Praktische Informatik I | WS 2012/13 | Vitalij Kochno
TutorIn: Jan Malburg | Gruppe: 3 | Yorick Netzer
Christophe Stilmant

Übungsblatt 6

Aufgabenlösung
Abgabe: 02.12.2012

Aufgabe 1 Der Herr der Tests

Aufgabe 1.1 Testfälle definieren

```
1  import static org.junit.Assert.*;
2  import org.junit.After;
3  import org.junit.Before;
4  import org.junit.Test;
5
6  /**
7   * Die Testklasse RingBufferTest.
8   *
9   * @author Vitalij Kochno - Yorick Netzer - Christophe Stilmant
10  * @version 27-11-2012
11  */
12 public class RingBufferTest
13 {
14     private RingBuffer ring;
15     int laenge;
16
17     /**
18      * Default constructor for test class RingBufferTest
19      */
20     public RingBufferTest()
21     {
22     }
23
24     /**
25      * Sets up the test fixture.
26      *
27      * Called before every test case method.
28      */
29     @Before
30     public void setUp()
31     {
32         laenge = 6;
33         ring = new RingBuffer(laenge);
34     }
35     /**
36      * Test1
37      * Die zu Beginn sollten keine Elemente in dem gerade erzeugten Ringbuffer drin sein
38      */
39     @Test
40     public void test1()
41     {
42         assertEquals(0,ring.size());
43     }
44     /**
45      * Test2
46      * Fügt ein Element dem Ringbuffer hinzu und überprüft dann die Größe des Ringbuffers
          . Diese sollte nun 1.
47      */
48     @Test
49     public void test2()
50     {
```

```
51          ring.push(1);
52          assertEquals(1,ring.size());
53      }
54      /**
55       * Test3
56       * Fügt ein neues Element dem Ringbuffer hinzu.
57       * Da der Ringbuffer gerade erzuegt wurde ist dieses zugleich auch das älteste Ele
           ment und wird mittels ring.pop() entnommen.
58       */
59      @Test
60      public void test3()
61      {
62          ring.push(2);
63          assertEquals(2,ring.pop());
64      }
65      /**
66       * Test4
67       * Füllt den Ringbuffer mit Zahlen in aufsteigender Reihenfolge.
68       * Danach wird einmal die toString Methode bei der Ausgabe in der Konsole getestet (
           nicht automatisiert)
69       * und es wird geschaut ob das älteste Element 0 ist ( da die 0 als erstes hinzugefü
           gt wurde).
70       */
71      @Test
72      public void test4()
73      {
74          //ring= new RingBuffer(laenge);
75          for(int i=0;i<laenge;i++){
76              ring.push(i);
77          }
78          System.out.println(ring);
79          assertEquals(0,ring.peek());
80
81      }
82      /**
83       * Test5
84       * Füllt den Ringbuffer mit Zahlen in aufsteigender Reihenfolge und fügt danach noch
           "11" und "12" hinzu.
85       * Der Ringbuffer sollte jetzt immernoch die länge laenge haben.
86       */
87      @Test
88      public void test5()
89      {
90          for(int i=0;i<laenge;i++){
91              ring.push(i);
92          }
93          ring.push(11);
94          ring.push(12);
95          //System.out.println(ring);
96          assertEquals(laenge,ring.size());
97      }
98      /**
99       * Test 6
100      * Diesmal wird ein Ringbuffer mit der Länge 25 verwendet.
101      * Diesem Ringbuffer werden nacheinander alle Zahlen von 0 bis 99 hinzugefügt.
102      * Hier sollte der Ringbuffer "überlaufen" und nur noch die letzten 25 Elemente
           enthalten.
103      * Dies wird in der zweiten for-Schleife auch überprüft, indem bei jedem
           Schleifendurchlauf geschaut wird, ob die Länge
104      * des Ringbuffers mit jedem Schleifendurchlauf abnimmt ( man nutzt bei der 2ten
           Anweisung in der Schleife pop(), wodurch
105      * das Element aus der Schleife entfernt werden sollte) und überprüft ob tatsächlich
           die 75 bis einschließlich 99 enthält.
106      *
107      * Zusätzlich wird zwischen den beiden Schleifen die toString() Methode getestet. Auß
           erdem kann man so schauen ob tatsächlich
108      * die Elemente im Ringbuffer sind, die da auch sein sollen.
109      */
110     @Test
111     public void test6()
112     {
```

```
113         ring = new RingBuffer(25);
114         for(int i = 0; i<100;i++)
115         {
116             ring.push(i);
117         }
118
119         System.out.println(ring);
120         for(int i=25;i>0;i--)
121         {
122             assertEquals(i,ring.size());
123             assertEquals(100-i,ring.pop());
124
125         }
126         //System.out.println(ring.pop());
127         /*
128          * Hier gibs n Problem, weil der Wert noch da ist bei ring . pop , aber das kann
              man theoretisch ignorieren
129          * oder man baut bei pop und peek ein if ein das da wegen der länge aufpasst
130          */
131     }
132
133     /**
134      * Tears down the test fixture.
135      *
136      * Called after every test case method.
137      */
138     @After
139     public void tearDown()
140     {
141     }
142 }
```

- Test1: Um zu sehen ob die Array wirklich leer ist.
- Test2: Schauen ob die Array grösser wird.
- Test3: Überprüft das älteste Element.
- Test4: Fügt eine Liste von 0 bis i dazu, und schaut ob 0 das älteste ist.
- Test5: Array wird überfüllt und die Länge sollte aber gleich bleiben
- Test6: Hier wollen die Reihefolge überprüfen, ob diese auch eingehalten wird.

Aufgabe 1.2 Implementierung

```
16 public class RingBuffer
17 {
18     /**
19      * Representiert der Buffer. In diese Tabelle werden alle Werte gespeichert.
20      */
21     private int[] ring;
22     /**
23      * Representiert das Element der Tabelle, mitdem wir gerade arbeiten. Diese Attribut
24      * erlaubt uns, der älteste Elemente der Tabelle zu finden.
25      */
26     private int i;
27
28     /**
29      * Definiert die Größe der Tabelle, und dann auch die maximale Anzahl von Einträgen,
30      * die gepuffert werden können.
31      */
32     private int capacity;
33
34     /**
35      * Definiert wieviele Elemente es schon in der Tabelle ring[] gibt.
36      */
```

```
37      private int lenght;
38
39      /**
40       * Erzeugt einen Ringpuffer.
41       * @param capacity Die maximale Anzahl von Einträgen, die gepuffert werden können.
42       */
43      public RingBuffer(int capacity)
44      {
45          ring = new int[capacity];
46          this.capacity = capacity;
47          i = 0;
48          lenght = 0;
49      }
50
51      /**
52       * Fügt ein neues Element in den Ringpuffer ein.
53       * @param value Der Wert, der eingefügt werden soll.
54       */
55      public void push(int value)
56      {
57          ring[i] = value;
58          i++;
59          if(lenght!=capacity)
60          {
61              lenght++;
62          }
63          checki();
64      }
65
66      /**
67       * Entnimmt das älteste Element aus dem Ringpuffer.
68       * @return Das Element, das entnommen wurde.
69       */
70      public int pop()
71      {
72          /* Da das Element von der Tabelle gelöcht sein wird, muss man das
73           * Element in ein Temporale Variable tmp hinzufügen, um den Wert
74           * zurücksenden. */
75          int tmp;
76
77          if(size() <= capacity)
78          {
79              tmp = ring[0];
80              for(int j = 0; j < lenght - 1; j++)
81              {
82                  ring[j] = ring[j+1];
83              }
84          }
85          else
86          {
87              tmp = ring[i];
88              for(int j = i; j < lenght - 1; j++)
89              {
90                  ring[j] = ring[j+1];
91              }
92              ring[lenght-1] = ring[0];
93              for(int k = 0; k < i; k++)
94              {
95                  ring[k] = ring[k+1];
96              }
97          }
98          i--;
99          checki();
100         lenght--;
101         return tmp;
102     }
103
104     /**
105      * Liefert das älteste Element aus dem Ringpuffer zurück, ohne es zu entnehmen.
106      * @return Das älteste Element im Ringpuffer.
107      */
```

```
108     public int peek()
109     {
110         if(size() <= capacity)
111         {
112             return ring[0];
113         }
114         else
115         {
116             return ring[i];
117         }
118     }
119 
120     /**
121      * Liefert die Anzahl der Elemente zurück, die sich im Puffer befinden, d.h. die
122      * mit {@link pop()} entnommen werden könnten.
123      * @return Die Anzahl der belegten Einträge im Puffer.
124      */
125     public int size()
126     {
127        return lenght;
128     }
129 
130     /**
131      * Überprüft, ob der Attribut i sich in ein gültige Intervall befindet.
132      */
133     private void checki()
134     {
135         if(i >= capacity)
136         {
137             i = 0;
138         }
139         if(i < 0)
140         {
141             i = capacity - 1;
142         }
143     }
144 
145     /**
146      * Liefert alle Elemente von der Buffer zurück.
147      * @return Eine String-Liste aller Elemente von der Buffer.
148      */
149     public String toString()
150     {
151         String answer = "{ ";
152         for(int j = 0; j < size(); j++)
153         {
154             answer =  answer + ring[j]+ ", ";
155         }
156         answer = answer.substring(0, answer.length() - 2) + " }";
157         return answer;
158     }
159 
160     /**
161      * Sagt, ob der Buffer leer ist oder nicht.
162      * @return true falls der Buffer leer ist, false sonst.
163      */
164     public boolean isEmpty()
165     {
166         return lenght==0;
167     }
168 }
```

Aufgabe 2 Spiel's noch einmal, Sam

Hier definieren wir drei RingBuffer für x,y und die Rotation.

```
77     /**
78      * Ringbuffer für Koordiaten und drehung
79      */
80     private RingBuffer ringX , ringY , ringRot;
```

```
81
82      /**
83       * Diese Konstante enthält die Grösse der 3 Ringpuffers von der Rackette.
84       */
85      private final int CAPACITY_RINGBUFFER = 1000;
86
87
88      public Rocket()
89      {
90          //init(getX(),getY());
91          super(false);
92          makePlaylist("bomb-1.wav,bomb-2.mp3,");
93          pose = new Pose(this);
94          inventory = new Inventory();
95
96          ringX =  new RingBuffer(CAPACITY_RINGBUFFER);
97          ringY =  new RingBuffer(CAPACITY_RINGBUFFER);
98          ringRot =  new RingBuffer(CAPACITY_RINGBUFFER);
99      }
100
101     /**
102      * Konstruktor. Herstellt ein Objekt mit eine benutzergewünschte Playliste.
103      */
104     public Rocket(String playlist)
105     {
106         super(false);
107         makePlaylist(playlist);
108         pose = new Pose(this);
109         inventory = new Inventory();
110
111         ringX =  new RingBuffer(CAPACITY_RINGBUFFER);
112         ringY =  new RingBuffer(CAPACITY_RINGBUFFER);
113         ringRot =  new RingBuffer(CAPACITY_RINGBUFFER);
114     }
```

Anschliessend führen wir sie aus. Mit ”r” wird das Replay abgespielt.

```
155         if(!Greenfoot.isKeyDown("r"))
156         {
157           ringX.push(getRealX());
158           //System.out.println("push "+ getX() + " real : " + getRealX());
159           ringY.push(getRealY());
160           ringRot.push(getRotation());
161         }
162         else if(!ringX.isEmpty())
163         {
164             setLocation(ringX.pop() + getScrWorld().getShiftX(), ringY.pop() +
                  getScrWorld().getShiftY());
165             setRotation(ringRot.pop());
166         }
```