Praktische Informatik I
TutorIn: Berghöfer/Senger

WS 2012/13
Gruppe: 8

Beate Ruffer
Mohamadreza Khostevan
Leopold Schulz-Hanke

Übungsblatt 9

Aufgabenlösung
Abgabe: 06.01.2013

Aufgabe 1 Java-Krise (30%)

If-else dient der bedingten Anweisung, also der Ausführung einer Anweisung unter einer bestimmten Bedingung (*if*). Die Bedingung ist dabei ein Ausdruck der als Ergebnis einen booleschen Wert (*true* oder *false*) liefern muss. Trifft die Bedingung nicht zu, wird eine alternative Anweisung ausgeführt (*else*). Der Anweisungskörper wird hierbei nur *ein Mal* ausgeführt.

Nehmen wir als Beispiel diese if-Verzweigung:

```
1     public boolean thisIsTrue = true;
2     public void ifCheckIf(){
3         if (thisIsTrue){
4             System.out.println("Something");
5         }
6         else {
7             System.out.println("Something ELSE");
8         }
9     }
```

Ist die Bedingung wahr, (also $thisIsTrue = true$) wird die Anweisung des *if*-Blocks ausgeführt, ist die Bedingung falsch ($thisIsTrue = false$), wird die Anweisung des *else*-Teils ausgeführt.

Möchten wir dies nun als *while Schleife* implementieren, müssen wir darauf achten, dass der Anweisungskörper ebenfalls nur einmal durchgeführt wird. Wir implementieren also gleich beim ersten Durchlauf die Abbruchbedingung. Die boolsche Variable i wird vor jeder Bedingungsprüfung auf true gesetzt und im Anweisungskörper sofort wieder auf false.

Im Beispiel:

```
1  public void ifCheckWhile(){
2          // if Zweig
3          boolean i = true;
4          while (i && thisIsTrue){
5              System.out.println("Something");
6              i = false;
7          }
8          // else Zweig
9          i=true;
10         while (i && !thisIsTrue){
11             System.out.println("Something ELSE");
12             i=false;
13         }
14     }
```

If lässt sich somit ohne größeren Aufwand durch *while* ersetzen.

Andersrum ist das etwas schwieriger und kann nicht ohne Einschränkungen umgesetzt werden. Eine *If*-Anweisung kann immer nur genau `ein Mal` ausgeführt werden. Möchten wir damit eine *while*-Schleife ersetzen, müssen wir einen Weg finden, die *if*-Anweisung mehrmals (so lange die Bedingung wahr ist) auszuführen.

Versuchen wir die folgende *while*-Schleife nur mit Hilfe von *if* zu implementieren:

```
1  public void whileCheckWhile(){
2          int i = 0;
3          while(i < 10){
4              System.out.println(i);
5              i++;
6          }
7      }
```

Der Wert von i wird also so lange ausgegeben, bis die Bedingung $i < 10$ nicht mehr wahr ist.

Da sowohl *while* als auch *if-else* von der Wahrheit einer Bedingung abhängen, ist es theoretisch möglich *while* durch *if* zu ersetzen. Um aber die wiederholte Bedingungsprüfung und Ausführung des Schleifenkörpers mit *if* zu erzwingen, müssen wir eine Methode schreiben, welche sich selbst rekursiv aufruft:

```
1 // Es muss whileChekIf(0) ausgeführt werden
2      public void whileCheckIf(int i){
3          if (i < 10){
4              System.out.println(i);
5              whileCheckIf(i+1);
6          }
7      }
```

Führt man nun `whileCheckIf(0)` auf, werden genau wie bei der *while*-Schleife alle Zahlen von 0 bis 9 Ausgegeben.

Hieraus ist ersichtlich, dass sich eine bedingte *if-else*-Anweisung durchaus durch mehrere *while*-Schleifen ersetzen lässt. Es muss lediglich, um eine unendliche Wiederholung der Schleife zu vermeiden entweder der zu prüfende Ausdruck (unpraktisch) oder ein zusätzlich eingeführter Parameter (wie im Beispiel) verändert werden, damit die Bedingung zur Ausführung nicht mehr wahr ist. Auch weitere Verzweigungen sollten sich theoretisch so darstellen lassen, wenngleich dies wahrscheinlich schnell relativ unübersichtlich würde.

Aufgabe 2 Delegationsspiel

Aufgabe 2.1 Vorbereiten (15%)

Die Klasse `Item` hat nun 2 Methoden bekommen, mit denen die Items selbst über ihr Verhalten bestimmen können.

Listing 1: Klasse *Item*

```
1 import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot und MouseInfo
  )
2
3 /**
4  * Ein Objekt, das aufgehoben werden kann. (Übungsblatt 8)
5  *
6  * @author Thomas Röfer
```

```
7  * @version 02.12.2012
8  */
9  public abstract class Item extends Actor
10 {
11     /**
12      * Standrad Methode um das Item zu nutzen, wenn das Item diese Methode
        nicht überschreibt
13      * @param robot Unser Roboter
14      * @return Das Item, welches das im Inventar liegende Item ersetzt
15      */
16     public Item useItem(Robot robot){
17         if (robot.collidesWith(Obstacle.class)){
18             Obstacle obstacle = (Obstacle) robot.getCollidingObject(Obstacle.
                class);
19             if (matches(obstacle)){
20                 robot.getWorld().removeObject(obstacle);
21                 return null;
22             }
23         }
24         return this;
25     }
26
27     /**
28      * Abstrakte Methode, die prüft ob das Item zum Hinderniss passt.
29      * @param obstacle Das zu prüfende Hinderniss
30      * @return true, wenn das Item zum Hinderniss passt
31      */
32     public abstract boolean matches(Obstacle obstacle);
33 }
```

Wir möchten im Inventar die Nutzung des Items anstoßen, ohne uns darum kümmern zu müssen, um welches Item es geht. Wir werden zu keiner Zeit Instanzen von Item direkt erzeugen, daher können wir abstrakte Methoden implementieren, die dann in den Subklassen überschrieben werden. Die Methode `matches()` ist abstrakt, da wir diese auf jeden Fall in den Subklassen überschreiben müssen, denn nur die Subklasse kann wissen, welches Hindernis zu diesem Item passt.
`useItem()` wurde nicht abstrakt implementiert, damit wir auf diese Weise ein Standardverhalten für Items haben. Wenn diese Methode also nicht überschrieben wird, kann man mit dem Item das passende Hindernis überwinden, es wird jedoch weder ein Sound abgespielt noch werden Punkte gutgeschrieben.

Aufgabe 2.2 Delegieren (25%)

Jede Subklasse von Item kann nun sein eigenes Verhalten definieren, indem sie `useItem(Robot robot)` überschreibt. Tut sie dies nicht, wird das Standardverhalten genutzt.
Mit der Methode `matches(Obstacle obstacle)` definieren wir welches Hindernis zum Item passt.

Listing 2: Klasse *Key*

```
1 /**
2  * Ein Schlüssel. Kann benutzt werden, um Türen zu öffnen. (Übungsblatt 3)
3  *
4  * @author Thomas Röfer
5  * @version 01.11.2012
6  */
7 public class Key extends Item
```

```
8  {
9      /**
10      * Methode um das Item zu nutzen
11      * @param robot Unser Roboter
12      * @return Das Item, welches das im Inventar liegende Item ersetzt
13      */
14     public Item useItem(Robot robot){
15         if (robot.collidesWith(Obstacle.class)){
16             Obstacle obstacle = (Obstacle) robot.getCollidingObject(Obstacle.
                class);
17             if (matches(obstacle)){
18                 robot.getWorld().removeObject(obstacle);
19                 robot.getScore().setScore(robot.getScore().getScore() + 100);
20                 Greenfoot.playSound("door-open.wav");
21                 return null;
22             }
23         }
24         return this;
25     }
26
27     /**
28      * Methode, die prüft ob das Item zum Hinderniss passt.
29      * @param obstacle Das zu prüfende Hinderniss
30      * @return true, wenn das Item zum Hinderniss passt
31      */
32     public boolean matches(Obstacle obstacle){
33         return obstacle instanceof Door;
34     }
35 }
```

Wir können nun die Methoden `blowUpWall()` und `openDoor()` aus der Klasse `Robot` entfernen. Stattdessen führen wir in jedem `act()`-Zyklus `inventory.useInventory(this);` aus.
Falls ein Item im Inventar ist, wird das Item nun genutzt.

Listing 3: Klasse *Inventory*, Methode *useInventory*

```
67 /**
68      * Benutzt das im Inventar leigende Item und ersetzt gegebenfalls das
         Item
69      * @param robot Der Roboter wird übergeben.
70      */
71     public void useInventory(Robot robot){
72         if (!isEmpty()){
73             Item item = contents.useItem(robot);
74             if (item != null){
75                 store(item);
76             } else {
77                 clear();
78             }
79         }
80     }
```

Das zurückgelieferte Item wird wieder im Inventar gespeichert. Standardverhalten - wenn das Item nicht genutzt werden kann - ist, dass sich das Item selbst zurückliefert.
Nach der Benutzung wird *null* zurückgeliefert. In diesem Fall wird das Inventar geleert.

Wir haben nun also 2 Varianten implementiert: Rückgabe des Items selbst und Rückgabe von *null*. Als dritte Variante sollen wir nun ein anderes Item zurückgeben (z.B. Durch Kombination

von Item und Hindernis).
Dazu haben wir nun zusätzlich zu dem Item `Bomb` ein weiteres Item `BombFire` und ein weiteres Hindernis `Fire`.
Um eine Wand sprengen zu können, benötigen wir nun nicht einfach eine Bombe, sondern müssen diese vorher anzünden indem wir (mit der Bombe im Inventar) mit dem Feuer kollidieren. Dann wird die Bombe im Inventar durch die brennende Bombe (BombFire) ersetzt.

Listing 4: Klasse *Bomb*, Methode *useItem()*

```
11 /**
12      * Methode um das Item zu nutzen
13      * @param robot Unser Roboter
14      * @return Das Item, welches das im Inventar liegende Item ersetzt
15      */
16     public Item useItem(Robot robot){
17         if (robot.collidesWith(Obstacle.class)){
18             Obstacle obstacle = (Obstacle) robot.getCollidingObject(Obstacle.
                class);
19             if (matches(obstacle)){
20                 robot.getWorld().removeObject(obstacle);
21                 Greenfoot.playSound("Bottle.aiff");
22                 return new BombFire();
23             }
24         }
25         return this;
26     }
```

Testen Durch die Änderungen wurde das Verhalten mit dem Schlüssel nicht beeinflusst. Nimmt man diesen ins Inventar auf, kann man nach wie vor keine Wand sprengen. Die korrekte Punkteanzahl wird gutgeschrieben ebenso auch der richtige Sound abgespielt. Mit der Bombe im Inventar können nun keine Wände mehr gesprengt werden. Kollidiert man aber nun mit dem Feuer, wird dieses aus der Welt entfernt und die Bombe im Inventar wird durch die brennende Bombe ersetzt. Damit lassen sich nun die Wände sprengen (Sound korrekt, Punkte korrekt). Das restliche Verhalten des Spiels bleibt weiterhin unverändert.

Aufgabe 2.3 Reflektieren (10%)

Durch die Auslagerung der Logik jedes Items in seine eigene Klasse haben wir die Wartbarkeit um einiges verbessert. Möchten wir das Verhalten eines Items nun ändern, wissen wir genau, dass es in dem Item selbst zu finden ist und müssen nicht mehr die entsprechende Methode in der Klasse Robot suchen. Außerdem hat sich dich die Übersichtlichkeit der Klasse `Robot` dadurch stark verbessert.
Wenn wir unser Spiel nun erweitern möchten (wie wir es z.B. mit der brennenden Bombe gemacht haben), müssen wir nur noch eine Subklasse von Item erzeugen und dessen Logik darin implementieren. Es können nun auch ohne viel Aufwand Items mit Standardverhalten hinzugefügt werden, es muss lediglich die Methode `matches()` überschrieben werden.
Es wäre ratsam auf gleiche Weise das Verhalten des Smileys und des Totenkopfs in sich selbst zu implementieren statt in der Klasse `Robot`. Das ist aktuell schon mit dem abspielen des Tons so definiert, doch auch der Punktewert (bei Smiley) und das Starten des Replays (Skull) sollten dort definiert sein. Gäbe es eine Superklasse dieser beiden, könnte man ein Standardverhalten für weitere Objekte festlegen, mit denen man Punkte sammelt.

Aufgabe 3 Buh! (20%)

Das Spielfeld besitzt eine labyrinthartige Form. Dies sorgt für eine starke Einschränkung der Manövrierfähigkeit des Spielers. Um einen Gegner zu erstellen, welcher trotz der genannten Umstände für den Spieler überwindbar sein sollte, musste dieser ebenfalls in seinen Fähigkeiten eingeschränkt werden.
Unsere Aufgabe war es ein Monster zu erstellen, welches in seinen Bewegungen eine Tendenz zum Spieler besitzt.

Eine der hilfreiche Methoden hierzu ist die Methode `turnTowards(x,y)`. Diese Methode ist ein Wegweiser zu jedem Objekt im Spiel, solange sich dieses in der selben Welt befindet und das Monster sich durch Wände bewegen kann. Aufgrund dieser Tatsache implementierten wir ein Monster, welches aus der Klasse `Ghost` besteht.
Es stellte sich bereits nach einem Versuch heraus, dass sich das Monster zu effektiv verhielt, es erfasste den ersten Spieler auf dem Spielfeld und wanderte mit einer geraden Linie durch alle Wände auf ihn zu.

Eine Kollision mit Instanzen der Klasse `Enemy` sorgt genau so wie eine Kollision mit der Klasse `Skull` für ein Replay der letzten Spielerbewegungen und das Ende des Spiels. Der Spieler hatte praktisch keine Chance zu entkommen. Die Herausforderung bestand somit nicht länger im Erstellen eines starken Gegners, sondern darin einen Gegner zu erstellen, der dem Spieler eine Chance ließ dem Gegner auszuweichen oder von diesem verfehlt zu werden.

Listing 5: Klasse *Ghost*

```
1  import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot und MouseInfo
   )
2  import java.util.*;
3  /**
4   * Beschreiben Sie hier die Klasse Ghost.
5   *
6   * @author Beate Ruffer (Bea), Mohamadreza Khostevan (Amir), Leopold schulz-
    Hanke (Leo)
7   * @version (Eine Versionsnummer oder ein Datum)
8   */
9  public class Ghost extends Enemy
10 {
11     private Robot target;
12
13     /**
14      * Act - Tue was immer Ghost tun möchte. Diese Methode wird aufgerufen,
15      * wenn die 'Act'- oder 'Run'-Knöpfe in der Umgebung gedrückt werden.
16      */
17     public void act()
18     {
19         if (newCollisionWith(Robot.class)) {
20             Greenfoot.playSound("sad-trombone.wav");
21             target = null;
22         }
23         move(1);
24
25         //Kollision mit Spielfeldgrenzen
26         int width  = getWorld().getWidth();
27         int height = getWorld().getHeight();
28         int iwidth  = getImage().getWidth();
29         int iheight = getImage().getHeight();
30         if (this.getX()-(iwidth/2)<=0 || this.getX()+(iwidth/2)>= width||
31         this.getY()-(iheight/2)<=0 || this.getY()+(iheight/2)>=height) {
32             turn(3);
```

```
33          }
34
35          //Warscheinlichkeit sich zum Spieler zu drehen
36          if(Greenfoot.getRandomNumber(150)==1) {
37              setNextTarget();
38              turnTowardsObject(target);
39          }
40          else{
41              turn(Greenfoot.getRandomNumber(6)-3);
42          }
43      }
44
45      private void setNextTarget(){
46          //erfasst den nächsten/neusten Spieler im Radius und setzt den Geist
             auf diesen an
47          Robot obj= (Robot) ObjectInRange(1000,Robot.class);
48          if(obj != target && obj != null) {
49              target = obj;
50          }
51      }
52  }
```

Zeilen 19-22 erkennen eine potentielle Kollision mit dem Spieler, das Spiel wird jedoch erst gestoppt, wenn der Spieler ebenfalls eine Kollision mit dem Geist erfasst.

Der Geist reagiert nicht auf Kollisionen mit Hindernissen, andererseits erfasst er jedoch Kollisionen mit den Spielfeldgrenzen. Dieser Vorgang findet in den Zeilen 25-33 statt. Sollte das Bild des Geistes sich mit der Grenze schneiden oder diese überschreiten, so beginnt der Geist sich im Uhrzeigersinn zu drehen (Zeile 32).

Der Geist selbst kann wie alle anderen Objekte das Spielfeld nicht verlassen. Der oben beschriebene Mechanismus verhindert jedoch, dass die Klasse dauerhaft am Spielfeldrand hängen bleibt oder sich in einer der Spielfeld-Ecken verfängt.

In den nächsten Zeilen folgt ein Mechanismus, der die zufällige Bewegung des Geistes erzeugt (Zeilen 35-42). Der Geist besitzt bei jedem Durchgang seines Programms eine Chance von 1/150 sich mittels der Methode `turnTowardsObject()` zu einem Spieler zu drehen, der unter dem Parameter target vermerkt ist.
Der jeweilige Spieler wird hierbei durch die Methode `setNextTarget()` bestimmt (Zeilen 45-50) Diese Methode erfasst ein Objekt, welches sich im genannten Radius des Geistes befindet. Sollte es ein anderes Objekt der Klasse `Roboter` sein als momentan für den Parameter *target* eingespeichert ist, so wird dieses für den Geist als neues Ziel gesetzt.
Der Radius lässt sich beliebig variieren solange der Geist bei dessen ersten Auslösung ein Objekt im Radius erfasst, das er jagen kann.

Wenn sich der Geist bei der Chance von 1/150 mal nicht zum Spieler dreht (Zeilen 40-42), so dreht er sich bei jeder Auslösung in einem zufälligen Winkel von +- 3 Grad

Dieser Vorgang sorgt für ein unregelmäßiges Bewegungsmuster des Geistes über das Spielfeld.

Möchte man den Schwierigkeitsgrad erhöhen, so erhöht man einfach die Chance das sich der Geist zum Spieler dreht.

Als zusätzlichen Gegner fürs Labyrinth haben wir zur Klasse `Enemy` die Klasse `Walker` hinzugefügt. Jene orientiert sich jedoch nicht Spieler selbst und bewegt sich per Zufall im Labyrinth. Sie stellt für den Spieler jedoch eine Gefahr dar, sollte sie ihn in eine Sackgasse drängen.

Aufgabe 4 Bonusaufgabe: Was war das für ein Geräusch? (5%)

Alle unsere Gegner (`Ghost` und `Walker`) erben von ihrer Superklasse die Methode `turnTowardsObject()`

Listing 6: Methode *turnTowardsObject()*

```
31 /**
32      * Dreht das Objekt in Richtung eine Actors. Im Moment werden die Objekte
33      * in Richtung des Roboters gedreht.
34      * @param actor in dessen Richtung sich der Gegner dreht
35      */
36     public void turnTowardsObject(Actor actor){
37         turnTowards(actor.getX(),actor.getY());
38     }
```

Nun möchten wir, dass unsere Gegner sich bei jedem Geräusch in Richtung des Roboters drehen. Unsere Klasse `Game` bekommt also eine statische Methode, die das für uns übernimmt.

Listing 7: Methode *playSound()* in der Klasse Game

```
43 /**
44      * Spielt einen Sound ab. Da aber alle Gegner sich bei jedem Sound in
45      * Richtung des Roboters drehen sollen, wird hier der Roboter als
         Argument
46      * erwartet.
47      * @param sound Der Abgespielt werden soll
48      * @param robot zu dem sich die Gengner Hinwenden sollen
49      */
50     public static void playSound(String sound, Robot robot){
51         Greenfoot.playSound(sound);
52         List<Enemy> enemys = robot.getWorld().getObjects(Enemy.class);
53             for (Enemy enemy : enemys) {
54                 enemy.turnTowardsObject(robot);
55             }
56     }
```

Sie bekommt als Argument den Namen der Sounddatei, die abzuspielen ist und auch unseren Roboter. Somit kann man anhand des Roboters die Welt ermitteln und ihn auch der `turnTowardsObject()`-Methode übergeben. Diese Methode wird für alle in der Welt vorhandenen `Enemy`-Instanzen ausgeführt.
Somit wird nun immer diese Methode (z.B. `Game.playSound("hooray.wav", robot);`) ausgeführt um Sound abzuspielen.