Robby Projekt

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

Adrian Schrader Moritz Jung Alexander Riecke

16. 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

Lösung

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.

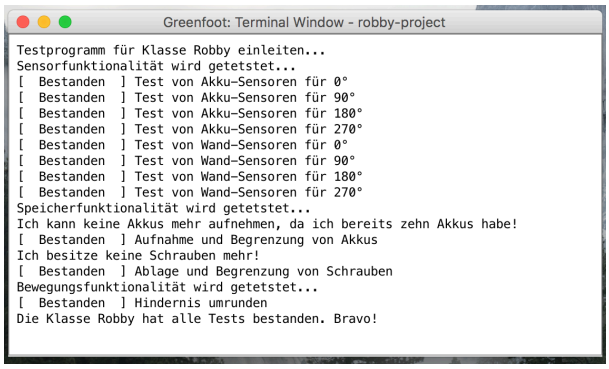

Abbildung 2.1: Erfolgreicher Testlog nach dem Kompilieren der Klassen.

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, 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.

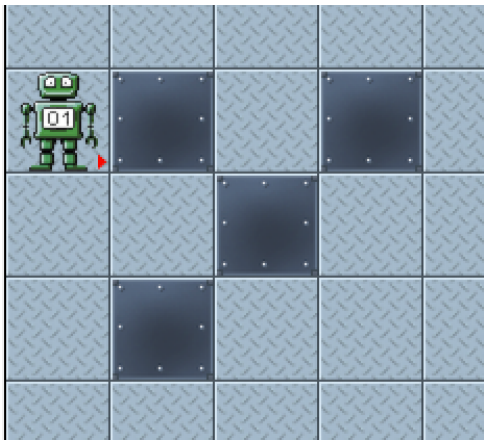

Abbildung 2.2: Das von der Testfunktion erstellte Hindernis.

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.

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

Sourcecode-Verwaltung

Um den Überblick über den aktuellen Stand zu behalten und die Sourcecodeversionen koordinert 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.

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 1.4
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     {
31         anzahlAkkus     = Robby.INIT_AKKUS;
32         anzahlSchrauben = Robby.INIT_SCHRAUBEN;
33     }
34
35     /**
36      * Gibt die aktuelle Anzahl an Schrauben zurück.
37      */
38     public int getAnzahlSchrauben() {
39         return this.anzahlSchrauben;
40     }
41
42     public int getAnzahlAkkus() {
43         return this.anzahlAkkus;
44     }
45
46     /**
47      * In der Methode "act" koennen Befehle / andere Methoden angewendet werden:
48      * Die Methoden werden dort nacheinander "aufgerufen", wenn man
49      * nach dem Kompilieren / uebersetzen den Act-Knopf drueckt.
50      */
51     @Override
52     public void act()
53     {
54
55     }
56
57     /**
58      * Robby soll hiermit einen Akku aufnehmen und im Inventar speichern. Vor
59      * der Akkuaufnahme wird auf dem Feld zunächst überprüft, ob sich hier ein
```

```
     * Akku befindet. Wenn dies der Fall ist, aber auch noch weniger als 10
     * Akkus im Inventar sind, nimmt Robby einen Akku auf und fügt dem Inventar
     * einen hinzu und speichert dies. Hat er bereits die Maximalanzahl von 10
     * Akkus im Inventar erreicht, meldet er dies und nimmt keinen weiteren Akku
     * auf. Wenn sich andernfalls auch kein Akku auf dem Feld befindet, meldet
     * er dies ebenfalls.
     */
    @Override
    public void akkuAufnehmen() {
        Akku aktAkku = (Akku)this.getOneObjectAtOffset(0, 0, Akku.class);
        if(aktAkku != null) {
            if(anzahlAkkus < Robby.MAX_AKKUS) {
                this.getWorld().removeObject(aktAkku);
                anzahlAkkus++;
            }
            else
                System.out.println("Ich kann keine Akkus mehr aufnehmen, da ich bereits zehn
                ↪  Akkus habe!");
        }
        else
            System.out.println("Hier ist kein Akku!");
    }

    /**
     * Robby soll hiermit eine Schraube vom Inventar ablegen und dies
     * abspeichern. Bevor Robby eine Schraube auf seinem Feld ablegt, prüft er
     * aber ob er überhaupt noch mindestens eine Schraube im Inventar besitzt.
     * Ist das der Fall, legt er eine Schraube ab und von der Anzahl der
     * Schrauben im Inventar wird eine abgezogen und dies abgespeichert.
     * Andernfalls meldet Robby, dass er keine Schrauben mehr hat.
     */
    @Override
    public void schraubeAblegen() {
        if(anzahlSchrauben > 0) {
            this.getWorld().addObject(new Schraube(),
                this.getX(), this.getY() );

            anzahlSchrauben--;
        }
        else
            System.out.println("Ich besitze keine Schrauben mehr!");
    }

    /**
     * Bewegt Robby um einen Schritt in die gewünschte, relative Richtung.
     * @param direction Relative Laufrichtung
     * @see Roboter#bewegen
     */
    public void bewegen(int direction) {
        int newDirection = (this.getRotation() + direction) % 360;
        if (newDirection != this.getRotation()) {
            this.setRotation(newDirection);
            Greenfoot.delay(1);
        }
        this.bewegen();
    }

    /**
     * Umrundet aus Wänden bestehende Hindernisse ohne den Kontakt zu diesem
     * zu verliehren und bleibt am Ausgangspunkt stehen. Zu Beginn muss Robby
     * auf eine Wand blicken. Wenn Robby auf die Weltgrenze trifft, behandelt
     * er diese als Teil des Hindernisses.
     * @see #hindernisUmrunden
     */
    public void hindernisUmrunden() {
        hindernisUmrunden(new Runnable() {
            @Override
```

```
126             public void run() {
127             }
128         });
129     }
130
131     /**
132      * Umrundet aus Wänden bestehende Hindernisse ohne den Kontakt zu diesem
133      * zu verliehren und bleibt am Ausgangspunkt stehen. Zu Beginn muss Robby
134      * auf eine Wand blicken. Wenn Robby auf die Weltgrenze trifft, behandelt
135      * er diese als Teil des Hindernisses.
136      * @param action Runnable-Aktion, die auf jedem Feld ausgeführt wird
137      * @see #istObjektNebendran
138      * @see #istGrenzeNebendran
139      */
140     public void hindernisUmrunden(Runnable action) {
141         int startX = this.getX(),
142             startY = this.getY();
143
144         dreheLinks();
145
146         do {
147             for (int direction = 450; direction > 90; direction -= 90) {
148                 if ( !istObjektNebendran(direction, Wand.class)
149                   && !istGrenzeNebendran(direction) ) {
150                     action.run();
151                     bewegen(direction);
152                     break;
153                 }
154             }
155         } while( startX != this.getX() || startY != this.getY() );
156
157         dreheRechts();
158     }
159
160     /**
161      * Der Sensor ueberprüft, ob sich in Laufrichtung des Roboters
162      * die Weltgrenze befindet
163      */
164     public boolean grenzeVorne() {
165         return this.istGrenzeNebendran(0);
166     }
167
168     /**
169      * Der Sensor ueberprüft, ob sich links der Laufrichtung des Roboters
170      * die Weltgrenze befindet
171      */
172     public boolean grenzeLinks() {
173         return this.istGrenzeNebendran(270);
174     }
175
176     /**
177      * Der Sensor ueberprüft, ob sich rechts der Laufrichtung des Roboters
178      * die Weltgrenze befindet
179      */
180     public boolean grenzeRechts() {
181         return this.istGrenzeNebendran(90);
182     }
183
184     /**
185      * Der Sensor ueberprüft, ob sich entgegen der Laufrichtung des Roboters
186      * die Weltgrenze befindet
187      */
188     public boolean grenzeHinten() {
189         return this.istGrenzeNebendran(180);
190     }
191
192     /**
```

```
193         * Der Sensor ueberprueft, ob sich in Laufrichtung des Roboters
194         * ein Akku befindet.
195         */
196        public boolean akkuVorne()
197        {
198           return this.istObjektNebendran(0, Akku.class);
199        }
200
201        /**
202         * Der Sensor ueberprueft, ob sich rechts der Laufrichtung des Roboters
203         * ein Akku befindet.
204         */
205        public boolean akkuRechts()
206        {
207            return this.istObjektNebendran(90, Akku.class);
208        }
209
210        /**
211         * Der Sensor ueberprueft, ob sich links der Laufrichtung des Roboters
212         * ein Akku befindet.
213         */
214        public boolean akkuLinks()
215        {
216            return this.istObjektNebendran(-90, Akku.class);
217        }
218
219        /**
220         * Der Sensor ueberprueft, ob sich links der Laufrichtung des Roboters
221         * ein Akku befindet.
222         */
223        public boolean akkuHinten()
224        {
225            return this.istObjektNebendran(180, Akku.class);
226        }
227
228        /**
229         * Der Sensor ueberprueft, ob sich entgegen der Laufrichtung des Roboters
230         * eine Wand befindet.
231         */
232        public boolean wandHinten()
233        {
234            return this.istObjektNebendran(180, Wand.class);
235        }
236
237        /**
238         * Der Sensor überprüft, ob sich neben der Laufrichtung von Robby ein
239         * anderer Actor befindet.
240         * @param direction Winkel von der Laufrichtung zum Suchfeld
241         * @param cl Klasse des gesuchten Actors
242         * @return boolean Gibt an, ob das Objekt mit den angegeben Eigenschaften existiert
243         * @see #akkuVorne
244         * @see #akkuRechts
245         * @see #akkuLinks
246         * @see #akkuHinten
247         * @see #wandHinten
248         */
249        protected boolean istObjektNebendran(int direction, Class<?> cl)
250        {
251           double angle = (this.getRotation() + direction) / 180.0 * Math.PI;
252
253           return (this.getOneObjectAtOffset(
254                   (int)Math.cos(angle),
255                   (int)Math.sin(angle), cl) != null);
256        }
257
258
259        /**
```

```
260         * Der Sensor überprüft, ob sich neben der Laufrichtung von Robby
261         * die Weltgrenze befindet.
262         * @param direction Winkel von der Laufrichtung zum Ende
263         * @return boolean Gibt an, ob die Weltgrenze neben Robby ist
264         * @see #grenzeVorne
265         * @see #grenzeRechts
266         * @see #grenzeLinks
267         * @see #grenzeHinten
268         */
269        protected boolean istGrenzeNebendran(int direction) {
270            direction = direction % 360;
271            if (( this.getX() + 1 >= this.getWorld().getWidth()
272                    && this.getRotation() == (360 - direction) % 360 )
273             || ( this.getY() + 1 >= this.getWorld().getHeight()
274                    && this.getRotation() == (450 - direction) % 360 )
275             || ( this.getX() <= 0
276                    && this.getRotation() == (540 - direction) % 360 )
277             || ( this.getY() <= 0
278                    && this.getRotation() == (630 - direction) % 360 ))
279                return true;
280            return false;
281        }
282
283    }
```

4.2 FeatureTest.java

```
1   import java.util.*;
2   import java.lang.reflect.*;
3
4   /**
5    * Basisklasse zum Testen einzelner Klassen in Greenfoot. Benötigt den Klassentyp.
    ↪  Implementierungen sollten eigene Subklassen verwenden.
6    * @author Adrian Schrader
7    * @version 1.0.0
8    */
9   public class FeatureTest<T> {
10      /* Statischer Text für Statusmeldungen */
11      public static final String MESSAGE_PASSED = "  Bestanden  ";
12      public static final String MESSAGE_FAILED = "Durchgefallen";
13
14      /* Attribute und zu testende Objektinstanz */
15      protected T object;
16      protected Class<T> cl;
17      protected String name;
18      protected boolean failed;
19
20      /**
21       * Instanziiert die Klasse über den Typ der Testklasse. Die zu testende
22       * Klasseninstanz wird automatisch erstellt.
23       * @param cl Typ der Testklasse
24       */
25      public FeatureTest(Class<T> cl) {
26          this.cl = cl;
27          this.name = cl.getName();
28          this.failed = false;
29
30          try {
31              this.object = cl.newInstance();
32          }
33          catch (Exception ex) {
34              System.err.println("Es konnte keine Instanz von " + this.name
35                  + " erstellt werden. ");
36          }
37      }
38
39      /**
```

```
40          * Instanziiert die Klasse über ein bestehendes Objekt.
41          * @param obj Objekt vom zu testenden Typ
42          */
43         public FeatureTest(T obj) {
44             this.cl = (Class<T>)obj.getClass();
45             this.name = cl.getName();
46             this.failed = false;
47             this.object = obj;
48         }
49
50         /**
51          * Gibt die erstellte Testinstanz der zu testenden Klasse zurück.
52          */
53         public T getInstance() {
54             return this.object;
55         }
56
57         /**
58          * Gibt den Namen der Testklasse zurück, der auch in den Statusmeldungen
59          * benutzt wird.
60          */
61         public String getName() {
62             return this.name;
63         }
64
65         /**
66          * Diese Funktion sollte von Unterklassen überschrieben werden, um alle
67          * Tests auszuführen und deren Ergebnisse zurückzugeben.
68          * @param welt Weltinstanz, in der die Tests ausgeführt werden sollen
69          */
70         public boolean testAllFeatures() {
71             return this.failed;
72         }
73
74         /**
75          * Gibt die Statusmeldungen für einzelne Tests aus
76          */
77         protected void sendStatus(String message, boolean passed) {
78             System.out.println("[" + (passed ? this.MESSAGE_PASSED : this.MESSAGE_FAILED) + "] " +
                ↪  message);
79         }
80
81         /**
82          * Gibt das reflexive Feld aus.
83          * @param name Name des Fields
84          * @param type Typ des Feldes
85          * @see testField
86          */
87         protected <F> F getField(String name, Class<F> type) {
88             try {
89                 return (F)(cl.getField(name).get(this.object));
90             } catch (Exception ex) {
91                 this.sendStatus(name, false);
92                 failed = true;
93
94                 System.err.println("Fehlerhaftes Testskript (testMethod): Feld in " + cl.getName() +
                    ↪  " konnte nicht geladen oder gecastet werden. (Nachricht: " + ex.getMessage() +
                    ↪  ")");
95                 return null;
96             }
97         }
98
99         /**
100          * Überprüft, ob das reflexive Feld der Klasse einem Sollwert entspricht.
101          * @param name Name des Fields
102          * @param target Sollwert für das angegebene Feld
103          * @return Gibt an, ob der Sollwert mit dem Feldwert übereinstimmt
```

```
104         * @see testMethod
105         */
106        protected boolean testField(String name, Object target) {
107            try {
108                return cl.getField(name).get(this.object) == target;
109            } catch (Exception ex) {
110                this.sendStatus(name, false);
111                failed = true;
112
113                System.err.println("Fehlerhaftes Testskript (testMethod): Feld in " + cl.getName() +
                       " konnte nicht geladen werden. (Nachricht: " + ex.getMessage() + ")");
114                return false;
115            }
116        }
117
118        /**
119         * Überprüft, ob die reflektiv angegebene Methode den Zielwert zurückgibt.
120         * Diese Überladung geht davon aus, dass die Methode keine Argumente
121         * benötigt.
122         * @param name Name der Methode
123         * @param returnValue Der erwartete Rückgabewert
124         * @return Gibt an, ob Erwartungswert mit dem Rückgabewert übereinstimmt
125         * @see #testMethod
126         */
127        protected boolean testMethod(String name, Object returnValue)
128        {
129            return this.testMethod(name, returnValue, new Class<?>[]{}, new Object[] {});
130        }
131
132        /**
133         * Überprüft, ob die reflektiv angegebene Methode den Zielwert zurückgibt.
134         * @param name Name der Methode
135         * @param expectedValue Erwartungswert für den Rückgabewert
136         *ä@param parameterTypes Array der Typen der Parameter, mit denen die Methode gefunden werden
    kann
137         * @param parameters Array der benötigten Parameter der gesuchten Methode
138         * @return Gibt an, ob die Methode den Test bestanden hat
139         */
140        protected boolean testMethod(String name, Object expectedValue, Class<?>[] parameterTypes,
               Object[] parameters) {
141
142            try {
143                Method method = this.cl.getMethod(name, parameterTypes);
144                Object returnValue = method.invoke(this.object, parameters);
145
146                if (expectedValue != null) {
147                    if (!returnValue.equals(expectedValue))
148                    {
149                        failed = true;
150                        return false;
151                    }
152                }
153
154                return true;
155            }
156            catch (Exception ex)
157            {
158                this.sendStatus(name, false);
159                failed = true;
160
161                System.err.println("Fehlerhaftes Testskript (testMethod): Methode in " + cl.getName()
                       + " konnte nicht geladen werden. ");
162                ex.printStackTrace();
163                return false;
164            }
165        }
166
```

```
167         /**
168          * Gibt einen Getter mit dem zugehörigen Namen zurück
169          * @param name Name der Methode
170          */
171         protected Object getReturnValue(String name) {
172             return getReturnValue(name, new Class<?>[] {}, new Object[] {});
173         }
174 
175         /**
176          * Gibt den Rückgabewert der angegeben Methode zurück
177          * @param name Name der Methode
178          */
179         protected Object getReturnValue(String name, Class<?>[] parameterTypes, Object[] parameters)
             ↪  {
180             try {
181                 Method method = this.cl.getMethod(name, parameterTypes);
182                 return method.invoke(this.object, parameters);
183             }
184             catch (Exception ex)
185             {
186                 this.sendStatus(name, false);
187                 failed = true;
188 
189                 System.err.println("Fehlerhaftes Testskript (getReturnValue): Methode in " +
                     ↪  cl.getName() + " konnte nicht geladen werden. ");
190                 ex.printStackTrace();
191                 return false;
192             }
193         }
194     }
```

4.3 RobbyTest.java

```
1   import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)
2   import java.util.*;
3   
4   /**
5    * Diese Klasses testet die Funktionalität von Sensoren, Speicher und
6    * Bewegungsapparat der Klasse Robby nach dem Programmstart.
7    * @author Adrian Schrader
8    * @version 1.5
9    */
10  public class RobbyTest extends FeatureTest<Robby>
11  {
12      private RoboterWelt world;
13  
14      /**
15       * Instanziiert die Klasse in der angegeben Welt.
16       */
17      public RobbyTest(RoboterWelt world)
18      {
19          super(Robby.class);
20          this.world = world;
21      }
22  
23      /**
24       * Gibt an, ob alle Funktionen von Robby einwandfrei funktionieren.
25       */
26      @Override
27      public boolean testAllFeatures() {
28          System.out.println("Testprogramm für Klasse Robby einleiten...");
29  
30          try {
31              this.world.addObject(this.object, 0, 0);
32  
33              boolean success = this.testSensors() && this.testMemory() && testMovement();
34  
```

```
35              this.world.removeObject(this.object);
36              return success;
37          } catch (Exception ex) {
38              System.err.println("Innerhalb der Testroutine ist ein Fehler aufgetreten. Sind alle
                ↪  Funktionen aufrufbar?");
39              ex.printStackTrace();
40              return false;
41          }
42      }
43
44      /**
45       * Testet Robbys Sensoren zum Aufspüren von Wänden und Akkus in einer Entfernung von einem
    ↪   Feld.
46       * @throws InstantiationException
47       * @throws IllegalAccessException
48       */
49      public boolean testSensors() throws InstantiationException, IllegalAccessException {
50          System.out.println("Sensorfunktionalität wird getetstet...");
51          this.failed = false;
52
53          // Füge Robby der Spielwelt hinzu
54          this.object.setLocation(1, 1);
55
56          // Überprüfe die Sensorfunktionen für Akkus
57          this.testRotationalObjectDetection("Akku-Sensoren", new String[] { "akkuVorne",
                ↪  "akkuLinks", "akkuRechts", "akkuHinten" }, Akku.class);
58
59          // Überprüfe die Sensorfunktionen für Wände
60          this.testRotationalObjectDetection("Wand-Sensoren", new String[] { "wandVorne",
                ↪  "wandLinks", "wandRechts", "wandHinten" }, Wand.class);
61
62          return !this.failed;
63      }
64
65      /**
66       * Testet Robbys Fähigkeit Schrauben abzulegen, Akkus aufzunehmen und
67       * dabei seine Statusanzeigen zu aktualisieren.
68       */
69      public boolean testMemory() {
70          boolean success = true;
71          System.out.println("Speicherfunktionalität wird getetstet...");
72
73          success &= testObjectAquisition("akkuAufnehmen", "getAnzahlAkkus", Robby.MAX_AKKUS,
                ↪  Akku.class);
74          this.sendStatus("Aufnahme und Begrenzung von Akkus", success);
75
76          success &= testObjectDeposition("schraubeAblegen", "getAnzahlSchrauben", 0,
                ↪  Schraube.class);
77          this.sendStatus("Ablage und Begrenzung von Schrauben", success);
78
79          return success;
80      }
81
82      /**
83       * Testet Robbys Fähigkeit ein geschlossenes Hindernis zu Umrunden,
84       * dabei anpassbare Aktionen auszuführen und zum Ausgangspunkt
85       * zurückzukehren.
86       */
87      public boolean testMovement() {
88          System.out.println("Bewegungsfunktionalität wird getetstet...");
89          this.object.setLocation(0, 1);
90          this.object.setRotation(0);
91
92          this.world.addObject(new Wand(), 1, 1);
93          this.world.addObject(new Wand(), 2, 2);
94          this.world.addObject(new Wand(), 1, 3);
95          this.world.addObject(new Wand(), 3, 1);
```

```
96
97          class MovementCheck implements Runnable {
98              boolean success = true;
99
100             @Override
101             public void run() {
102                 boolean isObstacleNearby = false;
103                 for (int x = -1; x < 2; x++) {
104                     for (int y = -1; y < 2; y++) {
105                         isObstacleNearby |= !getInstance().getWorld().getObjectsAt(
106                             getInstance().getX() + x,
107                             getInstance().getY() + y,
108                             Wand.class).isEmpty();
109
110                     }
111                 }
112                 if (!isObstacleNearby)
113                     sendStatus("Robby ist in [" + getInstance().getX() + "," +
                        ↪  getInstance().getY() + "] vom Weg abgekommen", false);
114                 success &= isObstacleNearby;
115             }
116         }
117
118         MovementCheck testRun = new MovementCheck();
119
120         this.testMethod("hindernisUmrunden", null, new Class<?>[] { Runnable.class }, new
            ↪  Runnable[] { testRun });
121         this.world.removeObjects(this.world.getObjectsAt(1, 1, Wand.class));
122         this.world.removeObjects(this.world.getObjectsAt(2, 2, Wand.class));
123         this.world.removeObjects(this.world.getObjectsAt(1, 3, Wand.class));
124         this.world.removeObjects(this.world.getObjectsAt(3, 1, Wand.class));
125
126         if (this.object.getX() != 0 || this.object.getY() != 1) {
127             this.sendStatus("Robby ist bei der Umrundung nicht wieder am Ausgangspunkt
                ↪  angekommen", false);
128             testRun.success = false;
129         }
130
131         this.sendStatus("Hindernis umrunden", testRun.success);
132         return testRun.success;
133     }
134
135     /**
136      * Testet, ob Robby Objekte aus seinem Speicher in die Welt platzieren kann und dabei Grenzen
    ↪  einhält.
137      * @param method Methode, die ein Objekt ablegen soll
138      * @param field Feld, dass dabei vermindert wird
139      * @param min Minimalwert für den Speicher (danach kann kein Objekt mehr platziert werden)
140      * @param cl Klasse des zu platzierenden Objekts
141      * @see #testObjectAquisition
142      */
143     protected boolean testObjectDeposition(String method, String field, int min, Class<? extends
        ↪  Actor> cl) {
144         this.object.setLocation(0, 0);
145         int max = (Integer)this.getReturnValue(field);
146          for (int x = max; x > min - 1; x--) {
147             this.testMethod(method, null);
148             if ((Integer)this.getReturnValue(field) < min) {
149                 return false;
150             }
151         }
152         this.world.removeObjects(this.world.getObjectsAt(0, 0, cl));
153         return true;
154     }
155
156     /**
157      * Testet, ob Robby Objekte aus der Welt in seinen Speicher laden kann und dabei Grenzen
    ↪  einhält.
```

```
158         * @param method Methode, die ein Objekt aufnehmen soll
159         * @param field Getter für einen Integer, der den erhöhten Wert zurückgeben soll
160         * @param max Maximalwert für den Speicher (danach kann kein Objekt mehr aufgenommen werden)
161         * @param cl Klasse des aufzunehmenden Objekts
162         * @see #testObjectDeposition
163         */
164        protected boolean testObjectAquisition(String method, String field, int max, Class<? extends
        ↪  Actor> cl) {
165            this.object.setRotation(0);
166
167            int startValue = (Integer)this.getReturnValue(field);
168
169            Akku[] akkus = new Akku[max + 1];
170            for (int x = 0; x < max + 1; x++) {
171                akkus[x] = new Akku();
172                this.world.addObject(akkus[x], x, 0);
173                this.object.setLocation(x, 0);
174
175                this.testMethod(method, null);
176
177                int newValue = (Integer)this.getReturnValue(field);
178                if (newValue < 0 || newValue > max) {
179                    this.sendStatus("Feld " + field + " blieb nicht im Bereich [ 0," + max + " ]",
                    ↪  false);
180                    return false;
181                }
182
183                if (x < max) {
184                    if (newValue != startValue + (x + 1)
185                     || !this.world.getObjectsAt(x, 0, cl).isEmpty()) {
186                        this.sendStatus("Feld " + field + " zählt nach Aufnehmen eines Akkus nicht
                        ↪  hoch oder sammelt ihn gar nicht erst ein. ", false);
187                        return false;
188                    }
189                } else {
190                    this.world.removeObject(akkus[x]);
191                }
192            }
193
194            return true;
195        }
196
197        /**
198         * Testet den Nachweis eines Objekts auf relativer Position zum Aktor.
199         * @param title Bezeichnung für den Test
200         * @param methods String-Array aus Methodennamen für die einzelnen Positionen (vorne, links,
    ↪  rechts, hinten)
201         * @param cl Klasse des nachzuweisenden Aktors
202         * @see #testObjektDetection
203         */
204        protected boolean testRotationalObjectDetection(String title, String[] methods, Class<?
        ↪  extends Actor> cl) {
205            if (methods.length < 4) {
206                System.err.println("Fehlerhaftes Testskript (testRotationalObjectDetection): Benötigt
                ↪  4 Methodennamen. ");
207            }
208            try {
209                boolean test0 = true;
210                this.object.setRotation(0);
211                test0 &= this.testObjectDetection(2, 1, methods[0], "0ř", cl);
212                test0 &= this.testObjectDetection(1, 0, methods[1], "0ř", cl);
213                test0 &= this.testObjectDetection(1, 2, methods[2], "0ř", cl);
214                test0 &= this.testObjectDetection(0, 1, methods[3], "0ř", cl);
215
216                boolean test90 = true;
217                this.object.setRotation(90);
218                test90 &= this.testObjectDetection(1, 2, methods[0], "90ř", cl);
```

```
            test90 &= this.testObjectDetection(2, 1, methods[1], "90ř", cl);
            test90 &= this.testObjectDetection(0, 1, methods[2], "90ř", cl);
            test90 &= this.testObjectDetection(1, 0, methods[3], "90ř", cl);

            boolean test180 = true;
            this.object.setRotation(180);
            test180 &= this.testObjectDetection(0, 1, methods[0], "180ř", cl);
            test180 &= this.testObjectDetection(1, 2, methods[1], "180ř", cl);
            test180 &= this.testObjectDetection(1, 0, methods[2], "180ř", cl);
            test180 &= this.testObjectDetection(2, 1, methods[3], "180ř", cl);

            boolean test270 = true;
            this.object.setRotation(270);
            test270 &= this.testObjectDetection(1, 0, methods[0], "270ř", cl);
            test270 &= this.testObjectDetection(0, 1, methods[1], "270ř", cl);
            test270 &= this.testObjectDetection(2, 1, methods[2], "270ř", cl);
            test270 &= this.testObjectDetection(1, 2, methods[3], "270ř", cl);

            this.sendStatus("Test von " + title + " für 0ř", test0);
            this.sendStatus("Test von " + title + " für 90ř", test90);
            this.sendStatus("Test von " + title + " für 180ř", test180);
            this.sendStatus("Test von " + title + " für 270ř", test270);

            return ( test0 && test90 && test180 && test270 );
        } catch (Exception ex) {
            System.err.println("Fehlerhaftes Testskript (testRotationalObjectDetection): Kann
             ↪  Methodennamen nicht auflösen (" + ex.getMessage() + ")");
            return false;
        }
    }

    /**
     * Gibt an, ob die Methode einen anderen Actor positiv und negativ
     * nachweisen kann. Wirft evtl. Fehler beim Instanziieren des Testobjekts.
     * @param x Horizontale Koordinate für das Objekt
     * @param y Vertikale Koordinate für das Objekt
     * @param method Name der Methode in der Klasse Robby
     * @param cl Klasse des gesuchten Actors
     * @returns Erfolg des Tests
     * @see FeatureTest#testMethod
     */
    protected boolean testObjectDetection(int x, int y, String method, String test, Class<?
     ↪  extends Actor> cl)
            throws InstantiationException, IllegalAccessException {
        // Sicherstellen, dass das Objekt nicht schon in der Spielwelt existiert
        this.world.removeObjects(this.world.getObjectsAt(x, y, cl));

        // Testen, ob die Methode keine false-positives zurückgibt
        boolean negative = this.testMethod(method, false);

        // Platzieren der neuen Objektinstanz in der Welt
        this.world.addObject(cl.newInstance(), x, y);

        // Testen, ob die Methode keine false-negatives zurückgibt
        boolean positive = this.testMethod(method, true);

        // Spielwelt für die nächsten Tests aufräumen
        this.world.removeObjects(this.world.getObjectsAt(x, y, cl));

        if (!positive) {
            this.sendStatus(method + "() konnte im Test " + test + " den Actor " + cl.getName() +
             ↪  "nicht positiv erkennen. ", false);
        }
        if (!negative) {
            this.sendStatus(method + "() konnte im Test " + test + " den Actor " + cl.getName() +
             ↪  " nicht negativ erkennen. ", false);
        }
```

```
282
283         return (positive && negative);
284     }
285 }
```

4.4 RoboterWelt.java

```
1  import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)
2
3  /**
4   * Die einzigen aktiven Akteure in der Roboterwelt sind die Roboter.
5   * Die Welt besteht aus 14 * 10 Feldern.
6   */
7
8  public class RoboterWelt extends World
9  {
10     private static int zellenGroesse = 50;
11
12     /**
13      * Erschaffe eine Welt mit 14 * 10 Zellen.
14      */
15     public RoboterWelt()
16     {
17         super(14, 10, zellenGroesse);
18         setBackground("images/Bodenplatte.png");
19         setPaintOrder(Roboter.class, Schraube.class, Akku.class, Wand.class);
20         Greenfoot.setSpeed(15);
21
22         RobbyTest robbyTest = new RobbyTest(this);
23         if (!robbyTest.testAllFeatures()) {
24             System.err.println("Die Klasse Robby hat nicht alle Tests bestanden. Bitte überprüfen
               ↪  sie den Log, um das Problem näher einzugrenzen. ");
25         } else {
26             System.out.println("Die Klasse Robby hat alle Tests bestanden. Bravo!");
27         }
28     }
29 }
```