Robby Projekt

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

Adrian Schrader Moritz Jung Alexander Riecke

17. Januar 2016

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 verstehen, 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.

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

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.

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

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

Die Klasse Robby sollte durch die in ihrer Funktion erweiterten Methoden akkuAufnehmen() und schraubeAblegen() mit zwei neu instanziierten globalen Variablen anzahlAkkus und anzahlSchrauben bestimmte Änderungen abspeichern können. Hierzu war vorgesehen, dass die Klasse Robby die vorher festgelegte Anzahl an Schrauben und Akkus nach Ausführung der Methoden jeweils um eins erniedrigt bzw. um eins erhöht. Sollten die Methoden nicht ausführbar sein, was durch ein leeres Feld ohne Akku oder zu wenig Schrauben verursacht werden könnte, war eine aussagekräftige Meldung vorgesehen, die dies beschreibt.

Problematik

Die ersten Schritte zur Herangehensweise an die Aufgabe waren zunächst die globalen Variablen. Anhand der vorgebenen Werte beider Variabeln, entstand hier bereits die Idee bei einer if-Abfrage diese Werte als Bedingung für das weitere Verfahren in der Methode zu verwenden. Mit Hilfe dieser Grundüberlegung entwickelten sich beide Methoden in der Planung zu jeweils einer einzigen if-Abfrage, in der mehre Bedingungen und Szenarien gleichzeitig abgedeckt werden. So war zum Beispiel vorgesehen in einer if-Abfrage ein leeres Feld UND eine noch nicht überschrittene Maximalanzahl an Akkus zu implementieren.

Dadurch entstand jedoch nach einigen Testdurchläufen ein Problem mit der getrennten Ausgabe der Fehlermeldung und der getrennten, nacheinander abfolgenden Ausführung in der if-Schleife.

Lösung

Hierfür war dann eine Verschachtelung der Befehlskette vorgesehen. Hierzu wurden dann die Befehle mit zweiten if-Abfragen ineinander verschachtelt und es entstand eine chronologische Vorgehensweise. Indem die Klasse Robby an dem bereits oben erwähnten Beispiel zunächst das Feld überprüft und dort einen Akku erkennt (Feld nicht leer), wird nun erst die Maximalanzahl von zehn Akkus überprüft. Ist diese auch noch nicht überschritten wird nun ein Akku aufgenommen und in den Speicher einbezogen. Bei Überlastung der Speichergrenze (gröSSer 10) wird nun in der gleichen Abfrage per else-Schleife eine Fehlermeldung ausgegeben. Falls sich auf dem überprüften Feld jedoch kein Akku befindet wird hier jetzt in der äuSSeren if-Schleife separat per else-Abfrage eine entsprechende Fehlermeldung gezeigt, was das aufgetretene Problem letztendlich gelöst hat.

Ein Test der beiden erweiterten Methoden zeigt nun eine erwartete Verringerung der Schrauben- und Erhöhung der Akku-Anzahl bezogen auf die Werte in den globalen Variablen. In entsprechenden Szenarien reagiert die Klasse Robby auch mit den Situationen entsprechenden Fehlermeldungen, wodurch die Speicherfunktion ihren Anforderungen entsprechend erfolgreich funktioniert.

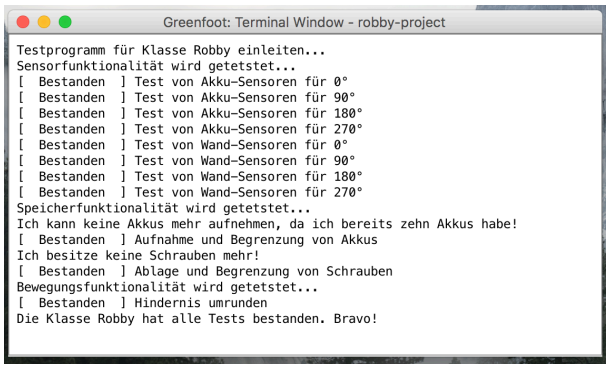

Abbildung 2.1: Erfolgreicher Testlog nach dem Kompilieren der Klassen.

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.

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.

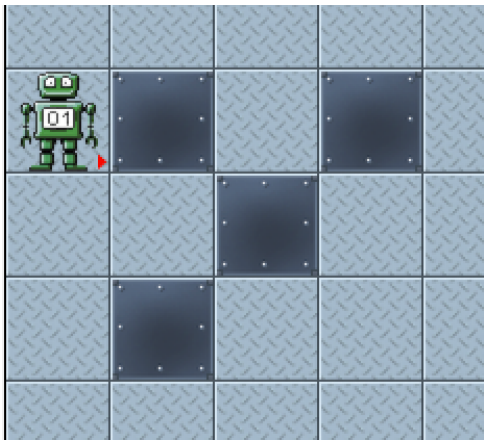

Abbildung 2.2: Das von der Testfunktion erstellte Hindernis.

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.

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.

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     {
31         anzahlAkkus     = Robby.INIT_AKKUS;
```

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

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

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

```
232         * Der Sensor ueberprueft, ob sich entgegen der Laufrichtung des Roboters
233         * eine Wand befindet.
234         */
235        public boolean wandHinten()
236        {
237            return this.istObjektNebendran(180, Wand.class);
238        }
239
240        /**
241         * Der Sensor überprüft, ob sich neben der Laufrichtung von Robby ein
242         * anderer Actor befindet.
243         * @param direction Winkel von der Laufrichtung zum Suchfeld
244         * @param cl Klasse des gesuchten Actors
245         * @return boolean Gibt an, ob das Objekt mit den angegeben Eigenschaften existiert
246         * @see #akkuVorne
247         * @see #akkuRechts
248         * @see #akkuLinks
249         * @see #akkuHinten
250         * @see #wandHinten
251         */
252        protected boolean istObjektNebendran(int direction, Class<?> cl)
253        {
254           double angle = (this.getRotation() + direction) / 180.0 * Math.PI;
255
256           return (this.getOneObjectAtOffset(
257                   (int)Math.cos(angle),
258                   (int)Math.sin(angle), cl) != null);
259        }
260
261
262        /**
263         * Der Sensor überprüft, ob sich neben der Laufrichtung von Robby
264         * die Weltgrenze befindet.
265         * @param direction Winkel von der Laufrichtung zum Ende
266         * @return boolean Gibt an, ob die Weltgrenze neben Robby ist
267         * @see #grenzeVorne
268         * @see #grenzeRechts
269         * @see #grenzeLinks
270         * @see #grenzeHinten
271         */
272        protected boolean istGrenzeNebendran(int direction) {
273            direction = direction % 360;
274            return ( this.getX() + 1 >= this.getWorld().getWidth()
275                    && this.getRotation() == (360 - direction) % 360 )
276                    || ( this.getY() + 1 >= this.getWorld().getHeight()
277                    && this.getRotation() == (450 - direction) % 360 )
278                    || ( this.getX() <= 0
279                    && this.getRotation() == (540 - direction) % 360 )
280                    || ( this.getY() <= 0
281                    && this.getRotation() == (630 - direction) % 360 );
282        }
283
284    }
```

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      protected T object;
18      protected Class<T> cl;
19      protected String name;
20      protected boolean failed;
21
22      /**
23       * Instanziiert die Klasse über den Typ der Testklasse. Die zu testende
24       * Klasseninstanz wird automatisch erstellt.
25       * @param cl Typ der Testklasse
26       */
27      public FeatureTest(Class<T> cl) {
28          this.cl = cl;
29          this.name = cl.getName();
30          this.failed = false;
31
32          try {
33              this.object = cl.newInstance();
34          }
35          catch (Exception ex) {
36              System.err.println("Es konnte keine Instanz von " + this.name
37                  + " erstellt werden. ");
38          }
39      }
40
41      /**
42       * Instanziiert die Klasse über ein bestehendes Objekt.
43       * @param obj Objekt vom zu testenden Typ
44       */
45      public FeatureTest(T obj) {
46          this.cl = (Class<T>)obj.getClass();
47          this.name = cl.getName();
48          this.failed = false;
49          this.object = obj;
50      }
51
52      /**
53       * @return Ertellte Instanz der zu testenden Klasse
54       */
55      public T getInstance() {
56          return this.object;
57      }
58
59      /**
60       * @return Name der Testklasse, der auch in den Statusmeldungen benutzt wird.
61       */
62      public String getName() {
63          return this.name;
64      }
65
66      /**
67       * Diese Funktion sollte von Unterklassen überschrieben werden, um alle
68       * Tests auszuführen und deren Ergebnisse zurückzugeben.
69       * @return Erfolg des Tests
70       */
71      public boolean testAllFeatures() {
72          return this.failed;
73      }
74
75      /**
76       * Gibt die Statusmeldungen für einzelne Tests aus
77       * @param message Nachricht für die Konsole/Log
78       * @param passed Erfolgreiche Meldung
```

```
79          * @see #MESSAGE_PASSED
80          * @see #MESSAGE_FAILED
81          */
82      protected void sendStatus(String message, boolean passed) {
83          System.out.println("[" + (passed ? FeatureTest.MESSAGE_PASSED :
            ↪  FeatureTest.MESSAGE_FAILED) + "] " + message);
84      }
85
86      /**
87       * Gibt das reflexive Feld aus.
88       * @param <F> Typ des Feldes
89       * @param name Name des Fields
90       * @param type Typ des Feldes
91       * @return Aktueller gecasteter Wert des Feldes
92       * @see testField
93       */
94      protected <F> F getField(String name, Class<F> type) {
95          try {
96              return (F)(cl.getField(name).get(this.object));
97          } catch (Exception ex) {
98              this.sendStatus(name, false);
99              failed = true;
100
101             System.err.println("Fehlerhaftes Testskript (testMethod): Feld in " + cl.getName() +
                ↪  " konnte nicht geladen oder gecastet werden. (Nachricht: " + ex.getMessage() +
                ↪  ")");
102             return null;
103         }
104     }
105
106     /**
107      * Überprüft, ob das reflexive Feld der Klasse einem Sollwert entspricht.
108      * @param name Name des Fields
109      * @param target Sollwert für das angegebene Feld
110      * @return Gibt an, ob der Sollwert mit dem Feldwert übereinstimmt
111      * @see testMethod
112      */
113     protected boolean testField(String name, Object target) {
114         try {
115             return cl.getField(name).get(this.object) == target;
116         } catch (Exception ex) {
117             this.sendStatus(name, false);
118             failed = true;
119
120             System.err.println("Fehlerhaftes Testskript (testMethod): Feld in " + cl.getName() +
                ↪  " konnte nicht geladen werden. (Nachricht: " + ex.getMessage() + ")");
121             return false;
122         }
123     }
124
125     /**
126      * Überprüft, ob die reflektiv angegebene Methode den Zielwert zurückgibt.
127      * Diese Überladung geht davon aus, dass die Methode keine Argumente
128      * benötigt.
129      * @param name Name der Methode
130      * @param returnValue Der erwartete Rückgabewert
131      * @return Gibt an, ob Erwartungswert mit dem Rückgabewert übereinstimmt
132      * @see #testMethod
133      */
134     protected boolean testMethod(String name, Object returnValue)
135     {
136         return this.testMethod(name, returnValue, new Class<?>[]{}, new Object[] {});
137     }
138
139     /**
140      * Überprüft, ob die reflektiv angegebene Methode den Zielwert zurückgibt.
141      * @param name Name der Methode
```

```
142         * @param expectedValue Erwartungswert für den Rückgabewert
143         * @param parameterTypes Array der Typen der Parameter, mit denen die Methode gefunden werden
    ↪  kann
144         * @param parameters Array der benötigten Parameter der gesuchten Methode
145         * @return Gibt an, ob die Methode den Test bestanden hat
146         */
147        protected boolean testMethod(String name, Object expectedValue, Class<?>[] parameterTypes,
        ↪  Object[] parameters) {
148            try {
149                Method method = this.cl.getMethod(name, parameterTypes);
150                Object returnValue = method.invoke(this.object, parameters);
151
152                if (expectedValue != null) {
153                    if (!returnValue.equals(expectedValue))
154                    {
155                        failed = true;
156                        return false;
157                    }
158                }
159
160                return true;
161            }
162            catch (Exception ex)
163            {
164                this.sendStatus(name, false);
165                failed = true;
166
167                System.err.println("Fehlerhaftes Testskript (testMethod): Methode in " + cl.getName()
                ↪  + " konnte nicht geladen werden. ");
168                return false;
169            }
170        }
171
172        /**
173         * Gibt den Wert eines Getters ohne Parameter mit dem zugehörigen Namen zurück
174         * @param name Name der Methode
175         * @return Rückgabewert der Methode
176         */
177        protected Object getReturnValue(String name) {
178            return getReturnValue(name, new Class<?>[] {}, new Object[] {});
179        }
180
181        /**
182         * Führt die angegebene Methode reflexiv aus und gibt seinen Rückgabewert zurück
183         * @param name Name der Methode
184         * @param parameterTypes Array der Typen der Parameter, mit denen die Methode gefunden werden
    ↪  kann
185         * @param parameters Array der benötigten Parameter der gesuchten Methode
186         * @return Rückgabewert der Methode
187         */
188        protected Object getReturnValue(String name, Class<?>[] parameterTypes, Object[] parameters)
        ↪  {
189             try {
190                Method method = this.cl.getMethod(name, parameterTypes);
191                return method.invoke(this.object, parameters);
192            }
193            catch (Exception ex)
194            {
195                this.sendStatus(name, false);
196                failed = true;
197
198                System.err.println("Fehlerhaftes Testskript (getReturnValue): Methode in " +
                ↪  cl.getName() + " konnte nicht geladen werden. ");
199                return false;
200            }
201        }
202    }
```

4.3 RobbyTest.java

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

/**
 * Diese Klasses testet die Funktionalität von Sensoren, Speicher und
 * Bewegungsapparat der Klasse Robby nach dem Programmstart.
 * @author Adrian Schrader
 * @version 1.5
 */
public class RobbyTest extends FeatureTest<Robby>
{
    private final RoboterWelt world;

    /**
     * Instanziiert die Testklasse und lässt neuen Robby erstellen
     * @param world Spielwelt, in der die Tests durchgeführt werden sollen.
     */
    public RobbyTest(RoboterWelt world)
    {
        super(Robby.class);
        this.world = world;
    }

    /**
     * Startet alle Tests für die Klasse Robby
     * @return Erfolg der Tests
     */
    @Override
    public boolean testAllFeatures() {
        System.out.println("Testprogramm für Klasse Robby einleiten...");

        try {
            this.world.addObject(this.object, 0, 0);

            boolean success = this.testSensors() && this.testMemory() && testMovement();

            this.world.removeObject(this.object);
            return success;
        } catch (Exception ex) {
            System.err.println("Innerhalb der Testroutine ist ein Fehler aufgetreten. Sind alle
            ↪  Funktionen aufrufbar?");
            return false;
        }
    }

    /**
     * Testet Robbys Sensoren zum Aufspüren von Wänden und Akkus in einer Entfernung von einem
↪   Feld.
     * @return Erfolg des Tests
     * @throws InstantiationException
     * @throws IllegalAccessException
     */
    public boolean testSensors() throws InstantiationException, IllegalAccessException {
        System.out.println("Sensorfunktionalität wird getetstet...");
        this.failed = false;

        // Füge Robby der Spielwelt hinzu
        this.object.setLocation(1, 1);

        // Überprüfe die Sensorfunktionen für Akkus
        this.testRotationalObjectDetection("Akku-Sensoren", new String[] { "akkuVorne",
        ↪  "akkuLinks", "akkuRechts", "akkuHinten" }, Akku.class);

        // Überprüfe die Sensorfunktionen für Wände
        this.testRotationalObjectDetection("Wand-Sensoren", new String[] { "wandVorne",
        ↪  "wandLinks", "wandRechts", "wandHinten" }, Wand.class);

```

```
63          return !this.failed;
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.object.setLocation(0, 1);
93          this.object.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                     }
114                 }
115                 if (!isObstacleNearby)
116                     sendStatus("Robby ist in [" + getInstance().getX() + "," +
                        ↪  getInstance().getY() + "] vom Weg abgekommen", false);
117                 success &= isObstacleNearby;
118             }
119         }
120
121         MovementCheck testRun = new MovementCheck();
122
123         this.testMethod("hindernisUmrunden", null, new Class<?>[] { Runnable.class }, new
            ↪  Runnable[] { testRun });
124         this.world.removeObjects(this.world.getObjectsAt(1, 1, Wand.class));
125         this.world.removeObjects(this.world.getObjectsAt(2, 2, Wand.class));
```

```
126         this.world.removeObjects(this.world.getObjectsAt(1, 3, Wand.class));
127         this.world.removeObjects(this.world.getObjectsAt(3, 1, Wand.class));
128 
129         if (this.object.getX() != 0 || this.object.getY() != 1) {
130             this.sendStatus("Robby ist bei der Umrundung nicht wieder am Ausgangspunkt
                ↪  angekommen", false);
131             testRun.success = false;
132         }
133 
134         this.sendStatus("Hindernis umrunden", testRun.success);
135         return testRun.success;
136     }
137 
138     /**
139      * Testet, ob Robby Objekte aus seinem Speicher in die Welt platzieren kann und dabei Grenzen
    ↪  einhält.
140      * @param method Methode, die ein Objekt ablegen soll
141      * @param field Feld, dass dabei vermindert wird
142      * @param min Minimalwert für den Speicher (danach kann kein Objekt mehr platziert werden)
143      * @param cl Klasse des zu platzierenden Objekts
144      * @return Efolg des Tests
145      * @see #testObjectAquisition
146      */
147     protected boolean testObjectDeposition(String method, String field, int min, Class<? extends
        ↪  Actor> cl) {
148         this.object.setLocation(0, 0);
149         int max = (Integer)this.getReturnValue(field);
150          for (int x = max; x > min - 1; x--) {
151             this.testMethod(method, null);
152             if ((Integer)this.getReturnValue(field) < min) {
153                 return false;
154             }
155         }
156         this.world.removeObjects(this.world.getObjectsAt(0, 0, cl));
157         return true;
158     }
159 
160     /**
161      * Testet, ob Robby Objekte aus der Welt in seinen Speicher laden kann und dabei Grenzen
    ↪  einhält.
162      * @param method Methode, die ein Objekt aufnehmen soll
163      * @param field Getter für einen Integer, der den erhöhten Wert zurückgeben soll
164      * @param max Maximalwert für den Speicher (danach kann kein Objekt mehr aufgenommen werden)
165      * @param cl Klasse des aufzunehmenden Objekts
166      * @return Erfolg des Tests
167      * @see #testObjectDeposition
168      */
169     protected boolean testObjectAquisition(String method, String field, int max, Class<? extends
        ↪  Actor> cl) {
170         this.object.setRotation(0);
171 
172         int startValue = (Integer)this.getReturnValue(field);
173 
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.object.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.object.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.object.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.object.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.object.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

```
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  }
```