Projektaufgabe OOSE WS14/15

Ein kleines Spielframework

Sven Eric Panitz
Hochschule RheinMain

Zum Abschluss des Moduls »Objektorientierte Softwareentwicklung« steht ein kleines Spieleprojekt, das in Einzelarbeit zu lösen ist.

Dieses Papier stellt zusammen mit der Projektaufgabe eine kleine Rahmenbibliothek vor, unter deren Benutzung das Spieleprojekt zu realisieren ist. Einige Klassen aus den bisherigen Übungsblättern finden in dieser Bibliothek Verwendung. Ein minimale Beispielapplikation ist auch in diesem Papier vorgestellt.

Inhaltsverzeichnis

Kapitel 1

Projektaufgabe

Sie sollen in dieser abschließenden Projektaufgabe eine kleine Spielapplikation entwickeln. Die Applikation soll mit einem `java.swing.Timer` animiert sein. Es soll auf den Klassen, die im Laufe des Semesters erstellt wurden, aufgebaut werden.

Das Spiel soll die grundlegende Spielidee von typischen Jump-and-Run-Spielen verfolgen. Als Beispiel können Sie zum Beispiel auf folgender Webseite ein solches Spiel spielen:

Beispielspiel (`http://www.zaga.de/spiele/abenteuer-spiele/ice-warrior/2822`)
.

Anforderungen und Bewertung: Sie können maximal 20 Punkte in diesem Projekt erreichen. Die Hälfte der Punkte erreichen Sie, wenn sie folgende Mindestanforderungen erfüllen:

- Das Spielfeld lässt sich aus einer Datei einlesen.
- Es gibt eine Spielfigur, die über die Tastatur gesteuert wird, sowie weitere Figuren, die sich selbstständig bewegen.
- Die Spielfigur und die weiteren Figuren können unter normalen Umständen nicht durch Wände laufen.
- Berührungen der Spielfigur mit Gegnerfiguren führen dazu, dass die Spielfigur Lebenskraft verliert.
- Es gibt Objekte auf dem Spielfeld, die die Spielfigur einsammeln muss, um Punkte zu bekommen.
- Eine Anzeige gibt an, wieviele Punktegesammelt wurden und wieviel Leben die Spielfigur noch hat.
- Figuren können springen und fallen.

Weitere 5 Punkte können Sie für Ausbauanforderungen erhalten:

- Die Spielfigur kann Kraftobjekte aufsammeln, mit der sie zeitweise immun gegen Gegner ist und diese zerstören kann, anstatt von ihnen zerstört zu

werden.

Alternativ kann die Spielfigur mit etwas auf die Gegner schießen, um sie unschädlich zu machen, oder durch einen gezielten Sprung auf einen Gegner, diesen unschädlich machen.

- Es gibt mehrere Level, die sich in Form des Labyrinth unterscheiden.
- Es gibt eine About-Box, in der Sie das Spiel und seine Schaffer beschreiben.

Weitere 5 Punkte können Sie mit der Profiaubaustufe erreichen. Diese umfasst mindestens:

- Ein Menu, in dem Einstellungen gesetzt werden können.
- Sound während des Spiels.
- Unterschiedliche Bilder für den Spieler, je nach Bewegungsrichtung
- Ein eigenes zusätzliches Feature.

Punktabzüge gibt es, wenn die Umsetzung zwar funktionale Anforderung erfüllt, aber weitere nicht-funktionale Anforerungen nicht erfüllt sind:

- Gute, sprechende Bezeichnernamen für Klassen, Felder und Methoden.
- Kleine übersichtliche Methoden.
- Durchgehender OO-Entwurf. Wenig Fallunterscheidungen, sondern mehr Unterklassen, die die verschiedenen Fälle unterscheiden.
- Sinnvolles und konsequentes Arbeiten mit Interfaces.
- Komplexer Code ist kommentiert.
- Wichtige Dinge sind dokumentiert.
- Schlechter oder gar nicht gehaltener Abschlussvortrag.
- Schlechtes (oder fehlendes) Übersichtspapier.
- Programm läuft nicht auf Linux-System (z.B. wegen Windows Pfadangaben oder nich Berücksichtigung von relevanter Groß-/Kleinschreibung in Dateinamen).
- Abgabemodalitäten (s.u.) nicht eingehalten. (z.B. Falscher Dateiname, keine tar-Datei).

Zusatzpunkte lassen sich erreichen durch:

- Besonders gelungene Spielideen.
- Funktionale Zusatzfeatures des Spiels und ein sehr komplexes Spiel.
- Ansprechende Optik.
- Dinge die sonst noch gefallen oder überraschen...

Abgabe:

- das Spiel ist in der letzten Praktikumsstunde (letzte Januarwoche) in einem 5 minütigen Vortrag vorzustellen. (unbedingt Länge einhalten und dafür sorgen, dass es keine Verzögerung gibt, das eigene Spiel am Beamer zu zeigen. Wir wollen und können die 90 Minuten der Praktikumsstunde nicht überziehen.)
- letzte Abgabemöglichkeit ist am Sonntag 1. Februar 20:15Uhr per Mail an den Lehrbeauftragten der Praktikumsstunde
- die Abgabe beinhaltet:
  - einen eclipse Projektordner mit alles Resourcen.

    Falls eine andere Entwicklungsumgebung oder ein Build-Tool (ant, maven, make) verwendet wurde alternativ der entsprechende Projektordner hierfür und ein Hinweis darauf in der PDF-Dokumentation.
  - eine kleine PDF-Datei, die die Besonderheiten der eigenen Lösung zeigen. In diesem Dokument sind alle zusätzlichen Quellen und Hilfen, die benutzt worden, genannt. Insbesondere die Urheber der Bilddateien, falls welche verwendet wurden. Es ist darin zu unterschreiben, dass die Aufgabe eigenständig gelöst wurde und keine anderen als die angegebenen Hilfen verwendet wurden.
  - der Projektordner und das PDF-Dokument sind in einer tar-Datei zu verpacken.

    Die tar-Datei hat den Namen, der aus Ihren Nachnamen gefolgt von Ihrem Vornamen besteht. Verwenden sie keine Umlaute oder sonstigen Sonderzeichen in dem Dateinamen, also z.B. `muellerMartin.tar`.

Kapitel 2

Das Spieleframework

Die Bibliothek erlaubt es Spiele zu entwickeln, die auf einer zweidimensionalen Spielfläche laufen. Es gibt Spielfiguren, die sich selbstständig bewegen, Spielfiguren, die durch Benutzereingaben gesteuert werden können und Spielelemente, die unbewegt das Spielfeld definieren.

2.1 Hilfsklassen aus dem Semesterverlauf

Aus dem siebten Übungsblatt kennen wir schon drei Schnittstellen, die ausdrücken, dass Spielfiguren Bewegungsschritte durchführen können und dass Spielfiguren sich grafisch darstellen lassen.

Für die Bewegung ist die Schnittstelle `Moveable` zuständig:

```
1 package name.panitz.game;
2
3 public interface Moveable {
4   void move();
5 }
```

Listing 2.1: Moveable.java

Die Schnittstelle `Paintable` drückt aus, dass ein Objekt sich grafisch darstellen kann:

```
1 package name.panitz.game;
2
3 import java.awt.Graphics;
4
5 public interface Paintable {
6   void paintTo(Graphics g);
7 }
```

Listing 2.2: Paintable.java

Die Schnittstelle `MoveAndPaintable` vereinigt diese beiden Schnittstellen.

```
1 package name.panitz.game;
2
3 public interface MoveAndPaintable extends Moveable, Paintable[]
```

Listing 2.3: MoveAndPaintable.java

Zur Darstellung von Punkten im zweidimensionalen Raum wurden bereits auf dem zweiten Übungsblatt die Klasse `Vertex` entwickelt.

```
1  package name.panitz.game;
2
3  public class Vertex {
4    public double x;
5    public double y;
6
7    public Vertex(double x, double y) {
8      super();
9      this.x = x;
10     this.y = y;
11   }
12
13   @Override
14   public String toString() {
15     return "("+x+", "+y+")";
16   }
17
18   public void move(Vertex that){
19     x += that.x;
20     y += that.y;
21   }
22
23   public void moveTo(Vertex that){
24     x = that.x;
25     y = that.y;
26   }
27
28   public Vertex mult(double d) {
29     return new Vertex(d*x,d*y);
30   }
31 }
```

Listing 2.4: Vertex.java

Als zentrale Klasse für unsere Spielobjekte wurde die Klasse `GeometricObject` entwickelt. Die hier angegebene Version ist noch abstrakt, da sie keine Implementierung der Methode `paintMeTo` enthält.

```java
package name.panitz.game;

public abstract class GeometricObject implements MoveAndPaintable{
  public Vertex corner;
  double width;
  double height;
  public Vertex movement;

  public GeometricObject(Vertex corner, double width, double height,
      Vertex movement) {
    super();
    this.corner = corner;
    this.width = width;
    this.height = height;
    this.movement = movement;
  }
  public boolean touches(GeometricObject that) {
    return !(   isAbove(that)  || isUnderneath(that)
             || isLeftOf(that) || isRightOf(that));
  }

  boolean isAbove(GeometricObject that) {
    return corner.y+height <= that.getCorner().y;
  }
  boolean isUnderneath(GeometricObject that) {
    return that.isAbove(this);
  }

  boolean isLeftOf(GeometricObject that) {
    return corner.x + width <= that.getCorner().x;
  }
  boolean isRightOf(GeometricObject that) {
    return that.isLeftOf(this);
  }

  @Override
  public void move() {
    corner.move(movement);
  }
```

Listing 2.5: GeometricObject.java

Wir spendieren dieser Klasse noch eine zusätzliche Methode, die prüft, ob das Objekt direkt oberhalb eines zweiten Spielobjektes steht.

```java
  public boolean isStandingOnTopOf(GeometricObject that){
    return  !(isLeftOf(that)||isRightOf(that))
          && isAbove(that)
          && corner.y+height+2 > that.getCorner().y;
  }

```

```
46   public Vertex getCorner(){return corner;}
47 }
```

Listing 2.6: GeometricObject.java

2.2 Das allgemeine Framework

In diesem Abschnitt folgen die Klassen, die die Eigenschaften eines Spiels beschreiben. Auch hier beginnen wir mit der Definition einer Schnittstelle.

Die Schnittstelle `GameFramework` beschreibt, was ein Spiel für einen generellen Aufbau hat:

```
1 package name.panitz.game;
2
3 public interface GameFramework extends Paintable{
```

Listing 2.7: GameFramework.java

Wie man sieht, wird verlangt, dass das Spiel sich selbst wieder komplett zeichnen lassen kann, dadurch dass die Schnittstelle `Paintable` erweitert wird.

Die erste Methode soll einen einzigen Spielschritt des Spiels durchführen. Es wird einen Ticker geben und pro Tick wird das Spiel mit Hilfe der Methode `step()` um einen Schritt weiterbewegt. Jedes bewegliches Spielobjekt nimmt dann genau einen Schritt mit seiner Bewegung vor. Eigentlich hätte man hier auch einfach die Schnittstelle `Moveable` als Oberschnittstelle nehmen können, denn im Prinzip drückt `step()` dasselbe aus wie `move()`. Um allerdings konzeptionell das Gesamtspiel von den einzelnen Spieleobjekten besser zu unterscheiden, wurde darauf verzichtet, die Schnittstelle `Moveable` zu bemühen.

```
4   void step();
```

Listing 2.8: GameFramework.java

Nach jedem Schritt finden Checks für das Spiel statt. Hier wird auf Kollisionen getestet und eventuell auf Grund einer Kollision der Spielzustand verändert, indem eventuell Spielobjekte gelöscht werden, oder Punktestände verändert werden. Hier befindet sich zumeist die komplette Spiellogik umgesetzt.

```
5   void checks();
```

Listing 2.9: GameFramework.java

Wir gehen davon aus, dass ein Spiel genau einen Spieler hat, der durch den Benutzer über die Tastatur gesteuert wird. Dieser Spieler kann von einem Spiel erfragt werden. Die entsprechende Schnittstelle `Player` ist weiter unten definiert.

```
6   Player getPlayer();
```

Listing 2.10: GameFramework.java

Ein Spiel muss nach seiner Höhe und Breite gefragt werden können.

```
7   int getHeight();
8   int getWidth();
```

Listing 2.11: GameFramework.java

Das Spiel muss angeben können, ob der Spieler bereits verloren hat:

```
9   boolean lost();
```

Listing 2.12: GameFramework.java

Oder ob der Spieler vielleicht gewonnen hat:

```
10  boolean won();
```

Listing 2.13: GameFramework.java

Das Spiel ist beendet, wenn der Spieler gewonnen oder verloren hat. Dieses drückt die Standardmethode `finished()` aus. Solche Standardmethoden können seit Java 1.8 definiert werden.

```
11  default boolean finished(){
12    return won()||lost();
13  }
14 }
```

Listing 2.14: GameFramework.java

Soweit die allgemeine Schnittstelle für das Spiel. Darin wurde bereits die Schnittstelle `Player` für die vom Benutzer geführte Spielfigur verwendet, deren Definition nun folgt:

```
1  package name.panitz.game;
2
3  import java.awt.event.KeyListener;
4
5  public interface Player [
```

Listing 2.15: Player.java

Die einzige wichtige Information über die vom Benutzer geführte Spielfigur ist, welche Tastatursteuerung diese Figur hat. Diese wird durch ein `KeyListener`-Objekt ausgedrückt.

```
6   KeyListener getKeyListener();
7   Vertex getCorner();
8 }
```

Listing 2.16: Player.java

2.2.1 Das Spiel Anzeigen und Starten

Die Schnittstellen im vorangegangenen Abschnitt definieren, was alles zu einem Spiel gehört. Klassen, die die Schnittstelle `GameFramework` implementieren, können nun in einem Fenster angezeigt werden und über einen Zeitticker in Form eines `javax.swing.Timer`-Objekts animiert werden.

Hierzu dient die Klasse: `GameScreen`, die eine Unterklasse der Swing Standardklasse `JPanel` ist.

```
1  package name.panitz.game;
2
3  import java.awt.Dimension;
4  import java.awt.Graphics;
5  import java.io.IOException;
6
7  import javax.swing.JPanel;
8  import javax.swing.Timer;
9
10 @SuppressWarnings("serial")
11 public class GameScreen extends JPanel {
```

Listing 2.17: GameScreen.java

Entscheidend für ein Objekt dieser Klasse ist, welches Spiel gespielt wird, sprich das Objekt, das die Schnittstelle `GameFramework` implementiert. Dieses wird ein einem Feld der Klasse gespeichert.

```
12   GameFramework game;
```

Listing 2.18: GameScreen.java

Der Konstruktor bekommt das Spiel als Objekt übergeben und führt dann eine Initiualisierung und den Start des Spiels durch. Zur Initalisierung wird das Spielerobjekt nach seiner Tastatursteuerung gefragt und diese dem `JPanel` zugefügt:

```
13   public GameScreen(GameFramework game) {
14     this.game = game;
15     initAndStart();
16   }
17   private void initAndStart(){
18     addKeyListener(game.getPlayer().getKeyListener());
19     timer.start();
20   }
```

Listing 2.19: GameScreen.java

Das Spiel erhält eine Zeitsteuerung über den Timer der Swing-Bibliothek. Pro Tick dieses Times sind die Spielfiguren einen Schritt weiter zu bewegen, die Checks auf Kollisionen durchzuführen, und zu überprüfen, ob das Spiel bereits beendet ist.

Schließlich ist die Szenerie neu zu zeichen und sicherzustellen, dass das Spiel noch im Fokus für die Tastatureingabe ist.

```
21   Timer timer = new Timer(1000 / 35, (ev) -> {
22     game.step();
23     game.checks();
24     checkForEndOfGame();
25
26     repaint();
27     java.awt.Toolkit.getDefaultToolkit().sync();
28
29     requestFocusInWindow();
30   });
```

Listing 2.20: GameScreen.java

Wenn das Spiel beendet ist, so wird in unserem Fall der Timer gestoppt.

```
31   void checkForEndOfGame(){
32     if (game.finished()){
33       timer.stop();
34     }
35
36   }
```

Listing 2.21: GameScreen.java

Die Größe des JPanel wird durch die vom Spiel angegebene Größe definiert.

```
37   @Override
38   public Dimension getPreferredSize() {
39     return new Dimension(game.getWidth(), game.getHeight());
40   }
```

Listing 2.22: GameScreen.java

Zum Zeichen des Spiels wird zunächst der Bildschirm geleert und dann die Spielkomponente aufgefordert sich auf den FPanel zu zeichnen.

```
41   @Override
42   protected void paintComponent(Graphics g) {
43     super.paintComponent(g);
44     game.paintTo(g);
45   }
46 }
```

Listing 2.23: GameScreen.java

2.2.2 Nützliche Klasse für Bildobjekte

In diesem Abschnitt folgen ein paar weitere nützliche Klassen, die hilfreich sind, um Spieleobjekte zu erzeugen. Zunächst lässt sich ein unteres und lebhafteres Spieleobjekt erzeugen, wenn hierzu Bilddateien benutzt werden. Wir stellen eine Unterklasse von `GeometricObject` zur Verfügung, die durch eine Bilddatei dargestellt wird. Zum Laden der Bilddateien wird die Klasse `javax.swing.ImageIcon` verwendet.

Gespeichert wird das Bild in einem Feld des Typs: `java.awt.Image`.

```
1  package name.panitz.game;
2
3  import java.awt.Graphics;
4  import java.awt.Image;
5  import java.net.URL;
6
7  import javax.swing.ImageIcon;
8
9  public class ImageObject extends GeometricObject {
10   Image img;
```

Listing 2.24: ImageObject.java

Entscheidend ist der Konstruktor. Dieser bekommt den Dateinamen der Bilddatei sowie je ein Vertex-Objekt für Position und Bewegung. Die Größe des Objekts wird erst durch die Größe der Bilddatei festgelegt. Solange diese Werte noch nicht bekannt sind, wird durch den Aufruf des Konstruktors der Obeklasse mit den Konstanten 0 Weite und Höhe auf 0 gesetzt.

```
11   public ImageObject(String imageFileName,Vertex corner,Vertex
       movement) {
12     super(corner, 0, 0, movement);
```

Listing 2.25: ImageObject.java

Um das Bild zu Laden, wird ein URL-Objekt erzeugt. Dieses geschieht durch einen Aufruf an den Klassenlader der aktuellen Klasse, der in der Lage ist eine Resource zu lokalisieren.

```
13
14      URL resource = this.getClass()
15                           .getClassLoader()
16                           .getResource(imageFileName);
```

Listing 2.26: ImageObject.java

Diese URL wird für den Konstruktor der Klasse `ImageIcon` verwendet.

```
17      ImageIcon icon = new ImageIcon(resource);
```

Listing 2.27: ImageObject.java

das Icon enthält schließlich das benötigte Image-Objekt:

```
18      img = icon.getImage();
```

Listing 2.28: ImageObject.java

Dieses hat jetzt die korrekte Weite und Höhe gesetzt.

```
19      width = img.getWidth(null);
20      height = img.getHeight(null);
21    }
```

Listing 2.29: ImageObject.java

Zum Zeichnen des Objekts stellt das Graphics-Objekt eine Methode zur Verfügung.

```
22  @Override
23    public void paintTo(Graphics g) {
24      g.drawImage(img, (int)corner.x,(int)corner.y, null);
25    }
26  }
```

Listing 2.30: ImageObject.java

2.2.3 Der freie Fall

Die nächste Hilfsklasse ermöglicht Spielfiguren, die springen und fallen können und dieses entsprechend der Erdbeschleunigung. Die Erdbeschleunigung beträgt konstant 9.81.

```
1  package name.panitz.game;
2
3  public class FallingImage extends ImageObject {
4    static final double G = 9.81;
```

Listing 2.31: FallingImage.java

Für fallende Figuren wird in einem Wahrheitswert festgehalten, ob die Figur gerade fällt oder auf festen Boden steht:

```
5    public boolean isJumping = false;
```

Listing 2.32: FallingImage.java

Wenn die Figur fällt, wird die Anfangsgeschwindigkeit festgehalten und die Zeitdauer, wie lange sie bereits fällt:

```
6    double v0;
7    int t = 0;
```

Listing 2.33: FallingImage.java

Ist die Anfangsgeschwindigkeit v0 negativ, so springt die Figur zunächst nach oben.

Der Konstruktor bekommt keine anderen Informtaionen, als der Konstruktor der Oberlkasse `ImageObject`.

```
8    public FallingImage(String imageFileName, Vertex corner, Vertex
       movement) {
9      super(imageFileName, corner, movement);
10   }
```

Listing 2.34: FallingImage.java

wenn sich das Objekt bewegen soll, muss zunächst geschaut werden, ob es gerade springt bzw. fällt. Dann ändert sich mit jedem Tick die Geschwindigkeit der vertikalen, also auf der y-Achse. Diese gerechnet sich aus $v_0 + G * t$. Wenn die y-Bewegung gesetzt wurde, dann kann die eigentliche bewegung durchgeführt werden.

```
11   @Override
12   public void move() {
13     if (isJumping) {
14       t++;
15       double v = v0 + G * t / 200;
16       movement.y = v;
17     }
18     super.move();
19   }
```

Listing 2.35: FallingImage.java

Wir sehen eine Methode vor, mit der ein Sprung gestartet wird. Dieser Sprung geht nach oben mit einer Startgeschwindigkeit von $3{,}5$.

```
20  public void jump() {
21    if (!isJumping) {
22      startJump(-3.5);
23    }
24  }
```

Listing 2.36: FallingImage.java

Zum Start eines Sprungs wird das Flag entsprechend auf true gesetzt, die Startgeschwindigkeit gesetzt und der Ticker, der angibt, seit welcher Zeit gesprungen wird auf 0 gesetzt.

```
25  public void startJump(double v0) {
26    isJumping = true;
27    this.v0 = v0;
28    t = 0;
29  }
```

Listing 2.37: FallingImage.java

Eine weitere Methode wird angeboten, mit der das Objekt egal ob es springt, fällt oder läuft in der Bewegung komplett gestoppt werden kann. Diese Methode setzt den letzten Schritt zurück. Dieses hat den Effekt, dass das Objekt, wenn es auf den Boden aufgekommen ist oder gegen eine Wand gelaufen ist, kurz wieder abprallt.

```
30  public void stop() {
31    corner.move(movement.mult(-1.1));
32    movement.x = 0;
33    movement.y = 0;
34    isJumping = false;
35  }
```

Listing 2.38: FallingImage.java

Objekte, die nicht gerade in der Luft sind, lassen sich nach links:

```
36  public void left() {
37    if (!isJumping) {
38      movement.x = -1;
39    }
40  }
```

Listing 2.39: FallingImage.java

Und auch nach rechts steuern:

```
41   public void right() {
42     if (!isJumping) {
43       movement.x = +1;
44     }
45   }
46 }
```

Listing 2.40: FallingImage.java

2.2.4 Einlesen von Dateien

Auf dem 9. Übungsblatt gab es eine kleine Hilfsklasse, mit deren Hilfe eine Textdatei zeilenweise eingelesen werden kann. Diese geben wir hier in leicht erweiterter Form:

```
1  package name.panitz.game.util;
2
3  import java.io.*;
4  import java.util.*;
5
6  public class FileUtil {
7    public static String[] readTextLines(String fileName){
8      try {
9         return readTextLines(new FileReader(fileName));
10     } catch (IOException e) {
11       e.printStackTrace();
12       throw new RuntimeException(e);
13     }
14   }
15
16   public static String[] readTextLines(InputStream fileStream) {
17     return readTextLines(new InputStreamReader(fileStream));
18   }
19
20   public static String[] readTextLines(Reader fileReader) {
21     try {
22       List<String> result = new LinkedList<>();
23       BufferedReader in = new BufferedReader(fileReader);
24       String line=null;
25       while ((line = in.readLine() )!=null){
26         result.add(line);
27       }
28       return result.toArray(new String[0]);
29     } catch (IOException e) {
30       e.printStackTrace();
31       throw new RuntimeException(e);
32     }
33   }
34
```

```
35   public static void main(String[] args) {
36     for (String l:readTextLines("test.txt")){
37       System.out.println(l);
38     }
39   }
40 }
```

Listing 2.41: FileUtil.java

Kapitel 3

Kleines Beispiel: Klaus sammelt Herzen

In diesem Kapitel wird ein minimales Beispiel gezeigt, wie mit Hilfe der im ersten Kapitel vorgestellten Bibliothek ein kleines Spiel erzeugt werden kann. In dem Spiel Hearts of Klaus wird ein kleiner Bär mit Namen Klaus Herzen einsammeln. Gestört wird er durch Apfelwein-Fässer, die durch das Spielfeld rollen und fallen und ihm jedes Mal verletzen, wenn er sie berührt.

3.1 Spielobjekte

Zunächst werden die Spielfiguren definiert. Hierzu gibt es zwei Spielelemente, die sich nicht bewegen. Zum einen die Wände und Fußböden/Decken des Spielfeldes:

```
1  package name.panitz.game.klaus;
2
3  import name.panitz.game.ImageObject;
4  import name.panitz.game.Vertex;
5
6  public class Wall extends ImageObject{
7    public Wall(Vertex corner) {
8      super("images/wall.png", corner, new Vertex(0,0));
9    }
10 }
```

Listing 3.1: Wall.java

Dann die einzelnen Herzen, die auf dem Spielfeld verteilt sind:

```
1  package name.panitz.game.klaus;
2
3  import name.panitz.game.ImageObject;
4  import name.panitz.game.Vertex;
5
6  public class Heart extends ImageObject {
```

```
7   public Heart(Vertex corner ) {
8     super("images/heart.png", corner, new Vertex(0,0));
9   }
10 }
```

Listing 3.2: Heart.java

Schließlich gibt es Fässer, die selbsttätig durch das Spielfeld fallen:

```
1  package name.panitz.game.klaus;
2
3  import name.panitz.game.FallingImage;
4  import name.panitz.game.Vertex;
5
6  public class Barrel extends FallingImage {
7
8    public Barrel(Vertex corner) {
9      super("images/fass.gif", corner, new Vertex(1, 0));
10   }
```

Listing 3.3: Barrel.java

Fässer können an einer zufälligen x-Position von oben neu gestartet werden.

```
11   public void fromTop(int wi) {
12     getCorner().moveTo(new Vertex(Math.random()*(wi-2*40) + 40,-40));
13   }
```

Listing 3.4: Barrel.java

Und können weiter laufen, wenn sie nach einem Fall auf einer Wand aufgetroffen sind.

```
14   public void restart() {
15     double oldX = movement.x;
16     corner.move(movement.mult(-1.1));
17     movement.x = -oldX;
18     movement.y = 0;
19     isJumping = false;
20   }
21 }
```

Listing 3.5: Barrel.java

Die Spielfigur heißt Klaus und ist ein `FallingImage` mit einer Tastatursteuerung. Die Pfeiltasten dienen zur Steuerung. Pfeil nach oben ist für das Springen, links rechts für das Laufen in diese Richting und Pfeil nach unten zum Stoppen. Allerdings kann die Figur nur gesteuert werden, wenn sie nicht gerade am Springen ist.

```java
package name.panitz.game.klaus;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.KeyAdapter;

import name.panitz.game.FallingImage;
import name.panitz.game.Player;
import name.panitz.game.Vertex;

public class Klaus extends FallingImage implements Player {

  private KeyListener keyListener = new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
      switch (e.getKeyCode()) {
      case KeyEvent.VK_LEFT:
        left();
        break;
      case KeyEvent.VK_RIGHT:
        right();
        break;
      case KeyEvent.VK_UP:
        jump();
        break;
      case KeyEvent.VK_DOWN:
        if (!isJumping) stop();
        break;
      }
    }
  };

  @Override public KeyListener getKeyListener() {
    return keyListener;
  }

  public Klaus(Vertex corner) {
    super("images/player.png", corner, new Vertex(0, 0));
  }
}
```

Listing 3.6: Klaus.java

3.2 Spiellogik

Mit diesen Spielfiguren lässt sich ein kleines Spiel erzeugen. Hierzu ist die Schnittstelle `GameFramework` zu implementieren:

```
1  package name.panitz.game.klaus;
2
3  import java.awt.Graphics;
4
5  import java.io.*;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  import javax.sound.sampled.*;
10 import name.panitz.game.*;
11 import name.panitz.game.util.FileUtil;
12
13 public class HeartsOfKlaus implements GameFramework {
```

Listing 3.7: HeartsOfKlaus.java

Felder dieser Klasse enthalten die Spielfiguren, zum einen die Spieler Klaus und desweiteren Listen von Wänden, Herzen und Fässern.

```
14   Klaus player;
15   List<Wall> walls = new ArrayList<>();
16   List<Heart> hearts = new ArrayList<>();
17   List<Barrel> barrels = new ArrayList<>();
```

Listing 3.8: HeartsOfKlaus.java

Die Spielfelddimensionen sind fest gesetzt durch zwei entsprechende Werte:

```
18   int width = 800;
19   int height = 600;
```

Listing 3.9: HeartsOfKlaus.java

Der Spielstand beinhaltet die Anzahl der gesammelten Herzen und die Lebensenergie des Spielers:

```
20   int energy = 5;
21   int collectedHearts=0;
```

Listing 3.10: HeartsOfKlaus.java

Im Konstruktor wird das Level aus einer Datei ausgelesen.

```
22   public HeartsOfKlaus() throws Exception {
23     loadLevel("./images/level1.txt");
24   }
```

Listing 3.11: HeartsOfKlaus.java

Zum Einlesen des Levels steht eine Hilfsmethode zur Verfügung:

```
25  void loadLevel(String levelDescription) {
26
27    String[] lines = FileUtil.readTextLines
28                       ( getClass()
29                        .getClassLoader()
30                        .getResourceAsStream(levelDescription));
31
32    int l = 0;
33    for (String ln:lines) {
34      int col = 0;
35      for (char c : ln.toCharArray()) {
36        switch (c) {
37        case 'w':
38          walls.add(new Wall(new Vertex(col * 40, l * 40)));
39          break;
40        case 'h':
41          hearts.add(new Heart(new Vertex(col * 40, l * 40)));
42          break;
43        case 'f':
44          barrels.add(new Barrel(new Vertex(col * 40, l * 40)));
45          break;
46        case 'p':
47          player = new Klaus(new Vertex(col * 40, l * 40));
48          break;
49        }
50        col++;
51      }
52      width = col * 40;
53      l++;
54    }
55    height = l * 40;
56  }
```

Listing 3.12: HeartsOfKlaus.java

Im jeden Schritt, also bei jedem Tick, werden die beweglichen Objekte, das sind in unserem Fall der Spieler und die Fässer, um einen Schrit bewegt.

```
57  public void step() {
58    player.move();
59    for (Moveable p : barrels)
60      p.move();
61  }
```

Listing 3.13: HeartsOfKlaus.java

Zum Zeichnen müssen alle Objekte gezeichnet werden. Zusätzlich wird der Süpielstand auch als String auf dem Bildschirm gezeichnet:

```
62   public void paintTo(Graphics g) {
63     for (Paintable p : walls)
64       p.paintTo(g);
65     for (Paintable p : hearts)
66       p.paintTo(g);
67     for (Paintable p : barrels)
68       p.paintTo(g);
69     player.paintTo(g);
70
71     g.drawString("Energy: "+energy, 50, 10);
72     g.drawString("Hearts: "+hearts.size(), 50, 30);
73   }
```

Listing 3.14: HeartsOfKlaus.java

Entscheidend sind schließlich die Kollisionschecks des Spiels. Dieses wird der Übersicht in Hilfsmethoden bewerkstelligt.

- zunächst wird gecheckt, ob ein Herz eingesammelt wird
- dann geschaut, ob der Spieler gegen eine Wand gelaufen ist
- dann, ob die Fässer fallen oder auf einen Boden auftreffen
- dann, ob die Spieler von einem Fass getroffen wird
- und schließlich, ob der Spieler unten aus dem Spielfeld gefallen ist. In diesem Fall wird er auf eine feste Position zurück gesetzt.

```
74   public void checks() {
75     collectHearts();
76     checkPlayerWallCollsions();
77     fallingBarrel();
78     playerBarrelCollision();
79
80     if (getPlayer().getCorner().y>height){
81       getPlayer().getCorner().moveTo(new Vertex(40,height-80));
82     }
83   }
```

Listing 3.15: HeartsOfKlaus.java

Es folgen die Checks im Einzelnen.

Für alle Herzen wird geschaut, ob der Spieler sie berührt. Wenn es solches gefunden wurde, wird dieses aus der Liste entfernt.

```
84   private void collectHearts() {
85     Heart removeMe = null;
86     for (Heart heart : hearts) {
87       if (player.touches(heart)) {
88         removeMe = heart;
```

```
89          collectedHearts++;
90          break;
91        }
92      }
93      if (removeMe!=null) hearts.remove(removeMe);
94    }
```

Listing 3.16: HeartsOfKlaus.java

In der nächsten Methode wird für alle Fässe getestet, ob sie den Spieler berühren. Ist dieses der Fall, wird ein Geräusch abgespielt und das Fass wieder von oben gestartet.

```
95    private void playerBarrelCollision() {
96      for (Barrel p : barrels){
97        if (p.touches(player)){
98          energy--;
99          playSound("./sounds/crash.wav");
100         p.fromTop(width);
101       }
102       if (p.getCorner().y>height){
103         p.fromTop(width);
104       }
105     }
106   }
```

Listing 3.17: HeartsOfKlaus.java

Die nächste Methode testet, ob alle Fässer mit den Wänden. Fässer, die nicht auf einer Wand stehen, fallen nach unten. Fässer, die auf einer Wand stehen hören auf zu fallen, falls sie im Fall sind.

```
107   private void fallingBarrel() {
108     for (Barrel p : barrels){
109       boolean isStandingOnTop = false;
110       for (GeometricObject wall : walls) {
111         if (p.touches(wall)) {
112           p.restart();
113         }
114         if (p.isStandingOnTopOf(wall)) {
115           isStandingOnTop = true;
116         }
117       }
118       if (!isStandingOnTop && !p.isJumping){
119         p.startJump(0.1);
120       }
121     }
122   }
```

Listing 3.18: HeartsOfKlaus.java

Die letzte Kollisionsmethode schaut, ob der Spieler mit einer Wand kollidiert.

```
123   private void checkPlayerWallCollsions() {
124     boolean isStandingOnTop = false;
125     for (GeometricObject wall : walls) {
126       if (player.touches(wall)) {
127         player.stop();
128         return;
129       }
130       if (player.isStandingOnTopOf(wall)) {
131         isStandingOnTop = true;
132       }
133     }
134
135     if (!isStandingOnTop && !player.isJumping)
136       player.startJump(0.1);
137   }
```

Listing 3.19: HeartsOfKlaus.java

Es folgen die von der Schnittstelle verlangten Getter-Metoden. Zunächst, um den Spieler zu erfragen:

```
138   @Override
139   public Player getPlayer() {
140     return player;
141   }
```

Listing 3.20: HeartsOfKlaus.java

Dann kann die Höhe des Spielfelds erfragt werden:

```
142   @Override
143   public int getHeight() {
144     return height;
145   }
```

Listing 3.21: HeartsOfKlaus.java

Und ebenso die Weite des Spielfelds:

```
146   @Override
147   public int getWidth() {
148     return width;
149   }
```

Listing 3.22: HeartsOfKlaus.java

Verloren hat der Spiele, wenn er keine Energie mehr hat.

```
150   @Override
151   public boolean lost() {
152     return energy<=0;
153   }
```

Listing 3.23: HeartsOfKlaus.java

Gewonnen hat er, wenn alle Herzen eingesammelt wurde.

```
154   @Override
155   public boolean won() {
156     return hearts.isEmpty();
157   }
```

Listing 3.24: HeartsOfKlaus.java

Eine Hilfsmethode ermöglicht es, eine Sounddatei abzuspielen.

```
158   void playSound(String soundFile){
159     try {
160         InputStream soundStream =
161            getClass()
162           .getClassLoader()
163           .getResourceAsStream(soundFile);
164
165         AudioInputStream stream
166           = AudioSystem.getAudioInputStream(soundStream);
167
168       Clip clip = AudioSystem.getClip();
169       clip.open(stream);
170       clip.start();
171       stream.close();
172     } catch (Exception e) {
173       e.printStackTrace();
174     }
175   }
176 }
```

Listing 3.25: HeartsOfKlaus.java

Die Klasse `GameScreen` ist in der Lage, dieses kleine Spiel anzuzeigen und zu starten:

```
1  package name.panitz.game.klaus;
2
3  import javax.swing.JFrame;
4  import name.panitz.game.GameScreen;
5
6  @SuppressWarnings("serial")
7  public class Start extends JFrame{
```

```
8   public static void main(String[] args) throws Exception {
9     Start f = new Start();
10    f.add(new GameScreen(new HeartsOfKlaus()));
11    f.pack();
12    f.setVisible(true);
13    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
14  }
15 }
```

Listing 3.26: Start.java

Abbildung 3.1 zeigt einen Screenshot des Spiels in Aktion. Die Spielfigur ist eine eigene Fotographie. Die Herzen wurden mit dem Bildbearbeitungsprogramm Gimp selbst erstellt, das Fass ist eine Fotographie, die mit Gimp zu einem animierten Gif bearbeitet wurde.

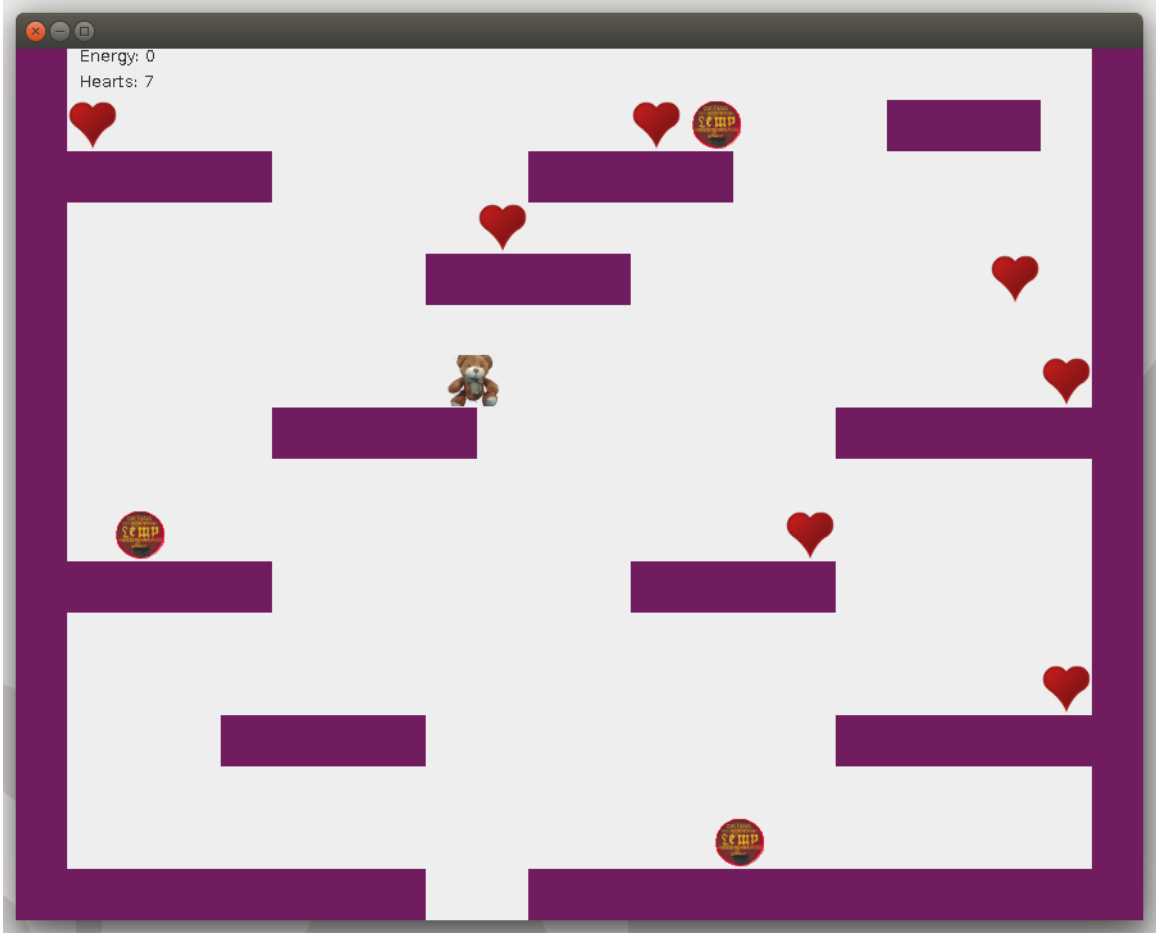

Abbildung 3.1: Heart of Klaus in Aktion