Übungsblatt 7

Musterlösung

Aufgabe 1 Nachbarschaften

In der Nachbarschaftssignatur werden die Zahlen 1 für Richtung 0, 2 für Richtung 1, 4 für Richtung 3 und 8 für Richtung 3 addiert, um die erreichbaren Nachbarn zu kodieren. Die Zahlen sind eigentlich Bits, d.h. sie lassen sich mit $1 << richtung$ berechnen. Um zu testen, ob das Bit für eine bestimmte Richtung gesetzt ist, muss es isoliert werden. Dies kann durch eine bitweise Und-Verknüpfung erreicht werden. $signatur \ \& \ (1 << richtung)$ berechnet eine Zahl, in der alle Bits, die nicht $1 << richtung$ sind, auf jeden Fall 0 sind, also ausmaskiert wurden. Übrig bleibt das Bit bei $1 << richtung$ aus der Signatur. Ist es dort gesetzt, ist das Ergebnis der Und-Verknüpfung $1 << richtung$, ist es das nicht, ist das Ergebnis 0. Also muss das Ergebnis der Und-Verknüpfung einfach mit 0 verglichen werden, um festzustellen, ob in einer bestimmten Richtung (im Code *direction*) ein Nachbar existiert:

```
114     /**
115      * Prüft, ob eine Zelle laut der Gitterstruktur in einer bestimmten
116      * Richtung einen Nachbarn hat.
117      * @param x Die x-Koordinate der geprüften Zelle.
118      * @param y Die y-Koordinate der geprüften Zelle.
119      * @param direction Die geprüfte Richtung (0 = rechts ... 3 = oben).
120      * @return Gibt es in der Richtung einen Nachbarn?
121      */
122     boolean hasNeighbor(final int x, final int y, final int direction)
123     {
124         return (getNeighborhood(x * 2, y * 2) & 1 << direction) != 0;
125     }
```

Die im nächsten Abschnitt beschriebenen Tests laufen alle erfolgreich durch.

Aufgabe 2 Nachbarschaftstest

```
1  // Importiert assertEquals usw. sowie Test-Annotationen
2  import static org.junit.jupiter.api.Assertions.*;
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.BeforeEach;
5  import org.junit.jupiter.api.Test;
6
7  /**
8   * Diese Klasse definiert die Tests für die Klasse Field.
9   * Es gibt keine Tests für Fälle, die nicht wirklich definiert
10  * sind, wie z.B. was passiert, wenn die Nachbarschaft einer
11  * Zelle überprüft wird, die selbst nicht im Feld liegt. Alle
12  * Tests bis auf einen verwenden ein leicht abgeändertes Feld
13  * aus Übungsblatt 5, das ja alle Nachbarschaftskombinationen
14  * enthält.
15  *
16  * @author Thomas Röfer
17  */
18 public class FieldTest
19 {
20     /** Das Feld, das von fast allen Tests verwendet wird. */
21     private Field field;
22
23     /**
24      * Erzeugen des Standardfeldes, das alle Kombinationen von
25      * Nachbarschaften enthält. Die ersten beiden Zeilen sind
26      * kürzer, so dass auch implizit getestet wird, ob die
27      * Klasse damit umgehen kann.
28      */
```

```
29      @BeforeEach
30      public void setUp()
31      {
32          field = new Field(new String[] {
33              "O-O-O-O",
34              "|   |",
35              "O O-O-O O",
36              "| | | | |",
37              "O-O-O-O-O",
38              "| | | | |",
39              "O O-O-O O",
40              "    |   |",
41              "  O-O-O O"
42          });
43      }
44
45      /**
46       * Testet den Zugriff in Richtungen außerhalb des Feldes
47       * auf einem 1x1-Feld. Dort sollte es keine Nachbarn geben.
48       */
49      @Test
50      public void testOutside()
51      {
52          final Field field = new Field(new String[] {"O"});
53          assertFalse(field.hasNeighbor(0, 0, 0));
54          assertFalse(field.hasNeighbor(0, 0, 1));
55          assertFalse(field.hasNeighbor(0, 0, 2));
56          assertFalse(field.hasNeighbor(0, 0, 3));
57      }
58
59      /** Testen einer Zelle ohne Nachbarn. */
60      @Test
61      public void testNone()
62      {
63          assertFalse(field.hasNeighbor(0, 4, 0));
64          assertFalse(field.hasNeighbor(0, 4, 1));
65          assertFalse(field.hasNeighbor(0, 4, 2));
66          assertFalse(field.hasNeighbor(0, 4, 3));
67      }
68
69      /**
70       * Testen von Zellen mit einem Nachbarn. Der Nachbar kann
71       * in vier Richtungen liegen, die nacheinander getestet
72       * werden.
73       */
74      @Test
75      public void testSingle()
76      {
77          assertTrue(field.hasNeighbor(1, 4, 0));
78          assertFalse(field.hasNeighbor(1, 4, 1));
79          assertFalse(field.hasNeighbor(1, 4, 2));
80          assertFalse(field.hasNeighbor(1, 4, 3));
81
82          assertFalse(field.hasNeighbor(4, 1, 0));
83          assertTrue(field.hasNeighbor(4, 1, 1));
84          assertFalse(field.hasNeighbor(4, 1, 2));
85          assertFalse(field.hasNeighbor(4, 1, 3));
86
87          assertFalse(field.hasNeighbor(3, 0, 0));
88          assertFalse(field.hasNeighbor(3, 0, 1));
89          assertTrue(field.hasNeighbor(3, 0, 2));
90          assertFalse(field.hasNeighbor(3, 0, 3));
91
92          assertFalse(field.hasNeighbor(0, 3, 0));
93          assertFalse(field.hasNeighbor(0, 3, 1));
94          assertFalse(field.hasNeighbor(0, 3, 2));
95          assertTrue(field.hasNeighbor(0, 3, 3));
96      }
97
98      /**
99       * Testen von Zellen mit gegenüber liegenden Nachbarn.
100      * Diese können zwei Ausrichtungen haben, die nacheinander
101      * getestet werden.
102      */
103     @Test
104     public void testStraight()
105     {
```

```
106         assertTrue(field.hasNeighbor(1, 0, 0));
107         assertFalse(field.hasNeighbor(1, 0, 1));
108         assertTrue(field.hasNeighbor(1, 0, 2));
109         assertFalse(field.hasNeighbor(1, 0, 3));
110
111         assertFalse(field.hasNeighbor(0, 1, 0));
112         assertTrue(field.hasNeighbor(0, 1, 1));
113         assertFalse(field.hasNeighbor(0, 1, 2));
114         assertTrue(field.hasNeighbor(0, 1, 3));
115     }
116
117     /**
118      * Testen von Zellen mit zwei Nachbarn in L-Form. Das L kann
119      * in vier Richtungen orientiert sein, die nacheinander getestet
120      * werden.
121      */
122     @Test
123     public void testL()
124     {
125         assertTrue(field.hasNeighbor(1, 1, 0));
126         assertTrue(field.hasNeighbor(1, 1, 1));
127         assertFalse(field.hasNeighbor(1, 1, 2));
128         assertFalse(field.hasNeighbor(1, 1, 3));
129
130         assertFalse(field.hasNeighbor(3, 1, 0));
131         assertTrue(field.hasNeighbor(3, 1, 1));
132         assertTrue(field.hasNeighbor(3, 1, 2));
133         assertFalse(field.hasNeighbor(3, 1, 3));
134
135         assertFalse(field.hasNeighbor(3, 3, 0));
136         assertFalse(field.hasNeighbor(3, 3, 1));
137         assertTrue(field.hasNeighbor(3, 3, 2));
138         assertTrue(field.hasNeighbor(3, 3, 3));
139
140         assertTrue(field.hasNeighbor(1, 3, 0));
141         assertFalse(field.hasNeighbor(1, 3, 1));
142         assertFalse(field.hasNeighbor(1, 3, 2));
143         assertTrue(field.hasNeighbor(1, 3, 3));
144     }
145
146     /**
147      * Testen von Zellen mit drei Nachbarn in T-Form. Das T kann
148      * in vier Richtungen orientiert sein, die nacheinander getestet
149      * werden.
150      */
151     @Test
152     public void testT()
153     {
154         assertTrue(field.hasNeighbor(2, 0, 0));
155         assertTrue(field.hasNeighbor(2, 0, 1));
156         assertTrue(field.hasNeighbor(2, 0, 2));
157         assertFalse(field.hasNeighbor(2, 0, 3));
158
159         assertFalse(field.hasNeighbor(4, 2, 0));
160         assertTrue(field.hasNeighbor(4, 2, 1));
161         assertTrue(field.hasNeighbor(4, 2, 2));
162         assertTrue(field.hasNeighbor(4, 2, 3));
163
164         assertTrue(field.hasNeighbor(2, 4, 0));
165         assertFalse(field.hasNeighbor(2, 4, 1));
166         assertTrue(field.hasNeighbor(2, 4, 2));
167         assertTrue(field.hasNeighbor(2, 4, 3));
168
169         assertTrue(field.hasNeighbor(0, 2, 0));
170         assertTrue(field.hasNeighbor(0, 2, 1));
171         assertFalse(field.hasNeighbor(0, 2, 2));
172         assertTrue(field.hasNeighbor(0, 2, 3));
173     }
174
175     /** Test einer Zelle mit vier Nachbarn. */
176     @Test
177     public void testX()
178     {
179         assertTrue(field.hasNeighbor(2, 2, 0));
180         assertTrue(field.hasNeighbor(2, 2, 1));
181         assertTrue(field.hasNeighbor(2, 2, 2));
182         assertTrue(field.hasNeighbor(2, 2, 3));
```

```
183     }
184
185     /**
186      * Testen, ob x und y innerhalb von {@link Field#getCell}
187      * vertauscht sind. Rechts unten ist das Feld nicht
188      * spiegelsymmetrisch.
189      */
190     @Test
191     public void testXY()
192     {
193         assertFalse(field.hasNeighbor(4, 4, 2));
194         assertTrue(field.hasNeighbor(4, 4, 3));
195     }
196 }
```

Aufgabe 3 In geregelten Bahnen

In der Klasse *PI1Game* wird das Feld konstruiert:

```
17          final Field field = new Field(new String[] {
18              "O-O-O-O",
19              "     |",
20              "O-O-O-O",
21              "     |",
22              "O-O-O-O-O",
23              "     |",
24              "O-O-O"
25          });
```

Die Definition ersetzt einen Großteil der bisher einzeln erzeugten Objekte. Ein paar bleiben noch erhalten (Ziel, Brücke, Bach), da sie bisher noch nicht von der Klasse *Field* erzeugt werden können.

Des Weiteren wurden lediglich die vier Richtungstests erweitert:

```
38              if (key == VK_RIGHT && field.hasNeighbor(player.getX(), player.getY(), 0)) {

42              else if (key == VK_DOWN && field.hasNeighbor(player.getX(), player.getY(), 1)) {

46              else if (key == VK_LEFT && field.hasNeighbor(player.getX(), player.getY(), 2)) {

50              else if (key == VK_UP && field.hasNeighbor(player.getX(), player.getY(), 3)) {
```

In der Klasse *Walker* wird nun statt der Attribute zum Schrittezählen das Feld gespeichert:

```
14      private final Field field;

17      Walker(final GameObject avatar, final Field field)

20          this.field = field;
```

Alle Zeilen, in denen Schritte gezählt wurden, entfallen. Geht es in Vorwärtsrichtung nicht mehr weiter, wird umgedreht.

```
47          // Wenn im Verfolgermodus und aufgezeichneter Schritt möglich,
48          // dann diesen verwenden.
49          if (stepsToFollow != null && field.hasNeighbor(avatar.getX(), avatar.getY(),
50                  stepsToFollow.peek())) {
51              avatar.setRotation(stepsToFollow.pop());
52          }
53          else {
54              // Wir sind nicht (mehr) im Verfolgermodus
55              stepsToFollow = null;
56
57              // Umdrehen, wenn nächster Schritt nicht mehr ausführbar
58              if (!field.hasNeighbor(avatar.getX(), avatar.getY(), avatar.getRotation())) {
59                  avatar.setRotation(avatar.getRotation() + 2);
60              }
```

In *PI1Game* ändern sich die Aufrufe der Konstruktoren:

```
31          final Walker walker1 = new Walker(new GameObject(1, 0, 2, "claudius"), field);
32          final Walker walker2 = new Walker(new GameObject(0, 1, 0, "laila"), field);
33          final Walker walker3 = new Walker(new GameObject(3, 2, 2, "child"), field);
```