Robby Projekt

Erweiterung des Greenfoot Roboter-Szenarios um Sensorik, Speicher und Hindernisumgehung

Adrian Schrader Moritz Jung Alexander Riecke

17. Januar 2016

Zusammenfassung

Inhaltsverzeichnis

1 Implementierung

1.1 Sensorik

Aufgabenstellung

Um Robby Anhaltspunkt für seine Aktionen zu geben, sollen zwei verschiedene Arten von Sensoren eingeführt werden, mit denen Robby seine Umgebung abtasten kann. Robby kann nicht durch eine Wand laufen, also sollte er erkennen können, ob in seiner Umgebung solch ein Hindernis auftaucht.

Eines der Hauptaktionen eines Roboters in diesem Szenario ist das Sammeln von Akkus für Energie. Um gezielter nach Akkus suchen muss Robby seine Umgebung nach ihnen abtasten.

wandHinten() Wenn, in Bezug auf Robbys Laufrichtung gesehen, ein Actor der Klasse Wand ein Feld hinter Robby steht, soll die Methode **`true`** zurückgeben, ansonsten **`false`**.

akkuVorne() [akkuRechts(), akkuLinks()] Wenn, in Bezug auf Robbys Laufrichtung gesehen, ein Actor der Klasse Akku ein Feld vor [rechts von, links von] Robby steht, soll die Methode **`true`** zurückgeben, ansonsten **`false`**.

Problematik

Diese vier Fälle können auf das Problem reduziert werden aus der Blickrichtung des Roboters und dem spezifischen Suchwinkel einen Vektor vom Roboter zum Suchfeld zu konstruieren, damit überprüft werden kann, ob die Methode `this.getOneObjectAtOffset(int v_x,int v_y,Class<?> c)` ein Objekt übergibt oder nicht. Die Mutterklasse Roboter löst die Aufgabe, in dem sie jeden einzelnen Suchvektor als einzelne Methode implementiert und in ihr die vier Blickrichtungen abfragt, um daraus einen fest einprogrammierten Vektor auszuwählen. Diese Herangehensweise funktioniert zwar, ist jedoch für eine schlanke, wiederverwendbare, nachvollziehbare und skalierbare Klasse nicht geeignet. Um die Klasse evtl. später um Abfragen zusätzlicher Aktoren erweitern zu können, muss die Abfrage in einer einzigen Methode stattfinden. Diese errechnet dynamisch aus den Faktoren den gewünschten Vektor.

Lösung

Um die Lösung für dieses Problem zu verstehen ist es hilfreich den gesuchten Vektor $\vec{v}$ als Zeiger zu verste-

hen, dessen Betrag immer auf $|\vec{v}| = 1$ genormt ist. Aus dem Winkel θ von der Horizontalen lässt sich dann die Komponente in x und y-Richtung mithilfe von Sinus und Kosinus errechnen.

$$v_x = |\vec{v}| \cdot \cos(\theta) \qquad v_y = |\vec{v}| \cdot \sin(\theta) \tag{1.1.1}$$

Für unseren Anwendungsfall interessieren uns nur ganzzahlige Werte von v_x und v_y zwischen -1 und 1. Daher können wir die Domäne für θ enger eingrenzen.

$$\theta \in \left\{ k \cdot \frac{\pi}{2} \,|\, k \in \mathbb{N}_0 \right\} \tag{1.1.2}$$

Für die in Abschnitt 1.1 besprochene Methode müssen wir jedoch zuerst den Winkel für die Laufrichtung und den Suchwinkel addieren und in das Bogenmaß umrechnen. Aufgespalten in seine Komponenten kann der Vektor in die Methode `this.getOneObjectAtOffset(int v_x,int v_y,Class<?> c)` eingegeben und damit die Existenz des Objekts überprüft werden. Dargestellt ist die implementierte Methode im Quellcode 1.1.

```
/**
 * Der Sensor überprüft, ob sich neben der
 ↪  Laufrichtung von Robby ein
 * anderer Actor befindet.
 * @param direction Winkel von der Laufrichtung
 ↪  zum Suchfeld.
 * @param class Klasse des gesuchten Actors
 * @return boolean
 */
public boolean istObjektNebendran(int
 ↪  direction, Class<?> cl)
{
   double angle = (this.getRotation() +
    ↪  direction) / 180.0 * Math.PI;

   return (this.getOneObjectAtOffset(
          (int)Math.cos(angle),
          (int)Math.sin(angle), cl) != null);
}
```

Quellcode 1.1: Implementation der Basismethode für die Sensorik aus Robby.java.

Nachdem wir diese Grundmethode implementiert haben, können die gesuchten Methoden durch eine einzelne Abfrage dargestellt werden. `akkuVorne()` verweist bspw. auf `istObjektNebendran(0, Akku.class)`. Da Greenfoot Winkel im Uhrzeigersinn misst, verwendet die Abfrage von `akkuRechts()` den Suchwinkel 90°. Eine Liste aller Implementierungen ist in Quellcode 4.1 zu finden.

1.2 Speicher

Aufgabenstellung

Problematik

Lösung

1.3 Hindernisse

Aufgabenstellung

Aufgabenstellung war es, Robby ein geschlossenes Hindernis aus Wänden umrunden zu lassen, so dass er nach erfolgreicher Umrundung wieder am Ausgangspunkt ankommt. Auch die Weltgrenze wird als Hindernis wahrgenommen.

Problematik

Als augenscheinlich einfachste Lösung stellte sich eine Verkettung von mehreren if-Abfragen heraus, in der nacheinander von rechts beginnend gegen den Uhrzeigersinn die einzelnen möglichen Bewegungsrichtungen abgefragt wurden. Diese Lösung erwies sich jedoch als sehr verschachtelt, da für jede Abfrage eine einzelne Reaktionsanweisung erstellt werden musste. Weiterhin konnten anfangs nur sehr begrenzte Tests durchgeführt werden, da die Funktion in einem Ablauf ausgeführt wird und somit keine Möglichkeit besteht, auch weitergehende Aufgaben währenddessen auszuführen.

Lösung

Durch Einführen einer do-while-Schleife und einer for-Schleife für die Abfragen der Umgebung, also der Wände und Weltgrenzen auf benachbarten Feldern, konnte der Code auf ein Minimum reduziert werden, in dem zwar immernoch von Rechts beginnend die Umgebung angefragt wird, jedoch die Funktion immer wieder mit anderem Winkel aufgerufen wird und nicht für jede Richtung eine neue Abfrage geschrieben werden muss. Auch eine Möglichkeit zum Ausführen von weiterem Code während der Umrundung wurde implementiert, um bspw. auch weitergehende Aufgaben, wie zum Beispiel das Ablegen von Schrauben zu ermöglichen, aber auch, um zu die zu Testzwecken benötigten Daten während der Umrundung zu erfassen.

Beschreibung der Funktion

Der endgültige Algorhitmus startet mit mehreren Vorbereitungen, zuerst wid die aktuelle Position, aufgeteilt in X- und Y-Koordinaten in die lokale Variablen startX und startY gespeichert. Diese werden später als Abbruchbedingung der Schleife gebraucht. Dann

wird der Roboter in die richtige Richtung gedreht, da er zu Anfang gegen das Hindernis blickt. Erst dann beginnt der eigentliche Algorhitmus, der immer wieder aufgerufen wird. Hier werden dann nacheinander, von rechts beginnend die einzelnen Bewegungsrichtungen abgefragt, sobald in einer Richtung kein Hindernis gefunden wird, kann zuerst eine Aktion ausgeführt werden, dann bewegt sich Robby auf das freie Feld. Wenn er sich auf das freie Feld bewegt hat, muss die Schleife abgebrochen und neu gestartet werden, damit wieder von rechts angefangen wird zu suchen. Die Schleife wird immer wieder aufgerufen, solange Robby nicht wieder auf seinem Startfeld, das am Anfang durch die beiden Koordinaten gespeichert wurde, steht. Wenn Robby dann wieder auf seinem Startfeld steht, dreht er sich wieder ein letztes Mal nach rechts, um wieder zum Hindernis zu blicken.

2 Funktionstests

Zum Überprüfen der Funktionalität haben wir eine Testengine entworfen. Die Klasse *FeatureTest* bietet mit der Funktion `boolean testAllFeatures()` die Funktionalität aller beschriebenen Funktionalitäten zu überprüfen.

Um eine automatische Funktionsüberprüfung nach dem kompilieren zu erreichen, wurde die Klasse `RoboterWelt` erweitert, damit Sie den Test starten kann. Im Prinzip des OOP wurden jedoch alle Tests in einer seperaten Klasse durchgeführt.

Die Klasse `FeatureTest` bietet das Grundgerüst für die Testengine. Hier sind die Methoden zu Hause, die reflexiv auf Robby zugreifen, um die Rückgabewerte seiner Methoden zu überprüfen oder Felder auszulesen.

In der `FeatureTest` erweiternden Klasse `RobbyTest` sind die unten beschriebenen Testabläufe implementiert. Die volle Umsetzung aller im Lastenheft angeforderten Funktionen ist gelungen und wird zu Beginn jedes Programms mit einem Testlog, der bspw. in Abbildung 2.1 zu sehen ist, bestätigt.

2.1 Sensorik

Robbys Sensorfunktionen für Akkus und Wände, die im Rahmen dieses Projekts erweitert wurden, sollen die jeweiligen Spielfiguren positiv, sowie negativ nachweisen können. Falsch positive und falsch negative Ergenisse müssen ausgeschlossen werden. Jede Funktion muss auSSerdem in unterschiedlichen Blickrichtungen getestet werden, um Zufallstreffer auszuschlieSSen.

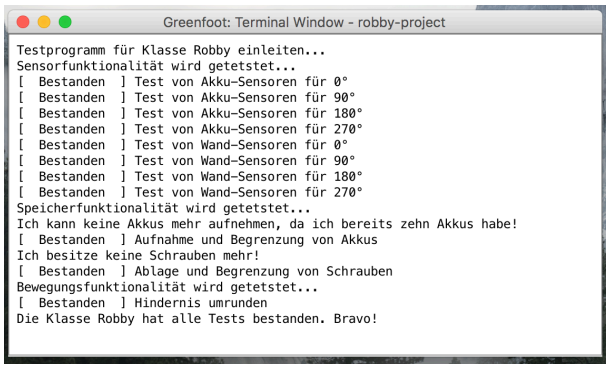

Abbildung 2.1: Erfolgreicher Testlog nach dem Kompilieren der Klassen.

Im Gegensatz zur Implementierung der Funktionen in Robby müssen die Testfunktionen wasserdicht sein. Deduktive mathematische Beweise werden daher durch das ausprobieren jedes möglichen Falls in jeder Ausgangssituation getestet.

Testablauf

Robby wird in Schleifen durch die verschiedenen Situationen/Winkel iteriert. Es wird für jeden Blickwinkel ($0°, 90°, 180°$ und $270°$) überprüft, ob die zuständige Funktion erkennt, dass kein Objekt neben Robby liegt und dass sich nach dem Einfügen des gesuchten Objekts neben Robby der Test positiv ausfällt. Die Objekte werden danach wieder aufgeräumt und aus der Welt entfernt.

Situationen:

Drehungen um $0°, 90°, 180°$ und $270°$

2.2 Speicher

Die Test sollen überprüfen, ob Robby beim Einsammeln oder Ablegen von Objekten auch sein Inventar im Speicher aktualisiert.

Testablauf

Akkus Robby wird immer wieder auf einen Akku gestellt und die Funktion zum einsammeln ausgeführt. Nach jedem Durchgang wird geprüft, ob Robby den Speicher um eins hochgezählt hat, die Variablengrenzen zwischen 0 und 10 nicht überschritten wurden und das Objekt tatsächlich entfernt wurde. Siehe `boolean testObjectAquisition(String, String, int, Class<?>)` in Quellcode 4.3.

Schraube Die Funktion zum Ablegen wird wiederhohlt ausgeführt. Nach jedem Durchgang wird geprüft,

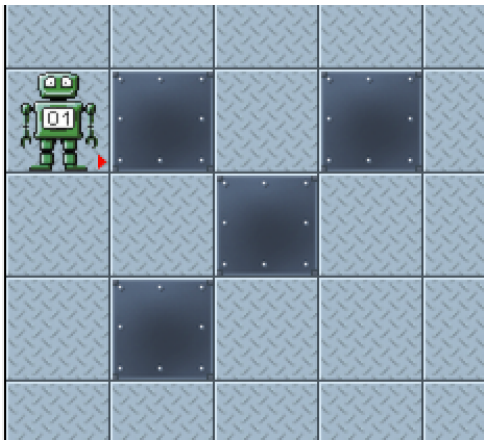

Abbildung 2.2: Das von der Testfunktion erstellte Hindernis.

ob Robby den Speicher um eins runtergezählt hat, die Variablengrenze von 0 nicht unterschritten wurden und das Objekt tatsächlich hinzugefügt wurde. Siehe `boolean testObjectDeposition(String, String, int, Class<?>)` in Quellcode 4.3.

2.3 Hindernisse und Bewegung

Die Test sollen überprüfen, ob Robby ein geschlossenes Hindernis aus Wänden umrunden kann ohne Kontakt zur Wand zu verliehren und wieder am Ausgangspunkt ankommt.

Testablauf

Robby wird in der Welt mit Blickrichtung zur Wand platziert und um ihn herum ein Hindernis, wie in Abbildung 2.2 dargestellt, aufgabaut. Da die Implementierung von `void hindernisUmrunden(Runnable)` ermöglicht, Benutzeraktionen während des Umrundens durchzuführen, wird getestet, ob sich in den acht Feldern um Robby eine Wand befindet. Wenn nicht ist der Test gescheitert. Auch muss gewährleistet sein, dass er am Ende wieder an der Ausgangsposition ankommt und die Spielfläche aufgeräumt wird.

3 Gruppenarbeit

Aus den Funktionsanforderungen des Lastenhefts für die Klasse Robby und die Dokumentation haben wir kleinere Aufgabenpakete zusammengestellt, die in der Gruppe verteilt werden können.

Sourcecode-Verwaltung

Um den Überblick über den aktuellen Stand zu behalten und die Sourcecodeversionen koordiniert zusammenführen zu können, haben wir gehostete git-Repositorys mit Issuetrackern und Milestones eingerichtet. Nach Abgabe der Dokumentation sind diese auch öffentlich verfügbar. Das Robbyprojekt ist unter `https://github.com/adrianschrader/robby-project` und die Dokumentation als LaTeX-Projekt unter `https://github.com/adrianschrader/robby-project-doc` zu finden.

Tabelle 3.1: Aufgabenübersicht und Zuständigkeiten des Projekts

Bereich	**Aufgabe**	**Zuständigkeit**
Sensorik	Funktionalität implementieren	Adrian
	Sourcecode kommentieren	Adrian
	Sourcecode aufräumen, anpassen und zusammenfügen	Adrian
	Schriftliche Dokumentation	Adrian
Speicher	Funktionalität implementieren	Alex
	Sourcecode kommentieren	Alex
	Sourcecode aufräumen, anpassen und zusammenfügen	Adrian
	Schriftliche Dokumentation	Alex
Hindernis	Funktionalität implementieren	Moritz
	Sourcecode kommentieren	Moritz
	Sourcecode aufräumen, anpassen und zusammenfügen	Adrian / Moritz
	Schriftliche Dokumentation	Moritz
Tests	Planung	Adrian
	Funktionalität implementieren	Adrian
	Sourcecode kommentieren	Adrian
	Sourcecode aufräumen, anpassen und zusammenfügen	Adrian
	Schriftliche Dokumentation	Adrian

4 Anhang: Vollständiger Quellcode

Alle geänderten oder hnzugefügten Klassen werden im Folgenden aufgeführt. Die Zeilenangaben links entsprechen denen im Greenfoot-Szenario.

4.1 Robby.java

```
1  import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)
2
3  /**
4   * Der Roboter Robby kann sich in der Welt orientieren, Hindernisse umrunden,
5   * Schrauben hinterlassen und Akkus aufnehmen.
6   * @author Adrian Schrader, Moritz Jung, Alexander Riecke
7   * @version 1.5
8   */
9  public class Robby extends Roboter
10 {
11     /* Konstanten */
12     /** Maximale Anzahl an Akkus, die Robby tragen kann (Standard: 10) **/
13     public static final int MAX_AKKUS      = 10;
14     /** Anzahl an Akkus, mit der Robby initialisiert werden soll (Standard: 0) **/
15     public static final int INIT_AKKUS     =  0;
16     /** Anzahl an Schrauben, mit der Robby initialisiert werden soll (Standard: 10) **/
17     public static final int INIT_SCHRAUBEN = 10;
18
19     /* Globale Variablen */
20     /** Anzahl an Schrauben, die Robby trägt **/
21     private int anzahlSchrauben;
22     /** Anzahl an Akkus, die Robby trägt **/
23     private int anzahlAkkus;
24
25     /**
26      * Der Konstruktor initialisiert die Speicherwerte für Robby aus den
27      * statischen Konstanten.
28      */
29     public Robby() {
30         anzahlAkkus     = Robby.INIT_AKKUS;
31         anzahlSchrauben = Robby.INIT_SCHRAUBEN;
```

```
32      }
33
34      /**
35       * @return Aktuelle Anzahl an Schrauben
36       */
37      public int getAnzahlSchrauben() {
38          return this.anzahlSchrauben;
39      }
40
41      /**
42       * @return Aktuelle Anzahl an Akkus
43       */
44      public int getAnzahlAkkus() {
45          return this.anzahlAkkus;
46      }
47
48      /**
49       * In der Methode "act" koennen Befehle / andere Methoden angewendet werden:
50       * Die Methoden werden dort nacheinander "aufgerufen", wenn man
51       * nach dem Kompilieren / uebersetzen den Act-Knopf drueckt.
52       */
53      @Override
54      public void act() {
55
56      }
57
58      /**
59       * Robby soll hiermit einen Akku aufnehmen und im Inventar speichern. Vor
60       * der Akkuaufnahme wird auf dem Feld zunächst überprüft, ob sich hier ein
61       * Akku befindet. Wenn dies der Fall ist, aber auch noch weniger als 10
62       * Akkus im Inventar sind, nimmt Robby einen Akku auf und fügt dem Inventar
63       * einen hinzu und speichert dies. Hat er bereits die Maximalanzahl von 10
64       * Akkus im Inventar erreicht, meldet er dies und nimmt keinen weiteren Akku
65       * auf. Wenn sich andernfalls auch kein Akku auf dem Feld befindet, meldet
66       * er dies ebenfalls.
67       */
68      @Override
69      public void akkuAufnehmen() {
70          Akku aktAkku = (Akku)this.getOneObjectAtOffset(0, 0, Akku.class);
71          if(aktAkku != null) {
72              if(anzahlAkkus < Robby.MAX_AKKUS) {
73                  this.getWorld().removeObject(aktAkku);
74                  anzahlAkkus++;
75              }
76              else
77                  System.out.println("Ich kann keine Akkus mehr aufnehmen, da ich bereits zehn
                    ↪  Akkus habe!");
78          }
79          else
80              System.out.println("Hier ist kein Akku!");
81      }
82
83      /**
84       * Robby soll hiermit eine Schraube vom Inventar ablegen und dies
85       * abspeichern. Bevor Robby eine Schraube auf seinem Feld ablegt, prüft er
86       * aber ob er überhaupt noch mindestens eine Schraube im Inventar besitzt.
87       * Ist das der Fall, legt er eine Schraube ab und von der Anzahl der
88       * Schrauben im Inventar wird eine abgezogen und dies abgespeichert.
89       * Andernfalls meldet Robby, dass er keine Schrauben mehr hat.
90       */
91      @Override
92      public void schraubeAblegen() {
93          if(anzahlSchrauben > 0) {
94              this.getWorld().addObject(new Schraube(),
95                  this.getX(), this.getY() );
96
97              anzahlSchrauben--;
```

```
98          }
99          else
100             System.out.println("Ich besitze keine Schrauben mehr!");
101     }
102
103     /**
104      * Bewegt Robby um einen Schritt in die gewünschte, relative Richtung.
105      * @param direction Relative Laufrichtung
106      * @see Roboter#bewegen
107      */
108     public void bewegen(int direction) {
109         int newDirection = (this.getRotation() + direction) % 360;
110         if (newDirection != this.getRotation()) {
111             this.setRotation(newDirection);
112             Greenfoot.delay(1);
113         }
114         this.bewegen();
115     }
116
117     /**
118      * Umrundet aus Wänden bestehende Hindernisse ohne den Kontakt zu diesem
119      * zu verliehren und bleibt am Ausgangspunkt stehen. Zu Beginn muss Robby
120      * auf eine Wand blicken. Wenn Robby auf die Weltgrenze trifft, behandelt
121      * er diese als Teil des Hindernisses.
122      * @see #hindernisUmrunden(Runnable)
123      */
124     public void hindernisUmrunden() {
125         hindernisUmrunden(new Runnable() {
126             @Override
127             public void run() {
128             }
129         });
130     }
131
132     /**
133      * Umrundet aus Wänden bestehende Hindernisse ohne den Kontakt zu diesem
134      * zu verliehren und bleibt am Ausgangspunkt stehen. Zu Beginn muss Robby
135      * auf eine Wand blicken. Wenn Robby auf die Weltgrenze trifft, behandelt
136      * er diese als Teil des Hindernisses.
137      * @param action Runnable-Aktion, die auf jedem Feld ausgeführt wird
138      * @see #istObjektNebendran
139      * @see #istGrenzeNebendran
140      */
141     public void hindernisUmrunden(Runnable action) {
142         int startX = this.getX(),
143             startY = this.getY();
144
145         dreheLinks();
146
147         do {
148             for (int direction = 450; direction > 90; direction -= 90) {
149                 if ( !istObjektNebendran(direction, Wand.class)
150                   && !istGrenzeNebendran(direction) ) {
151                     action.run();
152                     bewegen(direction);
153                     break;
154                 }
155             }
156         } while( startX != this.getX() || startY != this.getY() );
157
158         dreheRechts();
159     }
160
161     /**
162      * @return Überprüft, ob sich in Laufrichtung des Roboters
163      * die Weltgrenze befindet
164      */
```

```
165     public boolean grenzeVorne() {
166         return this.istGrenzeNebendran(0);
167     }
168 
169     /**
170      * @return Überprüft, ob sich links der Laufrichtung des Roboters
171      * die Weltgrenze befindet
172      */
173     public boolean grenzeLinks() {
174         return this.istGrenzeNebendran(270);
175     }
176 
177     /**
178      * @return Überprüft, ob sich rechts der Laufrichtung des Roboters
179      * die Weltgrenze befindet
180      */
181     public boolean grenzeRechts() {
182         return this.istGrenzeNebendran(90);
183     }
184 
185     /**
186      * @return Überprüft, ob sich entgegen der Laufrichtung des Roboters
187      * die Weltgrenze befindet
188      */
189     public boolean grenzeHinten() {
190         return this.istGrenzeNebendran(180);
191     }
192 
193     /**
194      * @return Überprüft, ob sich in Laufrichtung des Roboters
195      * ein Akku befindet.
196      */
197     public boolean akkuVorne() {
198        return this.istObjektNebendran(0, Akku.class);
199     }
200 
201     /**
202      * @return Überprüft, ob sich rechts der Laufrichtung des Roboters
203      * ein Akku befindet.
204      */
205     public boolean akkuRechts() {
206         return this.istObjektNebendran(90, Akku.class);
207     }
208 
209     /**
210      * @return Überprüft, ob sich links der Laufrichtung des Roboters
211      * ein Akku befindet.
212      */
213     public boolean akkuLinks() {
214         return this.istObjektNebendran(-90, Akku.class);
215     }
216 
217     /**
218      * @return Überprüft, ob sich links der Laufrichtung des Roboters
219      * ein Akku befindet.
220      */
221     public boolean akkuHinten() {
222         return this.istObjektNebendran(180, Akku.class);
223     }
224 
225     /**
226      * @return Überprüft, ob sich entgegen der Laufrichtung des Roboters
227      * eine Wand befindet.
228      */
229     public boolean wandHinten() {
230         return this.istObjektNebendran(180, Wand.class);
231     }
```

```
232
233     /**
234      * Der Sensor überprüft, ob sich neben der Laufrichtung von Robby ein
235      * anderer Actor befindet.
236      * @param direction Winkel von der Laufrichtung zum Suchfeld
237      * @param cl Klasse des gesuchten Actors
238      * @return boolean Gibt an, ob das Objekt mit den angegeben Eigenschaften existiert
239      * @see #akkuVorne
240      * @see #akkuRechts
241      * @see #akkuLinks
242      * @see #akkuHinten
243      * @see #wandHinten
244      */
245     protected boolean istObjektNebendran(int direction, Class<?> cl) {
246        double angle = (this.getRotation() + direction) / 180.0 * Math.PI;
247
248        return (this.getOneObjectAtOffset(
249               (int)Math.cos(angle),
250               (int)Math.sin(angle), cl) != null);
251     }
252
253     /**
254      * Der Sensor überprüft, ob sich neben der Laufrichtung von Robby
255      * die Weltgrenze befindet.
256      * @param direction Winkel von der Laufrichtung zum Ende
257      * @return boolean Gibt an, ob die Weltgrenze neben Robby ist
258      * @see #grenzeVorne
259      * @see #grenzeRechts
260      * @see #grenzeLinks
261      * @see #grenzeHinten
262      */
263     protected boolean istGrenzeNebendran(int direction) {
264         direction = direction % 360;
265         return ( this.getX() + 1 >= this.getWorld().getWidth()
266                 && this.getRotation() == (360 - direction) % 360 )
267                 || ( this.getY() + 1 >= this.getWorld().getHeight()
268                 && this.getRotation() == (450 - direction) % 360 )
269                 || ( this.getX() <= 0
270                 && this.getRotation() == (540 - direction) % 360 )
271                 || ( this.getY() <= 0
272                 && this.getRotation() == (630 - direction) % 360 );
273     }
274 }
```

4.2 FeatureTest.java

```
1  import java.lang.reflect.*;
2
3  /**
4   * Basisklasse zum Testen einzelner Klassen in Greenfoot. Benötigt den
5   * Klassentyp. Implementierungen sollten eigene Subklassen verwenden.
6   * @author Adrian Schrader
7   * @version 1.5
8   * @param <T> Typ der getesteten Klasse
9   */
10 public class FeatureTest<T> {
11     /** Statusmeldung für bestandene Tests **/
12     public static final String MESSAGE_PASSED = "  Bestanden  ";
13     /** Statusmeldung für nicht bestandene Tests **/
14     public static final String MESSAGE_FAILED = "Durchgefallen";
15
16     /* Attribute und zu testende Objektinstanz */
17     private T object;
18     private Class<T> cl;
19     private String name;
20
21     private boolean failed;
```

```
22
23      /**
24       * Instanziiert die Klasse über den Typ der Testklasse. Die zu testende
25       * Klasseninstanz wird automatisch erstellt.
26       * @param cl Typ der Testklasse
27       */
28      public FeatureTest(Class<T> cl) {
29          this.cl = cl;
30          this.name = cl.getName();
31          this.failed = false;
32
33          try {
34              this.object = cl.newInstance();
35          }
36          catch (Exception ex) {
37              System.err.println("Es konnte keine Instanz von " + this.name
38                  + " erstellt werden. ");
39          }
40      }
41
42      /**
43       * Instanziiert die Klasse über ein bestehendes Objekt.
44       * @param obj Objekt vom zu testenden Typ
45       */
46      public FeatureTest(T obj) {
47          this.cl = (Class<T>)obj.getClass();
48          this.name = cl.getName();
49          this.failed = false;
50          this.object = obj;
51      }
52
53      /**
54       * @return Ertellte Instanz der zu testenden Klasse
55       */
56      public T getInstance() {
57          return this.object;
58      }
59
60      /**
61       * @return Name der Testklasse, der auch in den Statusmeldungen benutzt wird.
62       */
63      public String getName() {
64          return this.name;
65      }
66
67      /**
68       * @return Klasse der Testinstanz
69       */
70      public Class<?> getTestClass() {
71          return this.cl;
72      }
73
74      /**
75       * @return Erfolg der Testergebnisse
76       */
77      public boolean hasFailed() {
78          return this.failed;
79      }
80
81      /**
82       * @param failed Erfolg der Testergebnisse
83       */
84      protected void hasFailed(boolean failed) {
85          this.failed = failed;
86      }
87
88      /**
```

```
89          * Diese Funktion sollte von Unterklassen überschrieben werden, um alle
90          * Tests auszuführen und deren Ergebnisse zurückzugeben.
91          * @return Erfolg des Tests
92          */
93         public boolean testAllFeatures() {
94             return this.failed;
95         }
96
97         /**
98          * Gibt die Statusmeldungen für einzelne Tests aus
99          * @param message Nachricht für die Konsole/Log
100         * @param passed Erfolgreiche Meldung
101         * @see #MESSAGE_PASSED
102         * @see #MESSAGE_FAILED
103         */
104        protected void sendStatus(String message, boolean passed) {
105            System.out.println("[" + (passed ? FeatureTest.MESSAGE_PASSED :
                ↪  FeatureTest.MESSAGE_FAILED) + "] " + message);
106        }
107
108        /**
109         * Gibt das reflexive Feld aus.
110         * @param <F> Typ des Feldes
111         * @param name Name des Fields
112         * @param type Typ des Feldes
113         * @return Aktueller gecasteter Wert des Feldes
114         * @see testField
115         */
116        protected <F> F getField(String name, Class<F> type) {
117            try {
118                return (F)(cl.getField(name).get(this.object));
119            } catch (Exception ex) {
120                this.sendStatus(name, false);
121                failed = true;
122
123                System.err.println("Fehlerhaftes Testskript (testMethod): Feld in " + cl.getName() +
                    ↪  " konnte nicht geladen oder gecastet werden. (Nachricht: " + ex.getMessage() +
                    ↪  ")");
124                return null;
125            }
126        }
127
128        /**
129         * Überprüft, ob das reflexive Feld der Klasse einem Sollwert entspricht.
130         * @param name Name des Fields
131         * @param target Sollwert für das angegebene Feld
132         * @return Gibt an, ob der Sollwert mit dem Feldwert übereinstimmt
133         * @see testMethod
134         */
135        protected boolean testField(String name, Object target) {
136            try {
137                return cl.getField(name).get(this.object) == target;
138            } catch (Exception ex) {
139                this.sendStatus(name, false);
140                failed = true;
141
142                System.err.println("Fehlerhaftes Testskript (testMethod): Feld in " + cl.getName() +
                    ↪  " konnte nicht geladen werden. (Nachricht: " + ex.getMessage() + ")");
143                return false;
144            }
145        }
146
147        /**
148         * Überprüft, ob die reflektiv angegebene Methode den Zielwert zurückgibt.
149         * Diese Überladung geht davon aus, dass die Methode keine Argumente
150         * benötigt.
151         * @param name Name der Methode
```

```
152         * @param returnValue Der erwartete Rückgabewert
153         * @return Gibt an, ob Erwartungswert mit dem Rückgabewert übereinstimmt
154         * @see #testMethod
155         */
156        protected boolean testMethod(String name, Object returnValue)
157        {
158            return this.testMethod(name, returnValue, new Class<?>[]{}, new Object[] {});
159        }
160
161        /**
162         * Überprüft, ob die reflektiv angegebene Methode den Zielwert zurückgibt.
163         * @param name Name der Methode
164         * @param expectedValue Erwartungswert für den Rückgabewert
165         * @param parameterTypes Array der Typen der Parameter, mit denen die Methode gefunden werden
    ↪  kann
166         * @param parameters Array der benötigten Parameter der gesuchten Methode
167         * @return Gibt an, ob die Methode den Test bestanden hat
168         */
169        protected boolean testMethod(String name, Object expectedValue, Class<?>[] parameterTypes,
    ↪  Object[] parameters) {
170            try {
171                Method method = this.cl.getMethod(name, parameterTypes);
172                Object returnValue = method.invoke(this.object, parameters);
173
174                if (expectedValue != null) {
175                    if (!returnValue.equals(expectedValue))
176                    {
177                        failed = true;
178                        return false;
179                    }
180                }
181
182                return true;
183            }
184            catch (Exception ex)
185            {
186                this.sendStatus(name, false);
187                failed = true;
188
189                System.err.println("Fehlerhaftes Testskript (testMethod): Methode in " + cl.getName()
    ↪  + " konnte nicht geladen werden. ");
190                return false;
191            }
192        }
193
194        /**
195         * Gibt den Wert eines Getters ohne Parameter mit dem zugehörigen Namen zurück
196         * @param name Name der Methode
197         * @return Rückgabewert der Methode
198         */
199        protected Object getReturnValue(String name) {
200            return getReturnValue(name, new Class<?>[] {}, new Object[] {});
201        }
202
203        /**
204         * Führt die angegebene Methode reflexiv aus und gibt seinen Rückgabewert zurück
205         * @param name Name der Methode
206         * @param parameterTypes Array der Typen der Parameter, mit denen die Methode gefunden werden
    ↪  kann
207         * @param parameters Array der benötigten Parameter der gesuchten Methode
208         * @return Rückgabewert der Methode
209         */
210        protected Object getReturnValue(String name, Class<?>[] parameterTypes, Object[] parameters)
    ↪  {
211            try {
212                Method method = this.cl.getMethod(name, parameterTypes);
213                return method.invoke(this.object, parameters);
```

```
214         }
215         catch (Exception ex)
216         {
217             this.sendStatus(name, false);
218             failed = true;
219 
220             System.err.println("Fehlerhaftes Testskript (getReturnValue): Methode in " +
                ↪  cl.getName() + " konnte nicht geladen werden. ");
221             return false;
222         }
223     }
224 }
```

4.3 RobbyTest.java

```
1  import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)
2  
3  /**
4   * Diese Klasses testet die Funktionalität von Sensoren, Speicher und
5   * Bewegungsapparat der Klasse Robby nach dem Programmstart.
6   * @author Adrian Schrader
7   * @version 1.5
8   */
9  public class RobbyTest extends FeatureTest<Robby>
10 {
11     private final RoboterWelt world;
12 
13     /**
14      * Instanziiert die Testklasse und lässt neuen Robby erstellen
15      * @param world Spielwelt, in der die Tests durchgeführt werden sollen.
16      */
17     public RobbyTest(RoboterWelt world)
18     {
19         super(Robby.class);
20         this.world = world;
21     }
22 
23     /**
24      * Startet alle Tests für die Klasse Robby
25      * @return Erfolg der Tests
26      */
27     @Override
28     public boolean testAllFeatures() {
29         System.out.println("Testprogramm für Klasse Robby einleiten...");
30 
31         try {
32             this.world.addObject(this.getInstance(), 0, 0);
33 
34             boolean success = this.testSensors() && this.testMemory() && testMovement();
35 
36             this.world.removeObject(this.getInstance());
37             return success;
38         } catch (Exception ex) {
39             System.err.println("Innerhalb der Testroutine ist ein Fehler aufgetreten. Sind alle
                ↪  Funktionen aufrufbar?");
40             return false;
41         }
42     }
43 
44     /**
45      * Testet Robbys Sensoren zum Aufspüren von Wänden und Akkus in einer Entfernung von einem
    ↪  Feld.
46      * @return Erfolg des Tests
47      * @throws InstantiationException
48      * @throws IllegalAccessException
49      */
50     public boolean testSensors() throws InstantiationException, IllegalAccessException {
```

```
51          System.out.println("Sensorfunktionalität wird getetstet...");
52          this.hasFailed(false);
53
54          // Füge Robby der Spielwelt hinzu
55          this.getInstance().setLocation(1, 1);
56
57          // Überprüfe die Sensorfunktionen für Akkus
58          this.testRotationalObjectDetection("Akku-Sensoren", new String[] { "akkuVorne",
            ↪  "akkuLinks", "akkuRechts", "akkuHinten" }, Akku.class);
59
60          // Überprüfe die Sensorfunktionen für Wände
61          this.testRotationalObjectDetection("Wand-Sensoren", new String[] { "wandVorne",
            ↪  "wandLinks", "wandRechts", "wandHinten" }, Wand.class);
62
63          return !this.hasFailed();
64      }
65
66      /**
67       * Testet Robbys Fähigkeit Schrauben abzulegen, Akkus aufzunehmen und
68       * dabei seine Statusanzeigen zu aktualisieren.
69       * @return Erfolg des Tests
70       */
71      public boolean testMemory() {
72          boolean success = true;
73          System.out.println("Speicherfunktionalität wird getetstet...");
74
75          success &= testObjectAquisition("akkuAufnehmen", "getAnzahlAkkus", Robby.MAX_AKKUS,
            ↪  Akku.class);
76          this.sendStatus("Aufnahme und Begrenzung von Akkus", success);
77
78          success &= testObjectDeposition("schraubeAblegen", "getAnzahlSchrauben", 0,
            ↪  Schraube.class);
79          this.sendStatus("Ablage und Begrenzung von Schrauben", success);
80
81          return success;
82      }
83
84      /**
85       * Testet Robbys Fähigkeit ein geschlossenes Hindernis zu Umrunden,
86       * dabei anpassbare Aktionen auszuführen und zum Ausgangspunkt
87       * zurückzukehren.
88       * @return Erfolg des Tests
89       */
90      public boolean testMovement() {
91          System.out.println("Bewegungsfunktionalität wird getetstet...");
92          this.getInstance().setLocation(0, 1);
93          this.getInstance().setRotation(0);
94
95          this.world.addObject(new Wand(), 1, 1);
96          this.world.addObject(new Wand(), 2, 2);
97          this.world.addObject(new Wand(), 1, 3);
98          this.world.addObject(new Wand(), 3, 1);
99
100         class MovementCheck implements Runnable {
101             boolean success = true;
102
103             @Override
104             public void run() {
105                 boolean isObstacleNearby = false;
106                 for (int x = -1; x < 2; x++) {
107                     for (int y = -1; y < 2; y++) {
108                         isObstacleNearby |= !getInstance().getWorld().getObjectsAt(
109                             getInstance().getX() + x,
110                             getInstance().getY() + y,
111                             Wand.class).isEmpty();
112
113                     }
```

```
                }
                if (!isObstacleNearby)
                    sendStatus("Robby ist in [" + getInstance().getX() + "," +
                     ↪  getInstance().getY() + "] vom Weg abgekommen", false);
                success &= isObstacleNearby;
            }
        }

        MovementCheck testRun = new MovementCheck();

        this.testMethod("hindernisUmrunden", null, new Class<?>[] { Runnable.class }, new
         ↪  Runnable[] { testRun });
        this.world.removeObjects(this.world.getObjectsAt(1, 1, Wand.class));
        this.world.removeObjects(this.world.getObjectsAt(2, 2, Wand.class));
        this.world.removeObjects(this.world.getObjectsAt(1, 3, Wand.class));
        this.world.removeObjects(this.world.getObjectsAt(3, 1, Wand.class));

        if (this.getInstance().getX() != 0 || this.getInstance().getY() != 1) {
            this.sendStatus("Robby ist bei der Umrundung nicht wieder am Ausgangspunkt
             ↪  angekommen", false);
            testRun.success = false;
        }

        this.sendStatus("Hindernis umrunden", testRun.success);
        return testRun.success;
    }

    /**
     * Testet, ob Robby Objekte aus seinem Speicher in die Welt platzieren kann und dabei Grenzen
 ↪  einhält.
     * @param method Methode, die ein Objekt ablegen soll
     * @param field Feld, dass dabei vermindert wird
     * @param min Minimalwert für den Speicher (danach kann kein Objekt mehr platziert werden)
     * @param cl Klasse des zu platzierenden Objekts
     * @return Erfolg des Tests
     * @see #testObjectAquisition
     */
    protected boolean testObjectDeposition(String method, String field, int min, Class<? extends
     ↪  Actor> cl) {
        this.getInstance().setLocation(0, 0);
        int max = (Integer)this.getReturnValue(field);
         for (int x = max; x > min - 1; x--) {
            this.testMethod(method, null);
            if ((Integer)this.getReturnValue(field) < min) {
                return false;
            }
        }
        this.world.removeObjects(this.world.getObjectsAt(0, 0, cl));
        return true;
    }

    /**
     * Testet, ob Robby Objekte aus der Welt in seinen Speicher laden kann und dabei Grenzen
 ↪  einhält.
     * @param method Methode, die ein Objekt aufnehmen soll
     * @param field Getter für einen Integer, der den erhöhten Wert zurückgeben soll
     * @param max Maximalwert für den Speicher (danach kann kein Objekt mehr aufgenommen werden)
     * @param cl Klasse des aufzunehmenden Objekts
     * @return Erfolg des Tests
     * @see #testObjectDeposition
     */
    protected boolean testObjectAquisition(String method, String field, int max, Class<? extends
     ↪  Actor> cl) {
        this.getInstance().setRotation(0);

        int startValue = (Integer)this.getReturnValue(field);

```

```
174         Akku[] akkus = new Akku[max + 1];
175         for (int x = 0; x < max + 1; x++) {
176             akkus[x] = new Akku();
177             this.world.addObject(akkus[x], x, 0);
178             this.getInstance().setLocation(x, 0);
179 
180             this.testMethod(method, null);
181 
182             int newValue = (Integer)this.getReturnValue(field);
183             if (newValue < 0 || newValue > max) {
184                 this.sendStatus("Feld " + field + " blieb nicht im Bereich [ 0," + max + " ]",
                    ↪  false);
185                 return false;
186             }
187 
188             if (x < max) {
189                 if (newValue != startValue + (x + 1)
190                  || !this.world.getObjectsAt(x, 0, cl).isEmpty()) {
191                     this.sendStatus("Feld " + field + " zählt nach Aufnehmen eines Akkus nicht
                        ↪  hoch oder sammelt ihn gar nicht erst ein. ", false);
192                     return false;
193                 }
194             } else {
195                 this.world.removeObject(akkus[x]);
196             }
197         }
198 
199         return true;
200     }
201 
202     /**
203      * Testet den Nachweis eines Objekts auf relativer Position zum Aktor.
204      * @param title Bezeichnung für den Test
205      * @param methods String-Array aus Methodennamen für die einzelnen Positionen (vorne, links,
    ↪  rechts, hinten)
206      * @param cl Klasse des nachzuweisenden Aktors
207      * @return Erfolg des Tests
208      * @see #testObjectDetection
209      */
210     protected boolean testRotationalObjectDetection(String title, String[] methods, Class<?
        ↪  extends Actor> cl) {
211         if (methods.length < 4) {
212             System.err.println("Fehlerhaftes Testskript (testRotationalObjectDetection): Benötigt
                ↪  4 Methodennamen. ");
213         }
214         try {
215             boolean test0 = true;
216             this.getInstance().setRotation(0);
217             test0 &= this.testObjectDetection(2, 1, methods[0], "0ř", cl);
218             test0 &= this.testObjectDetection(1, 0, methods[1], "0ř", cl);
219             test0 &= this.testObjectDetection(1, 2, methods[2], "0ř", cl);
220             test0 &= this.testObjectDetection(0, 1, methods[3], "0ř", cl);
221 
222             boolean test90 = true;
223             this.getInstance().setRotation(90);
224             test90 &= this.testObjectDetection(1, 2, methods[0], "90ř", cl);
225             test90 &= this.testObjectDetection(2, 1, methods[1], "90ř", cl);
226             test90 &= this.testObjectDetection(0, 1, methods[2], "90ř", cl);
227             test90 &= this.testObjectDetection(1, 0, methods[3], "90ř", cl);
228 
229             boolean test180 = true;
230             this.getInstance().setRotation(180);
231             test180 &= this.testObjectDetection(0, 1, methods[0], "180ř", cl);
232             test180 &= this.testObjectDetection(1, 2, methods[1], "180ř", cl);
233             test180 &= this.testObjectDetection(1, 0, methods[2], "180ř", cl);
234             test180 &= this.testObjectDetection(2, 1, methods[3], "180ř", cl);
235 
```

```
236             boolean test270 = true;
237             this.getInstance().setRotation(270);
238             test270 &= this.testObjectDetection(1, 0, methods[0], "270ř", cl);
239             test270 &= this.testObjectDetection(0, 1, methods[1], "270ř", cl);
240             test270 &= this.testObjectDetection(2, 1, methods[2], "270ř", cl);
241             test270 &= this.testObjectDetection(1, 2, methods[3], "270ř", cl);
242 
243             this.sendStatus("Test von " + title + " für 0ř", test0);
244             this.sendStatus("Test von " + title + " für 90ř", test90);
245             this.sendStatus("Test von " + title + " für 180ř", test180);
246             this.sendStatus("Test von " + title + " für 270ř", test270);
247 
248             return ( test0 && test90 && test180 && test270 );
249         } catch (Exception ex) {
250             System.err.println("Fehlerhaftes Testskript (testRotationalObjectDetection): Kann
                ↪  Methodennamen nicht auflösen (" + ex.getMessage() + ")");
251             return false;
252         }
253     }
254 
255     /**
256      * Gibt an, ob die Methode einen anderen Actor positiv und negativ
257      * nachweisen kann. Wirft evtl. Fehler beim Instanziieren des Testobjekts.
258      * @param x Horizontale Koordinate für das Objekt
259      * @param y Vertikale Koordinate für das Objekt
260      * @param method Name der Methode in der Klasse Robby
261      * @param test Name des durchgeführten Tests
262      * @param cl Klasse des gesuchten Actors
263      * @return Erfolg des Tests
264      * @throws java.lang.InstantiationException
265      * @throws java.lang.IllegalAccessException
266      * @see FeatureTest#testMethod
267      */
268     protected boolean testObjectDetection(int x, int y, String method, String test, Class<?
      ↪  extends Actor> cl)
269             throws InstantiationException, IllegalAccessException {
270         // Sicherstellen, dass das Objekt nicht schon in der Spielwelt existiert
271         this.world.removeObjects(this.world.getObjectsAt(x, y, cl));
272 
273         // Testen, ob die Methode keine false-positives zurückgibt
274         boolean negative = this.testMethod(method, false);
275 
276         // Platzieren der neuen Objektinstanz in der Welt
277         this.world.addObject(cl.newInstance(), x, y);
278 
279         // Testen, ob die Methode keine false-negatives zurückgibt
280         boolean positive = this.testMethod(method, true);
281 
282         // Spielwelt für die nächsten Tests aufräumen
283         this.world.removeObjects(this.world.getObjectsAt(x, y, cl));
284 
285         if (!positive) {
286             this.sendStatus(method + "() konnte im Test " + test + " den Actor " + cl.getName() +
                ↪  "nicht positiv erkennen. ", false);
287         }
288         if (!negative) {
289             this.sendStatus(method + "() konnte im Test " + test + " den Actor " + cl.getName() +
                ↪  " nicht negativ erkennen. ", false);
290         }
291 
292         return (positive && negative);
293     }
294 }
```

4.4 RoboterWelt.java

```java
import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Die einzigen aktiven Akteure in der Roboterwelt sind die Roboter.
 * Die Welt besteht aus 14 * 10 Feldern.
 */

public class RoboterWelt extends World
{
    private static int zellenGroesse = 50;

    /**
     * Erschaffe eine Welt mit 14 * 10 Zellen.
     */
    public RoboterWelt()
    {
        super(14, 10, zellenGroesse);
        setBackground("images/Bodenplatte.png");
        setPaintOrder(Roboter.class, Schraube.class, Akku.class, Wand.class);
        Greenfoot.setSpeed(15);

        RobbyTest robbyTest = new RobbyTest(this);
        if (!robbyTest.testAllFeatures()) {
            System.err.println("Die Klasse Robby hat nicht alle Tests bestanden. Bitte überprüfen
            ↪  sie den Log, um das Problem näher einzugrenzen. ");
        } else {
            System.out.println("Die Klasse Robby hat alle Tests bestanden. Bravo!");
        }
    }
}
```