Praktische Informatik II | SoSe 2017 | Jana Göken
TutorIn: Nikhil Bhardwaj | Gruppe: 09 | Alex Rink
Meira Iske

Übungsblatt 1

Aufgabenlösung
Abgabe: 30.04.2017

Aufgabe 1 PushPopy

Zuerst haben wir die Testklassen für ClassFiFo und ClassLiFo erstellt, die jeweils alle Methoden positiv und negativ testen.

```
1  package pi.uebung01.test;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Before;
6  import org.junit.Test;
7
8  import pi.uebung01.spec.ClassFiFo;
9  import pi.uebung01.spec.PushPopyException;
10 import pi.uebung01.spec.PushPopyInterface;
11
12 public class FifoTest {
13   private PushPopyInterface<String> ppi;
14
15   @Before
16   public void beforeCreateQueue() {
17     ppi = new ClassFiFo<String>(5);
18
19   }
20
21   /**
22    * positiver push Test
23    */
24   @Test
25   public void testPush() {
26     ppi.push("bla");
27     ppi.push("bla");
28     ppi.push("bla");
29   }
30
31   /**
32    * negativer push Test
33    */
34   @Test(expected = IllegalArgumentException.class)
35   public void testNullPush() {
36     ppi.push(null);
37   }
38
39   /**
40    * Overflow Test
41    */
42   @Test(expected = PushPopyException.class)
43   public void testOverflow() {
44     ppi.push("bla");
45     ppi.push("bla");
46     ppi.push("bla");
47     ppi.push("bla");
48     ppi.push("bla");
49     ppi.push("bla");
50   }
```

```
51
52   /**
53    * positiver pop Test
54    */
55   @Test
56   public void testPop() {
57     ppi.push("bla");
58     ppi.push("bli");
59     ppi.push("blub");
60     assertEquals("bla", ppi.pop());
61     assertEquals("bli", ppi.pop());
62   }
63
64   /**
65    * negativer pop Test
66    */
67   @Test(expected = PushPopyException.class)
68   public void testEmptyPop(){
69     assertEquals(null, ppi.pop());
70   }
71
72   /**
73    * positiver isFull test
74    */
75   @Test
76   public void testIsFull() {
77     ppi.push("bla");
78     ppi.push("bla");
79     ppi.push("bla");
80     ppi.push("bla");
81     ppi.push("bla");
82     equals(ppi.isFull());
83   }
84
85   /**
86    * negativer isFullTest
87    */
88   @Test
89   public void testIsNotFull(){
90     assertEquals(false, ppi.isFull());
91   }
92
93   /**
94    * positiver isEmptyTest
95    */
96   @Test
97   public void testIsEmpty() {
98     equals(ppi.isEmpty());
99   }
100
101  /**
102   * negativer isEmpty Test
103   */
104  @Test
105  public void testIsNotEmpty(){
106    ppi.push("hi");
107    assertEquals(false, ppi.isEmpty());
108  }
109
110 }
```

```
1  package pi.uebung01.test;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Before;
6  import org.junit.Test;
7
8  import pi.uebung01.spec.ClassLiFo;
9  import pi.uebung01.spec.PushPopyException;
10 import pi.uebung01.spec.PushPopyInterface;
```

```
11
12 public class LifoTest {
13
14   private PushPopyInterface<String> ppi;
15
16   @Before
17   public void beforeCreateStack() {
18     ppi = new ClassLiFo<String>(5);
19
20   }
21
22   /**
23    * positiver push Test
24    */
25   @Test
26   public void testPush() {
27     ppi.push("bla");
28     ppi.push("bla");
29     ppi.push("bla");
30   }
31
32   /**
33    * negativer push Test
34    */
35   @Test(expected = IllegalArgumentException.class)
36   public void testNullPush() {
37     ppi.push(null);
38   }
39
40   /**
41    * Overflow Test
42    */
43   @Test(expected = PushPopyException.class)
44   public void testOverflow() {
45     ppi.push("bla");
46     ppi.push("bla");
47     ppi.push("bla");
48     ppi.push("bla");
49     ppi.push("bla");
50     ppi.push("bla");
51   }
52
53   /**
54    * positiver pop Test
55    */
56   @Test
57   public void testPop() {
58     ppi.push("bla");
59     ppi.push("bli");
60     ppi.push("blub");
61     assertEquals("blub", ppi.pop());
62     assertEquals("bli", ppi.pop());
63   }
64
65   /**
66    * negativer pop Test
67    */
68   @Test(expected = PushPopyException.class)
69   public void testEmptyPop(){
70     assertEquals(null, ppi.pop());
71   }
72
73   /**
74    * positiver isFull test
75    */
76   @Test
77   public void testIsFull() {
78     ppi.push("bla");
79     ppi.push("bla");
80     ppi.push("bla");
81     ppi.push("bla");
```

```
82      ppi.push("bla");
83      equals(ppi.isFull());
84    }
85
86    /**
87     * negativer isFullTest
88     */
89    @Test
90    public void testIsNotFull(){
91      assertEquals(false, ppi.isFull());
92    }
93
94    /**
95     * positiver isEmptyTest
96     */
97    @Test
98    public void testIsEmpty() {
99      equals(ppi.isEmpty());
100   }
101
102   /**
103    * negativer isEmpty Test
104    */
105   @Test
106   public void testIsNotEmpty(){
107     ppi.push("hi");
108     assertEquals(false, ppi.isEmpty());
109   }
110
111 }
```

Die von uns erstelle Klasse ClassLiFo wird implementiert vom Interface LiFoInterFace und überschreibt sowohl die Methoden aus PushPopyInterFace als auch die Methode aus LiFoInterface. Bei der pop-Methode geht es in ClassLiFo darum, dass das Element, das als letzes hinzugefügt wurde, also das jüngste Element, als erstes wieder entfernt wird. Dies passiert in unsere Klasse, indem das Array, in dem sich alle Elemente vom Typ E befinden, überprüft wird. Das jüngste Element steht im Array immer an letzter Stelle (abgesehen von den unbelegten Plätzen im Array). Eine Schleife prüft, welches das erste Element ist, das null entspricht. Das Element vorher muss somit das Jüngste sein und wird ausgegeben und danach entfernt.

```
1  package pi.uebung01.spec;
2
3
4  public class ClassLiFo<E> implements LiFoInterface<E> {
5
6
7
8    private E[] array;
9    private final int size;
10
11   public ClassLiFo(final int bsize){
12
13     size = bsize;
14     /**@SuppressWarnings("unchecked")
15         final E[] array = (E[]) Array.newInstance(null, bsize);
16         this.array = array; **/
17
18         @SuppressWarnings("unchecked")
19         final E[] array = (E[])new Object[size];
20         this.array = array;
21   }
22    /**
23       * Fügt das Übergebene Element zu dieser Sammlung hinzu, falls noch Platz ist.
24       *
25       * @param pElement
26       *           Das einzufügende Element.
27       * @throws PushPopyException
28       *            Falls kein Platz mehr in dieser Sammlung vorhanden ist.
```

```
29        * @throws IllegalArgumentException
30        *             Falls das gegebene Element den Wert {@code null} hat.
31        */
32
33    @Override
34    public void push(E pElement) {
35      if(pElement == null){
36        throw new IllegalArgumentException("Ungültige Eingabe.");
37      }
38      if(this.isFull())
39      {
40        throw new PushPopyException("Die Sammlung ist voll.");
41      }
42      for(int i=0; i < array.length; i++){
43            if(array[i] == null){
44              array[i] = pElement;
45              return;
46            }
47      }
48
49    }
50
51
52
53    /**
54        * Gibt an, ob diese Sammlung leer ist.
55        *
56        * @return {@code true} falls diese Sammlung leer ist, ansonsten {@code false}.
57        */
58
59    @Override
60    public boolean isEmpty() {
61      for(int x = 0; x < array.length; x++){
62         if(array[x]!= null){
63           return false;
64         }
65      }
66      return true;
67    }
68
69    /**
70        * Gibt an, ob sich in dieser Sammlung bereits maximal viele Elemente befinden.
71        *
72        * return {code true} falls diese Sammlung maximal viele Elemente enthält,
73        *          ansonsten {code false}.
74        */
75
76    Override
77    public boolean isFull() {
78      int s =  array.length - 1;
79        if(array[s]== null){
80        return false;
81        }
82        else{
83      return true;
84        }
85    }
86      /**
87       * Die Methode entfernt einzelne Elemente aus der Sammlung. In dieser Methode wird
            immer das
88       * jüngste Elemente zuerst wieder entfernt.
89       *
90       * @return Das zu entfernende Element.
91       *
92       * @param juengstesElement
93       *         Das zu entferndende Element.
94       */
95    @Override
96    public E pop() {
97
98      E juengstesElement = (E) new Object();
```

```
99
100       if(this.isEmpty()){
101         throw new PushPopyException("Die Sammlung ist leer!");
102       }
103       for( int d = array.length - 1; d >= 0; d--){
104         if(array[d] != null){
105           juengstesElement = array[d];
106           array[d] = null;
107           return juengstesElement;
108         }
109     }
110       return juengstesElement;
111
112   }
113 }
```

Die Klasse ClassFiFo implementiert das Interface FiFoInterface und überschreibt ebenfalls die in PushPopyInterface gegebenen Methoden, sowie eine Methode aus FiFoInterface. Die pop-Methode soll das älteste Element zuerst entfernen. Eine Schleife durchläuft das Array und prüft ab der ersten Stelle des Arrays, welches Element zuerst nicht null ist. Dieses Element muss das bisher Älteste sein und wird entfernt.

```
1  package pi.uebung01.spec;
2
3  public class ClassFiFo<E> implements FiFoInterface<E> {
4    private E[] array;
5    private int size;
6
7    public ClassFiFo(final int bsize){
8
9      size = bsize;
10     /**@SuppressWarnings("unchecked")
11         final E[] array = (E[]) Array.newInstance(null, bsize);
12         this.array = array; **/
13
14         @SuppressWarnings("unchecked")
15         final E[] array = (E[])new Object[size];
16         this.array = array;
17   }
18    /**
19       * Fügt das Übergebene Element zu dieser Sammlung hinzu, falls noch Platz ist.
20       *
21       * @param pElement
22       *            Das einzufügende Element.
23       * @throws PushPopyException
24       *             Falls kein Platz mehr in dieser Sammlung vorhanden ist.
25       * @throws IllegalArgumentException
26       *             Falls das gegebene Element den Wert {@code null} hat.
27       */
28
29   @Override
30   public void push(E pElement) {
31     if(pElement == null){
32       throw new IllegalArgumentException("Ungültige Eingabe.");
33     }
34     if(this.isFull())
35     {
36       throw new PushPopyException("Die Sammlung ist voll.");
37     }
38     for(int i=0; i < array.length; i++){
39           if(array[i] == null){
40             array[i] = pElement;
41             return;
42           }
43     }
44
45   }
46
47
48
```

```
49   /**
50       * Gibt an, ob diese Sammlung leer ist.
51       *
52       * @return {@code true} falls diese Sammlung leer ist, ansonsten {@code false}.
53       */
54
55   @Override
56   public boolean isEmpty() {
57     for(int x = 0; x < array.length; x++){
58        if(array[x]!= null){
59          return false;
60        }
61     }
62     return true;
63   }
64
65   /**
66       * Gibt an, ob sich in dieser Sammlung bereits maximal viele Elemente befinden.
67       *
68       * @return {@code true} falls diese Sammlung maximal viele Elemente enthält,
69       *          ansonsten {@code false}.
70       */
71
72   @Override
73   public boolean isFull() {
74     int s =  array.length - 1;
75       if(array[s]== null){
76       return false;
77       }
78       else{
79     return true;
80       }
81   }
82     /**
83      * Die Methode entfernt einzelne Elemente aus der Sammlung. In dieser Methode wird
          immer das
84      * jüngste Elemente zuerst wieder entfernt.
85      *
86      * @return Das zu entfernende Element.
87      *
88      * @param juengstesElement
89      *         Das zu entferndende Element.
90      */
91   @Override
92   public E pop() {
93
94     E aeltestesElement = (E) new Object();
95
96       if(this.isEmpty()){
97         throw new PushPopyException("Die Sammlung ist leer!");
98       }
99       aeltestesElement = array[0];
100      for(int d = 1; d < array.length; d++ ){
101        array[d-1] = array[d];
102      }
103      array[array.length - 1] = null;
104      return aeltestesElement;
105
106  }
107 }
```

Aufgabe 2 Labyrinth

Aufgabe 2.1 Plättchen

```
1 package pi.uebung01.test;
2
```

```java
import static org.junit.Assert.assertEquals;

import org.junit.Test;

import pi.uebung01.Direction;
import pi.uebung01.Curve;

public class CurveTest {

  @Test
  public void testNorth() {
    Curve curve = new Curve(Direction.NORTH);
    assertEquals(curve.hasExit(Direction.NORTH), false);
    assertEquals(curve.hasExit(Direction.EAST), true);
    assertEquals(curve.hasExit(Direction.SOUTH), true);
    assertEquals(curve.hasExit(Direction.WEST), false);
  }

  @Test
  public void testEast() {
    Curve curve = new Curve(Direction.EAST);
    assertEquals(curve.hasExit(Direction.NORTH), false);
    assertEquals(curve.hasExit(Direction.EAST), false);
    assertEquals(curve.hasExit(Direction.SOUTH), true);
    assertEquals(curve.hasExit(Direction.WEST), true);
  }

  @Test
  public void testSouth() {
    Curve curve = new Curve(Direction.SOUTH);
    assertEquals(curve.hasExit(Direction.NORTH), true);
    assertEquals(curve.hasExit(Direction.EAST), false);
    assertEquals(curve.hasExit(Direction.SOUTH), false);
    assertEquals(curve.hasExit(Direction.WEST), true);
  }

  @Test
  public void testWest() {
    Curve curve = new Curve(Direction.WEST);
    assertEquals(curve.hasExit(Direction.NORTH), true);
    assertEquals(curve.hasExit(Direction.EAST), true);
    assertEquals(curve.hasExit(Direction.SOUTH), false);
    assertEquals(curve.hasExit(Direction.WEST), false);
  }
}
```

```java
/*
 * Copyright 2017 AG Softwaretechnik, University of Bremen, Germany
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 *   Unless required by applicable law or agreed to in writing, software
 *   distributed under the License is distributed on an "AS IS" BASIS,
 *   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *   See the License for the specific language governing permissions and
 *   limitations under the License.
 */
package pi.uebung01;

/**
 * Diese Klasse erbt von Tile und realisiert Kurvenplättchen.
 * Die Kurvenplättchen besitzen eine Orietierung, die die Ausrichtung
 * der Kurvenplättchen angibt.
 */
public final class Curve extends Tile {

   /**
```

```
26     * Erzeugt ein neues KurvenPlättchen mit der gegebenen Orientierung.
27     *
28     * @param pOrientation
29     *           Die Orientierung des neuen Kurvenplättchens.
30     */
31    public Curve(final Direction pOrientation) {
32       super(pOrientation);
33    }
34    /**
35     * Überprüft, ob das Kurvenplättchen in der aktuellen Orientierung einen
36     * Ausgang in die Übergebene Himmelsrichtung hat.
37     * Falls ja, wird (@code true} zurückegegeben, ansonsten {@code false}.
38     *
39     * @param pDirection
40     *       Die Himmelsrichtung, auf der geprüft wird, ob das Plättchen
41     *       dorthin einen Ausgang hat
42     * @return {@code true}, falls die angegebene Richtung entweder die nächste
43     *       oder Übernächste Richtung nach der aktuellen Orientierung ist,
44     *       sonst {@code false}
45     */
46    @Override
47    public boolean hasExit(final Direction pDirection) {
48      int orientation = getOrientation().getOrdinal();
49      if (pDirection.getOrdinal() == (orientation + 1) %4
50          || pDirection.getOrdinal() == (orientation + 2) %4) {
51         return true;
52      }
53      return false;
54    }
55
56 }
```

```
1  package pi.uebung01.test;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  import pi.uebung01.Junction;
8  import pi.uebung01.Direction;
9
10 public class JunctionTest {
11
12   @Test
13   public void testNorth() {
14     Junction junction = new Junction(Direction.NORTH);
15     assertEquals(junction.hasExit(Direction.NORTH), false);
16     assertEquals(junction.hasExit(Direction.EAST), true);
17     assertEquals(junction.hasExit(Direction.SOUTH), true);
18     assertEquals(junction.hasExit(Direction.WEST), true);
19   }
20
21   @Test
22   public void testEast() {
23     Junction junction = new Junction(Direction.EAST);
24     assertEquals(junction.hasExit(Direction.NORTH), true);
25     assertEquals(junction.hasExit(Direction.EAST), false);
26     assertEquals(junction.hasExit(Direction.SOUTH), true);
27     assertEquals(junction.hasExit(Direction.WEST), true);
28   }
29
30   @Test
31   public void testSouth() {
32     Junction junction = new Junction(Direction.SOUTH);
33     assertEquals(junction.hasExit(Direction.NORTH), true);
34     assertEquals(junction.hasExit(Direction.EAST), true);
35     assertEquals(junction.hasExit(Direction.SOUTH), false);
36     assertEquals(junction.hasExit(Direction.WEST), true);
37   }
38
39   @Test
```

```
40   public void testWest() {
41     Junction junction = new Junction(Direction.WEST);
42     assertEquals(junction.hasExit(Direction.NORTH), true);
43     assertEquals(junction.hasExit(Direction.EAST), true);
44     assertEquals(junction.hasExit(Direction.SOUTH), true);
45     assertEquals(junction.hasExit(Direction.WEST), false);
46   }
47 }
```

```
1  /*
2   * Copyright 2017 AG Softwaretechnik, University of Bremen, Germany
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *   http://www.apache.org/licenses/LICENSE-2.0
9   *
10  *   Unless required by applicable law or agreed to in writing, software
11  *   distributed under the License is distributed on an "AS IS" BASIS,
12  *   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  *   See the License for the specific language governing permissions and
14  *   limitations under the License.
15  */
16 package pi.uebung01;
17
18 /**
19  * Diese Klasse erbt von Tile und realisiert TPlättchen.
20  * Die TPlättchen besitzen eine Orietierung, die die Ausrichtung
21  * der TPlättchen angibt.
22  */
23 public final class Junction extends Tile {
24
25    /**
26     * Erzeugt ein neues TPlättchen mit der gegebenen Orientierung.
27     *
28     * @param pOrientation
29     *           Die Orientierung des neuen TPlättchens.
30     */
31    public Junction(final Direction pOrientation) {
32       super(pOrientation);
33    }
34    /**
35     * Überprüft, ob ein TPlättchen in der aktuellen Orientierung einen
36     * Ausgang in die gegebene Richtung hat.
37     * Falls dies der Fall ist, so wird {@code true} ausgegeben, sonst
38     * {@code false}.
39     *
40     * @param pDirection
41     *       Die Himmelsrichtung, bei der geprüft werden soll, ob
42     *       das TPlättchen einen Ausgang in diese Himmelsrichtung
43     *       besitzt
44     * @return {@code false}, falls die gegebene Himmelsrichtung die
45     *       aktuelle Orientierung des TPlättchens ist, ansonsten
46     *       wird {@code true} züruckgegeben.
47     */
48    @Override
49    public boolean hasExit(final Direction pDirection) {
50       return (pDirection.getOrdinal() != getOrientation().getOrdinal());
51    }
52
53 }
```

```
1 package pi.uebung01.test;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 import pi.uebung01.Straight;
8 import pi.uebung01.Direction;
```

```
9
10 public class StraightTest {
11
12   @Test
13   public void testNorth() {
14     Straight straight = new Straight(Direction.NORTH);
15     assertEquals(straight.hasExit(Direction.NORTH), true);
16     assertEquals(straight.hasExit(Direction.EAST), false);
17     assertEquals(straight.hasExit(Direction.SOUTH), true);
18     assertEquals(straight.hasExit(Direction.WEST), false);
19   }
20
21   @Test
22   public void testEast() {
23     Straight straight = new Straight(Direction.EAST);
24     assertEquals(straight.hasExit(Direction.NORTH), false);
25     assertEquals(straight.hasExit(Direction.EAST), true);
26     assertEquals(straight.hasExit(Direction.SOUTH), false);
27     assertEquals(straight.hasExit(Direction.WEST), true);
28   }
29
30   @Test
31   public void testSouth() {
32     Straight straight = new Straight(Direction.SOUTH);
33     assertEquals(straight.hasExit(Direction.NORTH), true);
34     assertEquals(straight.hasExit(Direction.EAST), false);
35     assertEquals(straight.hasExit(Direction.SOUTH), true);
36     assertEquals(straight.hasExit(Direction.WEST), false);
37   }
38
39   @Test
40   public void testWest() {
41     Straight straight = new Straight(Direction.WEST);
42     assertEquals(straight.hasExit(Direction.NORTH), false);
43     assertEquals(straight.hasExit(Direction.EAST), true);
44     assertEquals(straight.hasExit(Direction.SOUTH), false);
45     assertEquals(straight.hasExit(Direction.WEST), true);
46   }
47 }
```

```
1 /*
2  * Copyright 2017 AG Softwaretechnik, University of Bremen, Germany
3  *
4  * Licensed under the Apache License, Version 2.0 (the "License");
5  * you may not use this file except in compliance with the License.
6  * You may obtain a copy of the License at
7  *
8  *   http://www.apache.org/licenses/LICENSE-2.0
9  *
10 *   Unless required by applicable law or agreed to in writing, software
11 *   distributed under the License is distributed on an "AS IS" BASIS,
12 *   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 *   See the License for the specific language governing permissions and
14 *   limitations under the License.
15 */
16 package pi.uebung01;
17
18 /**
19  * Diese Klasse erbt von Tile und realisiert GeradenPlättchen.
20  * Die GeradenPlättchen besitzen eine Orietierung, die die Ausrichtung
21  * der GeradenPlättchen angibt.
22  */
23 public final class Straight extends Tile {
24
25    /**
26     * Erzeugt ein neues GeradenPlättchen mit der gegebenen Orientierung.
27     *
28     * @param pOrientation
29     *            Die Orientierung des neuen GeradenPlättchens.
30     */
31    public Straight(final Direction pOrientation) {
```

```
32        super(pOrientation);
33    }
34
35    /**
36     * Überpräft, ob ein GeradenPlättchen in der aktuellen Orientierung einen
37     * Ausgang in die gegebene Himmelsrichtung hat.
38     * Falls dies der Fall ist, so wird {@code true} ausgegeben, sonst
39     * {@code false}.
40     *
41     * @param pDirection
42     *        Die Himmelsrichtung, bei der geprüft werden soll, ob
43     *        das GeradenPlättchen einen Ausgang in diese Himmelsrichtung
44     *        besitzt
45     * @return {@code false}, falls die gegebene Himmelsrichtung die
46     *        aktuelle Orientierung oder die entgegen gesetzte Himmelsrichtung
47     *        des GeradenPlättchens ist, sonst {@code false}
48     */
49    @Override
50    public boolean hasExit(final Direction pDirection) {
51      int orientation = getOrientation().getOrdinal();
52      return (pDirection.getOrdinal() == orientation ||
53                pDirection.getOrdinal() == (orientation +2) %4);
54    }
55
56 }
```

Aufgabe 2.2 Spielfeld

Wir haben den Algorithmus, wie er geschildert war, implementiert. Hierbei werden die zwei Hilfsmethoden haspath und algorithmus verwendet. Die ArrayList fungiert hierbei als Kollektion.

```
1     @Override
2     public final boolean existsPath(final PieceInterface pPiece, final int pDestRow,
3           final int pDestCol) {
4        ClassLiFo<Tile> stack = new ClassLiFo<Tile>(49);
5        ClassFiFo<Tile> queue = new ClassFiFo<Tile>(49);
6        boolean a = this.algorithmus(stack, pPiece, pDestRow, pDestCol);
7        boolean b = this.algorithmus(queue, pPiece, pDestRow, pDestCol);
8        if(a != b){
9          throw new IllegalStateException();
10       }
11       else{
12         return a;
13       }
14
15    }
16
17    /**
18     * Die Methode erzeugt zunächst eine Sammlung vom Typ PushPopyInterface für die noch
        zu bearbeitenden Plättchen.
19     * Jetzt wird das Plättchen, auf dem sich die gegebene Spielfigur befindet, in die
        Sammlung Rand eingefügt.
20     * Nun beginnt eine Schleife, die so oft wiederholt wird, wie sich noch Elemente in
        der Sammlung Rand befinden.
21     * Innerhalb der Schleife wird mittels pop() ein Plättchen P aus dem Rand entnommen.
22     * Wenn dieses Plättchen bereits das Ziel ist, wird true zurückgegeben.
23     * Anderenfalls werden für P alle Nachbarplättchen, zu denen es einen Weg gibt, in die
         Sammlung Rand eingefügt.
24     * Falls das Zielplättchen nicht erreichbar ist, wird false zurückgegeben.
25     * Außerdem zählt ein Zähler die Schleifendurchläufe mit.
26     * @param rand die Sammlung Rand
27     * @param p das Feld, auf dem die Spielfigur zurzeit steht
28     * @param reihe die Reihe, in der sich das Zielfeld befindet
29     * @param spalte die Spalte, in der sich das Zielfeld befindet
30     * @return der boolsche Wert, der besagt, ob es einen Pfad von dem aktuellen Plättchen
         bis zum Zielfeld gibt
31     */
32    public boolean algorithmus(PushPopyInterface<Tile> rand, PieceInterface p, int reihe,
        int spalte){
33      rand.push(p.getTile());
```

```
34      int zaehler = 0;
35
36      List<Tile> checkedAlready = new ArrayList<>();
37
38      while(rand.isEmpty() == false){
39        ++zaehler;
40        Tile gepopt = rand.pop();
41        if((gepopt.getRow() == reihe) && (gepopt.getColumn()== spalte)) {
42          System.out.println("zaehler = " + zaehler);
43          return true;
44        }
45        else{
46          if((gepopt.getRow() != 0) && (this.hasPath(gepopt, tiles[gepopt.getRow() -1][
              gepopt.getColumn()]))){
47            if(!checkedAlready.contains(tiles[gepopt.getRow() -1][gepopt.getColumn()])){
48              rand.push(tiles[gepopt.getRow() -1][gepopt.getColumn()]);
49            }
50          }
51          if((gepopt.getColumn() != 0) && (this.hasPath(gepopt, tiles[gepopt.getRow() ][
              gepopt.getColumn() -1]))){
52
53            if(!checkedAlready.contains(tiles[gepopt.getRow() ][gepopt.getColumn() -1])){
54              rand.push(tiles[gepopt.getRow() ][gepopt.getColumn() -1]);
55            }
56          }
57          if((gepopt.getRow() != 6) && (this.hasPath(gepopt, tiles[gepopt.getRow() +1][
              gepopt.getColumn()]))){
58            if(!checkedAlready.contains(tiles[gepopt.getRow() +1][gepopt.getColumn()])){
59              rand.push(tiles[gepopt.getRow() +1][gepopt.getColumn()]);
60            }
61          }
62          if((gepopt.getColumn() != 6) && (this.hasPath(gepopt, tiles[gepopt.getRow() ][
              gepopt.getColumn() +1]))){
63            if(!checkedAlready.contains(tiles[gepopt.getRow() ][gepopt.getColumn() +1])){
64              rand.push(tiles[gepopt.getRow() ][gepopt.getColumn() +1]);
65            }
66          }
67
68          checkedAlready.add(gepopt);
69        }
70      }
71      System.out.println("zaehler = " + zaehler);
72      return false;
73    }
74
75    /**
76     * die Methode prüft, ob 2 Teile a und b eine Verbindung zueinander haben, indem sie
          die vier Fälle durchgeht,
77     * dass Plättchen b oberhalb, rechts von, unterhalb oder links von dem Plättchen a
          liegt.
78     * @param a ein beliebiges Plättchen
79     * @param b ein Nachbarplättchen von a
80     * @return der boolsche Wert, der besagt, ob es eine Verbindung von a nach b gibt
81     */
82
83    public boolean hasPath(Tile a, Tile b){
84      if ((a.getRow()  == b.getRow()) && (a.getColumn() +1 == b.getColumn()) && a.hasExit(
          EAST) && b.hasExit(WEST)){
85        return true;
86      }
87      if ((a.getRow() +1 == b.getRow()) && (a.getColumn() == b.getColumn()) && a.hasExit(
          SOUTH) && b.hasExit(NORTH)){
88        return true;
89      }
90      if ((a.getRow()  == b.getRow()) && (a.getColumn() == b.getColumn() +1) && a.hasExit(
          WEST) && b.hasExit(EAST)){
91        return true;
92      }
93      if ((a.getRow()  == b.getRow() +1) && (a.getColumn() == b.getColumn()) && a.hasExit(
          NORTH) && b.hasExit(SOUTH)){
94        return true;
```

```
95      }
96      else{
97        return false;
98      }
99    }
```

Es fällt auf, dass das LiFoInterface meist schneller zum Ziel gelangt, als das FiFoInerface. Dies liegt daran, dass der Stack sozusagen erstmal einen Weg ausprobiert und dann den nächsten wählt, wenn dieser nicht funktioniert hat. Die Queue probiert sozusagen alle Wege gleichzeitig. Bei längeren und sehr verzweigten Wegen kann dies sehr lange dauern.