Übungsblatt 9

Musterlösung

Aufgabe 1 Pro-aktiv

```java
/**
 * Diese abstrakte Klasse definiert die Basisklasse für alle aktiven
 * Spielobjekte, d.h. welche, die sich bewegen. Sie definiert eine
 * abstrakte Methode, die alle abgeleiteten Klassen überschreiben
 * müssen, um das Verhalten des Spielobjekts zu definieren. Sie
 * bietet außerdem eine Methode, über die abgefragt werden kann, ob
 * sich dieses Spielobjekt entsprechend der Gitterstruktur des Feldes
 * in eine bestimmte Richtung bewegen darf.
 *
 * @author Thomas Röfer
 */
abstract class Actor extends GameObject
{
    /** Das Spielfeld. */
    private final Field field;

    /**
     * Erzeugt eine neue Akteur:in.
     * @param x Die x-Koordinate der Akteur:in im Gitter.
     * @param y Die y-Koordinate der Akteur:in im Gitter.
     * @param rotation Die Rotation dieser Akteur:in (0 = rechts ... 3 = oben).
     * @param fileName Der Dateiname des Bildes, durch das diese Akteur:in
     *         dargestellt wird.
     * @param field Das Spielfeld, auf dem sich diese Akteur:in bewegt.
     */
    Actor(final int x, final int y, int rotation, final String fileName,
            final Field field)
    {
        super(x, y, rotation, fileName);
        this.field = field;
    }

    /**
     * Prüfen, ob die Akteur:in in eine bestimmte Richtung laufen darf.
     * @param direction Die geprüfte Richtung (0 = rechts ... 3 = oben).
     */
    boolean canWalk(final int direction)
    {
        return field.hasNeighbor(getX(), getY(), direction);
    }

    /**
     * Diese Methode muss überschrieben werden, um das Verhalten dieser
     * Akteur:in zu definieren.
     */
    abstract void act();
}
```

Aufgabe 2 Spielkram

```java
// Importieren der VK_*-Tastenkonstanten
import static java.awt.event.KeyEvent.*;

/**
 * Diese Klasse definiert die Figur, die von der Spieler:in gesteuert
 * wird. Die Figur bewegt sich dabei auf der Gitterstruktur des
 * Spielfeldes.
 *
 * @author Thomas Röfer
```

```
 */
class Player extends Actor
{
    /**
     * Erzeugen und Anzeigen einer neuen Spielfigur.
     * @param x Die x-Koordinate dieser Spielfigur im Gitter.
     * @param y Die y-Koordinate dieser Spielfigur im Gitter.
     * @param rotation Die Rotation dieser Spielfigur (0 = rechts ... 3 = oben).
     * @param field Das Spielfeld, auf dem sich diese Spielfigur bewegt.
     */
    Player(final int x, final int y, final int rotation, final Field field)
    {
        super(x, y, rotation, "woman", field);
    }

    /**
     * Die Spielfigur bewegt sich entsprechend der von der Spieler:in
     * bewegten Tasten. Diese Methode kehrt erst zurück, wenn ein
     * gültiger Zug gemacht wurde.
     */
    @Override
    void act()
    {
        while (true) {
            final int key = getNextKey();
            if (key == VK_RIGHT && canWalk(0)) {
                setRotation(0);
                setLocation(getX() + 1, getY());
            }
            else if (key == VK_DOWN && canWalk(1)) {
                setRotation(1);
                setLocation(getX(), getY() + 1);
            }
            else if (key == VK_LEFT && canWalk(2)) {
                setRotation(2);
                setLocation(getX() - 1, getY());
            }
            else if (key == VK_UP && canWalk(3)) {
                setRotation(3);
                setLocation(getX(), getY() - 1);
            }
            else {
                playSound("error");
                continue;
            }

            playSound("step");
            sleep(200);
            break;
        }
    }
}
```

Aufgabe 3 Weniger Fernsteuerung

Die Klasse *Walker* aus Übungsblatt 8 erbt nun von *Actor* und reicht die Parameter geeignet durch, wodurch die Attribut *avatar* und *field* entfallen. Zudem wurde der Parameter der Methode *act* in den Konstruktor verschoben und im Attribut *player* gespeichert. Aufrufe von *field.hasNeighbor* wurden durch Aufrufe der neuen Methode *canWalk* ersetzt. Ansonsten ändert sich in *act* nur, dass das Präfix *avatar.* überall entfernt werden musste.

```
import static java.lang.Math.abs;
import static java.lang.Math.max;

/**
 * Diese Klasse definiert eine Spaziergänger:in, die dieselbe Strecke immer
 * auf und ab läuft. Dabei werden bei der Konstruktion die Startposition und
 * -richtung angegeben, sowie das Spielfeld, auf dem sie sich bewegt.
 *
 * @author Thomas Röfer
```

```
 */
class Walker extends Actor
{
    /** Die Spielfigur. */
    final Player player;

    /** Die gepufferten Schritte der Spielfigur. */
    private RingBuffer stepsToFollow = null;

    /**
     * Erzeugt eine neue Spaziergänger:in.
     * @param x Die x-Koordinate der Spaziergänger:in.
     * @param y Die y-Koordinate der Spaziergänger:in.
     * @param rotation Die Rotation der Spaziergänger:in (0 = rechts ... 3 = oben).
     * @param field Das Spielfeld, auf dem sich die Spaziergänger:in bewegt.
     * @param player Die Spielfigur.
     */
    Walker(final int x, final int y, final int rotation, final String fileName,
            final Field field, final Player player)
    {
        super(x, y, rotation, fileName, field);
        this.player = player;
    }

    /**
     * Die Methode definiert das oben beschriebene Verhalten der Spaziergänger:in.
     */
    @Override
    void act()
    {
        // Wenn im Verfolgermodus, dann Schritt aufzeichnen
        if (stepsToFollow != null) {
            stepsToFollow.push(player.getRotation());
        }

        // Vorwärts bewegen
        if (getRotation() == 0) {
            setLocation(getX() + 1, getY());
        }
        else if (getRotation() == 1) {
            setLocation(getX(), getY() + 1);
        }
        else if (getRotation() == 2) {
            setLocation(getX() - 1, getY());
        }
        else {
            setLocation(getX(), getY() - 1);
        }

        // Sound dazu abspielen
        playSound("step");

        // Wenn im Verfolgermodus und aufgezeichneter Schritt möglich,
        // dann diesen verwenden.
        if (stepsToFollow != null && canWalk(stepsToFollow.peek())) {
            setRotation(stepsToFollow.pop());
        }
        else {
            // Wir sind nicht (mehr) im Verfolgermodus
            stepsToFollow = null;

            // Umdrehen, wenn nächster Schritt nicht mehr ausführbar
            if (!canWalk(getRotation())) {
                setRotation(getRotation() + 2);
            }

            // Wenn Spielfigur vor dieser Figur
            if (getRotation() == 0
                        && player.getX() > getX()
                        && player.getY() == getY()
                    || getRotation() == 1
                        && player.getX() == getX()
                        && player.getY() > getY()
```

```
83                      || getRotation() == 2
84                          && player.getX() < getX()
85                          && player.getY() == getY()
86                      || getRotation() == 3
87                          && player.getX() == getX()
88                          && player.getY() < getY()) {
89                  // Entfernung zur Spielfigur bestimmen.
90                  final int distance = max(abs(player.getX() - getX()),
91                          abs(player.getY() - getY()));
92
93                  // Wenn auch in Sichtweite, dann Puffer anlegen und Weg eintragen.
94                  if (distance <= 4) {
95                      stepsToFollow = new RingBuffer(distance);
96                      for (int i = 1; i < distance; ++i) {
97                          stepsToFollow.push(getRotation());
98                      }
99                  }
100             }
101         }
102
103         // Wenn gleiche Position wie Spielfigur, lasse diese verschwinden
104         if (getX() == player.getX() && getY() == player.getY()) {
105             player.setVisible(false);
106             playSound("go-away");
107         }
108     }
109 }
```

Aufgabe 4 Alles wieder zum Laufen bringen

In der Klasse *PI1Game* werden alle Akteur:innen in eine Liste eingetragen und dann immer wieder ihre Methode *act()* aufgerufen.

```
1  import java.util.ArrayList;
2  import java.util.List;

31          final Player player = new Player(0, 3, 0, field);
32          final List<Actor> actors = new ArrayList<>();
33          actors.add(player);
34          actors.add(new Walker(1, 0, 2, "claudius", field, player));
35          actors.add(new Walker(0, 1, 0, "laila", field, player));
36          actors.add(new Walker(3, 2, 2, "child", field, player));
37
38          while (player.isVisible()) {
39              for (final Actor actor : actors) {
40                  actor.act();
41              }
42          }
```