Spieleprogrammierung mit Java

Tobia Ippolito

10. Juli 2020

Inhaltsverzeichnis

1 Spieleprogrammierung

Die Grundlage für Videospiele bildet die Spieleprogrammierung. Sie ist der logische Kern, bildet den Kontext und definiert den Rahmen eines jeden Videospiels.
Es ist eigentlich logisch, aber hier sei noch der Grund hierfür erwähnt: Videospiele basieren auf dem Computer und dieser nimmt nur 1 und 0 wahr. Über komplexe Elektrotechnik kann hieraus noch einiges herausgeholt werden. Durch das Programmieren können wir Menschen, dem Computer Befehle geben. Dabei versteht der Computer an sich nichts von dem was wir da in eine Entwicklungsumgebung oder Texteditor schreiben. Wie gesagt nimmt er nur 1 und 0 wahr (schon ein bisschen mehr -> Stichwort Digital Logik). Jedoch muss man hier das Prinzip von Computern verstehen. Der Computer besitzt mehrere Ebenen. Die erste und unterste Schicht, ist die eben erwähnte. Das ist also die Hardware. Danach gibt es weitere Ebenen, welche immer abstrakter werden. Irgendwann kommt man bei den problemorientierten Sprachen (Java,...) an. Das tolle ist, dass wir nichts über die unteren Ebenen Wissen müssen, denn sie bauen aufeinander auf und sind in sich logisch -> können wieder 'reabstrahiert' werden. Falls unser Code nun ausgeführt und von dem Computer verstanden werden soll, so wird der Code in eine untere Ebene umgewandelt, bis man bei der untersten Ebene angekommen ist (digitale Logik / Hardware).
Das geschieht unter anderem durch das Interpretieren oder das Compilieren eines Codes.
Da wir etwas abgeschweift sind hier die Zusammenfassung:
Dadurch, dass man über das Programmieren dem Computer Befehle geben kann, ist die Spieleprogrammierung die Schnittstelle zum Zielobjekt (dem Computer). Dabei sei erwähnt, dass schon programmierte Programme ebenfalls so eine Schnittstelle darstellen können (aber um Sonderszenarien will ich mich hierbei nicht kümmern).
Bei der Entwicklung eines Videospiels fällt ein breites Spektrum von Aufgaben in den Bereich der Spieleprogrammierung .
Das wichtigste Dabei ist das Programmieren der Engine. Diese stellt den Antrieb eines jeden Spiels dar und entscheidet über Performance und auch wie schwierig die Entwicklung abläuft. Mehr zu dem Thema 'Spieleengine' findet sich in dem Kapitel 'Engine'.
Bei der Spieleentwicklung lassen sich jedoch noch einige mehr Bereiche finden.
Schließlich braucht es jemanden, der das Gameplay programmiert und Tools für die Entwicklung oder den Betrieb des Spiels programmiert.
Und noch weitere Bereiche, welche variieren und hier nicht weiter ausgeführt werden (Combat-Programmer, ...).
Außerhalb der Spieleprogrammierung gibt es natürlich noch viele weitere Köpfe. Da es bei diesem Projekt um die Spieleprogrammierung geht hier nur kurz erwähnt (erwähnen sollte man sie trotzdem): Game Designer, Grafiker(UI Designer, ...) , Leveldesigner, Komponisten, Sounddesigner, Autoren, Sprecher, Übersetzter, Tester, Game Director und Produzenten.
Die Liste könnte man natürlich noch erweitern.

Bei der Spieleprogrammierung ist die Programmiersprachenwahl eine wichtige Entscheidung. Dabei sollte man auf jeden Fall dazu sagen, dass eine Programmiersprache nur so effizient ist, wie sie eingesetzt wird. Bedeutet, dass man lieber zu Java greifen sollte anstatt zu C++, wenn man mehr mit Java anfangen kann.
Je nachdem welche Engine man verwendet, ist die Programmiersprache schon festgelegt.
Man könnte noch ein paar Worte zu Performance oder Bibliotheken sagen, jedoch halte ich das Ausgesprochene für das Wichtigste und damit würde ich es auch belassen.
Damit hätten wir die Spieleprogrammierung schon ziemlich ordentlich umrissen. Natürlich kann man in jedem Thema noch eindeutig tiefer gehen, aber das war auch gar nicht Ziel dieser PDF.
In den weiteren Kapiteln geht es nochmals um einzelne Themen, wie die Art der Programmierung. Danach werden wir Java für die Spieleprogrammierung kennenlernen und was es dabei zu wissen gibt.
(Die Grundkenntnisse werden dabei nicht behandelt)

1.1 Engine

Hier werden wir uns die Spielengine genauer anschauen. Dabei kann man schon eine bestehende Engine verwenden oder eine selber schreiben. Hierbei soll es es eher darum gehen, was eine Spieleengine ist und aus was sie besteht.
Eine Engine ist der Motor/Kern eines Spiels. Sie ist abstrakt, beinhaltet noch keine genauen Szenarios. Sie stellt also das Grundgerüst dar, auf welches man später bauten kann. Es handelt sich um eine abgeschlossene Einheit, welche aus verschiedenen einzelnen Systemen besteht, welche zur Problemlösung beitragen (zeichnen von etwas,...).
Eine Engine besitzt mehrere Bestandteile. Natürlich ist das Zeichnen von Dingen auf den Bildschirm sehr wichtig. Deswegen besitzt eine Engine ein Render-System.
Die Komplexität kann hier sehr variieren, je nachdem welche Grafiken angezeigt werden sollen. Sind die Grafiken 3D oder nur 2D?
Außerdem ist die Frage, wie man der GPU Daten übermittelt. Die mit Abstand bekanntesten Programmierschnittstellen sind hier OpenGL und DirectX.
Es gibt aber auch viele Bibliotheken, welche diese Schnittstellen verwenden und abstrahieren. Der Umgang ist somit vereinfacht.
Diese API's übermitteln der GPU Daten, welche in der CPU berechnet wurden.
Zum Render-System könnte man noch viel hinzufügen (Shader, Pipeline, GPU-Datas,...), aber wir wollen uns hier nicht in Details aufhängen. Zudem geht es uns hier nur um einen guten Überblick.
Ein Physik-System ist ebenfalls sehr wichtig. Dies wird beim Updaten der Spielobjekte benötigt. Auch dieses System kann sehr kompliziert werden, bei einer 2D-Grafik hält es sich jedoch stark in Grenzen.
Im 3D Bereich wird es schon eindeutig komplexer.
Für das Physik-System und das Render-System wird ein Mathe-System hilfreich. Dieses sollte mit allen Operationen der linearen Algebra, sowie der Geometrie umgehen können.
Außerdem sollte es einen Spielobjekte-Manager geben. So kann man den Überblick der Spielobjekte gewährleisten.
Als weitere Komponente kann man das Sound-System nennen. Denn irgendwie muss ja auch Musik und Sounds abgespielt werden. Damit eine virtuelle Spielwelt realistisch wirkt, ist Sound sehr wichtig. Der Feedback ist viel klarer und Sound kann das Game Design deutlich verstärken (in einem düsterem Spiel gehört düstere Musik,...).
Zu guter Letzt sollte es ein Zustandsmanagement geben. Wenn man den Zustand des Spiels speichert und darauf koordiniert reagiert, wird einem viel Arbeit abgenommen. Das Spiel ist so viel ordentlicher und wenn man später auf der Engine sein Spiel programmiert kann man einfach auf den Zustand des Spiels zugreifen und darauf reagieren. Das sollte bei einer Engine nicht fehlen.
All diese Systeme sind Abstrahierungen und Modularisierungen und sollten also getrennt umgesetzt werden.
Trotzdem arbeiten diese Systeme teils zusammen (Sound-System und Physik-System zum Beispiel -> Kollision -> Sound). Umso größer das Projekt, desto modularisierter sollte die Engine sein.
Diese Modularisierung bringt nämlich eine gewisse Ordnung und sie vereinfacht sehr vieles.
Hinzufügend sollte man erwähnen, dass Engines sehr unterschiedlich sein können und die Komponenten (aus welche eine Engine besteht) variieren können. Man kann beispielsweise auch nur eine Physik-Engine programmieren. Der Rest muss dann anders implementiert werden (Framework, Render-Engine, selbst programmieren,...).

Wir haben nun die verschiedenen Systeme einer Engine kennengelernt. Auch hier gibt es an jeder Ecke noch Dinge hinzuzufügen (Grafische Oberfläche, Performance,...), aber für einen soliden Überblick reicht es uns.
Zumindest fast, denn sehr wichtig ist noch der Engine Loop oder besser: Game Loop.
Dieser Beschreibt den Lifecircle der Spiele-Software. Also den internen workflow.
Der Engine Loop sieht so aus, dass zuerst alle Spiel-Objekte geupdatet werden und hier sollte der Spielobjekte-Manager seinen auftritt haben. Danach greift das Physik-System ein und prüft auf Kollisionen und deren Antwort/Reaktion (in welchem Winkel prallt der Ball ab).
Nun kommt das Render-System zum Einsatz und zeichnet die geupdateten Spielobjekte in Absprache des Spielobjekte-Managers.
Zum Schluss muss noch gewartet werden, zumindest vielleicht. Die FPS müssen unbedingt Konstant sein, sonst verändert sich das Komplette Spiel (für den Benutzer). Deswegen wartet man solange, wie der Durch-

gang brauchen sollte -> was davon noch übrig ist (Bei 60 Durchläufen, sollte ein Durchgang 1sek/60 dauern -> falls zu schnell, muss erst noch gewartet werden, bis diese Zeit erfüllt ist).
Für den Engine Loop ist auch noch der Zustand des Spiels interessant. Durch Abfragen können so passende Methoden aufgerufen werden (bei Gameover muss etwas anderes gezeichnet und vielleicht sogar geupdatet werden). Hier ist der besagte Zustandsmanager von Wichtigkeit.
Viel mehr gibt es nun nicht mehr zu dem Engine Loop zu sagen. Falls der Begriff 'Engine' noch nicht klar geworden ist, so wird er es vielleicht im nächsten Kapitel.

1.2 Framework's

Bei der Spieleprogrammierung muss als Grundlage nicht unbedingt eine Spieleengine zum Einsatz kommen. Es gibt auch andere Wege diesen Motor zu realisieren. Bei großen Projekten empfiehlt sich jedoch eine Engine, da diese ordentlicher, abstrakter und modularisierter ist. Sie nimmt einem also viel Arbeit ab (auch wenn das Programmieren erst einmal mehr Arbeit nimmt). Bei kleineren Spielen wäre der Aufwand größer eine eigene Engine zu programmieren. Hier könnte ein Framework das Richtige sein.
Klären wir zu Beginn, was ein Framework denn ist. Ein Framework ist eine Zusammenstellung aus Bibliotheken um ein Problem zu lösen. Framework's optimieren dabei die Verwendung für die Problemlösung. Anders ausgedrückt abstrahieren und vereinfachen Framework's die Bibliotheken für den Zweck (hier: Spieleprogramierung). Um ein Beispiel zu nennen könnte es ein Framework für 2D Spiele geben und damit sich der Programmierer nicht mit allen Features von OpenGL umschlagen muss, vereinfacht/optimiert das Framework die Verwendung.
Ein Framework stellt somit noch keinen Kern dar, sondern hilft einem bei der Programmierung eines solchen Kerns.
Mithilfe eines Frameworks kann man also die Grundstruktur eines Spiels und dann das Spiel selbst programmieren. Natürlich ist es dann auch möglich eine Engine mit einem Framework zu programmieren.
Eine Engine und ein Framework sind also schwer zu vergleichen, da sie Grundlegend andere Ziele verfolgen. Eine Engine will die Grundlage sein und ein Framework eher ein Starterkit: will also bei dem Programmieren einer Grundlage behilflich sein.
Damit ist ein Framework viel flexibler, aber nimmt eben weniger Arbeit ab und ist weniger modularisiert (als eine Engine).
Um den Unterschied zu Veranschaulichen, hier noch eine Metapher.
Eine Engine ist eine abgeschlossene Einheit, wie der Motor. Wenn man ein Auto bauen will, so kann man den Motor einfach nehmen und darauf sein Auto bauen. Der Motor besteht dabei aus vielen einzelnen Komponenten. Ohne so eine Engine müsste man alles neu bauen (Motor, Karosserie).
Frameworks sind wiederum nützliche Werkzeuge und einzelne Bauteile, welche bei der Entwicklung helfen. Den Motor muss man jedoch selbst basteln.

Bei meinem Spiel habe ich ebenfalls ein Framework verwendet. Es ist das JFC (Java Foundation Classes) - Framework. Es übernimmt für mich viele Render-Details und nimmt mir so viel Arbeit ab.
Damit hätten wir die Framework's gut durchquert.
Zusätzlich wollte ich kurz auf einen hypothetischen weiteren Fall hinweisen. Wenn man ein Spiel programmieren möchte könnte man rein theoretisch ohne Bibliotheken arbeiten.
Dies wäre die nächst untere Ebene und damit habe auch ich keinerlei Erfahrung. Der Aufwand muss immens sein und hier müsste man sich fragen, warum man sich so einen riesen Aufwand machen sollte. An sich gibt es nur den Vorteil der Flexibilität (und die Motivation -> Durst nach Wissen). Aber die hier vorgestellten Methoden der Spieleprogrammierung sind schon Flexibel genug, deswegen ist dies wohl kaum ein Argument. So eine Ineffizienz kann also argumentativ nicht nachvollzogen werden.
Ich wollte nur darauf Hinweisen und sehe es als unrealistisch an.
Damit springen wir nun in den Programmiertyp.

1.3 Warum OOP?

Hier soll es um die Art der Programmierung gehen. Dabei ist relativ klar, dass die objektorientierte Programmierung für die Spieleentwicklung am schlausten ist. Jedenfalls fällt mir kein Argument für einen anderen Programmiertyp ein. Warum dann dieses Kapitel? Dieses Kapitel soll verdeutlichen, was die OOP ist und warum sie so sinnvoll ist.
Die OOP ist ein Konzept, bei welcher Code in bestimmte Bereiche modularisiert wird und nur dort Geltung findet. Eine Software besteht dann aus Objekten, welche man selbst definiert und so zu einfacheren Umsetzungen imstande ist.
Zu diesem Konzept gesellen sich noch weitere Konzepte, wie Schnittstellen, Vererbung oder Polymorphie. Diese erweitern die Funktionalität der OOP und sind ein fester Bestandteil dieser. Somit ist mehr möglich.
Doch was bringt uns so eine Modularisierung? Es vereinfacht einiges. Und umso größer ein Projekt ist, desto mehr lohnt sich dieses Konzept.
Man muss nicht den Überblick über jede Variable oder Funktionalität haben, da sich jedes Modul selbst regelt.
Um das zu veranschaulichen: Man will prüfen, ob zwei Objekte kollidieren. Ohne OOP muss man erst mal alle nötigen Variablen heraussuchen und eventuell eine Methode schreiben. Diese Methode ist jedoch nur für ein Szenario zu verwenden, da man hier nichts abstrahieren/modularisieren kann.
Und wie gesagt, wird das immer komplizierter, je größer das Projekt wird.

Mit der OOP kann man einfach eine Methode in dem Objekt schreiben. Diese Methode ist dann im besten Fall abstrakt. Was meine ich damit? Heißt, ich könnte diese Methode auch für andere Kollisionen verwenden und hier kommt die große Stärke der Modularisierung und Abstrahierung zum Vorschein.
Ich habe nicht nur einen ordentlicheren Code, sondern muss keine genauen Daten und Fakten wissen, da ich 'relativ' programmieren kann. Man muss beispielsweise nur die X und Y-Koordinate als Parameter bei Methodenaufruf hinzugeben.
In welchen Szenario das passiert kann mir ziemlich egal sein.
So programmiert man nicht nur ein Szenario, sondern bildet eine Grundlage, welche einfach verwendet werden kann.
Auch das ändern von Daten ist einfacher. Ich muss nichts bedenken, ich kann einfach die set-Methode aufrufen. Falls es etwas zu beachten gilt, so wird das in dem Objekt/Methode beachtet.
Tatsächlich könnte dies einen an den Start dieser PDF erinnern. Dort war die Sprache von Computer-Ebenen und Abstraktionen. Dort kann man auch grob sehen, was für Vorteile Abstraktionen haben. Und im Kern ist das dasselbe wie hier beschrieben.
Nun weiter im Text. Für die Spieleprogrammierung ist die OOP perfekt. Schließlich hantieren wir mit verschiedenen Objekten (Spieler, Feinde,...) und können so jedes Objekt modularisieren. So wird das Programmieren viel einfacher. Auch Schnittstellen oder Vererbungen können uns hier viel abnehmen. Falls man mehrere ähnliche Objekte hat., kann man sich viel durch Allgemeinere Klassen sparen. Beispielsweise erben alle Feinde-Objekte von der Klasse 'Feind'. Hier können schon alle Grundlagen definiert werden und bei der Erstellung neuer Feinde muss man nur noch Details ändern (Angriffsart, ...).
Schnittstellen können zum Beispiel fürs Zeichnen vorteilig sein. Alle Objekte, welche gezeichnet werden müssen, implementieren eine Schnittstelle namens 'Paintable' und somit weiß jeder, dass dieses Objekt gezeichnet werden kann (eine Methode hierfür besitzt). In einem großen Studio kann dies beispielsweise hilfreich sein und Arbeit ersparen. Einfach ausgedrückt: Faule Menschen lieben die OOP. Durch ihre Modularisierung vereinfacht sie einiges. Erstrecht für das Erschaffen komplexer Welten und Systeme ist dies ein muss.

Damit haben wir nun die OOP durchgekaut, auch wenn man noch weiteres hierzu sagen könnte.
Kurz wollte ich noch auf den Zugriff der Variablen eingehen. In der OOP ist es wichtig, dass die Variablen eines Objektes nicht frei zugänglich sind. Wir haben das gerade eben schon ein bisschen angesprochen, aber hier noch einmal explizit: Dies hat mit der Modularisierung zu tun. Der Sinn dahinter ist, dass man nicht alles Wissen muss und somit einfach Dinge tun kann, ohne etwas 'falsch' zu machen. In einem kleinen Projekt ist dies zwar eigentlich nicht ganz so wichtig, aber bei größeren Projekten ist es sehr wichtig. Vor allem, weil Fremde etwas Programmieren müssen und dies sollte gehen, ohne das sie sich auskennen müssen. Das wichtige Stichwort ist hier 'Effizienz'.
Man sollte sich den Kerngedanken der OOP in Erinnerung rufen und dies dann auch so umsetzten, vor allem, weil es ja nicht aufwendiger ist. Im schlimmsten Fall vereinfacht es die Arbeit.

Dabei gibt es aber auch Ausnahmen. Bei manchen Variablen lohnt sich der öffentliche Zugriff. Beispielsweise beim Zustand des Spiels oder der Spielfenstergröße. Aber trotzdem sollte man bei Unsicherheit lieber mehr modularisieren als anders herum.

2 Java

2.0.1 Einleitung

In diesem Bereich soll es nun um die Spielepogrammierung mit Java gehen. Dabei können wir nicht alles umfassen, jedoch so viel um ein kleines Spiel zu programmieren.
Was wir nicht umfassen werden, sind die verschiedenen Engines und Framework's, welche es für Java gibt. Auch den 3-Dimensionalen Bereich werden wir hier nicht behandeln, könnte aber ein netter Zusatz sein. Das wären einfach zu viele Themen gewesen und ich wollte hier nur eine Art zeigen, Spiele programmieren zu können und zwar mit Java.
Wir werden für das Programmieren unseres Spieles nur die Klassenbibliothek von Java verwenden. Genau aus diesem Grund schauen wir uns nun unser Werkzeug einmal an.
Die Klassenbibliothek von Java enthält über 4000 Klassen. Um diese übersichtlich zu gestalten, sind diese in verschiedenen Packages unterteilt. Hier werde ich versuchen auf die Wichtigsten einzugehen. Je nach Spiel kann es eine hilfreiche Klasse geben. Die hier vorgestellte Klassenbibliothek ist von Java 8.

Den Anfang macht '**java.lang**'. Dies ist das Standartpackage schlecht hin, weswegen es auch automatisch importiert wird. Es lässt sich hier unter anderem die Klasse 'Object' finden. Von dieser Klasse erben alle anderen Klassen in Java. Auch die Standartklasse 'String' und die Klasse 'Number' (von welcher Integer, Float, Double,... erben) sind in diesem Package enthalten.
Des weiteren sind hier auch die sehr wichtigen Klassen 'System' und 'Thread' enthalten. Die letzte Erwähnung macht die Klasse 'Math', von welcher ich die Methode random() oft in meinem Projekt gebrauchen konnte.

Das nächste Package ist '**java.util**'. Dieses Package bietet einem eine Menge toller Klassen. Darunter Listen (ArryList) und die 'Scanner'-Klasse. Aber auch Klassen für den Umgang mit ZIP-Dateien oder generieren von Zufallszahlen.

Ein ebenfalls sehr wichtiges Package ist '**java.io**'. Die beinhalteten Klassen drehen sich um den Umgang mit Input und mit Output. Sprich der Umgang mit Dateien. Beispielsweise sind die Klassen 'BufferedReader', 'BufferedWriter' und 'File' aus diesem Package.

Zu diesem Package gibt es diese Alternative: '**java.nio**'. Dieses soll an sich die gleiche Aufgabe wie 'java.io' übernehmen, mit dem Unterschied, dass die Klassen aus diesem Package den aktuellen Thread nicht blockieren. Bei Operationen aus dem 'java.io'-Package muss der Thread auf das Ergebnis warten und bei vielen I/O-Operationen könnte das zum Problem werden.

'**java.math**' ist das kleinste Package in der Klassenbibliothek von Java. Es will mit den Grenzen von Zahlen aufräumen und erweitert so die Möglichkeiten. Auch wenn dieses Package wohl nur in Spezialfällen Gebrauch finden wird. In anderen Worten soll man Zahlen erzeugen können, welche keine Grenze in ihrer Größe und in ihrer Genauigkeit haben ('BigInteger', 'BigDecimal').

Für Netzwerkfeatures sorgt das Package '**java.net**'. Mithilfe dieses Package lässt es sich einen Server und Clients programmieren.

Mit dem Package '**java.secruity**' kann man verschiedene kryptografische Operationen durchführen (Verschlüsslung, Signierung, Key-Generierung). Auch lässt sich hierdurch den Zugriff auf Ressourcen abstimmen.

Um eine Java-Anwendung mit einer Datenbank anzubinden gibt es das Package '**java.sql**'. Man kann so SQL-Anfragen stellen.

'**java.time**' bietet verschiedene Operationen mit Zeitwerten. Dies könnte auch in der Spieleprogrammierung interessant sein.

Kommen wir nun zu den letzten Packages. Sie befassen sich alle mit der grafischen Benutzeroberfläche.
'**java.awt**' (Abstract Window Toolkit) ist das erste und älteste Package zur grafischen Darstellung. Es beinhaltet viele grundlegende Operationen. '**javax.swing**' und '**javafx**' sind zwei weitere GUI-Framework's. Wobei 'javafx' das jüngste ist.
AWT und Swing sind Teil des JFC (Java Foundation Classes)-Framework. Das JFC-Framework umfasst alle Packages (außer 'javafx'), welche zur Entwicklung von grafischen Oberflächen verwendet werden.
Die JFC beinhalten zudem noch die Java2D API. Sie ist eine Erweiterung des AWT-Framework's. Und bietet zusätzliche Features. In meinem Projekt wurde darauf verzichtet.
Im nächsten Kapitel werde ich genauer auf das JFC-Framework eingehen. JavaFX wird nicht weiter behandelt. Um kurz noch ein Wort über JavaFX zu verlieren: JavaFX besitzt eine eigene Struktur und zeichnet sich durch seinen ordentlicheren GUI-Aufbau aus.
Damit hätten wir alle wichtigen Packages angesprochen. Packages die für mein Projekt sehr wichtig waren, werde ich nun noch genauer besprechen.

2.0.2 Grafische Oberfläche

Allgemein

Um ein Spiel zu programmieren braucht man eine grafische Benutzeroberfläche. Dabei werden von dieser drei Dinge benötigt:
1. Zeichnen von Grundelementen (Linien, Rechtecke,...) in verschiedenen Farben -> sowie das Zeichnen von Bildern.
2. GUI-Komponenten (Fenster, Button,...)
3. Abhören von Aktivitäten (KeyListener,...)

Zeichnen und Abhören können wir mit AWT. Swing liefert uns GUI-Komponenten auf welche gezeichnet wird, weswegen das Rendersystem von Swing von Bedeutung ist.

Rendern

Für das Zeichnen gibt es verschiedene Varianten. Ich stelle die Variante aus meinem Projekt vor. Hierbei benutzte ich den Workflow des Swing GUI-Toolkits. Bedeutet, dass ich auf die Swing-Komponenten zeichne und zwar mithilfe von AWT (Graphics). Auf Java2D wurde dabei verzichtet.
Wie funktioniert nun das Zeichnen der GUI-Komponenten? Swing bedient sich einer Malcallback-Methode und erweitert dafür die Mal-Variante von AWT.
Mit der Methode 'paintComponent()' übergibt man Swing die grafischen Informationen über die jeweilige Komponente. Sie ist eine Erweiterung der 'paint()' Methode, welche nicht verwendet werden sollte. Zudem sollte man die 'paintComponent()'-Methode nicht selbst aufrufen. Swing hat diesbezüglich ein eigenes Mal-System.
Stattdessen ruft man die 'repaint()'-Methode auf. Dadurch wird im AWT-Event-Thread (oder auch: 'event dispatching thread') durch den Repaint-Manager ein Repaint-Event eingereiht.
Wenn das Repaint-Event dann im AWT-Event-Thread ausgeführt wird, wird der Repaint-Manager aufgefordert, die 'paintImmediately()'-Methode auf der Komponente aufzurufen. Diese stellt zu aller erst die 'root' Komponente fest (das ist der Top-Level-Container). Dann wird geprüft, ob DoubleBuffering aktiviert ist. Wenn ja wird das Graphics-Objekt in ein passendes offscreen Graphics umgewandelt. Auf diesem offscreen Graphics wird nun, durch den Aufruf von 'paint()' von der Root-Komponente, alle Komponenten gezeichnet. Zur Erinnerung: Mit 'paintComponent()' kann man festlegen, was die Komponente zeichnen soll.
Nun werden noch die offscreen Graphics auf das ursprüngliche onscreen Graphics der Root-Komponente kopiert.
Die Methode update(Graphics g) wird im Swing-Worklfow also nicht aufgerufen und sollte also nicht überschrieben werden. Im AWT-Mal-Workflow ist das anders, aber um diesen soll es nicht gehen (er wird ja auch nicht verwendet).
Wir haben nun gelernt, wie das Zeichnen in Swing durch 'repaint' gehandelt wird und dass wir deswegen der Komponente nur sagen müssen, was sie zeichnen soll (und nicht wann genau). Mit der Methode 'repaint()' lassen wir dann alles zeichnen.
Hinzufügend haben wir gleich noch gesehen, dass Swing von Haus aus mit einem Doppelpuffer-System daher kommt und uns somit Arbeit erspart. DoubleBuffering ist übrigens automatisch aktiviert.

Mögliche Kritik an dieser Variante wäre, dass das Neuzeichnen aller Komponenten eh schon genug Arbeit ist und wenn man repaint() nun öfters aufruft, wird die Schlange im AWT-Event-Thread langsam voll. Es kann irgendwann zu Problemen beim Zeichnen oder beim Abhören von User-Eingaben kommen.
Eine Alternative wäre das Aktive Rendern. Das bedeutet, zeichnen unabhängig vom AWT-Event-Thread. Also unabhängig von dem Swing-Toolkit -> beziehungsweise nicht ganz, denn irgendwo müssen wir die Graphics herbekommen.
Ich habe diese Variante ausprobiert. Aber man will hier nicht nur den AWT-Event-Thread entlasten, sondern will auch ruckelfreie Grafiken. Hier müssen wir uns selbst ein Doppelpuffer-System zusammenstellen.
Dies habe ich mit 'BufferStrategy' und einem Canvas-Objekt gemacht. Erst wird alles auf die BufferStrategy gezeichnet und danach erst angezeigt.
Mein Resultat: In meinem Spiel kann ich keine Änderung vermerken. Es kann zwar sein, dass der AWT-Event-Thread entlastet wird, aber Auswirkungen scheint dies in der Praxis nicht zu haben.
Eine Mögliche Erklärung dafür, dass es keinen Unterschied gibt, werde ich hier versuchen zu erläutern. Es könnte gut sein, dass die Kritik von eben nicht berechtigt ist und Swing doch alles im Griff hat. Der Repaint-Manager könnte eine Überfüllung im AWT-Event-Thread stabil halten. Er fasst nämlich Repaint-Events zusammen. Dies geschieht bei sehr vielen und überfüllten Anfragen. Was bei meinem Projekt durchaus der Fall sein kann. Dies könnte die Erklärung sein, warum der Swing-Workflow genauso gut funktioniert wie das aktive Rendern.
Außerdem gibt es gute Gründe gegen das aktive Rendern (welche trotzdem weit verbreitet ist und deswegen auch zur Sprache kommt). Denn es könnte Swing in seinem Workflow stören. Wie wir gesehen haben, arbeitet Swing asynchron mit dem Zeichnen, um etwas performanter zu sein (so können Anträge zusammengefasst werden). Wenn wir jedoch nun synchron/direkt zeichnen könnte dies Schwierigkeiten geben.

Ich rate also eindeutig dazu, in dem normalen Swing-Workflow zu bleiben. Damit ist man auf der sicheren Seite. ich denke die Erwähnung des anderen Systems ist trotzdem sinnvoll gewesen.

Abhören mit AWT

Um ein Spiel zu programmieren, muss man mit dem Input des Users umgehen können. In Swing/AWT gibt es deswegen ein Event-System. Jede Eingabe erzeugt ein Event-Objekt und kann von dem jeweiligen Event-Listener im AWT-Event-Thread empfangen und verarbeitet werden.
So funktioniert das Event-System:
Falls das System ein Event empfängt, ruft es im AWT-Event-Thread automatisch die jeweilige Methode eines Listeners auf und führt diesen aus. Ein Listener muss deswegen eine bestimmte Schnittstelle implementieren (jenachdem mit welchen Events er umgehen soll). Zudem muss man den Listener dann noch einer Komponente hinzufügen. Am Schluss muss man den Listener von der Root-Komponente aus erreichen können.
Sehr wichtig sind die ActionEvents, MouseEvents und die WindowEvents. Dazu gibt es dann die jeweiligen Schnittstellen: ActionListener, MouseListener/MouseMotionListener und den WindowListener.
Man sollte aber darauf achten, den Code so klein wie möglich zu halten, denn sonst kann es zu Überfüllung im AWT-Event-Thread kommen. Schließlich wird der Code eines Listeners im AWT-Event-Thread ausgeführt und kann in dieser Zeit nichts anderes mehr wahrnehmen.

Swing und AWT

Swing und AWT sind so eine Sache. Zwar baut Swing auf AWT auf, aber ist trotzdem sehr verschieden.
Und es gibt einen guten Grund warum man nicht nur noch AWT benutzt. AWT Komponenten sind nämlich **heavyweight**. Das Bedeutet, dass AWT mit Peer-Klassen arbeitet und so eine Verbindung zum Betriebssystem herstellt. Ein Button ist dann ein Button in Windows oder Mac OS Form. AWT gibt also nur einen Auftrag an das Betriebssystem und hat dann nicht mehr viel mit der Komponente zu tun.
Doch hier entstehen Probleme, denn so ändern sich Eigenschaften des Programms (Textfelder bieten unterschiedliche Größen,...). Außerdem können diese Komponenten nicht Transparent sein und alle Komponenten müssen rechteckig sein. Zudem ist deswegen die Anzahl an unterschiedlichen Komponenten gering.
Aus diesen Gründen muss man Swing-Komponenten benutzen. Diese verwenden keine Peer-Klassen und sind somit unabhängig vom Betriebssystem. Und somit liegen die Komponenten in der Speicherverwaltung von Java. Die Swing-Komponenten werden deswegen **lightweight** genannt.

Swing-Komponenten

Wie schon beschrieben sind Swing-Komponenten *lightweight* und erben alle von 'JComponent'.
Diese Komponenten geben den grafischen Rahmen des Programms und auf ihnen wird auch mit 'Graphics' gezeichnet.
Die Swing-Komponenten kann man hierbei in Container-Klassen und in Komponenten-Klassen einteilen.
Mithilfe der Container-Klassen kann man andere Komponenten Aufnehmen und so seine Anwendung ordentlich strukturieren. Die wichtigsten sind hierbei das JFrame und das JPanel.
Um nun noch mehr Ordnung in seine Anwendung zu bekommen, hat eine Komponente ein Layout. Dieses kann gewechselt werden. Bei meinem Programm hat beispielsweise das 'CardLayout' gute Verwendung gefunden. Hiermit war ich in der Lage zwischen drei JPanel's zu tauschen. Damit immer nur eins aktiv ist.
Wie gesagt kann man die Swing-Komponenten in die eben erwähnte Container-Klassen und in die jetzt präsentierte Komponenten-Klassen unterteilen.
Hier sind die wichtigsten Klassen der JButton, die JCheckBox, der JRadioButton, JTextField, das JLabel und das JMenu.
Soviel zu den Swing-Komponenten, jetzt schauen wir uns noch an wie ein Programm mit Swing-Komponenten grob aufgebaut sein könnte.

Aufbau mit Swing-Komponenten

Hier will ich eine Beispielstruktur vorstellen und zwar die von meinem Projekt. Im Bereich 3 kann man hierzu auch noch weitere Details finden.
Die oberste Komponente (Top-Level-Container) ist das JFrame. Dieses besitzt als Inhalt ein JPanel. In einem CardLayout kann man dann weitere JPanel ordnen, zwischen welchen man dann tauschen kann.
Nun haben wir schon einen ordentlichen Aufbau mit verschiedenen Layern und die angezeigten Komponenten sind die im CardLayout hinzugefügten JPanels.
In meinem Fall füge ich diesen JPanels jeweils ein JPanel hinzu, welches Events abhört und von der 'repaint'-Konstruktion aufgerufen wird. Dies ist ordentlich und kann natürlich auch anders realisiert werden.
Damit konnten wir einen groben Aufbau der Swing-Komponenten sehen. Ich verweise nochmals auf mein Projekt in Bereich 3.

2.0.3 Wichtige Klassen

Wir konnten nun in Erfahrung bringen, wie man in Java grafische Komponenten erstellt, grafisch etwas zeichnen lassen kann und wie man mit User-Input umgehen kann.
Nun werden noch weitere interessante und wichtige Klassen in Java durchgegangen.

Math

Natürlich braucht man eine 'Math'-Klasse um verschiedene mathematische Probleme lösen zu können. In Java befindet sich diese Klasse in 'java.lang.Math', ist also schon standardmäßig implementiert. Nicht zu verwechseln mit java.Math, was auch hilfreich sein kann, aber nur wenig bietet und 'nur' spezielle Datentypen anbietet.
'java.lang.Math' bietet dagegen alles was das Herz begehrt. Von Cosinus, bis Wurzeln ziehen. Auch Umrechnungen von Gradmaß in Bogenmaß oder PI. Ich denke dazu gibt es an sich nicht mehr zu sagen.
Nur auf eine Methode möchte ich etwas genauer eingehen, denn sie hat in meinem Projekt mehrfach Verwendung gefunden und ich halte sie für sehr hilfreich. Die Rede ist von 'Math.random()'.
Sie generiert eine zufällige Zahl vom Datentyp Double zwischen dem Wert **0.0 und 1.0**. Klären wir zunächst wie einem das was nützt. Um ein Maximum festzulegen, multipliziert man die zufällige Zahl mit dem Maximum. Dadurch ist die größte Zahl: 1.0 * Maximum = Maximum. Dabei können nun auch alle Zahlen zwischen Maximum und 0.0 dran kommen. Dazu zwei Beispiele: 0.32876 * Maximum oder 0.852456 * Maximum. Es kommt also eine Zahl zwischen 0.0 und Maximum heraus. Hierzu sollte man die Zahl in einen Integer casten (meistens braucht man eine zufällige Ganzzahl): (int) (Math.Random() * Maximum);
Nun will man oft auch eine untere Grenze. Auch das ist hiermit möglich. Dazu kann man den entsprechenden Wert zum Schluss hinzu addieren/subtrahieren (subtrahieren geht auch, aber man muss mit denken).

Also (int) (Math.random() * Maximum) + Minimum. Ein Test zeigt uns die Richtigkeit: 0.0 + Minimum = Minimum.
Nun muss man das Minimum noch von dem Maximum abziehen, denn sonst ist das größtmögliche Ergebnis: 1.0*Maximum+Minimum. Also unser Endresultat sieht so aus:
(int) (Math.random() * (Maximum -Minimum)) + Minimum und dabei bleiben keine Wünsche offen. Das Einzige vielleicht verwirrende wären negative Werte.
Hier sieht es so aus: **(int) (Math.random() * (pGrenze - nGrenze)) + nGrenze**
Also die gleiche Gleichung wie davor, nur mit **positivster Grenze** (pGranze) und **negativster Grenze** (nGrenze) um die 'richtige' Zahl einzusetzen und einer Verwirrung zu entgehen. Falls man das schon oben gut nachvollziehen kann, hat man kein Problem.

Für die Praxis ist man zwar nun bereit, aber da gäbe es noch ein bisschen Theorie. Wie kommt diese zufällige Zahl zwischen 0.0 und 1.0 denn eigentlich zustande?
Hier muss man sagen, dass der Computer an sich nicht in der Lage ist eine wirklich zufällige Zahl zu generieren. Um die Definition zu klären: Ein Zufall ist eine **unvorhersehbare** und **unwillkürliche** Aktion. Uns genügt jedoch ein Pseudozufall, also eine Aktion bei welcher man keine kausalen Zusammenhänge erkennen kann (obwohl es die gibt). Je nachdem wie genau das gemacht wird, ist für uns kein wirklicher Unterschied zu erkennen.
Übrigens: nach meinem Wissensstand gibt ausschließlich Pseudozufälle, außer in der Quantenwelt.
Java besitzt mit java.util.Random einen Pseudozufallsgenerator, welcher willkürlich Zahlen generiert, welche trotzdem schwer (bis gar nicht) vorhersehbar sind.
Dieser Pseudozufallsgenerator verwendet hierfür einen **seed**, welcher übergeben werden muss. Dieser seed beeinflusst die Formel zur Berechnung der Pseudozahl. Hierdurch entstehen schon mal verschiedene Zahlen, aber das Muster bleibt gleich und würde irgendwann vorhersehbar werden. Damit das Muster der Generierung sich trotzdem noch ändert, wird der seed verändert. Hierzu wird die aktuelle Systemzeit genommen, da diese sich ständig ändert und man somit unterschiedliche Werte erhält.
Wenn wir Math.random() aufrufen, erzeugt diese Methode ein Objekt von 'java.util.Random'. Und wie dieser funktioniert haben wir eben gesehen.
Damit hätten wir das wichtige Thema der Zufälligkeit theoretisch und praktisch besprochen.

Threads

In der Spieleprogrammierung brauchen wir unbedingt mehrere Ausführungsstränge. Beispielsweise für die Benutzeroberfläche. Diese wartet nur darauf, bis der Benutzer etwas eingibt und kann dann darauf reagieren. Oder den schon bekannten AWT-Event-Thread, welcher sich 'nur' um die eingehenden Events kümmert. In meinem Spiel gibt es dann noch das Spiel, welches die ganze Zeit updatet und zeichnen lässt, bis das Spiel zu Ende ist. Da ist kein Platzt für weiteren Code.
Früher war das nochmal schwieriger (Stichwort: Quasi-Parallelität), aber heute besitzen Prozessoren mehrere Kerne und können damit mehrere Prozesse gleichzeitig durchführen. Das kommt uns nur allzu gelegen, denn wie wir gerade gesehen haben, führt kein Weg an mehreren Prozessen vorbei.
Hierfür gibt es in Java die Klasse 'Thread', welche in 'java.lang.Thread' zu finden ist. Der Lebenszyklus eines Threads ist wie folgt: Zuallererst wird ein Objekt von der Klasse 'Thread' erzeugt. Zudem wird ihm ein Objekt mit der Schnittstelle 'Runnable' übergeben (bedeutet, dass es eine run()-Methode gibt).
Danach wird der Thread mit 'start()' gestartet. Dadurch bildet sich ein neuer Ausführungsstrang und man gelangt zur 'run()'-Methode (isAlive() = true).
Sobald der Thread am Ende dieser Methode angelangt ist stirbt er. Der Ausführungsstrang schließt sich und kann nicht mehr mit 'start()' erzeugt werden. Man muss einen neuen Thread erzeugen.

Damit hätten wir auch schon das Wichtigste besprochen. Natürlich könnte man noch über Locks oder über den Executer berichten, jedoch reicht dieses Wissen schon völlig für die grundlegende Spieleprogrammierung aus.

Strings

Strings werden bei allen Projekten mehrfach gebraucht und ist ein Datentyp zum speichern von Text. An sich will ich hierzu nicht allzu viel sagen - da ich die Grundlagen nicht behandeln wollte- aber es gibt ein paar nützliche Methoden.
Ein String besteht aus einem Array aus dem primitiven Datentyp Char (Primitive Datentypen besitzen kein Objekt). Aber ein String ist ein Objekt und bietet viele nützliche Methoden, welche einem auch bei der Spieleprogrammierung behilflich sein können.
Mit 'length()' erhalten sie die Anzahl an Chars. 'charAt(int index)' gibt einen Char aus einem String aus.
Für die Spieleprogrammierung könnte noch 'toUpperCase' / 'toLowerCase' sehr interessant sein, da man bei einer Usereingabe nicht auf Groß- und Kleinschreibung achten muss und will.
Mit 'contains(String s)' kann man einen String auf seinen Inhalt überprüfen ('startsWith(String s)' und 'endsWith(String s)' könnten spezieller noch verwendet werden).
Die letzte wichtige Sache zu Strings ist das Vergleichen von Strings. Dies geht nicht über logische Vergleichsoperatoren (==), sondern über 'matches(String s)' oder 'equals(String s)'.

Enumerations

Sehr wichtig könnten auch Aufzählungen sein. Hierfür gibt es die spezielle Klasse 'java.lang.Enum'. Ein Enum beschreibt ein Konstrukt auf der Ebene von Klassen und Interfaces und alle Aufzählungsobjekte erben von dieser Klasse.
Mithilfe dieser Klasse kann man einfach eine Liste mit enumerierten Datentypen erstellen. Diese beinhalten einen Namen (String) und eine Position (Ordinalzahl).
In der Spieleprogrammierung kann man dadurch den Zustand des Spiels anzeigen. Gut daran ist, dass man die Zustände dann miteinander vergleichen kann.

ArrayLists

In der Spieleprogrammierung ist es oft sehr hilfreich eine Liste mit gleichartigen Objekten zu erstellen. Oder noch besser: Eine Liste mit Objekten, welche ein bestimmtes Interface implementiert haben.
Beispielsweise könnte es eine Liste für alle zu zeichnenden Objekten geben oder aber auch eine für alle Gegner.
Hier haben wir in Java zwei Möglichkeiten: Arrays oder Collections.
Arrays sind primitiver, aber auch performanter. Sie können beispielsweise primitive Datentypen aufnehmen. Collections müssen was das angeht einen Umweg gehen, aber dafür handelt es sich bei Collections um richtige Objekte.
Arrays sind tatsächlich nur einzelne Objekte oder Daten (int, float,...), jedoch sind diese in einem festgelegten Speicherbereich (aus diesem Grund kann man ihre Größe nicht mehr verändern).
Da ein Array alle seine Elemente in einem Block beinhaltet ist der Zugriff sehr schnell und effizient.
Collections besitzen dagegen nur Referenzen auf die beinhalteten Objekte und somit ist der Zugriff auch langsamer. Dafür besitzt eine Collection mehr Features und ist in seiner Größe nicht festgelegt.
Für die Spieleprogrammierung empfehle ich Collections, denn ich würde behaupten, dass man diesen Performance-Unterschied nicht merkt und lieber die Features mitnehmen sollte. Wie wir sehen werden gibt es auch Collections, welche Vorteile von Array und Collection ziehen.
Deswegen gehe ich hier noch genauer auf Collections ein.
Collections nehmen an sich jedes Objekt vom Typ Object auf, durch eine Notation mit <Typ> kann man die Liste in ihrem Inhalt jedoch begrenzen. Zu finden sind Collections hier: 'java.util.Collection'
Collection ist an sich aber nur ein Interface. Zwei Arten von Collections sind Lists und Sets. Ich werde aber nur auf Lists eingehen. Eine List ist ebenfalls nur ein Interface und die Klasse ArrayList verwendet diese.
Eine ArrayList ist einem Array ziemlich ähnlich, nur das es mehr Komfort bietet. So benutzt eine ArrayList ein Array um seine Daten zu speichern, jedoch bietet es noch hilfreiche Methoden an. So erzeugt eine ArrayList automatisch ein größeres Array, falls dieses ihr Maximum erreicht hat.
Außerdem kann man mit remove() ein Element entfernen werden und die anderen Elemente rutschen automatisch auf.
Mit set() kann man ein Element ersetzten.
ArrayLists nehmen sich also von dem Array und der Collection ihren Vorteil und sind so performant und komfortabel. Sie sind nicht wegzudenken und natürlich kann man auch eine ArrayList erstellen, welche ArrayLists beinhaltet.

Hierzu dieses Beispiel:
private **ArrayList< ArrayList< String > >** map = new ArrayList< ArrayList< String > >();

Innere Klassen

Klassen können nicht nur als eigene Datei/Klasse existieren. Sie können auch in/unter eine Klasse geschrieben werden. Diese Klassen nennt man innere Klassen. Dabei gibt es drei verschiedene Arten: Öffentliche statische innere Klassen, nichtstatische innere Klassen und anonyme innere Klassen.
Die öffentlich statische innere Klasse kann von außen aus verwendet werde: new äußereKlasse.innereKlasse();
Nach meiner Meinung bringt diese Variante am wenigsten und sollte eher als nichtinnere klasse umgesetzt werden.
Sinnvoller wird es bei der nichtstatischen inneren Klasse. Diese kann übrigens in Abhängigkeit ihrer gebundenen Klasse auch einzeln verwendet werden:

ÄußereKlasse a = new ÄußereKlasse();
InnereKlasse i = **a.new** InnereKlasse();

Die inneren Klassen haben Zugriff auf die Variablen/Methoden der angebunden Klasse. Falls man nun eine Klasse braucht, welche auf all diese Methoden zugreifen muss und sonst nirgends zu gebrauchen ist, macht diese Umsetzung schon Sinn. Sonst kann man diese Umsetzung nur mit Stil und Ordnung begründen.
Anders sieht es bei der anonymen inneren Klasse aus. Diese ist sehr hilfreich. Sie wird nicht wie eine Klasse definiert, sondern wird dort definiert, wo sie Verwendung findet.
Dadurch ist sie gänzlich an der umgebenden Klasse gebunden. So warum sollte das nützlich sein?
Naja damit kann man sich sehr viel Arbeit sparen. Man muss nicht extra eine Klasse schreiben. Am besten kann man das an einem Beispiel sehen:

```
btnOptions = (new Button(width/2 - 100, 350, 200, 50, "Options") {
    @Override
    public void clickAction() {
        musicManager.playCheck();
        showOptionMenu();
    }
});
```

Abbildung 2.1: anonyme innere Klasse

```
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        while(timer.isWorking()) {
            System.out.println("Waiting for Timer");
        }
        timer.setHowLong(time);
        timer.shouldWork();
    }
});
t.start();
```

Abbildung 2.2: anonyme innere Klasse 2

Übrigens werden innere Klasse in eigene Dateien gespeichert, mit dem Unterschied, dass sie den Namen der gebundenen Klasse + $ 1..2... tragen.
Man konnte recht gut sehen, dass innere Klassen Aufwand eingrenzen können. An sich würde ich trotzdem immer versuchen nichtinnere Klasse zu schreiben. Nur in Spezialfällen halte ich es für sinnvoll und vertretbar.

BufferedReader / BufferedWritter

In der Spieleprogrammierung ist das Einlesen und das Schreiben von/in Dateien sehr wichtig. Es sei nur schon an die Scores gedacht. Außerdem kann man so auch Spielstände auslesen und speichern.
Hierfür ist java.io sehr wichtig. Die erste wichtige Grundklasse ist 'File'. Sie kann zum finden, erzeugen, prüfen und löschen von Dateien verwendet werden.
Mit 'URL' und dem ClassLoader, kann man noch relativ zum Projekt navigieren, was wichtig für große Projekte ist, welche auch auf anderen Rechnern funktionieren sollen.
Um nun Textdateien lesen zu können braucht man einen Reader und zum schreiben einen Writer (für Binärdaten wäre das Input- und Outputstreams.).
Mithilfe eines FileReaders und einem File, kann man einer Datei ihrer Daten entnehmen. Dann muss man Char für Char lesen und in einer Datei laden.
Oder man macht es sich bequem und verwendet einen BufferedReader. Dieser benötigt ebenfalls eine File und den FileReader. Dann wird der Text aber automatisch und teilweise geladen/gepuffert. Nun kann man Zeile für Zeile durch gehen und erspart sich einiges. Sobald der gepufferte Text ausgeht, wird geschaut ob noch mehr Text da ist und dann wieder gepuffert.
Zwar ist diese Variante nicht ganz so performant, aber wegen so einer Kleinigkeit würde ich lieber den Komfort wählen. Aber das ist Ihnen überlassen.
Bei dieser Variante ist jedoch eines zu beachten. Zum Schluss muss (durch *finally*) noch BufferedReader geschlossen werden, falls er es denn noch nicht ist. Außerdem kann es zu Fehlern kommen, weswegen man einen try-Block um diese Operation machen muss. Leider muss man dann den BufferedReader davor definieren, denn finally und try besitzen unterschiedliche Zugriffsrechte (Scopes).
Ansonsten steht dem Auslesen nichts im Weg.
Nun noch zum Schreiben in Dateien. Hierzu kann ein BufferedWriter verwendet werden und an sich bleibt alles wie gerade besprochen.
Der BufferedWriter kann einfach Strings annehmen und wandelt/puffert diese in Chars um. Außerdem kann man einfach 'newLine()' verwenden. Sehr praktisch, so wie der BufferedReader.
Hierzu gibt es eigentlich nicht mehr zu sagen.

Exceptions

Wenn wir es gerade eben eh schon von Exceptions hatten, soll es hier kurz um diese gehen.
Fehler gibt es leider wie Sand am Meer und deswegen sollte man hier einiges Wissen.
Fehler werden Java wie Objekte behandelt. In einem try-Block kann so ein Fehler-Objekt geworfen werden. Wenn dieser try-Block einen catch-Block mit dem selben Fehler-Typ enthält, wird der Fehler gefangen und man kann auf ihn reagieren. Dies ist für das Trouble-Shooting wichtig (Fehler analysieren) und falls man nicht will, dass gleich alles abstürzt (beim Laden von Dateien). Denn wenn eine Exception weitergereicht wird, wird das Programm abstürzen. So bleibt es im catch-Block.
Natürlich kann man auch mehrere catch-Blocks einsetzten oder man kann auch mit logischem Oder arbeiten.
Zusätzlich kann noch ein finally-Block hinzugefügt werden. Dieser wird immer nach dem try-Block durchgeführt. Dies kann gut für Aufräumarbeiten verwendet werden, welche unbedingt durchgeführt werden müssen, egal ob es einen Fehler in dem try-Block gab oder nicht.
Verwendung findet sich beispielsweise beim Laden von Dateien.
Ein try-Block muss zwangsweise einen dieser vorgestellten Blöcke enthalten.

So nun noch zu den Fehlertypen. Es gibt drei Fehlergruppen. Die unchecked Exceptions, die checked Exceptions und die Errors (die Exceptions liegen in java.lang).
Die unchecked Exceptions sind die Fehler, welche von dem Programmierer verursacht wurden und eigentlich vermieden werden könnten. Sprich **IndexOutOfBoundsException**, **NullPointerException** oder auch **IllegalArgumentException** (hier nur die Wichtigsten).
Die checked Exceptions sind im Gegenzug die Fehler, auf welche sie keinen Einfluss haben. Die wichtigste Exception ist hierbei die **IOException**.
Errors sind Fehlerklassen, welche ein ernsthaftes Problem bei der Ausführung des Programms darstellen. Hier ist der wichtigste Error: **OutOfMemoryError**. Damit hätten wir die wichtigen Grundlagen der Exceptions durch.

Java Sound

Natürlich darf der Sound in einem Spiel nicht fehlen und damit haben wir dann alle grundlegende Features eines Games mit den zugehörigen Klassen abgedeckt.
Um Sound nun einlesen und abspielen zu können, gibt es die API 'Java Sound'.
Zu finden ist diese Programmierschnittstelle unter **javax.sound**. Dabei wird hier nur javax.sound.sampled benötigt. Diese Schnittstelle ist für das Abspielen und Aufnehmen von digitalem Sound wichtig.
Die Klasse 'AudioSystem' verarbeitet Sound-Dateien und kann ihr Daten entnehmen. Diese Daten übergibt sie 'Clip', welcher sich mit diesen Daten in einem neuen Thread öffnet. Ein Objekt von 'Clip' erhält man vom AudioSystem.
Ich empfehle noch den Pfad der Sound-Datei mit einem 'URL' relativ zum Projekt anzugeben.
Außerdem kann man mit **clip.setMicrosecondPosition(0);** den Sound von Beginn abspielen lassen. Mit **clip.start();** startet man den Sound/Clip.
Beachten sollte man auch, dass man den Clip nur einmal erzeugt und nur einmal die Sound-Datei hinzufügt. Sonst werden immer mehr Clip-Threads geöffnet und schließen sich nicht mehr.
Hilfreich ist auch das wiederholte Abspielen von Sound. Hierzu gibt es ebenfalls eine Methode und eine Konstante: **clip.loop(clip.LOOP_CONTINUOUSLY);** .

In meinem Projekt habe ich einen MusikManager geschrieben. Dieser beinhaltet einzelne Sound-Klassen. Die Sound-Klassen besitzen eine Methode zum Öffnen eines Clips und zum Laden der Sound-Datei, sowie zum einmaligen und wiederholten Abspielen des Sounds. Nun muss man nur noch ein Objekt von 'Musik-Manager' haben und kann über ihn alle Sounds abspielen. Zu Beginn werden alle Clips erstellt und geöffnet. Für ein besseres Verständnis empfehle ich das Klassendiagramm 3.1 im Bereich 3.

Damit haben wir das letzte Puzzleteil zusammen und können nun ein Spiel oder eine Engine in Java programmieren. In Bereich 3 kann man noch ein Beispielprojekt von mir sehen.
Natürlich muss ich darauf hinweisen, dass es noch sehr viel mehr Nützliches gibt und es auch sonst noch viele Alternativen gibt. Bei jedem Projekt lernt man dazu und das ist auch gut so.

3 Beispiel: "Falling Champion"

3.1 Einleitung

Mein Projekt „Falling Champion“ wurde mit der Programmiersprache Java und der Java Version 1.8 programmiert. Für die grafische Oberfläche habe ich die alte Standartbibliothek, javax.swing, benutzt.
In diesem Projekt wurde ein Videospiel programmiert. Bei diesem ist man ein fallender Kreis (Champion) und muss versuchen, so lange wie nur möglich zu überleben.
Um dies zu schaffen bewegt man sich per Maussteuerung. Und ich empfehle kein Touchpad zu benutzen, denn mit Maus ist eindeutig angenehmer.
Als Hindernis erscheinen zufällig Objekte (Steine) mit zufälliger Position und Größe.
Um den Spieler mehr unter Stress zu setzen, existiert noch ein Strahl. Dieser bewegt sich langsam nach unten und stört damit den Spieler und nimmt ihm Raum (er tut sehr weh).
Durch Items kann der Spieler den Abstand vergrößern. Und diese werden durch die linke Maustaste aktiviert (manche aktivieren sich auch automatisch).
Wenn der Spieler dann doch mal getroffen wird, ist es erst mal halb so wild. Der Spieler hat nämlich 3 Leben.
Vor dem Spielbeginn wird der Schwierigkeitsgrad gewählt. Diese Wahl finde ich super smooth. Man sieht dabei keinen Mauszeiger und ist automatisch auf einen Button. Finde ich persönlich sehr ansprechend. Danach versucht man solange durchzuhalten wie nur möglich. Im GameOver Bereich wird der Score automatisch gespeichert (wenn er denn hoch genug war) und kann dann durch einen Klick auf einen Button angeschaut werden. Fairer weise werden die Scores durch den Schwierigkeitsgrad getrennt. Den Score stellt übrigens die überstandene Zeit dar.
Von dem GameOver Menü kommt man wieder zum Start-Menü(darüber berichte ich noch).
Während des Spiels kann man per ’Space’ das Spiel freezen (das ist eigentlich nur eine Entwickler-Hilfe). Wenn man die rechte Maustaste drückt, so kommt man in das Pausen-Menü (auf das hässlichen JMenu wurde hierbei gänzlich verzichtet). Das Pausen-Menü ist schick umleuchtet und bietet verschiedene Funktionen. Man kann das Spiel fortsetzen (auch wenn man nochmals auf die rechte Maustaste drückt), zum Start-Menü zurück kehren, das Spiel schließen oder auch die Optionen öffnen. Hier kann man eine erweiterte Ansicht aktivieren. Dabei sieht man mehr Details zu den Kollisionen.
Das war es fast zum Spiel. Man sollte nur noch die anderen Items erwähnen. Es gibt einen Schutzschild (selbst erklärend), zusätzliche Leben und zusätzliche Zeit (einen höheren Score). Von jedem Item gibt es vier verschiedene Stufen, welche über eine bestimmte Wahrscheinlichkeit ausgemacht wird.
Welches Item generiert wird wird ebenfalls über Wahrscheinlichkeiten entschieden. Dabei wurde java.lang.math (random) verwendet. Diese Methode wurde aber schon besprochen.
Damit hätten wir den Kern meiner Software durchgesprochen (das Spiel an sich). Wenn man mein Spiel startet wird man jedoch feststellen, dass mein Programm noch einen schicken Rahmen bietet und dieser wird nun besprochen.
Zu Beginn wird man von dem Studio-Intro begrüßt (’Timerift Studio’ ist keine öffentlich eingetragene Marke/Firma - mein Anwalt hat mir empfohlen dies anbei zu erwähnen). Danach wird man aufgefordert einen Button der Maus zu drücken. Die Musik ändert sich hierbei und der Hintergrund auch. Wenn man einen Mausbutton gedrückt hat, so gelangt man in das Start-Menü. Dieses besteht aus drei Button (Start - Credits - Exit).
Was auf dem Start-Button passiert haben wir anfänglich hinreichend erläutert und der Exit-Button erklärt sich von selbst. Somit kommen wir zum letzten Part meines Programms: die Credits.
Hierbei nimmt man lustigerweise die Gegenposition ein und muss versuchen Spieler mithilfe von Steinen zu treffen. Der Spieler sieht dabei nur einen roten Strich, welcher von oben nach unten eine Linie bildet. Wenn man nun die linke Maustaste drückt, so entsteht hier ein Stein und dieser fliegt nach oben.
Von oben kommen Champions (Kreise) und diese können von den Steinen getroffen werden. Also wie bereits gesagt, nimmt man hier die Gegenposition ein.
Aber es handelt sich hierbei immer noch um die Credits, nur das meine etwas kreativer ausfallen (die meisten Credits sind wirklich enttäuschend). Wenn man einen Champion/Spieler mit einem Stein getroffen hat, so ploppt aus ihm ein zufälliger Titel und wer diesen in diesem Projekt trägt.
Super ist daran auch, dass diese sich aus dem Weg gehen und auch Abstand zu den Wänden halten. Dies sieht manchmal ziemlich nice aus.

Der Spieler hat dabei übrigens immer nur maximal drei aktive Steine. Und per rechten Mausklick kommt man wieder ins Start-Menü.
Außerdem ist alles mit Sound untermauert und der Feedback ist so eindeutiger.
Damit haben wir mein Programm komplett durchgearbeitet.

Aus Gründen der Übersicht wurden die Klassen in verschiedenen Packages untergebracht. Jedes Package steht für einen 'Modus' (StartMenü-Spiel-Credits).
Zusätzlich sollte erwähnt werden, dass die Klasse 'GameControl' im StartMenü-Package untergebracht wurde, aber eigentlich nicht dort rein gehört. Aber dazu im nächsten Kapitel mehr.

3.2 Klassenübersicht und Struktur

3.2.1 Schriftliche Beschreibung

Die Kern-Klasse bildet die Klasse 'GameControl'. Sie erbt von JPanel und wird beim starten dem JFrame als Inhalt hinzugefügt. An sich dient dieses JPanel nur als Verbindung der einzelnen JPanel. Bedeutet, dass es ein CardLayout besitzt und zwischen verschiedenen anderen JPanels tauscht. Also eine rein organisatorische Klasse, welche damit ein tragendes JPanel ist.
Die Klasse beinhaltet drei JPanel, von welchen immer nur eins gleichzeitig aktiv sein kann. Schauen wir uns hierbei zu aller erst das Spiel an.
Hier ist die Klasse 'Game' die Kern-Klasse. Gezeichnet und Aktivitäten belauscht aber ein hinzugefügtes Objekt von der Klasse 'GameCanvas'. Falls GameControl zu dem Game-Objekt schaltet startet dieses einen neuen Thread und lässt dort den Gameloop abspielen.
Dieser Gameloop hat eine festgelegte Zeit (Frequenz) und ruft je nach Zustand eine Methode zum updaten der Objekte und die repaint()-Methode auf.
Gezeichnet wird nur das hinzugefügte JPanel ('GameCanvas'). Dieses ruft aber Methoden von der 'Game'-Klasse auf. Hier sind schließlich alle Spiel-Objekte.
Die verschiedenen Zustände werden durch Enumerations dargestellt.
Für das Spiel sind sonst noch die Methoden: Player, TimeBeam, Background, Stone und die verschiedenen Items von großer Bedeutung. Sie sind schließlich die Spielobjekte und werden meist in ArrayList's zusammengefasst. Sie werden in der Klasse 'Game' erstellt und verwaltet. Die Spiel-Objekte implementieren die Schnittstelle 'GameObject' und müssen damit eine Methode zum Zeichnen und eine zum Updaten des Objektes besitzen.
Kommen wir zu den Credits. Hier ist die Klasse, die alles steuert, die 'Credits'-Klasse. Sie erstellt Objekte zu den einzelnen anderen Klasse. Unter anderem ist dabei auch die Klasse 'Listener' dabei. Diese erbt von JPanel und ist für das Zeichnen und das Abhören der Aktionen (MouseEvent,...) zuständig. Falls das Toolkit beschließt zu zeichnen, so ruft diese Klasse Methoden der 'Credits'-Klasse auf. Falls ein Event empfangen wird, so wird dieses ebenfalls an die Klasse 'Credits' weitergeleitet.
Wenn das Cardlayout in 'GameControl' zu 'Credits' wechselt, wird zusätzlich die Methode 'creditsStart' aufgerufen. Diese startet einen Loop in einem neuen Thread. Damit man noch mit der UI interagieren kann.
Weitere Klassen sind 'Player', 'Stone', 'Text'. Die Objekte der 'Player'- Klasse kommen von oben herunter und können durch ein Objekt von der Klasse 'Stone' zerstört werden. Ein Objekt der Klasse 'Stone' wird durch einen Linksklick erzeugt. Falls ein Objekt der 'Player'-Klasse getroffen wird, so wird in dem 'Credits'-Objekt ein 'Text'-Objekt erzeugt. Dieses erhält intern einen zufälligen Text und passt sich seiner Umgebung in seinem Update an.
Damit kommen wir auch schon zu dem Start-Menü. Hier ist die Kern-Klasse 'StartScreen'. Auch diese Klasse besitzt sein eigenes JPanel, welches zum Belauschen und Zeichnen Verwendung findet. Diese Klasse heißt 'StartListener'. Die Klasse besitzt zudem noch JButtons um das Game oder die Credits zu starten.
Ansonsten sind hier nur noch zwei Klassen relevant: 'Titel' und 'Titel2'. Beim Start des Spiel kommt nämlich erst ein Intro und dafür werden jeweils ein Objekt der beiden Klassen benötigt.

Damit hätten wir einen kleinen Rundgang durch die Struktur und die Klassen dieses Spieles gemacht. Es fehlen noch ein paar kleine und unnötige Klassen. Die Klassen 'MusicLoader' und 'MusicManager' sind jedoch noch sehr wichtig. Das Objekt von der Klasse 'MusicManager' lädt alle Sounds per 'MusicLoader' (jeder Soundtrack ist ein Objekt von 'MusicLoader) und kann dann mit einem entsprechenden Methodenaufruf den richtigen Soundtrack abspielen.

3.2.2 Klassendiagramm

Um eine bessere Übersicht zu erlangen werden hier Klassendiagramme gezeigt. Ich nenne nicht alle Variablen und nicht alle Methoden und versuche nur die aller Wichtigsten zu nennen, da dies sehr viele sind. Außerdem unterteile ich mein Klassendiagramm in vier Stücke.
Ein Klassendiagramm um Übersicht zu schaffen. Bedeutet die Kern-Klasse und die drei verschiedenen JPanel-Klassen. Danach gehe ich auf jedes JPanel und seine wichtigen Klassen ein (Spiel-Credits-StartMenü).

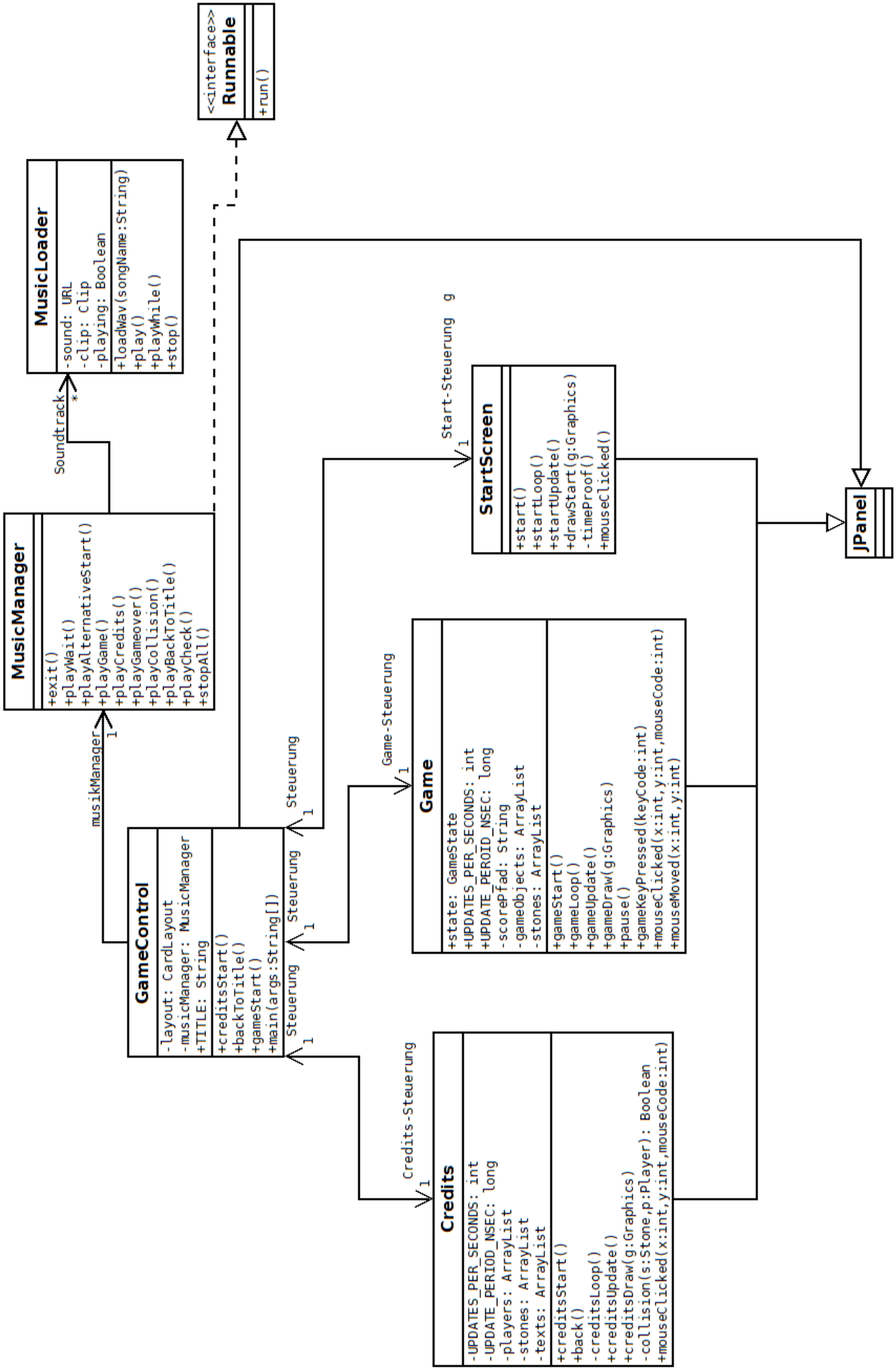

Abbildung 3.1: Übersicht - Klassendiagramm

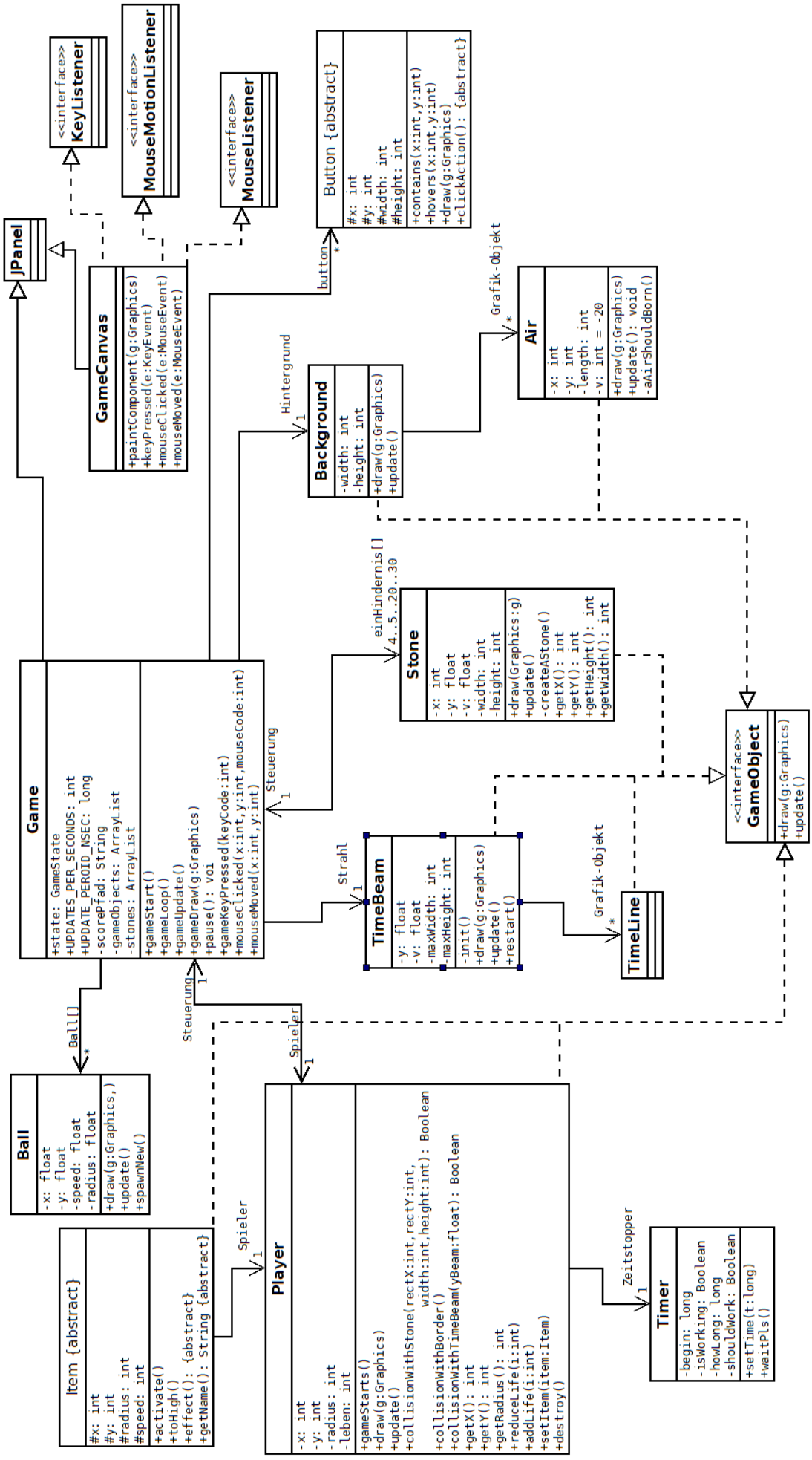

Abbildung 3.2: Game - Klassendiagramm

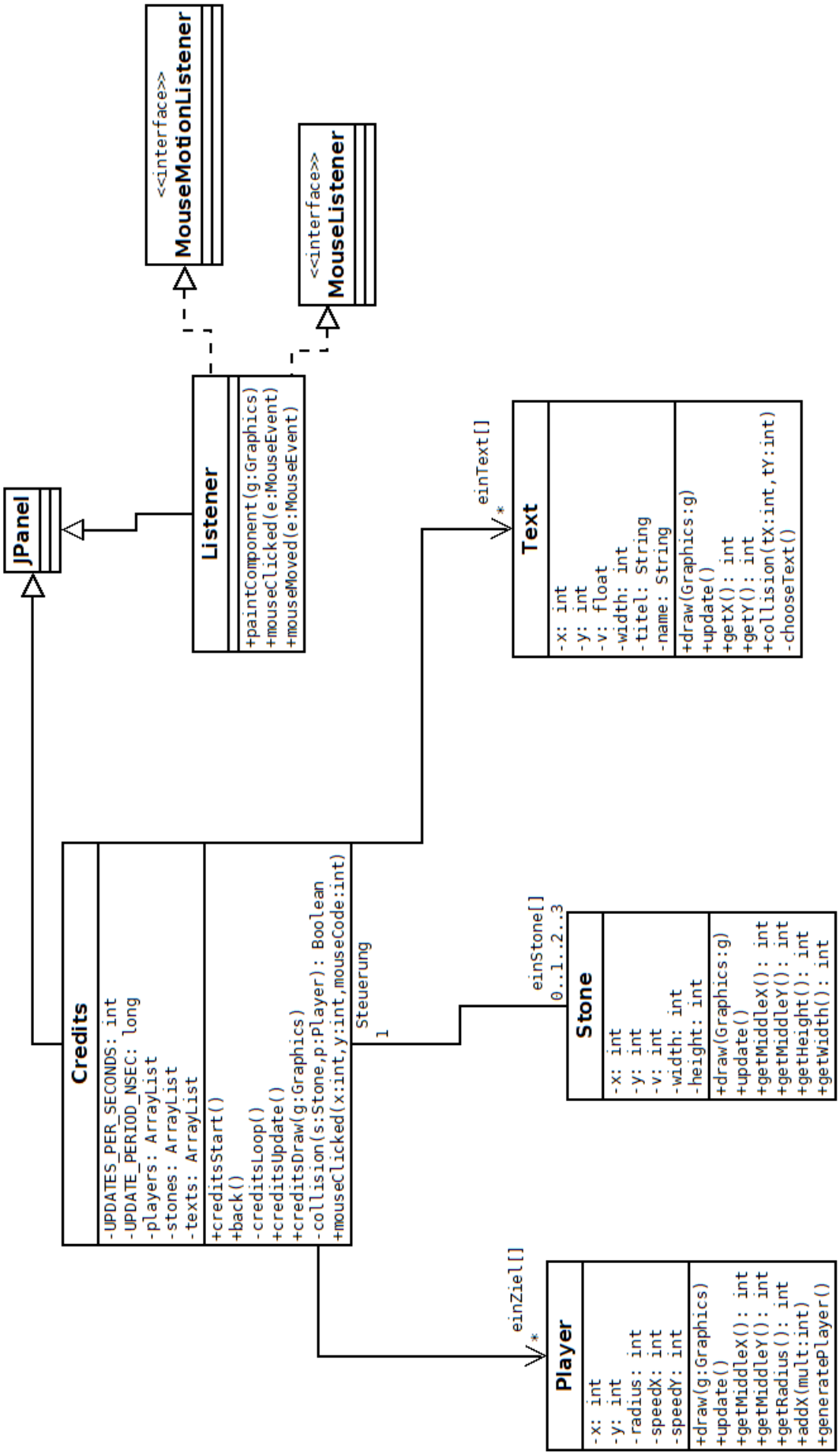

Abbildung 3.3: Credits - Klassendiagramm

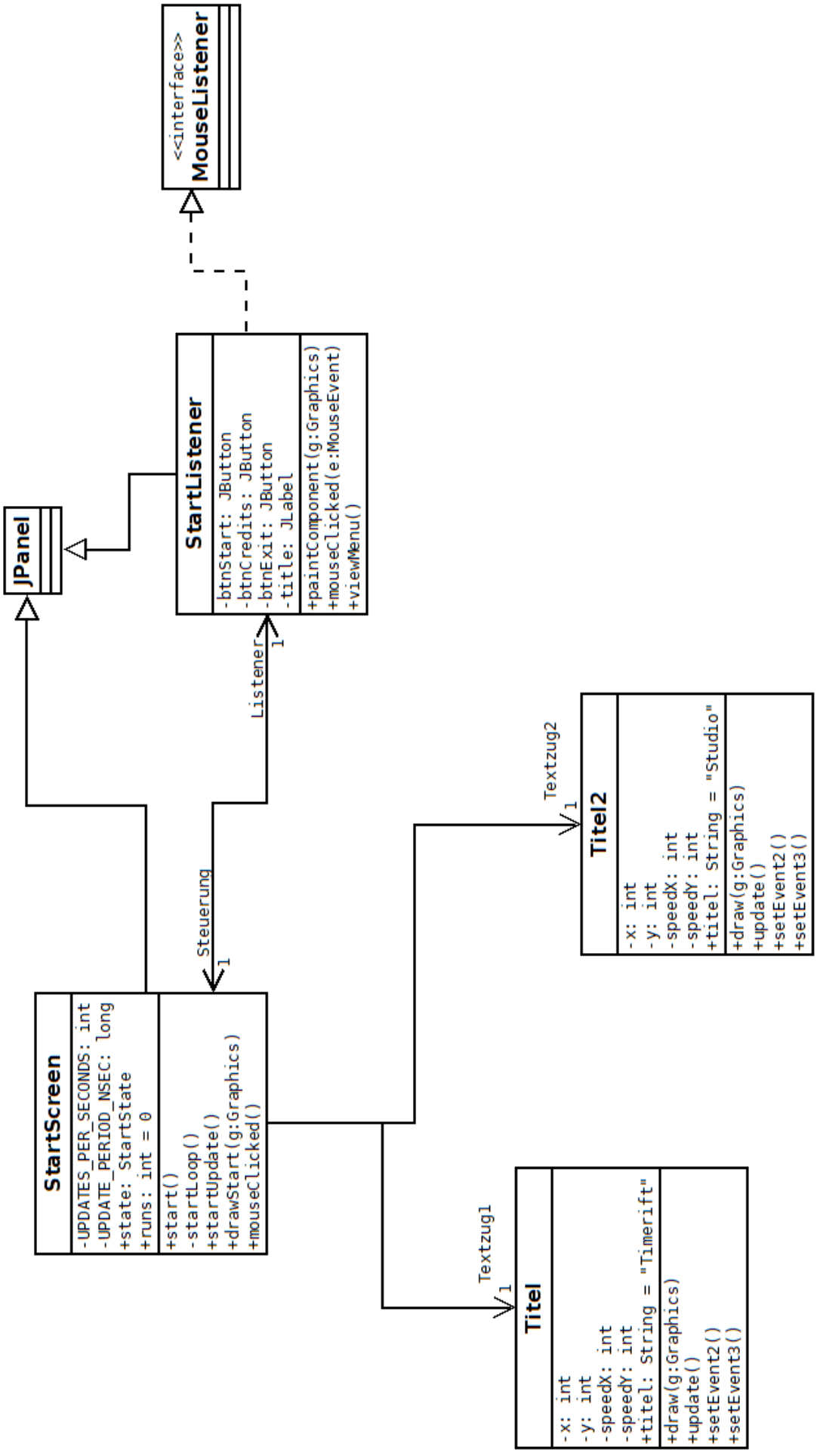

Abbildung 3.4: Start - Klassendiagramm

3.3 Einzelne Software Strukturen erklärt

3.3.1 Gameloop

In dem Gameloop werden die Spielobjekte geupdatet und dann gezeichnet. Das wird solange wiederholt, bis das Game seinen Zustand ändert.
Dabei möchte man eine (relativ) konstante Wiederholungsrate. Denn sonst ist der Spielablauf unterschiedlich und das ist nicht gerade angenehm.
Um dies gewährleisten zu können, prüfe ich die Zeit eines Durchgangs und lege fest, wie lange ein Durchgang dauern soll, wenn ich in einer Sekunde 60 Durchgänge haben möchte. Dafür teile ich einfach 1 Sekunde durch die Anzahl der präferierten Anzahl an Durchgängen. Damit weiß ich, wie lange ein Durchgang dauern sollte.
Falls nun geupdatet und gezeichnet wurde und man unter dieser Zeit ist, so muss man warten bis diese Soll-Zeit des Durchgangs erfüllt ist. Damit sind die FPS stabil. Außerdem gibt man in dieser Wartezeit Ressourcen für andere Operationen frei. Beispielsweise fürs Zeichnen, denn mit der 'repaint()'-Methode wird nicht sofort gezeichnet. Es wird viel eher vermittelt, dass das Toolkit sobald wie möglich zeichnen soll.

3.3.2 Enumerations

Durch einen enum namens 'state' kann man recht einfach definieren, in welchen Zustand sich das Spiel befindet. Natürlich könnte man einfach Integer genutzen (0 steht für...), aber so versteht man den zustand gleich. Ziemlich praktisch.
Falls man nun ingame umgebracht wurde kann man beispielsweise den Zustand des Spiels auf GAMEOVER schalten und im Gameloop wird dort durch Abzweigungen eine andere Update-Methode ausgeführt.
Wie Enumerations funktionieren wird im allgemeinen Part zu Java erklärt.

3.3.3 Kollision

Sehr wichtig ist natürlich die Kollision, da ohne sie nicht viel gehen würde. Bei meinem Spiel gibt es den Fall, dass ein Kreis auf einen Rechteck geprüft werden muss und auch ein Kreis zu einem Kreis.
Fangen wir bei ersterem an. Hierbei bilde ich sozusagen einen virtuellen Rahmen um das Rechteck und zwar mit der zusätzlichen Länge von dem Radius des Spielers. Wie in Abbildung 3.5 zu sehen ist.
Der Spieler kann mit dem Rechteck nur innerhalb dieses Rahmens kollidieren.

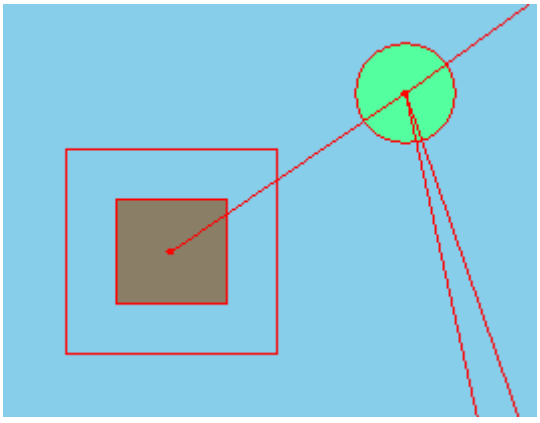

Abbildung 3.5: Virtueller-Rahmen

Deswegen teste ich ob sich der Spieler innerhalb des Rahmens befindet. Hierzu berechne ich den Abstand der X und Y Koordinaten. Dabei ist es wichtig den Betrag hiervon zu nehmen (Math.abs(x - rectX)).
Nun kann man den AbstandX und den AbstandY mit den Rahmen vergleichen, welcher die Breite/2 bzw. die Höhe/2 + den Radius ist.
Falls der Spielers sich nicht in diesem Bereich ist, so muss man auch nicht weiter prüfen.
Wenn dies doch der Fall ist, so prüft man, ob die Distanz auf X oder Y-Ebene kleiner ist als als dieser Rahmen.
In der Abbildung 3.6 kann man den Code noch sehen. Diese Kollisionsabfrage ist nicht zu 100% genau,

aber schon ziemlich genau. Es gibt Szenarien, wo es an Genauigkeit mangelt. Wie gesagt ist das bei diesem Projekt so mehr als nur ausreichend.

```
int distanceX = Math.abs(xMiddle - rectX);
int distanceY = Math.abs(yMiddle - rectY);

if(distanceX > (width/2 + radius))  return false;
if(distanceY > (height/2 + radius)) return false;

if(distanceX <= (width/2 + radius)) return true;
if(distanceY <= (height/2 + radius)) return true;

return false;
```

Abbildung 3.6: Collision-Code

In dem Fall Kreis-Kreis gehe ich etwas anders vor. Hierbei verwende ich den Satz des Pythagoras, um den Abstand zu ermitteln (mithilfe des X und Y-Abstandes wie im letzten Beispiel). Und vergleiche dies mit den Radien der beiden Kreise, denn der Abstand darf nicht kürzer sein als die Radien zusammen groß sind. Ansonsten kollidieren die Kreise.

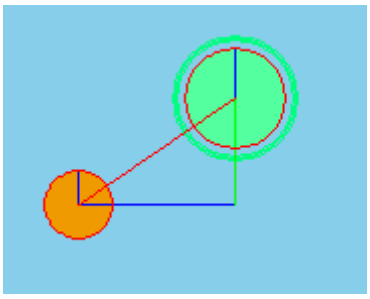

Abbildung 3.7: Collision-Kreis-Kreis

```
int distanceX = Math.abs(pX - iX);
int distanceY = Math.abs(pY - iY);

int distance = (int) Math.sqrt( Math.pow(distanceX, 2) + Math.pow(distanceY, 2) );
int distanceRadius = pR + iR;

if(distance <= distanceRadius) {
    //kollision
}
```

Abbildung 3.8: Collision-Kreis-Kreis-Code

3.4 Zustände

3.4.1 Schriftliche Beschreibung

Hier werde ich die Zustände meines Spiels systemisch abzuarbeiten. Dabei kann sich mein Spiel (grob gesehen) in drei Zuständen befinden. Entweder im StartScreen, im Spiel selbst oder in den Credits.
Zu Beginn wird jedoch noch alles initialisiert. Und einen ’Destroyed’ Zustand könnte man ebenfalls noch nennen.

Schauen wir uns jetzt die Zustände der einzelnen Zustände an. Fangen wir beim ersten Zustand an - Start-Screen.
Hier befindet man sich zu Beginn in der 'Preview', wo das Studio-Intro angesiedelt ist. Danach gelangt man zu dem Zustand 'Waiting', bei welchem man auf die Mauseingabe des Spielers wartet. Anschließend gelangt man in den Zustand 'StartMenu'. Hier gelangt man bei einem Zustandswechsel immer wieder zurück und kann zu den anderen Zuständen navigieren (Spielen, Credits).
Die Zustände 'Preview' und 'Waiting' können dann übrigens nicht noch einmal erreicht werden.
Kommen wir nun zu dem Spiel. Dieses besitzt ein enum, welches die Zustände schon darstellt. Diese sind 'INITIALIZED', 'DIFFICULTY', 'PLAYING', 'PAUSED', 'GAMEOVER' und 'DESTROYED'.
Zu Beginn werden die Spiel-Objekte initialisiert und man befindet sich im 'INITIALIZED' Status. Dieser Zustand wird jedoch nur bei Start der Software erreicht. Danach sind ja die Objekte schon initialisiert und werden nicht noch einmal neu erzeugt.
Wenn man ein Spieldurchgang startet (also der Gesamtzustand auf 'Spiel' wechselt) gelangt man zu dem Zustand 'DIFFICULTY'. Hier muss man einen Schwierigkeitsgrad wählen.
Wenn dies vollbracht ist, startet das Spiel und der Zustand geht zu 'PLAYING' über. Bei dem betätigen der rechten Maustaste oder der Space-Taste (unterschiedliche Pausen!) wechselt der Zustand zu 'PAUSED'.
Falls der Spieler sterben sollte, so wird zu dem Zustand 'GAMEOVER' gewechselt. Von diesem Zustand gelangt man, sowie auch aus dem 'PAUSED'-Zustand, zu dem Start-Menü.
Übrigens: Abfragen in dem Gameloop über den Zustand des Spieles entscheiden über die Update-Methode und in den Methoden zum Zeichnen geschieht dies ebenso.
Soviel zu den Zustände in dem Zustand 'SPIEL'. Schlussendlich gelangen wir zu dem Zustand 'Credits'. Dieser Zustand hat eigentlich nur drei Zustände: 'INITIALIZED', 'ACTIVE', 'DESTROYED'/'PAUSED'. Der 'INITIALIZED'-Zustand wird hier wieder nur einmal erreicht. Bei 'ACTIVE' wird gezeichnet und die Mausposition und der Mausklick abgehört. Außerdem sind die Spielobjekte aktiv.
Aus dem 'ACTIVE'-Zustand gelangt man wieder zum Zustand 'StartMenu'. Hierbei wird der Thread, in dem der Loop lief, durch eine Änderung einer Boolean-Variable beendet. Die Spiel-Objekte bleiben jedoch und so wirkt es so, als würden die Credits nur pausiert gewesen. Mehr oder weniger stimmt das ja auch.

3.4.2 Zustandsdiagramm

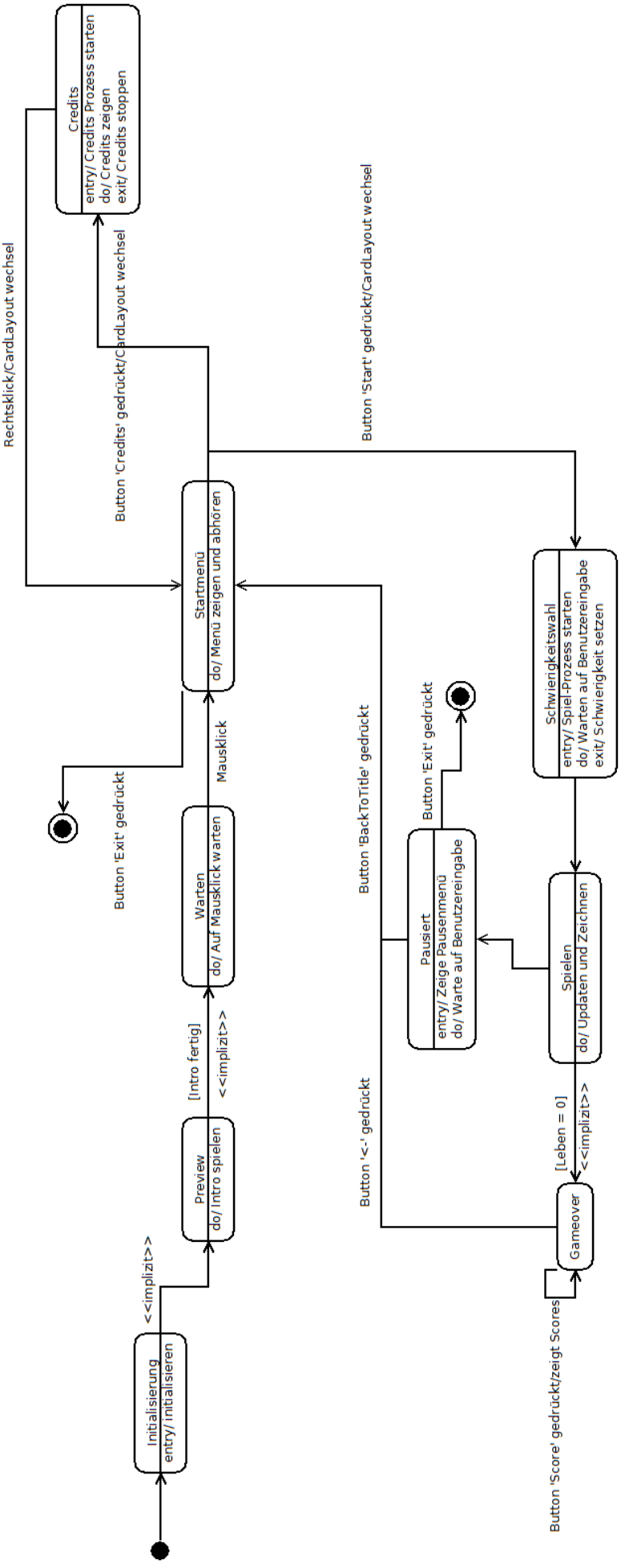

Abbildung 3.9: Zustandsdiagramm

3.5 Probleme

3.5.1 Es fehlen Unterteilungen

Ganz am Anfang hatte ich ein Problem mit der Unterteilung meines Programmes. Dieses besitzt drei JPanels (Spiel - Credits - StartScreen). Alle JPanels besitzen ihren eigenen Loop und ihre eigene Komponente, welche zum Zeichnen fungiert. Deswegen darf mein Programm immer nur eines der JPanel aktiviert haben. Zusätzlich muss ich zwischen dem aktiven und den inaktiven JPanels wechseln können.
Nach längerem Recherchieren habe ich meine Lösung gefunden. Mein Haupt-JPanel (welches dem JFrame als inhaltliche Komponente hinzugefügt wird) besitzt diese drei JPanel und es muss ein CardLayout hinzugefügt werden. Dort muss ich die drei JPanels nur noch hinzufügen. Das CardLayout wirkt dabei, wie ein Kartenstapel. Dabei ist immer eine Karte/Komponente aktiv. Genau das ist die richtige Lösung meines Problems.
Bei der Umsetzung gab es noch ein paar Schwierigkeiten, welche aber schnell gelöst waren. Dieses Problem ist damit abgehackt.

3.5.2 Die Zeit macht Probelme

In dem Spiel gibt es einen unzerstörbaren Modus, indem man für eine gewisse zeit unverwundbar ist. Dieser hatte an einigen Ecken Fehler.
Am längsten hatte ich mit dem Fehler zu kämpfen, wenn der Spieler schon unsterblich war und ein weiteres Schild aktiviert wurde. Leider habe ich es nicht hinbekommen die zeit einfach zu adden. Ich habe sehr viele Varianten geprüft, aber dies schien nicht so realisierbar zu sein. Immer einen neuen Thread mit einem Timer wollte ich ebenso wenig umsetzen. Dies wäre vielleicht machbar gewesen, jedoch sind Threads kostspielige Operationen.
Zudem kam ich irgendwann auf einen sehr guten Gedanken, der dann auch sehr gut klappte.
Wenn man einen Schild aktiviert, während man wartet, so wartet das Programm in einem externen Thread, bis der Timer wieder zu Verfügung steht.
Zwar muss man hier ebenfalls mit Threads arbeiten, jedoch ist der Unterschied groß, denn hier existiert der Thread nur kurz. Diese Lösung halte ich für sehr gut und sehe eigentlich keinen anderen vernünftigen Weg.

3.5.3 Wenn man seine Dinge gut gemacht haben will, muss man sie selbst machen

Recht früh fingen die Probleme mit den JButtons an. Sie wollten mit dem restlichen Gezeichnetem einfach nicht arbeiten. Erst dachte ich, dass ich awt mit swing zu sehr mische und darin der Fehler liegt.
Der Fehler ist jedoch simpler. Ich zeichne auf einem JPanel, welcher in einem JPanel ist, welcher in einem weiteren JPanel ist und dieser ist die inhaltliche Komponente des Jframes.
Problem ist, dass ich die JButtons auf dem vorletzten JPanel habe. Ich denke, dass der unterste JPanel den oberen überdeckt, vielleicht würde es mit dem Aufruf super(g) funktionieren. Oder man transferiert die Buttons auf das untere JPanel, was am schlausten wäre, denn dies habe ich ja wegen der Struktur/Übersicht gemacht. Und in meinem Programm habe ich dies auch schon an einer anderen Stelle getestet.
Aktuell habe ich noch selbst geschriebene Buttons. Dies ist ebenfalls eine Lösung, da sie gut funktionieren. Aber für eine Umsetzung mit Swing habe ich ja ebenfalls eine Lösung parat.

3.5.4 Music sollte relativ sein

In meinem letzten Projekt habe ich mich sehr geärgert. Ich konnte es nämlich nicht so gut exportieren. Die ausführbare Datei hatte nämlich keinen Schimmer, wo die Ressourcen (Soundclips, Bilder) sind und so konnte man das Spiel dann nur ohne Bilder und ohne Sound spielen.
Dieses Problem gehört der Vergangenheit an, denn Java bietet den sehr tollen und relativen Klassenlader. Über diesen kann man relativ zu dem Projekt und auf die Ressourcen zugreifen. Der Standort, wo das Projekt ist, ist dann völlig egal.
Um es genau zu beschreiben erstellt man ein Objekt von URL (Uniform Resource Locater). Hier hangelt man sich von der aktuellen Klasse und erhält einen relativen Datenpfad. Hier noch in Code: getClass().getClassLoader().getResource(X);

3.5.5 Dateien schützen sich und wie ich damit umging

Gegen Ende meines Projektes brachte ich das neue Feature des Scores mit ein. Es ist ein sehr wichtiges Feature, denn dieses sorgt für Motivation beim Spielen.
An sich hatte ich den dafür benötigten Code schnell in die Tastatur getippt, jedoch wollte es einfach nicht klappen. Die IOExceptions ägerten sich und reklammierten, dass sie diesen eingegebenen Pfad nicht finden. Ich hatte den Pfad extra relativ zum Projekt angegeben (siehe 'Musik sollte relativ sein') und war mir nicht sicher ob das Problem daher kommt. Ich probierte also fast alle möglichen Arten diesen Code zu schreiben aus und benutzte dabei unterschiedlichste Objekte. Irgendwann war mir klar, dass dies nicht das Problem sein konnte. Es musste irgendetwas anderes sein. Bis ich herausfand, dass die Dateien innerhalb meines Eclipse Ordners, alle schreibgeschützt sind. Und nun hatte ich den 'Fehler' endlich gefunden.
Um Eclipse nicht in seinem workflow zu stören, habe ich mich dazu entschieden die Sache weniger relativ zu lösen. Man muss nun stumpf einen Dateipfad angeben. Dort erstellt mein Programm automatisch die benötigten Text-Dateien. Falls der Dateipfad nicht angepasst wird, muss man wohl auf Scores verzichten. Das Spiel sollte dies trotzdem nicht beeinträchtigen (es sollten also keine Fehler dadurch entstehen). So habe ich es programmiert.
Auch wenn ich sehr viel Zeit für etwas unnötiges geopfert habe, so hat man daraus auch gelernt.

3.6 Feedback

Dieses Projekt hat mir ziemlich gefallen und ist mein bisher größtes Projekt. Tatsächlich war es schon immer mein Traum mein eigenes Spiel zu programmieren. Nun habe ich, dank dieser 3 Jahre auf dem technischen Gymnasium, das nötige Wissen um diesen Traum verwirklichen zu können. Und genau das habe ich bei diesem Projekt gemacht.
Ich habe auch allgemein viel über Java, und um genauer zu sein über javax.swing gelernt. Ich musste mich viel informieren, damit ich ein Spiel programmieren konnte, dass sauberen Swing-Code besitzt. Fragen wie Double-Buffering mussten noch geklärt werden, auch wenn ich schon etwas Erfahrung hatte.
In diesem Projekt konnte ich all meine kreativen Ströme freien Lauf lassen und mir überlegen wie ich diese realisieren wolle. Wie man vielleicht sehen konnte lag meine Hauptintention gar nicht auf ein perfektes Spiel, sondern darin, dass das Gesamtpaket stimmt. Ich rede von dem Intro und den Credits und das es überhaupt ein Menü gibt. Auch das man ingame pausieren kann (in zwei Arten).
Auch das mein Projekt diesmal relativer war fand ich super (ich rede von der Ressourcen-Einbindung). Insgesamt kann man sagen, dass ich es schön fand, dass ich all meine Ideen umsetzten konnte und dabei noch einiges gelernt habe.
Ich denke, dass ich meine Fortschritte an diesem Projekt gut sehen konnte. Natürlich gibt es noch viel zu lernen. Gerne würde ich langsam Schritte zur 3D Spielentwicklung machen, aber auch im 2D Bereich kann ich mich noch verbessern.
Java im allgemeinen ist mir nun viel näher. Ich besitze einen viel weiteren Überblick über die Klassenbibliothek und verstehe die Funktionsweise besser.
Auch die Spieleprogrammierung ist mir allgemein näher gekommen. Ich habe verinnerlicht welche Arten zu einem Spiel führen können. Dabei habe ich auch Lust bekommen meine eigene Engine zu programmieren.

Mit dieser PDF wollte ich ein Werk schreiben, welches alles beinhaltet was man von null auf für die Spieleprogrammierung gebrauchen könnte. Natürlich konnte ich dabei meist nur einen Überblick gewähren, aber schon das ist nach meiner Meinung sehr wertvoll. Man verliert nicht selten den Überblick über ein Thema und verweigert dem Verständnis zu verstehen. Hier sollte eben mein Werk ins Spiel kommen, aber es gäbe noch einiges hinzuzufügen und zu glätten. Auch eine weitere Programmierpsrache wäre sehr nett und könnte ein zukünftiges Projekt sein.
Insgesamt bin ich trotzdem sehr stolz auf diesen Text. Das ich alle Themen ansprechen konnte, welche ich für erwähnenswert hielt und auch für den Überblick wichtig waren. Dabei hoffe ich, dass es mehr Qualität als Quantität hat.

4 Quellen

4.1 Für den Bereich 1

1) 'Rechnerarchitektur' geschrieben von Todd Austin und Andrew S. Tanenbaum über das grundlegende gerüst eines Computers.
ISBN: 978-3-86894-238-5.

Bewertung: Ein fundiertes Grundlagenwerk über Computer. Der Schreibstil ist sachlich und der Inhalt theoretisch, aber realitätsnah.

Verwendung: In der Einleitung erkläre ich, warum Programmierer die Schnittstelle für die Spieleprogrammerierung und den Computern sind. Dabei nehme ich von diesem Werk die Definition der verschiedenen Ebenen eines Rechners.

2) 'Game Engine Architecture' geschrieben von Jason Gregory über den Aufbau einer Engine.
ISBN: 978-1-138-03545-4.

Bewertung: Ein riesiges Werk über Spieleengines. Strukturiert führt einem der Autor durch die Welt der Engines.

Verwendung: Von hier stammt unter anderem mein Wissen über Engines.

3) 'Components of an Game Engine' geschrieben von Harold Serrano über den Aufbau einer Engine.
https://gumroad.com/l/componentsofagameengine?recommended_by=library

Bewertung: Eine überschaubare PDF, welche sehr viel erklärt und Inhalte anschaulich näher führt. Für das Verständnis sehr hilfreich.

Verwendung: Von hier stammt unter anderem mein Wissen über Engines.

4) Wikipedia Eintrag: 'Spiel-Engine' erstellt und editiert von mehreren Autoren über Engines allgemein.
https://de.wikipedia.org/wiki/Spiel-Engine (stand 06.07.2020)

Bewertung: Für eine Übersicht sehr hilfreich. Teils etwas zu theoretische Formulierungen.

Verwendung: Von hier stammt unter anderem mein Wissen über Engines.

4.2 Für die Bereiche 2 & 3

1) Sachliteratur geschrieben von Sven Eric Panitz über das grundlegende Programmieren mit Java.
ISBN: 978-3-8348-1410-4, 'Java will nur Spielen'.

Bewertung: Der Autor geht sehr sachlich, aber auch sehr nachvollziehbar an die einzelnen Themen heran. Er erklärt jeden Schritt und sein Sachtext besitzt eine ordentliche Struktur. Heißt: Seine Beispiele bauen aufeinander auf und der Schwierigkeitsgrad steigert sich langsam. Hervorragendes Werk für Anfänger!
Sein Lebenslauf bestätigt einem sein Können auf diesem Gebiet.

Verwendung: Hier habe gelernt, wie ich mit den Swing und Swing-Komponenten umzugehen habe.

2) Sachliteratur geschrieben von Neos Thanh über die Spielprogrammierung mit Java.
ISBN: 9781686037634, 'Java Game Programming'.

Bewertung: Dieser Titel besitzt einige Stärken, jedoch mindestens genauso viele Schwächen. Der Autor erklärt meist nur sehr wenig und man muss es zwingend im Internet recherchieren. An manchen Stellen wird er ausführlicher, jedoch ist dann meist das Thema uninteressanter.
Was er mir jedoch gut zeigen konnte, war der generelle Aufbau eines Spiels mit javax.swing. Die Struktur konnte ich dadurch besser verstehen. Leider fehlt da noch einiges und auch diese Struktur musste ich noch nachbessern.
Für mich war dieses Werk aber auch nur hilfreich, da ich schon Erfahrung mitbrachte. Ohne diese wäre man bei diesem Werk verloren. Schade, denn der Titel verspricht anderes.

Verwendung: Hier habe ich die Grundstruktur meines Projektes her.

3) Sachliteratur geschrieben von Kai Günster über fast alle Themen was Java anbelangt.
ISBN: 978-3-8362-4095-6, 'Einführung in Java'

Bewertung: Bei diesem Werk handelt es sich um ein tolles Werkzeug für Anfänger. Aber auch Fortgeschrittene können hieraus noch einiges lernen. Der Sachtext ist gut strukturiert. Außerdem wird hier viel von Erklärungen gehalten und das ist super. So versteht man was man da eigentlich macht. In meinem Projekt hat es an mehreren Stellen Verwendung gefunden. Beispielsweise für das Einlesen und das Neubeschriften der Text-Dateien für die Scores.

Verwendung: Dieses Werk konnte ich an mehreren Stellen zur Rate ziehen. Bei allen Ungereimtheiten habe ich hier nachgeschlagen. Direkt entnommen habe ich Informationen über das Ein-/Auslesen von Text-Dateien.

4) 'Java ist auch eine Insel'-Openbook geschrieben von Christian Ullenboom über die GUI Swing.
http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_19_001.htm (stand 03.07.2020)
(So wie weitere Kapitel des Openbooks)

Bewertung: Dieses Werk gibt einen übersichtlichen und fundierten Blick über die Programmierung der grafischen Oberflächen mit Swing. Es wird dabei eindeutig mehr als nur an der Oberfläche gekratzt. Für das Verständnis, wie auch für die Praxis sehr Hilfreich.

Verwendung: Der Umgang und der Überblick über Swing und dessen Komponenten.

5) Öffentliche Debatte zum Thema Kollision von Kreisen.
https://stackoverflow.com/questions/401847/circle-rectangle-collision-detection-intersection (stand 03.07.2020)

Bewertung: In dieser Debatte trifft man auf viele unterschiedliche Ansätze zur Kollisionsberechnung. Teils sind die Lösungen nicht vertretbar, aber wenn man mit Verstand dabei ist macht das nichts.

Verwendung: In meiner Kollisionsberechnung von dem Spieler und den Steinen (siehe Projekt).

6) Online Video über die Kollision Kreis mit Kreis
https://www.youtube.com/watch?v=gHUVVQCDzkg (stand 26.06.2020)

Bewertung: Der Youtuber erklärt anschaulich und nachvollziehbar seine Strategie bei dieser Kollisionsprüfung.

Verwendung: In meiner Kollisionsberechnung von dem Spieler und den Items (siehe Projekt).

7) Online Video über die verschiedenen Layouts in Swing.
https://www.youtube.com/watch?v=JMkHA2ndook (stand 21.06.2020)

Bewertung: 'Easy Engeering Classes' erklärt hier recht übersichtlich die Layouts, welche es in Swing gibt. Dabei präsentiert er die Layouts relativ praxisbezogen.

Verwendung: Verständnis des Layout Systems in Swing und welche es gibt.

8) Online Artikel/Tutorial über das Ändern des Cursors in Swing.
https://javabeginners.de/Swing/Look_And_Feel/Eigenen_Cursor_definieren.php (stand 27.06.2020)

Bewertung: Hier wird einem sehr einfach gezeigt, wie man den Cursor eines JFrames ändern kann.

Verwendung: Ändern des Cursors, ingame.

9) Online Tutorial von Oracle über Swing und seine Anwendung.
https://docs.oracle.com/javase/tutorial/uiswing/ (stand 06.07.2020)

Bewertung: Hier werden sehr viele Tipps und Grundlagen vermittelt. Schön ist auch, dass es übersichtlich und detailreich übermittelt wird.

Verwendung: Viel Wissen über Swing.

10) Online Openbook von Stefan Middendorf, Reiner Singer und Jörn Heid über Java allgemein.
https://www.dpunkt.de/java/index.html (stand 06.07.2020), Java Programmierhandbuch und Referenz.

Bewertung: Ein altes aber umfassendes Werk über fast alle Bereiche von Java.

Verwendung: Für mich hat das Kapitel 'Oberflächenprogrammierung' am meisten Verwendung gefunden. Ich konnte hier fundiertes Wissen über javax.swing erlangen.

11) Online Sammlung von Artikeln über Java.
https://wiki.byte-welt.net/wiki/Kategorie:Java (stand 06.07.2020)

Bewertung: Hier lassen sich viele interessante und leicht verständliche Artikel zu Java Themen finden.

Verwendung: Bei dieser Quelle konnte ich sehr viel unterschiedliches lernen. Am meisten konnte ich Informationen über die Mal-Funktionsweise von Swing erhalten.

12) Online Artikel von mehreren Autoren über Java Sound.
https://de.wikipedia.org/wiki/Java_Sound (stand 10.07.2020)

Bewertung: Ein kleiner, aber übersichtlicher Artikel über die Java Sound API.

Verwendung: Diesen Artikel konnte ich zum Verständnis von Java Sound verwenden.

13) DIA: Diagramm Editor -Software
http://dia-installer.de/ (stand 10.07.2020)

Bewertung: Eine einfach zu bedienende Software zum digitalen Zeichnen von UML-Diagrammen. Sie bietet nicht viele Features, aber es reicht um gut UML-Diagramme zu erstellen und ist vor allem kostenlos.

Verwendung: Ich konnte mithilfe dieser Software die Klassendiagramme und das Zustandsdiagramm erstellen.