**Arkanoid**

**Prüfungsstudienarbeit zur Vorlesung „Java Programmierung“**Prüfer: Prof. Dr. Andreas Berl

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Daniel Hafeneder, 00743021,** [**daniel.hafeneder@stud.th-deg.de**](mailto:daniel.hafeneder@stud.th-deg.de)**, WI-B**

**Semester:** SS 2020

**Datum der Abgabe: 30.07.2020**

**Erreichte Punktezahl:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Note:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Erstkorrektur Unterschrift:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Zweitkorrektur Unterschrift:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inhalt

[Kurzbeschreibung des Spiels 2](#_Toc39237783)

[Implementierung des Spiels 4](#_Toc39237784)

[Statistiken 6](#_Toc39237785)

[Beschreibung einer besonderen Herausforderung bei der Umsetzung des Spiels 7](#_Toc39237786)

# Kurzbeschreibung des Spiels

**Wesentliche Features und ausführliche Erklärung des Spiels:**

Wie im Original steuert der Spieler einen Schläger auf dem Spielbildschirm. Dieser kann mit Hilfe der Tasten „Pfeiltaste Links“ und „Pfeiltaste Rechts“ sowohl nach links als auch nach rechts bewegt werden. Die Aufgabe dieses Schlägers ist es, einen Ball durch das Spielfeld zu schießen und den Ball vor dem Herunterfallen zu bewahren. Zu Beginn des Spiels, und auch wenn der Ball heruntergefallen ist, „klebt“ der Ball am Schläger fest. Um ihn erstmalig zu schießen, muss der Spieler die Taste „Leertaste“ drücken. Sobald der Ball fliegt, reicht es, ihn mit dem Schläger zu berühren, um ihn wieder nach oben zu schießen. Der Ball kann von dem Schläger in fünf verschiedene Richtungen geschossen werden, je nachdem, wo der Ball am Schläger auftrifft.

Der Ball kann vom Schläger nach

* Steil links, wenn links außen getroffen
* Links, wenn zwischen links außen und Mitte getroffen
* Mitte, wenn mittig getroffen
* Rechts, wenn zwischen rechts außen und Mitte getroffen
* Steil rechts, wenn rechts außen getroffen

geschossen werden.

Das Ziel des Spiels ist es nun, Blöcke, welche in jedem Level unterschiedlich angeordnet sind, abzuschießen. Diese Blöcke lenken den Ball bei Kollision ebenfalls um. Entscheidend ist hierbei, die X- und Y-Richtung in die sich der Ball beim Zeitpunkt der Kollision bewegt.

Sollte ein Block auf seiner oberen bzw. unteren Seite getroffen werden, wird die Bewegung in Y-Richtung umgekehrt, die X-Richtung bleibt allerdings bestehen. Sollte ein Block auf seiner linken bzw. rechten Seite getroffen werden, wird die X-Richtung umgekehrt und die Y-Richtung bleibt bestehen.

Ein weiteres Hindernis, welches den Ball ablenkt, ist der Rand des Spielfeldes. Wenn der Ball mit dem linken bzw. rechten Rand kollidiert, wird die Bewegung in X-Richtung des Balles umgekehrt, die Y-Richtung bleibt bestehen. Kollidiert der Ball mit dem oberen Rand, wird die Y-Richtung umgekehrt und die X-Richtung bleibt bestehen.

In manchen Blöcken verstecken sich Bonusobjekte, welche beim zerstören der Blöcke herunterfallen. Wenn man diese Bonusobjekte nun mit dem Schläger auffängt, wird der jeweilige Bonus aktiviert.

Folgende Boni wurden implementiert:

* Schläger vergrößern
* Schläger verkleinern
* Geschwindigkeit des Balles erhöhen
* Geschwindigkeit des Balles verringern

Es gibt die drei Geschwindigkeitsstufen **langsam**, **normal** und **schnell**. Die Boni die die Geschwindigkeit ändern, tun dies immer stufenweise. Angenommen man befindet sich in der **langsamen** Geschwindigkeit und sammelt ein **Geschwindigkeit erhöhen** Bonusobjekt auf. Die Geschwindigkeit wäre nun wieder **normal**. Sammelt man erneut **Geschwindigkeit erhöhen** auf, wäre die Geschwindigkeit nun **schnell**.

Wenn alle Blöcke in einem Level zerstört werden, gelangt der Spieler in das nächste Level, oder wenn alle Level durchgespielt werden an den Endbildschirm. Jeder getroffene Block erhöht den Score Counter. Sollte der Spieler den Ball mit dem Schläger nicht fangen und der Ball den Bildschirm nach unten verlässt, verliert er ein Leben. Wenn die Leben auf 0 sinken, wird der Endbildschirm angezeigt.

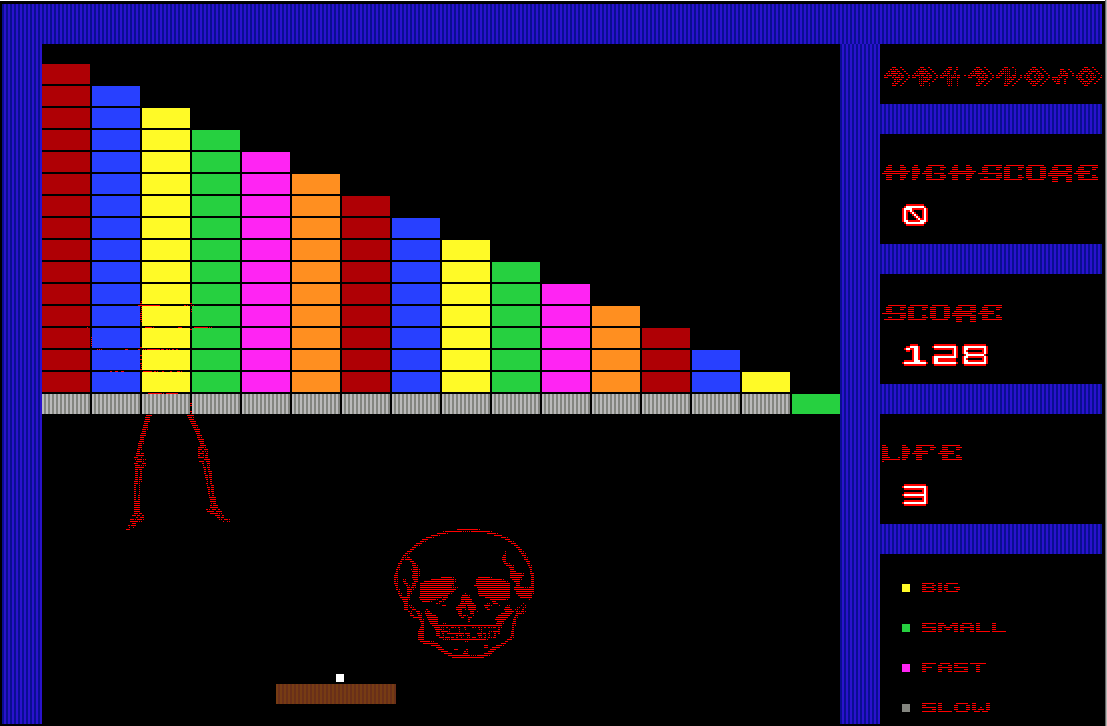
**Unterschiede zum Original:**

* Es wurden nur die ersten beiden Level implementiert (Diese sind allerdings aufgebaut wie im Original)
* Ich habe unterschiedliche und mehr Sounds eingebaut als im Original
* Mein Spiel ist von der Grafik/Darstellung nicht originalgetreu
* Es wurden nicht alle Boni des Originals implementiert
* Es wurden keine Aliens / Bewegliche Gegner implementiert

**Screenshots:**



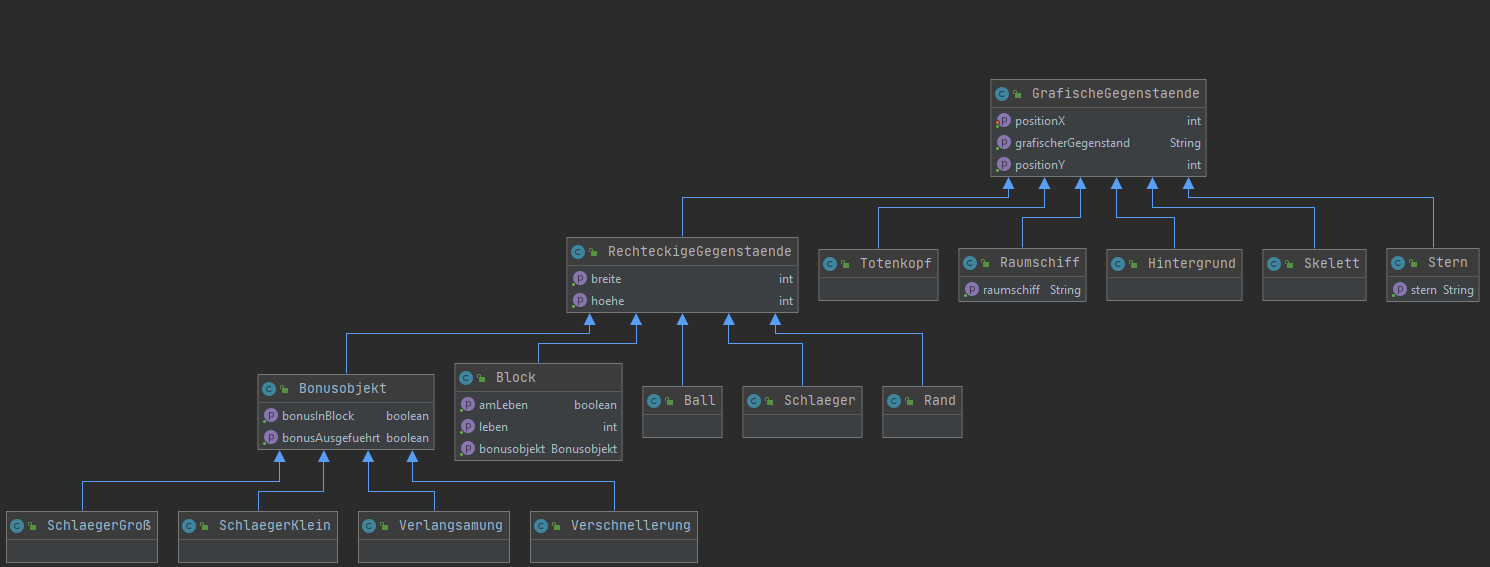
**Startbildschirm**



**Ingame, Level2**

# 

# Implementierung des Spiels



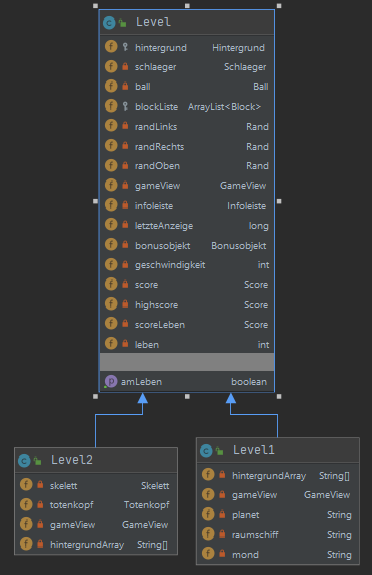
Sämtliche einzelne Gegenstände, die auf dem Canvas dargestellt werden sollen, haben als Vaterklasse die Klasse **GrafischeGegenstaende**. Diese vererbt die zur grafischen Darstellung notwendigen Attribute, wie die Position auf der X-Achse, die Position auf der Y-Achse und den String, der dargestellt werden soll an seine Child-Klassen **RechteckigeGegenstaende, Totenkopf, Raumschiff, Hintergrund, Skelett** und **Stern**. Die relevanten Methoden wie Getter und Setter, welche die Position und den String des Objekts zurückgeben werden ebenfalls vererbt.

Die Gegenstände **Bonusobjekt, Block, Ball, Schlaeger** und **Rand** sind in meinem Spiel rechteckige Gegenstände. Aus diesem Grund erben sie die Eigenschaften wie die Breite und Höhe, welche zur Erstellung des Strings und auch zur Abfrage der Hitbox notwendig sind von der Klasse **RechteckigeGegenstaende**. Zum Erstellen der Strings gibt es in der Klasse **RechteckigeGegenstaende** eine einzige Methode, die sämtliche Strings aller Child-Klassen zusammensetzt.

Da es in meinem Spiel unterschiedliche **Bonusobjekte** gibt, wurde hier ebenfalls eine Vaterklasse angelegt, welche an die Child-Klassen **SchlaegerGroß, SchlaegerKlein, Verlangsamung** und **Verschnellerung** Attribute und Methoden vererbt. Essenziell ist hier die Methode *bonusAusfuehren*, welche in jeder der Child-Klassen überschrieben wird und damit die gewünschte bonusspezifische Wirkung erreicht.

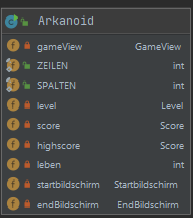
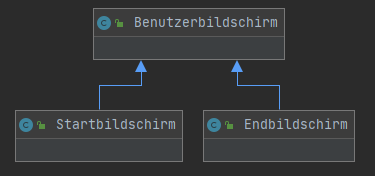
Natürlich wird der grafische Aufbau der Bonusobjekte wieder von den Vaterklassen **RechteckigeGegenstaende** und letztendlich **GrafischeGegenstaende** übernommen.

Diesen Vorgang möchte ich am Beispiel des Anlegens eines **Ball**s verdeutlichen. Beim Anlegen eines neuen **Ball**s werden an seinen Konstruktor eine X-Position, eine Y-Position, eine Höhe, eine Breite und ein String übergeben. Anschließend wird der Konstruktor seiner Vaterklasse **RechteckigeGegenstaende** aufgerufen, welcher über die Methode super() ebenfalls diese Übergabeparameter erhält. Die Höhe, Breite und der String werden nun den hierfür angelegten Attributen in dieser Klasse zugewiesen. Der Konstruktor führt nun die Methode aus, welche uns unseren rechteckigen String mit Hilfe der übergebenen Höhe, Breite und dem String zusammensetzt. Auch der Konstruktor der Klasse **RechteckigeGegenstaende** führt die Methode super() aus, welche die Attribute positionY und positionX an seine Vaterklasse **GrafischeGegenstaende** übergibt. Der Konstruktor dieser Klasse weist diese Werte nun zu.



Sämtliche Objekte die für den Ablauf eines **Level**s notwendig sind, habe ich in der Klasse **Level** angelegt. Relevant sind hier die meisten **GrafischeGegenstaende**, wie **Schlaeger, Ball, Rand** etc. Hier werden dann die Konstruktoren der jeweiligen Objekte mit ihren Übergabeparametern aufgerufen. In dieser Klasse werden die Levelinternen Spielabläufe und das Zusammenspiel der einzelnen Objekte untereinander geregelt. So werden zum Beispiel Kollisionen des **Ball**s mit dem **Rand**, dem **Schlaeger** und **Block**en ausgewertet und verarbeitet, indem unterschiedliche Sounds eingespielt werden oder auch das Leben des Spielers reduziert wird. Des Weiteren werden die **Bonusobjekt**e die mit dem **Schlaeger** kollidieren ausgewertet und verarbeitet.

Ich habe in dem Spiel zwei Level implementiert, **Level1** und **Level2**, welche beide von **Level** erben. Essenziell ist hier die jeweils überschriebene Methode *aufbauLevel()*, welche für die unterschiedliche Anordnung der Blöcke in den Leveln sorgt.

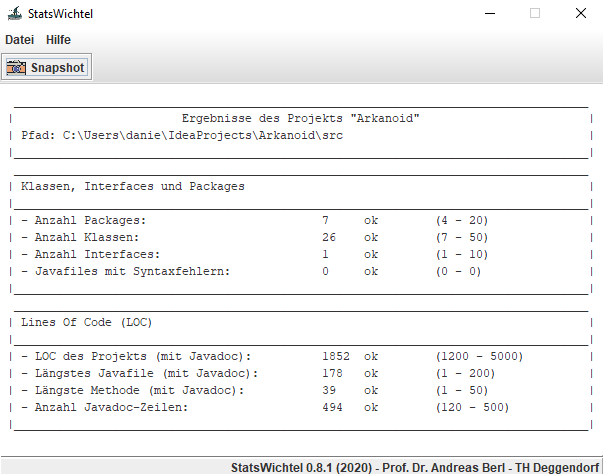


Die Klasse in der dann der komplette Spielablauf stattfindet, ist die Klasse **Arkanoid**. Hier werden Objekte vom Typ **Level, Score, Startbildschirm** und **Endbildschirm** angelegt. Über eine Dauerchleife werden nach und nach **Startbildschirm**, **Level1**, **Level2** und **Endbildschirm** angelegt und ausgegeben.

Da auf Grund der objektorientierten Prinzipien das **Level1/Level2** zugleich ein **Level** ist, reicht es, wenn man ein **Level** anlegt und diesem dann über den *new* Operator ein neues **Level1/Level2** zuweist.

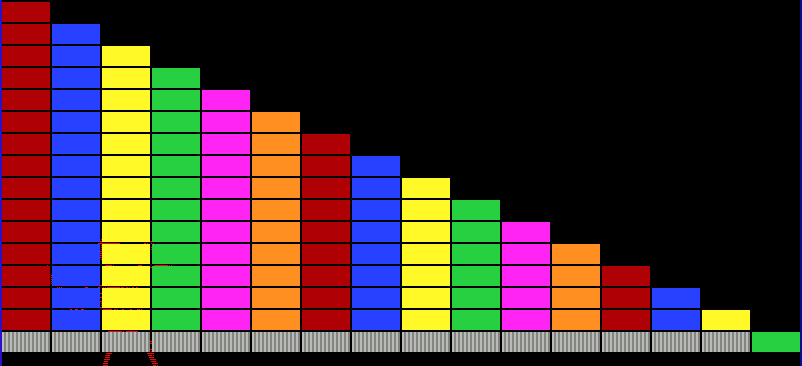
Der **EndBildschirm** und der **Startbildschirm** erben von der Klasse **Benutzerbildschirm**. Sie erben zum einen den String, welcher „Press Enter“ ausgibt und zum anderen wird die Methode *bildschirmStarten()* überschrieben, die dann die einzelnen Abläufe im jeweiligen Bildschirm regelt.

# Statistiken (ohne GameView.java)



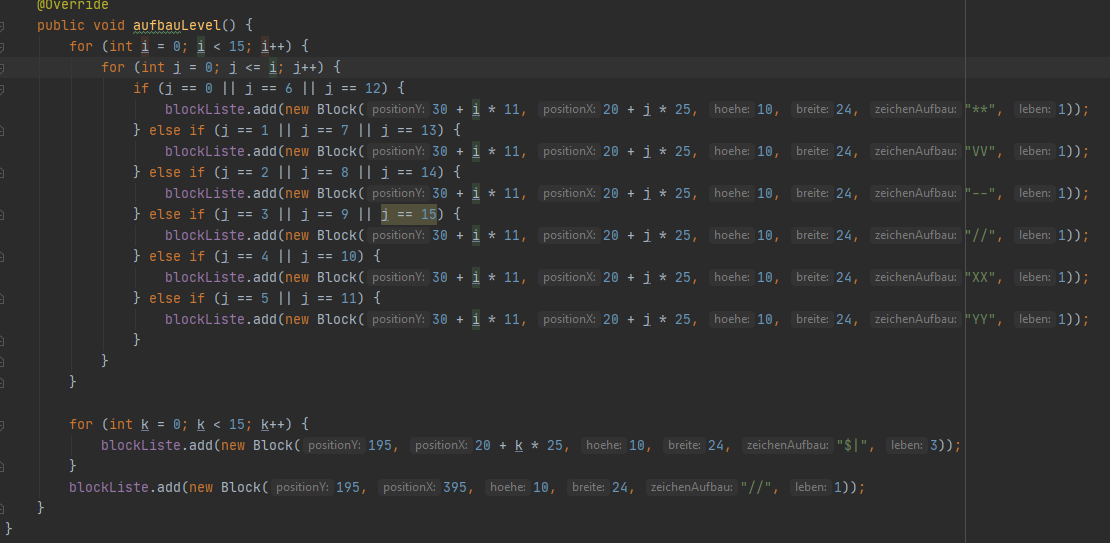
|  |  |
| --- | --- |
| **Umsetzung von Inhalten der Vorlesung** | **Wie/Wo/Warum eingesetzt?** |
| Selbst geschriebene Interfaces | **Wie:**  Es wurde ein neues Interface **TastenAuswertbar** in einem package interfaces angelegt. Die Methode *tastenAuswerten()* ist automatisch als abstract deklariert, was bedeutet, dass Klassen die dieses Interface implementieren diese Methode übernehmen müssen.  **Wo:**  Das Interface kommt bei den Klassen **Startbildschirm**, **Endbildschirm** und **Benutzerbildschirm** zum Einsatz.  **Warum**:  In jeder dieser Klasse müssen zwingend Tasten ausgewertet werden können. Mit @Override lässt sich die Methode *tastenAuswerten()* des Interfaces dann in der jeweiligen Klasse spezifizieren. In der Klasse **Startbildschirm** wird so zum Beispiel ausgewertet, ob die Enter-Taste gedrückt worden ist, um die Anzeige des Startbildschirms zu beenden. |
| Exceptions | **Wie:**  Es wurde eine neue Klasse **LevelException** angelegt. Die Methoden bei denen ein bestimmter Fehler auftreten kann, können nun diese Exception werfen. Beim Aufrufen dieser Methoden wird der mögliche Fehler über ein try/catch Konstrukt abgefangen.  **Wo:**  Die **LevelException** kommt in der Klasse **Arkanoid** bei der Methode *levelAnlegen()* zum Einsatz.  **Warum:**  Diese Exception dient dazu, andere Programmieren die eventuell an diesem Projekt weiterarbeiten darauf aufmerksam zu machen, dass beim Anlegen eines neuen Levels das übergebene Leben größer als 0 sein muss, da dieses sonst nicht startet. Wenn das Leben kleiner oder gleich 0 ist, wird eine Konsolennachricht ausgegeben. |
| toString() | **Wie:**  Die Methode *toString()* wird aus der Klasse **Object** überschrieben. Hier werden die Strings textArkanoid und textEnter mit einigen Leerzeilen zu einem String zusammengefasst und von der Methode zurückgegeben  **Wo:**  In der Klasse **Startbildschirm**  **Warum:**  Würde ich die *toString()* Methode nicht nutzen, müsste ich die *addToCanvas()* Methode der **GameView** für jeden String einzeln aufrufen. Hier reicht ein einzelner Aufruf. |
| equals() | **Wie:**  Die Methode *checkGewinnerScore()* vergleicht die Strings aus dem String Array mit den zuvor definierten Strings *zwei, neun* und  *vier*. Der Rückgabewert der *equals()* Methode ist ein boolscher Ausdruck  **Wo**:  In der Klasse **Score** in der Methode *checkGewinnerScore*  **Warum:**  Es wird überprüft ob das String Array aus den Strings *zwei, neun* und *vier* besteht, da dies dem maximalen Score entspricht, der erreicht werden kann. Ich benutze hier die *equals()* Methode, da man mit „==“ nur primitive Datentypen und Referenzen vergleichen kann. *Equals()* hingegen überprüft die Gleichheit von Objekten. |
| clone() | **Wie:**  Die Klasse die ein Objekt Clonen soll, muss das Interface Cloneable implementieren. Die Methode *clone()* erzeugt eine echte Kopie des Objektes. Die Methode wird aus dem Interface überschrieben und hat das geklonte Objekt als Rückgabewert.  **Wo:**  Die Methode wurde in der Klasse **Stern** deklariert und in der Klasse **Startbildschirm** aufgerufen.  **Warum:**  Auf dem Startbildschirm sollen drei Sterne zu sehen sein. Diese sollen exakt die Selben Eigenschaften haben, inklusiver einer gleichen Y-Position. Sie unterscheiden sich anhand ihrer X-Position. Anstatt nun drei Objekte anlegen zu müssen, bei denen immer jeweils die (teilweise gleichen) Übergabeparameter angegeben werden müssen, kann man das Objekt einfach klonen und die jeweiligen Attribute über Methoden abändern. |
| Nutzung von Comparator oder Comparable | **Wie:**  Die Klasse, welche die *compareTo()* Methode enthalten soll, muss das Interface **Clonable** implementieren. Anschließend kann die Methode aus dem Interface überschrieben werden. Hier sollen die Counter Werte von zwei Score Objekten verglichen werden und deren Differenz zurückgegeben werden.  **Wo:**  Die Methode wurde in der Klasse **Score** deklariert und in den Klassen **Arkanoid** und **Endbildschirm** aufgerufen.  **Warum:**  Ich habe für den Score, den der Spieler im laufenden Spiel erreicht, und den Highscore zwei Unterschiedliche Objekte. Wenn der Spieler „stirbt“ soll über *compareTo()* überprüft werden, ob der vom Spieler erreichte Score höher als der bisherige Highscore ist, damit der Highscore gegebenfalls angepasst werden kann. |
| StringBuilder | **Wie:**  Zuerst wird ein neues Objekt der Klasse **StringBuilder** angelegt. Diesem Objekt kann man anschließend über die Methode *append()* neue Strings anfügen. Über die Methode *toString()* werden alle angefügten Strings zu einem String zusammengeführt.  **Wo:**  In der Klasse **Endbildschirm** und der Klasse **Hintergrund**  **Warum:**  Der StringBuilder hat den Vorteil, dass ich meine Strings dynamisch zusammensetzen kann. In der Klasse **Endbildschirm** wird zum Beispiel immer der Text „Game Over“ ausgegeben, allerdings unterscheidet sich die Ausgabe, je nachdem ob ein neuer Highscore erzielt wurde oder nicht. Ein weiterer Vorteil ist die höhere Performance im Vergleich zu String. |

# Beschreibung einer besonderen Herausforderung bei der Umsetzung des Spiels



Eine Herausforderung war es, den Aufbau des 2. Levels (siehe Screenshot) möglichst effizient und knapp zu gestalten. Zuerst habe ich mir überlegt wie ich die große Anzahl an Blöcke überhaupt speichere. Einzelne Objekte kamen verwaltungstechnisch nicht in Frage. Eine Möglichkeit wäre ein Array gewesen. Dies wäre allerdings nicht dynamisch erweiterbar und das Arbeiten mit den Indexen schien mir auch unnötig kompliziert. Aus diesem Grund fiel meine Wahl auf eine Arraylist.

Nun musste ich mir überlegen, wie ich die Blöcke am besten in der Arraylist einfüge und vorallem, wie ich ihnen die richtige Position zuweisen kann. Im Folgenden meine entwickelte Lösung:



Ich habe das Problem mit Hilfe einer geschachtelten For-Schleife gelöst. Das Konstrukt wird von oben nach unten hin aufgebaut. Die äußere For-Schleife bestimmt die Zeile in der wir uns befinden und die innere For-Schleife baut die Blöcke dann spaltenweise auf. Die if-Abfragen sind dazu da, den Blöcken eine andere Farbe zu geben, denn die Farbe ist abhängig von der Position des Blocks. Die Farbe wird über die String Übergabe geregelt. Ein Beispiel für die erste Zeile:

1. Durchgang:
   * i = 0
   * j = 0

Da die Bedingung der inneren For-Schleife j <= i lautet, führt die innere Schleife einen Durchgang für j = 0 aus. Hier wird nur die erste if-Abfrage mit j=0 greifen und nur **einen** Block anlegen. Die Y-Position sowie die X-Position wird dynamisch mit Hilfe der Schleifenvariablen festgelegt.

Das Beispiel für die zweite Zeile:

1. Durchgang:
   * i = 1
   * j = 0

Hier wird nun ebenfalls ein Block in der ersten Spalte angelegt, allerdings eine Zeile **unterhalb** des Blocks aus dem ersten Durchgang, da i auf 1 gesetzt ist und damit die Y-Position beeinflusst.

Im nächsten Schritt wird j auf 1 erhöht, während i gleich bleibt. Dies bedeutet, dass ein Block in der gleichen Zeile, allerdings in der nächsten Spalte angelegt wird.

Lässt man diese Schleife nun komplett durchlaufen erhält man diese Treppenform.