Praktische Informatik I | WiSe 2021/22
Tutor:in: Leander Staack | Bearbeiter:in: Julius Walczynski [6113829]

Übungsblatt 12

Lösungsvorschlag

Aufgabe 1 Ferngesteuert

```
9  public class ControlledPlayer extends Player {
10
11      private ServerSocket server;
12      private Socket client;
13      final private int PORT = 1234;
14      public ControlledPlayer(final int x, final int y, int rotation, final Field field) {
15          super(x, y, rotation, field);
16          try{
17              this.server = new ServerSocket();
18              this.server.bind(new InetSocketAddress(this.PORT));
19
20              this.client = this.server.accept();
21          }catch(BindException e){
22              System.out.println("port is already in use");
23          }catch(SecurityException e){
24              System.out.println("not allowed to open server on given port");
25          }catch(IOException e){
26              System.out.println("error while creating server");
27          }
28      }
```

Die Klasse *ControlledPlayer* erbt von *Player* und gibt in ihrem Konstruktor alle erforderlichen Parameter an die Player Klasse weiter. Ein neuer *ServerSocket* wird auf einem spezifizierten Port (1234) geöffnet und es wird auf die Verbindung des ersten Clients/*Socket*s gewartet. Da der *ControlledPlayer* im Konstruktor der *Level* Klasse instanziiert wird, wird das Spiel, auf Grund des blockierenden Verhaltens der *ServerSocket::accept* Methode, erst nach einer erfolgreichen Verbindung vollständig gestartet. Sowohl der *ServerSocket* (*server*), als auch der erste *Socket* (*client*), der sich verbindet, werden als Attribute abgespeichert um im Folgenden weiter auf sie zugreifen zu können.

Beim erstellen des *ServerSocket*s - z.B. während versucht wird ihn an einen freien Port zu binden - können diverse Probleme/*Exception*s auftreten. In diesem Fall insbesondere *IOException, BindException, SecurityException* Sollten diese auftreten, werden sie abgefangen und entsprechende Fehlernachricht in die Konsole ausgegeben.

```
30      @Override
31      public void act() {
32          try{
33              InputStream in = this.client.getInputStream();
34
35              byte[] buffer = new byte[32];
36              if (in.read(buffer) > 0){
37                int rotation = (int) buffer[0];
38                super.Move(rotation);
39              }
40          }catch(IOException e){
41              System.out.println("error while reading data");
42          }
43      }
```

Die *act* Methode (von *Actor* bzw *Player* vererbt) wird überschrieben. Dabei wird ihre Basis-Logik nicht ausgeführt; es wird also nicht auf Userinput gewartet. Stattdessen werden über den *InputStream* des *client*s die über das Netzwerk gesendeten Daten ausgelesen. Hierbei blockiert die *InputStream::read* Methode solange, bis neue Daten vorliegen und schreibt diese in den als Parameter gegebenen *byte[ ]*.

Die Bewegung wird als Bewegungsrichtung (0-3) als einziges Byte übertragen.

Sollte beim auslesen der Daten ein Problem/eine *IOException* auftreten, wird diese abgefangen und eine entsprechende Fehlernachricht wird in die Konsole ausgegeben.

Sollte das erste *Byte* erfolgreich ausgelesen worden sein, wird dieses benutzt, um den Spieler mit Hilfe der *Player::Move* Methode zu bewegen.

```
45      @Override
46      void setVisible(final boolean visible){
47          super.setVisible(visible);
48          if(!visible){
49              try{
50                  this.client.close();
51                  this.server.close();
52              } catch (IOException e){
53                  System.out.println("unable to close socket");
54              }
55          }
56      }
```

Die von *GameObject* geerbte Methode *setVisible* wird überschrieben. Ihre Basis-logik, wird mit Hilfe von *super.setVisible* jedoch weiterhin ausgeführt. Sollte das *GameObject* hierbei auf unsichtbar gesetzt werden (in unserem simplen Spiel tritt dies nur auf, falls das Spiel beendet wird/ der Spieler einen NPC berührt und stirbt), werden sowohl die Verbindung zum Client über den *Socket* und anschließend der *SocketServer* selbst geschlossen.

Sollte beim Schließen ein problem/eine *IOException* auftreten, wird diese abgefangen und als sinvolle Fehlermeldung in der Konsole ausgegeben.

Aufgabe 2 Fernsteuernd

```
9  public class RemotePlayer extends Player {
10
11     private final String IP = "127.0.0.1";
12     private final int PORT = 1234;
13
14     private Socket socket;
15     public RemotePlayer(final int x, final int y, int rotation, final Field field) {
16         super(x, y, rotation, field);
17
18         try{
19             this.socket = new Socket();
20             this.socket.connect(new InetSocketAddress(IP, PORT));
21         }catch(ConnectException e){
22             System.out.println("unable to connect to server");
23         }catch(IOException e){}
24     }
```

Ähnlich zu *ControlledPlayer* erbt die Klasse *RemotePlayer* von *Player* und übergibt durch seinen eigenen Konstruktor alle essenziellen Parameter an die *Player* Klasse.

Um eine Verbindung zu einer Instanz des *ControlledPlayer* herzustellen, wird ein neuer *Socket* instanziiert und geöffnet. Um diesen mit dem Server zu verbinden, muss die *Socket::connect* Methode aufgerufen werden. Diese benötigt die IP-Adresse und den Port des Servers. Demnach wird die selbe Portnummer, die zuvor im *ControlledPlayer* festgelegt wurde übergeben: 1234. Da das Spiel zum Testen in mehreren Instanzen auf dem gleichen Gerät geöffnet wurde, wird "127.0.0.1"bzw. localhost als IP-Adresse verwendet.

Sollten beim Verbindungsaufbau Probleme/eine *IOException* oder *ConnectException* auftreten, wird diese abgefangen und eine entsprechende Fehlernachricht in der Konsole ausgegeben.

```
26     @Override
27     public void act(){
28         super.act();
29
30         try{
31             OutputStream out = this.socket.getOutputStream();
32             out.write(new byte[]{ (byte)super.getRotation() });
33             out.flush();
34         }catch(IOException e){
35             System.out.println("unable to write data");
36         }
37     }
```

Ähnlich zu *ControlledPlayer* wird die *Player::act* Methode überschrieben. Hierbei wird die Basis-Logik jedoch weiterhin durch *super.act()* aufgerufen uns ausgeführt. Da diese solange blockiert bis der User eine Bewegungstaste gedrückt/der *Player* sich wirklich bewegt hat, wird der nachfolgende Code erst danach ausgeführt.

Im folgenden wird eine die momentane Rotation des *Player*s zu einem *byte* konvertiert und in einem *byte[ ]* in den *OutputStream* des *Socket*s geschrieben. *OutputStream::flush* zwingt den *OutputStream* die Daten unverzüglich über das Netzwerk zu senden.

Anmerkung: Die Rotation des *Player*s nach dem Ausführen von *Player.act()*, muss zu diesem Zeitpunkt zwingend die vom User gewählte Richtung sein.

Sollte beim senden der Daten ein Problem/eine *IOException* auftreten, wird diese abgefangen und als sinnvolle Fehlernachricht in die Konsole ausgegeben.

```
39     @Override
40     void setVisible(final boolean visible){
41         super.setVisible(visible);
42         try{
43             if(!visible) this.socket.close();
```

```
44          }catch(IOException e){
45              System.out.println("unable to close socket");
46          }
47      }
```

Genau wie in *ControlledPlayer*, wird die *GameObject::setVisible* Methode überschrieben, ihre Basis-Logik weiterhin ausgeführt und schließlich die Verbindung des *Socket*s im Falle der Unsichtbarkeit des *Player*s geschlossen.

Sollten hierbei Probleme/eine *IOException* auftreten, wird diese abgefangen und als sinnvolle Fehlermeldung in die Konsole ausgegeben.

Aufgabe 3 Spielend

```
29      public Level(String filename, boolean isRemote) {
30          this.isRemote = isRemote;
```

Um zu bestimmen, ob das *Level* mit einem *ControlledPlayer* oder einem *RemotePlayer* geladen werden soll, wird der *Level* Klasse im Konstruktor als Parameter die *boolean isRemote* übergeben und als Attribut abgespeichert.

```
55                          if(this.isRemote) actors.add(0, new RemotePlayer(x/2, y/2,
                              rotation, this.field));
56                          else actors.add(0, new ControlledPlayer(x/2, y/2, rotation, this.
                              field));
```

Sollte diese wahr sein, wird ein *RemoteSpieler*, ansonsten ein *ControlledPlayer* an der entsprechenden Stelle des *Level*s erschaffen.

```
21      public static void main(boolean isRemote) {
22
23          level = new Level("Map.txt", isRemote);
```

Da jede Instanz von *Level* nun zusätzliche eine *boolean isRemote* als Parameter benötigt, muss diese in der Spielhauptklasse übergeben werden. Diese bezieht den Wert der Variable als Parameter der *RPGGame::main* Methode. Der Benutzer muss *isRemote* somit beim Starten des Programms als Argument mit angeben.