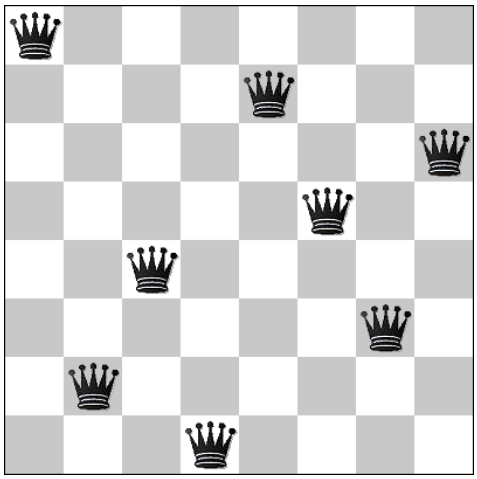


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

Sudoku-Simulator

## *10.9 Theaterstück damen*

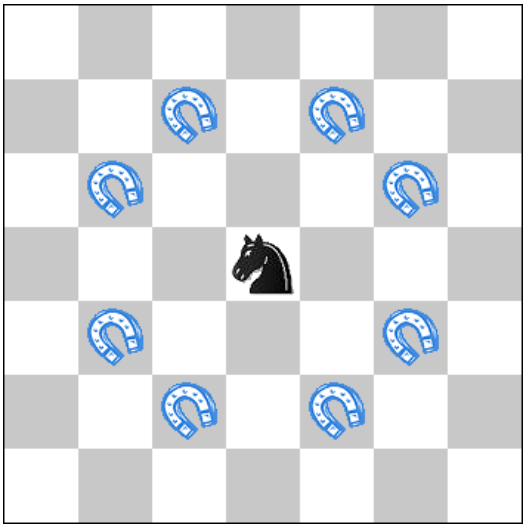
Das Theaterstück damen ermöglicht, Programme zu entwickeln, die das bekannte Damen-Problem lösen (8 Damen sind auf einem Schachbrett so zu platzieren, dass sie sich gegenseitig nicht schlagen können). Genauso wie beim sudoku-Theaterstück lässt sich hierdurch das Prinzip der Rekursion und des Backtrackings sehr schön veranschaulichen.



Damen-Simulator

## *10.10 Theaterstück springer*

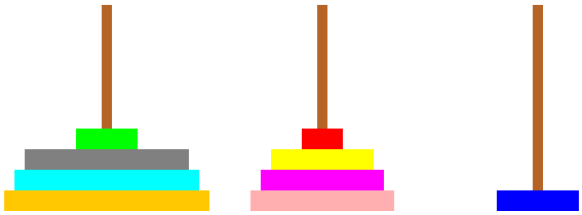
Das Theaterstück springer ermöglicht, Programme zu entwickeln, die das bekannte Springer-Problem lösen (ein Springer beim Schach muss alle Felder eines Schachbretts genau einmal besuchen). Genauso wie beim sudoku- und beim damen-Theaterstück lässt sich hierdurch das Prinzip der Rekursion und des Backtrackings sehr schön veranschaulichen.



Springer-Simulator

## *10.11Theaterstück hanoi*

Das Theaterstück springer ermöglicht, Programme zu entwickeln, die das bekannte Spiel Türme-von-Hanoi spielen Genauso wie beim sudoku-, beim damen-und beim springer-Theaterstück lässt sich hierdurch das Prinzip der Rekursion und des Backtrackings sehr schön veranschaulichen.

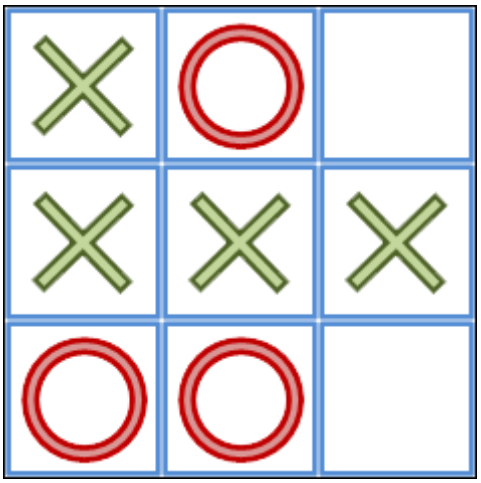


hanoi-Simulator

In diesem Theaterstück wird insbesondere demonstriert, wie man in Solist Animationen realisieren kann.

## *10.12Theaterstück tictactoe*

Das Theaterstück tictactoe ermöglicht, Programme zu entwickeln, die gegen Menschen und/oder andere Programme das bekannte Spiel TicTacToe spielen.

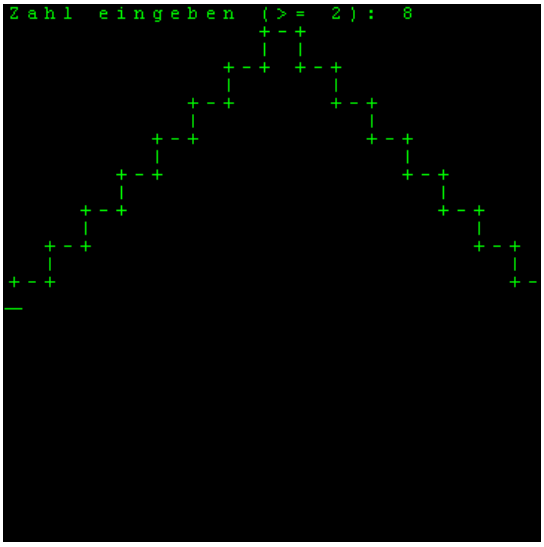


tictactoe-Simulator

In diesem Theaterstück wird insbesondere demonstriert, wie man in Solist Interaktionen mit dem Benutzer realisieren kann. Das Theaterstück kann auch als Ausgangspunkt für die Umsetzung anderer 2-Personen-Strategiespiele (Schach, Reversi, 4-Gewinnt, …) genutzt werden.

## *10.13Theaterstück terminal*

Das Theaterstück terminal simuliert ein ASCII-Terminal, auf das Daten (via System.out) ausgegeben und Daten vom Benutzer abgefragt( via System.in) werden können.




terminal-Simulator

In diesem Theaterstück wird demonstriert, wie man in Solist Interaktionen mit dem Benutzer realisieren kann.

## *10.14Theaterstück fruechte*

Das Theaterstück fruechte ermöglicht, Programme zu entwickeln, die ein gegebenes Optimierungsproblem lösen. Programme steuern einen Einkaufswagen und müssen dabei möglichst viele vom Himmel fallende Früchte einfangen.


fruechte-Simulator

## *10.15Weitere Theaterstücke*

Es werden regelmäßig neue Theaterstücke von mir entwickelt. Diese können Sie von der Solist-Website http://www.programmierkurs-java.de/solist herunterladen. Wenn Sie eigene Theaterstücke entwickelt haben, können Sie mir diese auch gerne zukommen lassen, so dass ich sie, wenn Sie wünschen, ebenfalls auf der Solist-Website anderen Solist-Nutzern zugänglich mache.