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*).

Ein Beispiel verdeutlicht dies:

```
1     public boolean thisIsTrue = false;
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.

*While* wiederum wiederholt eine Anweisung solange die Bedingung wahr ist, wobei die Bedingung vor der Anweisung geprüft wird.

Im Beispiel:

```
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 eine Anweisung durch die andere zu ersetzen:

```
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      }
```

Hieraus ist ersichtlich, dass sich eine bedingte *if-else*-Anweisung durchaus durch mehrere *while*-Schleifen ersetzen lässt, wenn man die jeweilige Bedingung des *if*-Teils als Bedingung der Durchführung einer *while*-Schleife formuliert. Ist die Bedingung wahr, werden die entsprechenden Anweisungen ausgeführt, ist die Bedingung falsch, werden die Anweisungen der alternativ für den *else*-Teil formulierten *while*-Schleife ausgeführt. 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.

Auch umgekehrt lässt sich eine *while*-Schleife durch eine *if-else* Anweisung ersetzen:

```
1  // Es muss whileChekIf(0) für dasselbe Ergebnis wie bei whileCheckWhile()
   ausgeführt werden
2  public void whileCheckIf(int i){
3           if (i < 10) {
4               System.out.println(i);
5               whileCheckIf(i+1);
6           }
7           else {
8           ;
9           }
10      }
```

Hier wird in einer Methode wie bei einer *while*-Schleife durch *if* geprüft, ob die Bedingung zur Ausführung des folgenden Anweisungsblocks wahr ist und für diesen Fall einfach die Methode erneut rekursiv aufgerufen und funktioniert dementsprechend genauso wie eine Schleife. Ebenso wird für den Fall, dass die Bedingung nicht (mehr) wahr ist, der rekursive Aufruf abgebrochen bzw. gar nicht erst durchgeführt. Dazu muss sich die *if-else*-Anweisung allerdings in einer Methode befinden (denn nur so kann sie sich selbst aufrufen).

# 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 auch ohne viel Aufwand nun 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%)

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