Praktische Informatik I
TutorIn: Berghöfer/Senger

WS 2012/13
Gruppe: 8

Beate Ruffer
Mohamadreza Khostevan
Leopold Schulz-Hanke

Übungsblatt 8

Aufgabenlösung
Abgabe: 16.12.2012

Aufgabe 1 Alles super hier (20%)

In unserem Spiel sind 2 Klassen implementiert dessen Instanzen Aufgehoben werden können: Hammer und Scissor. Sie besaßen schon die Superklasse `Collectable` seit einigen Abgaben. Auch die Klassen der Hindernisse: Bush, Stone und Comb erbten schon von der Superklasse `Obstacle`. Da wir aber in vorherigen Aufgaben mit Hilfe vom *instanceof*-Operator auf ein passendes Inventar prüfen sollten, war es nicht möglich das passende Inventar als Attribut zu hinterlegen und die Methode in der Superklasse die Prüfung zu überlassen. Nun haben wir eine Objektattribut `isBeatenBy` vom Typ Class eingeführt, mit dessen Hilfe der Vergleich durchgeführt wird. Somit können wir die prüfende Methode in der Superklasse Obstacle verschieben. Dadurch haben wir 3-Fach duplizierten Code in nur eine Methode zusammengeführt.

Listing 1: Klasse *Obstacle*

```
3  /**
4   * Oberklasse für alle Hindernisse. Diese Klasse vereint alle Hindernisse,
5   * so dass man leichter auf ein Hindernis prüfen kann.
6   *
7   * @author Beate Ruffer (Bea), Mohamadreza Khostevan (Amir), Leopold Schulz-
     Hanke (Leo)
8   * @version 2.0.0
9   *
10  */
11 public class Obstacle extends Actor
12 {
13     /**
14      * Sound Datei, die Abgespielt wird, wenn dieses Hindernisse überwunden
         wurde.
15      */
16     public String beatenSound;
17
18     /**
19      * Definiert durch Objekte welcher Klasse das Hindernis überwunden werden
          kann
20      */
21     protected Class isBeatenBy;
22
23     /**
24      * Konstruktor von Obstacle wird von der Subklasse aufgerufen.
25      * @param isBeatenBy Die Klasse des Objekts, welches das Hindernis ü
         berwinden kann.
26      */
27     public Obstacle(Class isBeatenBy){
28         this.isBeatenBy = isBeatenBy;
29     }
30
31     /**
```

```
32      * Prüft ob das sich im Inventar befindende Objekt eine Instanz von
         isBeatenBy ist. Falls ja, wird das Hindernis von dem Objekt
33      * im Inventar geschlagen und die Heldin kann passieren.
34      *
35      * @param collectable Das zu prüfende Objekt.
36      * @return true, wenn Hindernis überwunden ist.
37      */
38     public boolean isBeaten(Actor collectable)
39     {
40         return collectable.getClass() == isBeatenBy ;
41     }
42 }
```

Nun müssen die Klassen der Hindernisse, nur noch das passende Inventar (über den Konstruktor der Superklasse) und den Sound setzen.

Listing 2: Konstruktor von *Stone*

```
18 public Stone(){
19         super(Hammer.class);
20         beatenSound = "explosion.wav";
21     }
```

Tests. Da nun mal jedes kleine Refactoring Regression-Bugs hervorbringen kann, haben wir erneut getestet. Wir haben einen Hammer in unser Inventar aufgenommen, konnten wie erwartet keinen Bush oder Comb passieren. Nur der Stone konnte überwunden werden (der richtige Sound wurde auch abgespielt). Mit der Schere im Inventar konnten wir den Stone nicht überwinden, dafür aber Comb und Bush (ebenfalls mit den richtigen Sounds).

Aufgabe 2 Massenkarambolage (30%)

1. Methode

Listing 3: *getCollidingObject(Class cls)*-Methode

```
25 /**
26      * Liefert das Objekt der Klasse cls mit dem eine Kolliosion besteht
         .
27      * @param cls die Klasse, nach dessen Kollision geprüft wird
28      * @return das Objekt, mit dem Kollidiert wurde
29      */
30     public Actor getCollidingObject(Class cls){
31         return getOneIntersectingObject(cls);
32     }
```

2. Methode

Listing 4: *collidesWith(Class cls)*-Methode

```
25 /**
26      * Prüft ob gerade eine Kollision besteht
27      * @param cls die Klasse, nach dessen Kollision geprüft wird
28      * @return ob gerade eine Kollision mit einer Instanz der Klasse cls
          besteht
29      */
```

```
30      public boolean collidesWith(Class cls){
31          return getCollidingObject(cls) != null;
32      }
```

3. Methode

Listing 5: *newCollisionWith(Class cls)*-Methode

```
25 /**
26       * Prüft, ob gerade eine neue Kollision mit einer Instanz
27       * der Klasse cls begonnen hat.
28       * @param cls die Klasse, nach dessen Kollision geprüft wird
29       * @return ob gerade eine neue Kollision mit einer Instanz der
          Klasse cls begonnen hat
30       */
31      public boolean newCollisionWith(Class cls) {
32          Crashable obj =  null;
33          if ( collidesWith(cls) ){
34              if (collisions.get(cls) == null || collisions.get(cls) !=
                 obj ){
35                  collisions.put(cls, obj);
36                  return true;
37              }
38          }
39          return false;
40      }
```

Erneut musste einiges an Refactoring gemacht werden. Alle Klassen, mit dessen Instanzen die Biene nun kollidieren kann, erben nun von `Crashable`. Jede Subklasse von Crashable hat nun auch eine Methode `handleCrash(Bee bee)`, mit der sie auf (von der Biene registrierten) Kollisionen reagieren kann. Registriert die Biene nun eine Kollision, ruft sie den Crash Hanlder des Objektes auf, mit dem sie kollidiert. Dabei übergibt sie sich selbst als Paramter (für eventuelle Zugriffe auf das Inventar oder Scoreboard der Biene).

Aufgabe 3 Richtig kollidieren (50%)

Da nun alle Klassen, die mit der Biene kollidieren können, Subklassen von `Crashable` sind, somit braucht sie nicht mehr zu wissen, mit welcher Subklasse von Crashable kollidiert wird, sondern braucht nur noch dessen Crash-Handler aufzurufen.

Listing 6: *checkCollisions()*-Methode

```
156 /**
157       * Prüft, ob die Biene mit einem Hindernis kollidiert. Falls eine
           Kollision existiert, wird
158       * der Crash Handler des Objekts aufgerufen.
159       */
160      private void checkCollisions()
161      {
162          if (newCollisionWith(Crashable.class))  {
163              Crashable crashable = (Crashable) getCollidingObject(Crashable.
                  class);
164              crashable.handleCrash(this);
165          }
166      }
```

Einzige Ausnahme dabei sind die *Collectables*. Dies sind die Objekte, die in das Inventar aufgenommen werden können. Da nicht mit jedem act()-Aufruf auf Kollisionen mit Collectables geprüft werden soll, sondern nur auf Tastendruck, erben Collectables nicht von Crashable (haben somit auch keinen Crash Handler). Es wird weiterhin auf Tastendruck die Methode takeOrFreeCollectable() aufgerufen. Diese nutzt aber zur Kollisionsprüfung nun unsere neuen Methoden.

Listing 7: *takeOrFreeCollectable()*-Methode

```
168 /**
169      * Wenn ein Collectable in der Nähe und das Inventar leer ist, wird das
          Collectable eingesammelt.
170      * Beindet sich ein Actor im Inventar, wird dieser an der aktuellen
          Position der Biene in die
171      * Welt gesetzt. Andernfalls wird der Fehlerton abgespielt.
172      */
173     private void takeOrFreeCollectable()
174     {
175         if (newCollisionWith(Collectable.class) && myInventory.isEmpty() ){
176             Collectable collectable = (Collectable) getCollidingObject(
                  Collectable.class);
177             myInventory.pushToInventory(collectable);
178         } else if (!myInventory.isEmpty()){
179             myInventory.removeFromInventory(this.getX(), this.getY());
180         } else {
181             Greenfoot.playSound("out.wav");
182         }
183     }
```

Kommen wir nun zu der Implementation der 3 Methoden zur Kollisionsprüfung. Im Vergleich zu Aufgabe 2, haben wir nur die erste Methode erweitert.
`getCollidingObject(Class cls)` gibt uns das Objekt zurück, mit dem Tatsächlich (also nicht nur die Bounding Box) kollidiert wird.

Listing 8: *getCollidingObject(Class cls)*-Methode

```
22 /**
23      * Liefert das Objekt der Klasse cls, mit dem eine Kolliosion besteht.
24      * @param cls die Klasse, nach dessen Kollision geprüft wird
25      * @return das Objekt, mit dem Kollidiert wurde
26      */
27      public Actor getCollidingObject(Class cls){
28         List<Actor> objects = getIntersectingObjects(cls);
29
30         for (Actor object : objects){
31                 for (int x = 0; x < this.getImage().getWidth(); x++){
32                     for (int y = 0; y < this.getImage().getHeight(); y++){
33                         if (this.getImage().getColorAt(x, y).getAlpha() > 0){
34                             double rotation = Math.toRadians(this.getRotation
                                ());
35                             double dx = x - this.getImage().getWidth() / 2;
36                             double dy = y - this.getImage().getHeight() / 2;
37                             int xWorld = (int) (this.getX() + dx * Math.cos(
                                rotation) - dy * Math.sin(rotation));
38                             int yWorld = (int) (this.getY() + dx * Math.sin(
                                rotation) + dy * Math.cos(rotation));
39                             if (pixelsWithinImageBounds(object, xWorld,
                                yWorld) && visiblePixelAt(xWorld, yWorld, object
                                )){
40                                 if (testMode){
```

```
41                                          this.getImage().setColorAt(x, y, Color.
                                             RED);
42                                      } else {
43                                          return object;
44                                      }
45                                  }
46                              }
47                          }
48                      }
49
50          }
51          return null;
52      }
```

Zunächst speichern wir eine Liste mit allen Kollidierten Objekten (noch Bounding Box). Diese Liste wird durchlaufen und darin wird jeweils jedes Pixel des Bildes durchlaufen (Zeilen 31 & 32).
Im Falle eines nicht-transparenten Pixels (Zeile 33) wird dieses Pixel in die Weltkoordinaten umgerechnet. Anhand dieser Koordinaten können wir dann die Koordinaten im kollidierendem Objekt errechnen. Doch vorher müssen wir prüfen, ob diese Koordinaten überhaupt innerhalb des kollidierten Objekts liegen. Dies geschieht mit `pixelsWithinImageBounds(Actor obj, int x, int y)`.

Listing 9: *pixelsWithinImageBounds(Actor obj, int x, int y)*-Methode

```
97  /**
98       * Prüft ob ein die Koordinaten innerhalb des Bildes eines Objektes
          liegen
99       * @param obj Das zu Prüfende Objekt
100      * @param x X-Koordinate
101      * @param y Y-Koordinate
102      * @return true, wenn die Koordinaten innerhalb des Bildes liegen
103      */
104     private boolean pixelsWithinImageBounds(Actor obj, int x, int y){
105         int xStart = obj.getX() - obj.getImage().getWidth() / 2;
106         int xEnd = xStart + obj.getImage().getWidth();
107         int yStart = obj.getY() - obj.getImage().getHeight() / 2;
108         int yEnd = yStart + obj.getImage().getHeight();
109         return x >= xStart && x < xEnd && y >= yStart && y < yEnd;
110     }
```

Liegen die Koordinaten also innerhalb des kollidierenden Objekts, können die Weltkoordinaten in die Bildkoordinaten dieses Objekt umgerechnet werden. Dann wird geprüft, ob auch dieses Pixel nicht-transparent ist.

Listing 10: *visiblePixelAt(int xWorld, int yWorld, Actor obj)*-Methode

```
80  /**
81       * Rechnet Weltkoordinaten in die Bildkoordinaten des Actors um und prüft
82       * ob dort ein Sichrbarer sichtbarer Pixel ist
83       * @param xWorld x-Weltkoordiante
84       * @param yWorld y-Weltkoordinate
85       * @param obj Das Objekt in dessen Bildkoordinaten umgerechnet werden
          soll
86       * @return true, wenn an den Koordinsten ein Sichtbarer Pixel im Bild ist
87       */
88      private boolean visiblePixelAt(int xWorld, int yWorld, Actor obj){
89          double rotation = Math.toRadians(obj.getRotation());
```

```
90          int dx = xWorld - obj.getX();
91          int dy = yWorld - obj.getY();
92          int xInObject = (int) (obj.getImage().getWidth() / 2 + dx * Math.cos(
             rotation) + dy * Math.sin(rotation));
93          int yInObject = (int) (obj.getImage().getHeight() / 2 - dx * Math.sin
             (rotation) + dy * Math.cos(rotation));
94          return (obj.getImage().getColorAt(xInObject, yInObject).getAlpha() >
             0 );
95      }
```

Liefern pixelsWithinImageBounds() und visiblePixelAt() beide true zurück, wird das Objekt, mit dem man dann tatsächlich Kollidiert ist zurückgeliefert. Dies geschieht aber nicht im testMode. In dem Fall wird lediglich das entsprechende Pixel rot gefärbt (Listing 8 Zeile 40).
Der testMode ist ein Objektattribut in Collider. Dieser kann auf Wunsch einfach auf true gesetzt werden.