Praktische Informatik I
TutorIn: Berghöfer/Senger

WS 2012/13
Gruppe: 8

Beate Ruffer
Mohamadreza Khostevan
Leopold Schulz-Hanke

Übungsblatt 7

Aufgabenlösung
Abgabe: 09.12.2012

Aufgabe 1 Der TutorInnen Leid... (70%)

Methode 1 (10%). Ganzzahlige Wurzel

```
3  public int method(int i)
4      {
5          int a = i;
6          int b = 1;
7          while (a - b > 1) {
8              a = (a + b) / 2;
9              b = i / a;
10         }
11         return (a + b) / 2;
```

Die Methode berechnet die Wurzel einer eingegebenen Zahl und gibt diese Zahl zurück. Berechnet wird die Wurzel hier nach dem Heron-Verfahren, welches eine Näherung der Quadratwurzel einer Zahl im ganzzahligen Bereich (da alle i, a und b als Integer deklariert werden) mit der Formel

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}$$

berechnet. Auf die Methode hier angewendet wird also zunächst, wenn i-1 $>$ 1 ist, x_0 festgelegt:

$$a_0 = \frac{a+b}{2} = x_0 = \frac{i+1}{2}$$

Wenn danach a - b $>$ 1 folgt

$$a_1 = \frac{a_0 + \frac{i}{a_0}}{2} = x_1 = \frac{x_0 + \frac{i}{x_0}}{2}$$

Solange a - b $>$ 1 wird nun

$$a_{n+1} = \frac{a_n + b_n}{2}$$

mit b $= \frac{i}{a_n}$ und $a_n = \frac{a_{n-1}+b_{n-1}}{2}$ berechnet.

Wenn das Abbruchkriterium a - b ≤ 1 erfüllt ist, wird der Wert der letzten Berechnung von $\frac{a_n+b_n}{2}$, der der ganzzahligen Näherung der Wurzel von i entspricht, zurückgegeben. Da die Methode jedoch offenbar nicht weiß, dass es keine negativen Quadratzahlen gibt, berechnet sie allerdings natürlich auch Werte für eingebene negative Zahlen.

Methode 2 (10%). Fakultät

```
14 public int $(int _)
15     {
16         return _ == _ + _ ? -~_ : _ * $(_ - _ / _);
17     }
```

Teilen wir zunächst das return Statement in verständliche Abschnitte. Der Ausdruck `_ == _ + _` wird auf true oder false getestet. Da nur die 0 die Summe einer Addition mit sich selbst ist, ist dieser Ausdruck eine Abfrage ob `_ == 0` ist.
Nun wird im true Fall der Teil zwischen `?` und `:` ausgeführt und im false Fall, der Teil nach dem `:`. Betrachten wir den true Fall: `-~_`. Wir wissen nun, dass unsere Zahl (der Unterstrich) in diesem Fall nur 0 sein kann. Die Tilde bildet das bitwise complement der 0, also -1. Dieses wird wiederum durch das Minus negiert, was 1 ergibt. Im true Fall wird also immer 1 zurückgegeben. Nun der false Fall: `_ * $(_ - _ / _)` Die Methode ruft sich selber rekursiv auf und multipliziert das Ergebnis mit sich selbst vermindert um 1. Der Ausdruck des übergebenen Parameters lässt sich vereinfachen, da aufgrund der Erstausführung von Punktrechnung vor Strichrechnung die Zahl durch sich selbst geteilt 1 ergibt: `$(_ - 1)`

Nun haben wir die Methode in ihre Einzelheiten zerlegt und können sie jetzt eine leserlichere Form bringen. Dadurch ist zu erkennen, dass die Methode die Fakultät einer Zahl berechnet, denn unsere Zahl wird so lange mit sich selbst -1 multipliziert bis die Abbruchbedingung `number == 0` erreicht ist.

Listing 1: Methode factorial()

```
65 public int factorial(int number)
66     {
67         if (number == 0)
68         {
69             return 1;
70         }
71         else
72         {
73             return number * factorial(number - 1);
74         }
75     }
```

Methode 3 (10%). Maximum Die Methode `max(int[] values)` bestimmt das Maximum der Werte eines Arrays, nur dann, wenn das Maximum eine positive Zahl ist. Da die 0 bereits als Maximum definiert wird, werden negative Werte ignoriert.

Listing 2: Korrigierte max() Methode

```
81 public int max2(int[] values)
82     {
83         int maximum = 0;
84         if (values.length != 0){
85             maximum = values[0];
86         }
87         for (int value : values) {
88             if (value > maximum) {
89                 maximum = value;
90             }
91         }
92         return maximum;
93     }
```

Da wir noch keine Exceptions implementieren können, belassen wir die Initialisierung von maximum mit 0. Dies ist der Wert, der zurückgegeben wird, wenn wir ein leeres Array übergeben. Bei einem nicht-leerem Array nutzen wir nun das erste Element als Vergleichswert. So stellen wir sicher, dass auch negative Elemente verglichen werden.

Methode 4 (15%). Matrix transponieren

Im Prinzip wird das zweidimensionale Array in zwei ineinander geschachtelten for-Schleifen durchlaufen. Die Matrix hat also die Form:

[0][0] [0][1]

[1][0] [1][1]

Dabei wird der aktuelle Wert des Arrayelements mit dem Index [x][y] in "temp" gespeichert und an das Element mit dem Index [y][x] übergeben, während das Element mit dem Index [x][y] den Wert des Elements mit dem Index [y][x] zugewiesen bekommt. Das heißt, im ersten Durchlauf bleibt das Element das gleiche, da x und y beide 0 sind ([0][0] ist das gleiche wie [0][0]). Für das Element mit dem Index [1][1], also y=1 und x=1 gilt dies analog. Da jedoch das komplette, zweidimensionale Array durchlaufen wird, werden also zunächst im zweiten Durchlauf für x=1 und y=0 die Elemente [1][0] und [0][1] vertauscht, in Bezug auf die Matrix also wie gewünscht an der Diagonale gespiegelt, jedoch werden im nächsten Durchlauf für y=1 und x=0 die beiden Elemente erneut getauscht, diesmal [0][1] mit [1][0], sodass die Ausgangsamtrix wiederhergestellt wird.

Um dies zu verhindern dürften die beiden Elemente [1][0] und [0][1] also nur einmal vertauscht werden, z.B. indem bei einem zweidimensionalen Array die innere for-Schleife nur x<1 ausgeführt wird (also einmal für y=0 und einmal für y=1).

Listing 3: *transpose2()*: modifizierte transpose() Methode

```
98      public int[][] transpose2(int[][] matrix)
99      {
100         for (int y = 0; y < matrix.length; ++y) {
101             for (int x = 0; x < matrix[y].length-1; ++x) {
102                 int temp = matrix[x][y];
103                 matrix[x][y] = matrix[y][x];
104                 matrix[y][x] = temp;
105             }
106
107         }
108         return matrix;
109     }
```

Methode 5 (25%). Array sortieren

Listing 4: *sort2()*: Korrigierte sort() Methode

```
127 public int[] sort2(int[] array)
128     {
129         int i = array.length;
130         while (i > 1) {
131             int j = 1;
132             while (j < i) {
133                 if (array[j - 1] > array[j]) {
134                     int temp = array[j];
135                     array[j] = array[j - 1];
136                     array[j - 1] = temp;
```

```
137                     }
138                     j++;
139                 }
140                 i--;
141             }
142             return array;
143         }
```

Durch das Beheben von zwei groben Fehlern, lässt sich die Methode kompilieren. Sie hätte ohne weitere Korrekturen trotzdem nicht ihren eigentlichen Zweck erfüllt. Ein Compiler-Fehler entstand durch die Signatur der Methode (Zeile 43 - Die Zeilennummern beziehen sich in diesem Abschnitt auf die ursprüngliche, fehlerhafte Methode). Hier wurde als Übergabeparameter ein Integer namens “array“ deklariert statt einem Array aus Integern. Diese Variable wird jedoch später im code als Array behandelt. Das zweite Hauptproblem (Zeile 45) war, dass man versucht hatte den Operator “array.length“ wie einen Methodenaufruf auszuführen (welcher natürlich nicht existierte).
Die Klasse war jetzt kompilierbar, funktionierte dennoch nicht wie beabsichtigt. Bei der Testübergaben von Arrays wie z.B. {4,2,1,3}reagierte das Programm mit out of Bound exceptions. Der Rest des Programmabschnittes war mit scheinbaren Flüchtigkeitsfehlern durchsetzt. So startete der Index j mit der Zahl 0 (Zeile 47) wodurch direkt beim ersten Austauschverfahren ein Wert außerhalb des Arrays abgefragt wurde (-1), um dass zu beheben muss Der Wert von j bei jedem erneuten Durchlauf auf den Wert 1 gesetzt werden, das war auch die Ursache der Exception.
Nach weiterem durchlesen wurde einem klar, dass das Grundgerüst der Schleifen im grundlegenden in Ordnung war, jedoch sämtliche Parameter verwechselt wurden. Es wurde beispielsweise der falsche Wert unter dem Parameter temp gespeichert (Zeile 50), außerdem wurden die Indizes verwechselt, welche jeweils für die Verschiebung der While-Schleifen verantwortlich sind.
In der inneren While-Schleife wird der Array hierbei mit Hilfe des Index j durchlaufen (Zeile 54 (ersetzen von i++ durch j++)), bis j letztendlich größer-gleich i ist. Die äußere schleife definiert mit i den Bereich, welcher von j durchlaufen wird. Nach jedem Durchlauf von j verkürzt sich dieser Bereich um 1 (Zeile 56 (ersetze von j-- durch i--)). Ist i kleiner als 1 so wird der Array sortiert zurückgegeben.

Aufgabe 2 . . . ist der Studierenden Spiel (30%)

”Java lernen mit BlueJ”(D.J. Barnes,M.Kölling;4.Auflage;Pearson;2009)(Kapitel 7).

`Kopplung`: Die Kopplung definiert den Grad der Verknüpfung, welche Programmabschnitte untereinander besitzen. Angestrebt wird hierbei eine möglichst unabhängige Funktionsweise.
Bei der Kommunikation zwischen Programmen wird somit versucht möglichst minimale Schnittstellen zu erreichen, über welche die Programme miteinander kommunizieren, eine so genannte `lose Kopplung`.

Der Grad der Kopplung ist außerdem entscheidend, wenn es um mögliche Änderungen an einem Programm geht.
Bei einer losen Kopplung oder einer gekapselten Klasse, können Änderungen meistens ohne größeren Aufwand durchgeführt werden. Starke Kopplungen hingegen erschweren diese, da sich Änderungen schnell auf die Funktion anderer Klassen auswirken können.

`Kohäsion`: Kohäsion beschreibt den Grundgedanken, dass jede Klasse und jeder Programmabschnitt für eine fest definierte Aufgabe verantwortlich ist. Feste Definitionen von Aufgaben

ermöglichen einen besseren Überblick über die Funktion des Quelltextes, sowie eine präzise Implementierung.
Ein Gegenbeispiel hierfür aus dem Projekt wäre die Aufgabe der Paddle-Klassen. Hätte man hier die Funktion der Klasse "LeftPaddel" spezifiziert, dann wäre es möglich gewesen diese eine Klasse für die Modellierung beider Paddles auf den jeweiligen Spielfeldseiten zu verwenden.
Man hätte nicht nur eine Klasse einsparen können, sondern auch einen Haufen unnötiger Kopplungen zwischen den Klassen verhindert und somit nur eine Klasse für beide Seiten des Spielfeldes einsetzen können.
Ein weiterer Fakt, welcher hier zu nennen ist, ist die Verwendung der act() Methode in der Klasse Ball (Zeilen 66-93). Diese sollte man in kleinere Abschnitte aufteilen, da sie nicht nur eine spezifische Aufgabe übernimmt, sondern beinahe das gesamte Verhaltensspektrum der Klasse Ball. Es wäre in Bezug auf die Kohäsion also überlegenswert, diese Methode in ihre einzelnen Aufgabenbereiche zu unterteilen.

Listing 5: act() Methode der Klasse Ball

```
66 public void act()
67     {
68         if (!crushed) {
69             boolean collided = colliding;
70             colliding = getOneIntersectingObject( LeftPaddle.class ) != null
                ||
71                         getOneIntersectingObject (RightPaddle.class) != null;
72
73             if (colliding&&!collided) {
74                 xSpeed = -xSpeed;
75                 Greenfoot.playSound("tock.wav");
76             }
77
78
79
80             int width = getImage().getWidth();
81             int height = getImage().getHeight();
82
83             if (getX() <= width / 2 ||
84                     getX() >= getWorld().getWidth() - width / 2) {
85                 crushManager.doIt(this);
86             } else if (getY() <= height / 2 ||
87             getY() == getWorld().getHeight() - height / 2) {
88                 ySpeed = -ySpeed;
89             }
90
91             setLocation(getX() + xSpeed, getY() + ySpeed);
92         }
93     }
```

Das Problem wird deutlich, wenn man sich über die Kopplung und die eigentlichen Aufgaben der Paddles bewusst wird.
Die Aufgabe der jeweiligen Klassen ist in jenem Fall praktisch identisch, trotzdem werden hier zwei Klassen implementiert, welche voneinander in ihrer Funktion trotzdem abhängig sind. Es wäre hier klug gewesen eine Klasse zu spezifizieren, welche die gesamte Paddle-Funktionalität übernimmt.

`Code-Duplizierung`: Quelltext sollte grundlegend so verfasst werden, dass jeder Abschnitt nicht öfter als einmal implementiert werden muss. Tauchen Abschnitte von Quelltexten mehrfach auf, ist dass oft nicht nur ein Hinweis auf eine schlechte Implementierung oder zu wenig Überlegung

durch den Programmierer, es ist außerdem ein Ärgernis, wenn man später versucht diese Abschnitte zu modifizieren. Änderungen müssen in solchen Fällen an mehreren Stellen durchgeführt werden und kosten zusätzliche Zeit und Mühe. Änderungen, die an einer der entsprechenden Stellen vergessen werden, sind zumeist auch noch zusätzliche Fehlerquellen.
Bei der Original-Version des Pong-Spiels wird ein solcher Fall der Code-Duplizierung deutlich, wenn man sich die Weltklasse "Pong" ansieht (Zeilen 22-45):

Listing 6: Ausschnitt der Pong() Methode der Klasse Pong

```
22          // Der Ball bekommt ein zufälliges Bild
23          String[] images =
24          {
25              "apple1.png",
26              "apple2.png",
27              "bananas.png",
28              "bread.png",
29              "cherries.png",
30              "chips-1.png",
31              "chips-2.png",
32              "flan.png",
33              "grapes.png",
34              "hamburger.png",
35              "lemon.png",
36              "muffin.png",
37              "orange.png",
38              "pear.png",
39              "pizza_cheese.png",
40              "plum.png",
41              "pumpkin.png",
42              "stawberry.png",
43              "strawberry2.png"
44          };
45          ball.setImage(images[Greenfoot.getRandomNumber(images.length)]);
```

Ein identischer Abschnitt lässt sich in der alten Klasse "LeftPaddle"finden, welcher für das Erstellen eines neuen Balls verantwortlich ist. Es gibt keinen Unterschied zwischen beiden Quelltexten, sowohl in Bezug auf die Form als auch hinsichtlich ihres Verwendungszwecks.
Ein weiterer Fall ist die doppelte Implementierung der Paddle-Klassen.

`Entwurf nach Zuständigkeit`: Ein großer Makel des Pong-Spiels besteht außerdem hinsichtlich des Kriteriums "Entwurf nach Zuständigkeit". Entwurf nach Zuständigkeit heißt, dass jede Klasse eine Verantwortung für einen bestimmten Code oder Anwendungsbereich besitzt. Durch das Sortieren der Methoden nach ihrer Zuständigkeit werden die Zusammenhänge zwischen Methoden und Klassen deutlich. Es lassen sich so später auch schneller Codesegmente ausfindig machen, indem man sie in jenen Klassen sucht, welche die jeweilige Zuständigkeit besitzen. Die Ordnung der Klassen beeinflusst außerdem stark den Grad der gesamten Kopplung, wie dies hier im Spiel deutlich zu erkennen ist.
Fast alle Klassen enthalten Methoden für welche sie momentan nicht zuständig sind, so kümmert sich Die Klasse "LeftPaddle" um die Erstellung eines neuen Balls, so wie sich die Klasse "Ball" um die Bewegung der Paddles kümmert (Zeilen 95-129). Hierdurch entsteht eine unnötige Kopplung zwischen den Klassen, da diese jeweils untereinander auf die notwendigen Methoden zugreifen müssen.

Listing 7: Klasse Ball Zeile 95-129

```
95          /**
96       * Tastatursteuerung für das linke Paddle.
97       * Ein Druck auf die linke Pfeiltaste bewegt es nach oben, einer auf die
98       * rechte nach unten.
99       * @param paddle Das Paddle, das gesteuert wird.
100      */
101     void handleLeftPaddle(LeftPaddle paddle)
102     {
103         if ( Greenfoot.isKeyDown("q") ) {
104             paddle.setRotation(-90);
105             paddle.move(1);
106         } else if(Greenfoot.isKeyDown("a") ) {
107             paddle.setRotation(90);
108             paddle.move(1);
109         }
110     }
111
112     /**
113      * Tastatursteuerung für das rechte Paddle.
114      * Ein Druck auf die obere Pfeiltaste bewegt es nach oben, einer auf die
115      * untere nach unten.
116      * @param paddle Das Paddle, das gesteuert wird.
117      */
118     void handleRightPaddle(RightPaddle paddle)
119     {
120         if(Greenfoot.isKeyDown("up")) {
121             paddle.setRotation(-90);
122             paddle.move(1);
123
124
125         } else if ( Greenfoot.isKeyDown("down")) {
126             paddle.setRotation(90);
127             paddle.move(1);
128         }
129     }
```

Insgesamt ist Werners Abgabe ein perfektes Beispiel dafür wie man jene Kriterien nicht erfüllt. Sein Entwurf funktioniert zwar einwandfrei, jedoch reihen sich hier massenhaft Verstöße gegen jedes einzelne der oben genannten Kriterien aneinander.
Der eigentliche Quelltext des Projektes ist im Grunde nicht aufwendig oder schwer zu verstehen, trotzdem gerät man durch die unübersichtliche Aufstellung des Quelltextes schon beim Versuch der Analyse in Rage.
Wäre seine Aufgabe gewesen einen Verstoß gegen jene Kriterien zu entwerfen, so hätte er diese Aufgabe vorbildlich erfüllt. Hätte die Aufgabe jedoch darin bestanden die Kriterien zu erfüllen, so wäre es eine Scham gewesen einen derartigen Entwurf einzusenden.

Aufgabe 3 Bonusaufgabe: Macht es anders (5%)

Die originale Version des Pong-Spiel war ein wahres Grauen hinsichtlich des dazugehörigen Quelltextes.
Aufgabe war es nicht, die grundlegende Funktionalität des Programms zu verändern, sondern Refactoring zu betreiben. Es geht beim Refactoring primär um die Veränderung des Quelltextes zu einer möglichst leserlichen, übersichtlichen Form, so dass dieser später auch von anderen leicht verstanden und modifiziert werden kann.

Während des Vorhabens waren regelmäßige Tests notwendig um zu gewährleisten, dass das Programm seine ursprüngliche Funktionalität in vollem Umfang beibehält.

Zu Beginn der Arbeiten konnte man das Programm durchaus mit einem großen Scherbenhaufen vergleichen, von dem man zwar wusste, dass sich die Splitter zu ganzen Flaschen zusammenfügen lassen, trotzdem war es aufgrund der Formatierung des Projekts sehr mühsam die Teile passend zu sortieren.

Konzentrieren wir uns bei den ersten Umbauten zunächst hauptsächlich auf das Kriterium "Entwurf nach Zuständigkeit".
Wir verschieben die `doIt()` Methode der Klasse "Leftpaddle" in die Klasse "Ball", welche eigentlich für sie zuständig ist. Danach entfernen wir die Methode zum Setzen des Bildes für den Ball aus der Weltklasse "Pong" und verschieben dafür den identischen zweiten Abschnitt aus der doIt() Methode in den Konstruktor von "Ball".

Listing 8: Konstruktor von Ball

```
17 /**
18      * Konstruktor von Ball. Das Bild des Balles wird zufällig aus dem
19      * Array der verfügbaren Bilder gewählt.
20      */
21     public Ball()
22     {
23         // Der Ball bekommt ein zufälliges Bild
24         String[] images =
25         {
26             "apple1.png",
27             "apple2.png",
28             "bananas.png",
29             "bread.png",
30             "cherries.png",
31             "chips-1.png",
32             "chips-2.png",
33             "flan.png",
34             "grapes.png",
35             "hamburger.png",
36             "lemon.png",
37             "muffin.png",
38             "orange.png",
39             "pear.png",
40             "pizza_cheese.png",
41             "plum.png",
42             "pumpkin.png",
43             "stawberry.png",
44             "strawberry2.png"
45         };
46         setImage(images[Greenfoot.getRandomNumber(images.length)]);
47
48     }
```

Der Konstruktor ist für das Setzen eines zufälligen Bildes für den Ball verantwortlich. Als nächstes verschieben wir den Bewegungsmanager für die Paddles aus der Klasse "Ball" in die Klasse "LeftPaddle". Spätestens hier wird deutlich, dass die zweite Paddle-Klasse hinsichtlich des Kriteriums der Kohäsion überflüssig ist.
Somit können wir uns auf nur eine Paddle-Klasse beschränken, welche jetzt zur Simulation bei-

der Paddles verwendet werden kann.

Da diese Klasse nun beide Funktionen übernimmt, ist es notwendig ihre Handlungsfähigkeit für die Funktion als unabhängige Paddles auszulegen.
Das Bild der Klasse so wie die Steuerung müssen flexibel sein. Wir entschlossen uns also die Bilder und Steuerung der jeweiligen Instanz über Parameter festzulegen. Diese sollten der jeweiligen Instanz direkt nach ihrer Erstellung übergeben werden.

Listing 9: Ausschnitt der Weltenklasse Pong

```
17 // Erzeuge die Spielobjekte
18         // und setzt die Bilder und steuerung der Paddles
19         Paddle leftPaddle = new Paddle("turtle2.png","w","s");
20         Paddle rightPaddle = new Paddle("lobster.png","up","down");
21         Ball ball = new Ball();
```

Listing 10: Konstruktor Paddle

```
24 /**
25      * Der Konstruktor dreht das Paddle nach oben
26      * setzt das Bild des Paddels ein.
27      * @param image: Das Bild des Paddles. up: Taste für die Aufwärtsbewegung
         . down: Taste für die Abwärtsbewegung.
28      */
29     public Paddle(String image,String up,String down)
30     {
31         setRotation(-90);
32         setImage(image);
33         this.up  =up;
34         this.down=down;
35     }
```

Der Konstruktor von Paddle sollte hierbei alle für die Instanz notwendigen Parameter für das Paddle übernehmen und für die spätere Verwendung abspeichern.

Die jeweilige Steuerung der Instanzen wird jetzt mit einer modifizierten Form des "Keyboard-Manager" durchgeführt, in diesem Fall in Form der Methode `paddleControl`.

Listing 11: Methode *paddleControl()* der Klasse Paddle

```
47 /**
48      * Je nach Tastendruck werden die Paddles hoch oder runter bewegt
49      */
50     public void paddleControl()
51     {
52         if(Greenfoot.isKeyDown(this.up)) {
53             setRotation(-90);
54             move(1);
55
56
57         } else if ( Greenfoot.isKeyDown(this.down)) {
58             setRotation(90);
59             move(1);
60         }
61     }
```

Durch diese Modifikationen wurden nach und nach sämtliche Kopplungen unnötig, so sind jetzt

beispielsweise die Klassen "Paddle" und "Ball" komplett voneinander unabhängig. Die einzige noch vorhandene Schnittstelle besteht in der Übergabe der Bilder und Steuerung an die Klasse "Paddle" bei deren Erstellung durch die Weltenklasse "Pong". (Zeilen 19-20 Pong).

Der letzte Feinschliff entstand nun durch eine Kapselung der aller unabhängigen Methoden, so wie einer Zerteilung einiger Abschnitte, welche zuvor mehr als nur eine Aufgabe übernahmen. So zerteilen wir beispielsweise die Aufgaben der act() Methode in der Klasse "Ball", sodass jetzt beispielsweise der Test für die Spielfeldgrenzen in der Methode `testForBorder()` aufzufinden ist.

Listing 12: *testForBorder()* Methode in Ball

```
123 /**
124      * Prüft, ob der Ball die Bildgrenze Berührt und zerquetscht den Ball
          falls dem so ist.
125      */
126     private void testForBorder(){
127         int width = getImage().getWidth();  //Ballbreite
128         int height = getImage().getHeight();    //Ballhöhe
129         // stellt fest ob die Darstellungen des Balls die Bildgrenze berührt
130         if (getX() <= width / 2 ||
131         getX() >= getWorld().getWidth() - width / 2) {
132             doIt();
133         } else if (getY() <= height / 2 ||
134         getY() == getWorld().getHeight() - height / 2) {
135             ySpeed = -ySpeed;
136         }
137     }
```

Das Spiel verhält sich nun immernoch genauso wie es zu Beginn, jedoch ist der Quelltext jetzt wesentlich übersichtlicher und lesbarer.

Man kann das Refactoring folgendermaßen zusammenfassen:

- Entfernen einer Klasse (Right Paddle)
- Sortierung aller Methoden nach deren Klassen und Funktionen
- Entkopplung und Kapselung fast aller Klassen und ihrer Methoden
- Entfernen unnötiger Methoden, welche vorher jene Kopplung verursachten
- Aufteilung einiger multifunktionaler Methoden.