Übungsblatt 12

Musterlösung

Aufgabe 1 Fernsteuernd (40%)

Die Klasse *RemotePlayer* setzt die Aufgabenstellung wie auf die Übungsblatt beschrieben um. Ihr Konstruktor bekommt zusätzlich zu den Parametern, die der Konstruktor der Basisklasse *Player* hat, die IP-Adresse übergeben, unter der er das Spiel mit dem *ControlledPlayer* finden kann, sowie die Portnummer übergeben, auf der dieses auf eine Kontaktaufnahme wartet. In der Methode *act* wird einfach das *act* der Klasse *Player* ausgeführt. Danach zeigt die Rotation an, in welche Richtung sich die Figur bewegt haben muss. Dies wird per Netzwerk übertragen. Alle auftretenden Ausnahmen werden so gefangen, dass eine Fehlermeldung auf der Konsole ausgegeben (außer beim Schließen) und die Figur unsichtbar gemacht wird. Das unsichtbar Machen bewirkt, dass der Socket geschlossen wird und das Spiel letztendlich anhält.

```
1  import java.io.IOException;
2  import java.net.InetSocketAddress;
3  import java.net.Socket;
4
5  /**
6   * Diese Klasse definiert die Figur, die von der Spieler:in gesteuert
7   * wird. Die Figur bewegt sich dabei auf der Gitterstruktur des
8   * Spielfeldes. Zusätzlich wird ihre Bewegung per Netzwerk zu übertragen,
9   * um dieselbe Figur in einer entfernten Instanz des Spiels fernzusteuern.
10  *
11  * @author Thomas Röfer
12  */
13 class RemotePlayer extends Player
14 {
15     /** Der Socket, über den die Befehle geschickt werden. */
16     private final Socket socket = new Socket();
17
18     /**
19      * Erzeugen und Anzeigen einer neuen Spielfigur.
20      * @param x Die x-Koordinate dieser Spielfigur im Gitter.
21      * @param y Die y-Koordinate dieser Spielfigur im Gitter.
22      * @param rotation Die Rotation dieser Spielfigur (0 = rechts ... 3 = oben).
23      * @param field Das Spielfeld, auf dem sich diese Spielfigur bewegt.
24      * @param address Die IP-Adresse des Rechners mit der ferngesteuerten Figur.
25      * @param port Der Portauf dem Rechner mit der ferngesteuerten Figur.
26      */
27     RemotePlayer(final int x, final int y, final int rotation, final Field field,
28             final String address, final int port)
29     {
30         super(x, y, rotation, field);
31         try {
32             socket.connect(new InetSocketAddress(address, port));
33         }
34         catch (final IOException e) {
35             System.err.println("Kann mit nicht mit " + address + ":" + port
36                     + " verbinden.");
37             setVisible(false);
38         }
39     }
40
41     /**
42      * Die Spielfigur bewegt sich entsprechend der von der Spieler:in
43      * bewegten Tasten. Diese Methode kehrt erst zurück, wenn ein
44      * gültiger Zug gemacht wurde. Sie überträgt zudem die Bewegungsrichtung
45      * per Netzwerk.
46      */
47     @Override
```

```
48      void act()
49      {
50          super.act();
51          try {
52              socket.getOutputStream().write(getRotation());
53              socket.getOutputStream().flush();
54          }
55          catch (final IOException e) {
56              System.err.println("Kann Bewegung nicht senden.");
57              setVisible(false);
58          }
59      }
60
61      /**
62       * Macht dieses Spielobjekt sichtbar bzw. versteckt es. Wenn es
63       * versteckt wird, wird zudem die Netzwerkverbindung geschlossen.
64       * @param visible Soll das Objekt sichtbar in der Zeichenfläche sein?
65       */
66      @Override
67      public void setVisible(final boolean visible)
68      {
69          super.setVisible(visible);
70          if (!visible) {
71              try {
72                  socket.close();
73              }
74              catch (final IOException e) {
75                  // Ignorieren
76              }
77          }
78      }
79  }
```

Aufgabe 2 Ferngesteuert (40%)

Der *ControlledPlayer* ist ganz ähnlich implementiert, insbesondere was die Fehlerbehandlung angeht. Die Überladung von *setVisible* ist fast identisch. Einziger Unterschied ist, dass *socket* hier auch *null* sein könnte, wenn noch keine Verbindung aufgebaut wurde. Da hier ein Server implementiert wird, benötigt der Konstruktor keine IP-Adresse. Zum Öffnen des Server-Sockets wird *try-with-resources* benutzt, weshalb dieser bereits vor Ende des Konstruktors wieder geschlossen wird. In der Methode *act* werden die Bewegungsrichtungen einfach als Bytes mit den Werten 0-3 empfangen und die Figur dann entsprechend bewegt. Zumindest auf dem Mac liefert das Lesen aus einer von der Gegenseite geschlossenen Verbindung -1 zurück. Dies wird auch durch das Verstecken der Figur behandelt.

```
1  import java.io.IOException;
2  import java.net.InetSocketAddress;
3  import java.net.ServerSocket;
4  import java.net.Socket;
5
6  /**
7   * Diese Klasse definiert eine Figur, die von einer Instanz der
8   * Klasse RemotePlayer per Netzwerk ferngesteuert wird.
9   *
10  * @author Thomas Röfer
11  */
12 class ControlledPlayer extends Player
13 {
14      /** Der Socket, über den die Befehle empfangen werden. */
15      private Socket socket = null;
16
17      /**
18       * Erzeugen und Anzeigen einer neuen Spielfigur.
19       * @param x Die x-Koordinate dieser Spielfigur im Gitter.
20       * @param y Die y-Koordinate dieser Spielfigur im Gitter.
21       * @param rotation Die Rotation dieser Spielfigur (0 = rechts ... 3 = oben).
22       * @param field Das Spielfeld, auf dem sich diese Spielfigur bewegt.
```

```
23      * @param port Auf diesem Port wird auf eine Verbindung gewartet.
24      */
25     ControlledPlayer(final int x, final int y, final int rotation, final Field field,
26             final int port)
27     {
28         super(x, y, rotation, field);
29         try (final ServerSocket server = new ServerSocket()) {
30             server.bind(new InetSocketAddress(port));
31             socket = server.accept();
32         }
33         catch (final IOException e) {
34             System.err.println("Akzeptieren von Verbindung auf Port " + port
35                     + " fehlgeschlagen.");
36             setVisible(false);
37         }
38     }
39
40     /**
41      * Die Spielfigur bewegt sich entsprechend der aus dem Netzwerk
42      * empfangenen Richtung.
43      */
44     @Override
45     void act()
46     {
47         try {
48             final int direction = socket.getInputStream().read();
49             if (direction == -1) {
50                 System.err.println("Verbindung wurde beendet.");
51                 setVisible(false);
52             }
53             else {
54                 setRotation(direction);
55                 final int[][] offsets = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
56                 setLocation(getX() + offsets[getRotation()][0], getY() + offsets[
                            getRotation()][1]);
57                 playSound("step");
58                 sleep(200);
59             }
60         }
61         catch (final IOException e) {
62             System.err.println("Kann Bewegung nicht empfangen.");
63             setVisible(false);
64         }
65     }
66
67     /**
68      * Macht dieses Spielobjekt sichtbar bzw. versteckt es. Wenn es
69      * versteckt wird, wird zudem die Netzwerkverbindung geschlossen.
70      * @param visible Soll das Objekt sichtbar in der Zeichenfläche sein?
71      */
72     @Override
73     public void setVisible(final boolean visible)
74     {
75         super.setVisible(visible);
76         if (!visible && socket != null) {
77             try {
78                 socket.close();
79             }
80             catch (final IOException e) {
81                 // Ignorieren
82             }
83         }
84     }
85 }
```

Aufgabe 3 Spielend (20%)

Der Konstruktor der Klasse *Level* bekommt nun zusätzlich die IP-Adresse und den Port übergeben. Ist die IP-Adresse *null*, wird als *Player* ein *ControlledPlayer* erzeugt, ansonsten ein *Remo-*

tePlayer:

```
54      Level(final String fileName, final String address, final int port)

76          final Player player = address == null
77                  ? new ControlledPlayer(-1, 0, 0, field, port)
78                  : new RemotePlayer(-1, 0, 0, field, address, port);
```

In der Klasse *PI1Game* hat die Methode *main* nun ebenfalls die IP-Adresse und den Port als Parameter, mit derselben Bedeutung. Diese werden an den *Level* durchgereicht:

```
18      static void main(final String address, final int port)

21          final Level level = new Level("levels/1.lvl", address, port);
```

Dadurch kann nun in zwei BlueJs das Spiel gestartet werden. Beim ersten wird *null* als Adresse übergeben (und z.B. 9999 als Port). Beim zweiten wird 127.0.0.1 als Adresse verwendet (*localhost*) und derselbe Port. Mit der zweiten Instanz können nun beide gesteuert werden, die sich identisch verhalten. Wird die zweite Instanz vorzeitig beendet, liefert sie die entsprechende Fehlermeldung und beendet sich auch. Wird hingegen die erste Instanz vorzeitig beendet, können noch ein paar Bewegungen in der zweiten ausgeführt werden, bevor sich diese ebenfalls mit einer Fehlermeldung beendet.