Konzeption und Entwicklung eines iOS Spieles in Swift auf der Basis von SpriteKit und dem MVCS-Entwurfsmuster

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Eugen Waldschmidt

2024311

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Prof. Dr. Edmund Weitz

Zweitprüfer: Prof. Dr. Andreas Plaß

19. Oktober 2015

Inhaltsverzeichnis

Abstract

This thesis employs the design and development of an iOS game, written in Swift and work with the SpriteKit framework. The application implements a software architecture based on models, views, controllers and services. Furthermore, to understand the software architecture it explains other design pattern like observer, singleton and MVC. Afterwards it shows the operation and the important algorithm of the application. At the last point it draws the conclusion about SpriteKit, Swift and the MVCS pattern.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Konzeption und Entwicklung eines iOS Spieles, welches in Swift geschrieben wird und das SpriteKit Framework als Basis nutzt. Zudem wird eine Architektur mit den Bestandteilen Model, View, Controller, Services entwickelt und in dem Spiel implementiert. Hierfür werden erstmal vorhandene Entwurfsmuster vorgestellt und evaluiert. Hinterher wird das Spielprinzip und die Architektur vorgestellt, anschließend die Funktionsweise und die wichtigen Algorithmen des Programms erklärt. Zuletzt wird ein Fazit bzgl. SpriteKit, Swift und dem MVCS-Entwurfsmuster gezogen.

1 Einleitung

1.1 Motivation

Das Unternehmen nodapo Software GmbH (im weiteren Verlauf nodapo) wurde 2015 von Alexander Poschmann gegründet. Das Unternehmen beschäftigt mittlerweile vier Mitarbeiter. nodapo erstellt für Kunden Web-basierte Software und nebenher eigene Applikationen für den mobilen Markt. Zum ersten Mal seit der Gründung kam die Idee von einem Spiel als Produkt, woraufhin das Spiel **Drawix** entstand. Mithilfe des Spiels soll ein Einstieg in den Spiele-Markt erreicht und in Zukunft weitere Spiele produziert werden.

„Die verschiedenen Funktionen eines Programms in verschiedenen Teilen einer Lösung zu realisieren, ist ein generelles Bestreben in der Programmentwicklung [...].“, (Joachim Goll 2013: S. 377)

In der Softwareentwicklung ist eine modulare Software immer ein generell angestrebtes Ziel. Durch eine modular aufgestellte Architektur, kann eine Software effizienter erweitert werden. Eine klare Schichtentrennung in einer Architektur fördert die Übersichtlichkeit von einem Programmcode. Dieses bestreben wird in der Architektur von Drawix eine große Rolle spielen.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung des Spieles in Apples neuer Programmiersprache Swift. Da diese Sprache relativ jung (veröffentlichung 2014 [Stefan Popp & Ralf Peters (2015)]) ist, kann es stets zu Problemen kommen, wie z.B. eine unvollständige Dokumentation. Des Weiteren ist es gewünscht, ein Entwurfsmuster zu verwenden, welches bisher in keinem iOS-Projekt des Unternehmens bis zu diesem Zeitpunkt zum Einsatz kam.
Folgende Schwerpunkte werden für diese Arbeit definiert:

1. Anwendung verschiedener Entwurfsmuster, um eine übersichtliche Architektur zu schaffen. Diese soll modular sein, um zukünftige Erweiterungen zu ermöglichen.

2. Die Programmierung soll weitestgehend unabhängig von vorhandenen Frameworks realisiert werden.

3. Die Architektur soll so konzipiert werden, dass eine möglichst einfache Portierung auf andere Systeme stattfinden kann.
4. Eigens konzipierte Algorithmen sollen implementiert werden.
5. Am Ende der Arbeit soll ein spielbarer Prototyp vorhanden sein.

1.3 Aufbau

In diesem Kapitel wird erklärt, weshalb das Spiel entwickelt wird und welche Ziele zum Ende dieser Arbeit erfüllt sein müssen.

Im zweiten Kapitel wird auf die Zielgruppe eingegangen. Des Weiteren werden einige vorhandene Spiele analysiert. Die Analyse soll einen Überblick über den Spiele-Markt geben. Das Ziel ist, ein grobes Bild von der Gestaltung der Konkurrenzprodukte zu erhalten. Außerdem könnte die Spielmechanik Aufschluss geben, welche Merkmale bei der Entwicklung beachtet werden sollten.

Im dritten Kapitel werden die für das Spiel gewählten Technologien erläutert. Dabei wird Swift vorgestellt und die wesentlichen Punkte erklärt, die im Verlauf der Arbeit erwähnt werden. Hinterher wird auf SpriteKit[1] eingegangen und der Grund für diese Entscheidung erläutert. Des Weiteren werden die für die Arbeit gewählten Snippets erklärt

Im vierten Kapitel werden Design-Pattern vorgestellt, die ihre Anwendung in Drawix finden. Dabei werden das Observer-Pattern, das Singleton-Pattern und MVC als Basis für das MVCS-Entwurfsmuster eingesetzt.

Im fünften Kapitel dieser Arbeit wird das Spielprinzip, die Gestaltung und die Anwendungsarchitektur beschrieben. Bei der Gestaltung werden alle Entwürfe erstellt, die in Drawix vorhanden sein sollen. Dabei handelt es sich nicht um das endgültige Design, sondern nur um jenes für den Prototypen. Bei der Anwendungsarchitektur werden die einzelnen Module, die in Drawix Verwendung finden, als Klassen abgebildet, es wird erklärt welche Aufgabe diese haben und auf welche Art die Kommunikationswege aufgebaut sind.

Im sechsten Kapitel wird die Umsetzung des Spiels vorgestellt. Dabei wird erläutert wie das Programm und die drei konzipierten Algorithmen im Detail funktioniert.

Im siebten Kapitel wird der Prototyp vorgestellt. Dabei werden die Ziele aus 1.2 abgeglichen und auf deren Vollständigkeit geprüft. Des Weiteren werden die Wünsche

[1] SpriteKit ist ein 2D Spiele-Framework von Apple.

von nodapo aufgelistet, wie die weitere Entwicklung des Spieles stattfinden soll und welche Erweiterungen in Zukunft geplant werden.

Im letzten Kapitel dieser Arbeit wird ein Fazit von der Entwicklung in Swift, dem Entwurfsmuster MVCS und SpriteKit gebildet.

2 Analyse

2.1 Marktanalyse

Der Markt für Puzzle-Spiele[1] ist groß und unübersichtlich. Häufiger sind diese Spiele reine Denkspiele. Manche Spiele fügen, für die gewisse Spannung und Herausforderung, Zeitdruck hinzu. Hierbei hat der Spieler pro Zug wenig Zeit und muss sich schnell entscheiden, um beispielsweise Punktabzüge zu vermeiden.

Zusätzlich können Puzzle-Spiele jederzeit unterbrochen werden. Meistens entscheidet der Spieler, ob und wann er das Spiel spielen möchte. Diese Option ist im Genre von beispielsweise Simulationen nicht immer enthalten. Durch die klein gehaltenen Level in Puzzle-Spielen benötigt der Spieler nicht unbedingt viel Zeit, um ein Level zu meistern. Aus dem Grund können diese Spiele auf der Fahrt zur Arbeit, Uni oder Schule gespielt und jederzeit unterbrochen werden. Des Weiteren müssen Puzzle-Spiele keine Geschichte enthalten, die dem Spieler erzählt werden soll. In vielen Fällen könnte ein trickreiches Spielprinzip für Spannung sorgen.

Um nun einen so großen Markt spezieller zu betrachten, müssen zunächst Kriterien festgelegt werden. Zum einen ist es wichtig zu verstehen, warum die ausgewählten Puzzle-Spiele so erfolgreich sind und zum anderen können gestalterische Aspekte viel über das Spiel und seine Zielgruppe aussagen.
Nach einigen Gesprächen mit dem Geschäftsführer von nodapo wurden folgende Fragestellungen definiert:

- Welche Spiele besitzen ein ähnliches Spielprinzip [5.1] wie Drawix?
- Was ist die Motivation bei den verglichenen Spielen?
- Besitzen die Spiele besondere Eigenschaften, die das Spiel interessant für den Spieler machen?
- Wie sind die Konkurrenzprodukte gestaltet?

[1] Puzzle-Spiele sind Geduldsspiele, welche z.B. Rätsel enthalten können.

2.1.1 Candy Crush Saga

Candy Crush Saga[2] von King.com Limited eine Unternehmen aus England wurde am 14. November 2012 für mobile Endgeräte veröffentlicht und hat seitdem einen der höchsten Anteile an Downloads und Umsätze im Apple App-Store [Wikipedia (2015)]. Zum Zeitpunkt der Analyse befand sich das Spiel auf Platz 3 im Apple App-Store - Kategorie: Spiele - Puzzle[3].

Das Spielprinzip besteht darin, zufällig angeordnete Süßigkeiten durch Wechselgesten so anzuordnen, dass mindestens drei gleiche Süßigkeiten in einer Reihe oder Spalte sind. Sobald das erfüllt ist, werden diese aufgelöst und von oben rücken weitere Süßigkeiten nach. Bei einer Wechselgeste kann nur eine Süßigkeit horizontal oder vertikal mit einer anderen getauscht werden. Das Ziel ist durch das Auflösen eine gewisse Punktzahl zu erreichen und/oder eine Aufgabe zu erfüllen. Dabei existiert nur eine bestimmte Anzahl an Zügen. Eine Aufgabe kann darin bestehen, eine andere Süßigkeit bis zum Rand des Spielfeldes zu treiben und diese dadurch aufzulösen.

Beim erstmaligen Start des Spiels wird dem Spieler eine Anleitung präsentiert. Hierbei und während des weiteren Spielgeschehens begleitet ein Charakter (oder Maskottchen) den Spieler. Sobald der Spieler eine gewisse Dauer nicht reagiert, greift der Charakter in das Spiel ein und schlägt dem Benutzer einen Zug vor. Sobald das Level durch die oben genannten Kriterien erfüllt wurde, wird ein weiteres Level freigeschaltet. Wenn die Kriterien nicht erreicht wurden, gilt das Level als verloren und dem Spieler wird eine Auswahl zwischen Goldbarren[4] ausgeben oder ein Leben zu opfern. Hat der Spieler sein letztes Leben geopfert, kann nicht weiter gespielt werden. Die Leben regenerieren sich halbstündig, d.h. jede halbe Stunde kommt ein Leben hinzu, bis es fünf sind. Außerdem können Leben über das soziale Netzwerk Facebook[5] verschenkt werden.

Das Spiel ist sehr bunt gestaltet und jede Süßigkeit hat dabei seine eigene Farbe. Im Hintergrund befindet sich immer eine Grafik, die zu der jeweiligen Spielwelt passt. Im oberen Teil des Spieles sind Optionen, Lebensanzahl, Bonusgegenstände und die Zielpunktzahl zu sehen; im unteren Teil die Anzahl der noch vorhandenen Spielzüge, aktuelle Punktzahl und eine Fortschrittsanzeige, welche anhand der Punktzahl eine Bewertung in Sternen anzeigt. Grundsätzlich wirkt das Spiel sehr kindlich und knall bunt, was offensichtlich aufgrund des Themas Süßigkeiten so festgelegt wurde.

[2] oder kurz: Candy Crush
[3] Aktuelle Version: 1.23.1 iPhone, 1.22 Android, Stand: 03.08.2015
[4] Goldbarren sind bei Candy Crush eine Währung, die man in der App kaufen kann.
[5] http://www.facebook.com/

2.1.2 1010!

Das Spiel mit dem Titel „1010!“ besitzt eine ähnliche Spielmechanik wie bei Drawix geplant ist. Entwickelt wurde das Spiel von grams[6], welcher seinen Sitz in der Türkei hat.

Das Spielprinzip ähnelt dem von Tetris. Es unterscheidet sich jedoch darin, dass der Spieler aus drei vorgegebenen Steinen einen auswählt und diesen in das Spielfeld hineinzieht. Das Spielfeld besteht aus 10x10 Feldern und die Steine aus drei bis neun Quadraten. Wenn alle drei Steine im Spielfeld eingesetzt wurden, erscheinen drei neue. Wie auch in Tetris, werden vollständige Reihen und zusätzlich Spalten entfernt und hierfür Punkte vergeben. Das Auflösen mehrerer Reihen und Spalten zugleich, wird mit Bonuspunkten ergänzt. Durch Probieren hat sich ergeben, dass die Formel hierfür offensichtlich wie folgt aussieht:

$$\sum_{n=1}^{a} 10 \cdot n | n, a \in \mathbf{N}$$

In dieser Formel entspricht **a** der Anzahl der vollen Zeilen bzw. Spalten und **n** mindestens einer vollen Zeile bzw. Spalte. Wenn ein Stein gelegt wird, werden hierfür auch Punkte angerechnet.

Verschiedene Level gibt es zum Zeitpunkt der Analyse nicht. Aus diesem Grund ist das Besondere an dem Spiel, ist das Hineinziehen von Steinen an eine beliebige Stelle im Spielfeld. Das Spiel wirkt bunt, jedoch aufgeräumt. Das Spielfeld besteht aus grauen Steinen und die einzelnen Steine verfügen jeweils über fest zugeteilte Farben. Zudem sind alle Spielelemente auf einem weißen Hintergrund platziert.

2.1.3 1010! World

Das Spiel „1010! World“ ist von den selben Entwicklern wie **1010!** [2.1.2]. Das Spielprinzip ist leicht abgeändert im Vergleich zu **1010!**, jedoch ist die Steuerung gleich geblieben. Statt eines endlosen Spielerlebnisses, wie in **1010!**, verfügt das Spiel über einzelne Level. In jedem Level müssen Aufgaben erfüllt werden, um in das nächste Level zu gelangen. Es gibt wie bei **Candy Crush** 2.1.1 ein Lebenssystem. Des Weiteren müssen wie in **1010!** Reihen/Spalten befüllt werden. Zusätzlich müssen gewisse Aufgaben erledigt werden. Dabei stehen dem Spieler jedoch nur eine bestimmte Anzahl an Spielzügen zur Verfügung.

Im Gegensatz zu **1010!** wirkt **1010! World** bunt und stark verniedlicht. Außerdem wurde ein Maskottchen hinzugefügt, welches einen im Spielverlauf begleitet. Zusätzlich stellt das Maskottchen Aufgaben, die der Spieler meistern soll oder erklärt dem Spieler neue Spielelemente. Trotz der bunten Farben wirkt das Spiel aufgeräumt und übersichtlich.

[6] http://www.grams.gs

2.1.4 Zusammenfassung

Tabelle 2.1: Eigenschaftstabelle

	Charakter	Level	Spielwelt	Anleitung
Candy Crush Saga	ja	ja	ja	ja
1010!	nein	nein	nein	nein
1010! World	ja	ja	ja	ja

Diese Tabelle dient dazu, eine Übersicht über die Eigenschaften zu schaffen. Alle drei Spiele sind bunt gestaltet. Bis auf **1010!** verfügen alle anderen Spiele über einen Charakter, ein Level-System, eine Spielwelt und eine Anleitung. Die Anleitung dient als Beschreibung der Steuerung des Spieles.

Alle drei Spiele besitzen ein ähnliches Spielprinzip, d.h. sowohl in Reihen als auch in Spalten müssen Elemente gesammelt und aufgelöst werden. Bei **1010!** muss eine möglichst hohe Punktzahl erreicht werden, was die einzige Motivation hierbei ist. Hingegen müssen bei **1010! World** und bei **Candy Crush** auch zum Teil Aufgaben erfüllt werden.

2.2 Zielgruppe

Für die Ermittlung einer Zielgruppe, wird zunächst auf die Studie von Ingo Kamps (2014) verwiesen. Laut dieser Studie sind knapp 31% der Frauen eher bereit für In-App-Käufe[7] Geld auszugeben und verbringen 35% mehr Zeit als männliche Spieler auf mobilen Endgeräten. Selbst das erstmalige Öffnen der Spiele innerhalb der ersten Woche nach dem Download, werden mit einer 42% höheren Wahrscheinlichkeit getätigt.

Die für Puzzle-Spiele gespielte Zeit ist geringer als bei anderen Genres. Sowohl männliche und weibliche Spieler spielen Puzzle-Spiele im gleichen Verhältnis. Männliche Spieler spielen eher Karten-, Rollenspiele oder „Tower-Defense“[8] Spiele. Die weiblichen Spieler bevorzugen eher Spiele vom Genre Management bzw. Simulation [Ingo Kamps (2014)].

Aus diesem Grund ist eine Geschlechtertrennung bei einem Spiel wie Drawix nicht zwingend nötig. Offensichtlich sollten andere Aspekte, wie Altersbeschränkung und Endgeräte-Wahl betrachtet werden. Da das Spiel keine Gewaltszenen o.ä. enthalten wird, kann das Spiel ohne Altersbeschränkung veröffentlicht werden. Zudem wird das

[7] Softwaregüter oder Features, die durch Kauf in der App oder im Spiel freigeschaltet werden.
[8] Tower-Defense oder zu Deutsch Turm Verteidigung sind Spiele wo der Spieler Türme aufstellen muss um seine Basis vor Angreifern zu verteidigen.

Spiel in Swift geschrieben, von daher können auch hier die Endgeräte von Apple, wie das iPad und das iPhone, festgelegt werden. Diese Anforderung führt dazu, dass das Betriebssystem iOS8 oder neuer als Rahmenbedingung definiert werden kann.

Zusätzlich können Gelegenheitspieler festgelegt werden, da das Spiel in der ersten Version über zwei Spielarten verfügen soll, Zeit beschränkt und unbeschränkt. Wie in der Marktanalyse[2.1] vorgestellt kann das Spiel bei kurzer Spielzeit gespielt werden und jederzeit unterbrochen werden.

Aus dieser Analyse geht hervor, dass die Zielgruppe offensichtlich den Gelegenheitsspieler mit einem iPhone oder iPad entspricht, ohne Alters- oder Geschlechterbeschränkung.

3 Technologiewahl

3.1 Swift

Drawix wird komplett in Swift geschrieben. Swift ist eine recht junge Sprache (veröffentlicht 2014), die auf der WWDC[1] vorgestellt wurde. Grund für die neue Programmiersprache ist, dass Objektive-C aus den 80er Jahren stammt und eine Grunderneuerung nötig war [(Stefan Popp & Ralf Peters 2015: S. 3)]. Bei der Entwicklung von Swift die Vorzüge aus C und Objective-C zu übernehmen, jedoch auf deren Einschränkungen zu verzichten. Das bedeutet, es werden neue Sprachtypen wie Closures, generic Types und Protokolle(Objective-C) vorhanden sein. Darüber hinaus gibt es multiple Rückgabewerte etc..

Für Swift wird die LLVM (Low Level Virtual Machine) genutzt. Dies ist ein modularer Compiler mit einer virtuellen Maschine[2]. Dieser Compiler-Unterbau kann derzeit Programmcode in unterschiedliche Programmiersprachen, darunter auch Swift und Objektive-C kompilieren [(Stefan Popp & Ralf Peters 2015: S. 2,3)].

3.1.1 Enumerator

Enumeratoren, kurz **enum** oder zu Deutsch *Aufzählung*, ist eine Liste von konstanten Werten. In der Programmiersprache C enthalten diese symbolische Konstanten. In Swift besitzen Enumeratoren die selben Fähigkeiten und können darüber hinaus Integer, Strings, einen anderen Datentyp[3] oder Nachkommastellen enthalten. Zusätzlich können in Enumeratoren Funktionen, Konstruktoren oder sogar Protokolle[4] definiert werden. Solche Möglichkeiten sind normalerweise ein Bestandteil von Klassen[(Stefan Popp & Ralf Peters 2015: S. 89)]. Jede Aufzählung enthält in Swift Fälle oder sogenannte **Cases**. Jedem **Case** kann ein Wert von den genannten Datentypen zugewiesen werden. Aus dem Grund ähneln Enumeratoren stark einer *Switch*-Syntax.

Die Werte, die ein **enum** in sich trägt, werden Raw-Values, zu Deutsch *rohe Werte*, genannt. Um diese zu lesen, müssen Enumeratoren ausgepackt oder in Swift *unwrapped* werden. Erst nach dieser Tätigkeit können die Werte, die in einem Case definiert

[1] Apple Worldwide Developers Conference https://developer.apple.com/wwdc/
[2] Besitzt einen virtuellen Befehlssatz, einen Prozessor (GPU, CPU)
[3] Datentypen können z.B. Klassen entsprechen
[4] Protokolle sind vergleichbar mit Java-Interfaces

wurden, in Variablen von dem Typ übertragen werden, der im **Case** deklariert wurde. Ein interessanter Punkt ist, dass Enumeratoren in Swift sich selbst rekursiv aufrufen können. In Drawix werden Enumeratoren ihren Einsatz in den Farben [5.4] bekommen.

3.1.2 Strukturen

In Swift gibt es die sogenannten Strukturen oder auch **struct** genannt. Strukturen ähneln sich in der Syntax stark den Klassen, besitzen jedoch einige Eigenschaften von Klassen nicht [(Stefan Popp & Ralf Peters 2015: S. 94)].

- Beide besitzen **Properties**, in den Werte gespeichert werden können.
- Sie können beide Funktionen in sich tragen.
- Es besteht die Möglichkeit in beiden Konstruktoren zu erstellen.
- Beide können außerhalb erweitert werden, d.h. für alle Objekte vom Typ eine Änderung durchführen, ohne vom Objekt zu erben.
- Beide können Protokollen entsprechen.

Im Gegensatz dazu können Strukturen folgende Dinge:

- Strukturen können keine **Properties** als auch Methoden vererben
- Strukturen können nicht deinitialisiert[5] werden.
- Bei Strukturen kann ein und die selbe Instanz an mehren Stellen gleichzeitig nicht genutzt werden.
- Strukturen können in Methoden nicht als *Reference Type* genutzt werden, d.h. wenn diese in Methoden genutzt werden, werden Strukturen als Kopie übergeben und nicht als Referenz [(Stefan Popp & Ralf Peters 2015: S. 105)].

3.2 Frameworks & Snippets

Alle Frameworks und Snippets, die für das Spiel benötigt wurden, werden in diesem Kapitel erklärt. Einige Funktionen sind in Swift schwer zu implementieren oder gar nicht vorhanden. Die hier verwendeten Snippets sollten einen überschaubaren Code besitzen. Unter anderem bedeutet es, dass auch die Implementierung ohne Schwierigkeiten ablaufen sollte. SpriteKit ist von Apple und würde von daher auch für auf allen Apple-Geräten unterstützt werden. Andere Snippets wie das für den Array2D und für die Obserable Funktion, könnten in Zukunft Probleme mit sich tragen, z.B. nicht rückwärtskompatibel Uptates von Swift.

[5] Speicher freigeben und Objekt entfernen

3.2.1 SpriteKit

SpriteKit ist eine Spiele-Engine für 2D Spiele. Es bietet eine große Infrastruktur für Grafiken[6] und Animationen. Des Weiteren liefert SpriteKit eine Physik-Simulation, die hier jedoch keine Verwendung finden wird. Jeder Frame wird neu dargestellt, dabei folgt SpriteKit im Unterbau einer Schleife:

1. Update
2. Aktionen
3. Aktionen ausgeführt
4. Physik
5. Physik ausgeführt
6. Randbedingung
7. Randbedingung ausgeführt
8. Update beenden
9. Anzeigen

Im Gesamtumfang dient diese Schleife dazu, dass in jedem Frame eine gewünschte Änderung durchgeführt wird.

Sobald ein Frame geladen werden soll, führt die Schleife zunächst die Update-Methode aus. In dieser können Simulationen ausgeführt, Spiellogik angewendet oder benötigte Aktionen definiert werden.

Im nächsten Schritt werden alle erstellten Aktionen[7], auf die Objekte in der Szene angewendet. Wenn alle Aktionen ausgeführt wurden, gibt SpriteKit nochmal die Möglichkeit, vor dem nächsten Frame, Einfluss auf die Aktionen zu nehmen. Man kann diese löschen oder nochmals verändern.

Als nächstes wird die Physik von SpriteKit angewendet, sofern Objekte damit versehen wurden. Sobald dies geschehen ist, gibt SpriteKit auch hier die Möglichkeit auf die Physik für den nächsten Frame Einfluss zu nehmen.

Haben Objekte gewisse Bedingungen definiert, werden diese nun angewendet. Darunter ist zu verstehen, dass z.B. zwei Objekte immer nur einen bestimmten Abstand voneinander haben sollen. Auch hier kann nach der Anwendung der Bedingungen

[6] Eine Grafik wird im SpriteKit als Sprite bezeichnet.
[7] Animationen, Soundeffekte o.ä.

Einfluss auf diese genommen werden.

Als vorletzten Punkt ermöglicht SpriteKit dem Entwickler zum letzten Mal, vor dem nächsten Frame, Einfluss auf die Szene zu nehmen. Wenn alle Punkte abgearbeitet wurden, wird die Veränderung auf der Szene dargestellt [About SpriteKit (2014)]. Diese Schleife wird zur Laufzeit aufgerufen. Aus dem Grund muss bei der Entwicklung darauf geachtet werden, dass nicht zu viele unnötige Aktionen aufgerufen werden, denn sonst kann das Spiel eine kleinere Frame-Rate erreichen und dies kann zu Verzögerungen im Programm führen.

Jegliches Objekt, welches bzw. das die Szene darstellt, erbt in SpriteKit von der Klasse **SKNode**. Diese ermöglicht die Positionierung, Rotation etc.. Die Klasse **SKScene** ist die Klasse, die für die Darstellung auf der View verantwortlich ist. Die Szene bekommt Kind-Elemente, die auf dieser dargestellt werden. Die Darstellung geschieht in der Regel in einer **SKView**. Eine SKView ist eine View, die speziell von einer SKScene mit Anzeigeelementen versehen wird. Eine Szene teilt der View mit, was dargestellt bzw. aktualisiert werden soll. Wie im MVC [4.3] beschrieben, entspricht die Szene demnach einem Controller. Um ein Bild darzustellen, nutzt man in SpriteKit die **SKSpriteNode**, welche auch von der **SKNode** erbt. Eine **SKSpriteNode** ist eine Sub-Klasse von **SKNode**. Dadurch ermöglicht es eine **SKSpriteNode** zu positionieren, drehen oder andere Eigenschaften, die das Objekt **SKNode** anbietet. Zusätzlich besteht die Möglichkeit, eine **SKSpriteNode** mit einem Bild, einer Farbe oder einer Textur zu versehen. Außerdem kann man auf beide Objekte Aktionen ausführen, z.B. um diese zu bewegen.

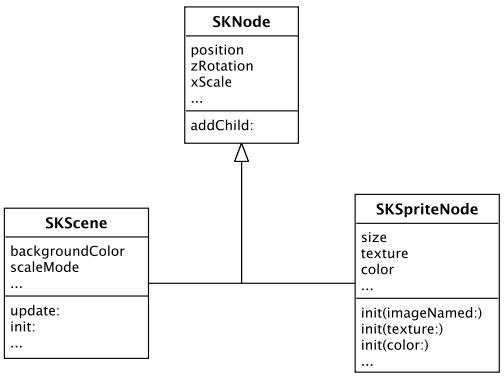

Abbildung 3.1: Erbverhalten von SpriteKit-Klassen

Wenn ein Element in der Szene dargestellt werden soll, wird es über die Methode *addChild*, auf das gewünschte Element angewendet, als Kind-Element angehängt. Durch dieses Verfahren kann, je nach Anzahl, ein großer Baum entstehen. Jedoch müssen Kind-Elemente nicht nur in der Szene vorhanden sein. Wenn mehrere **SKSpriteNodes** zusammengehören, kann man diese in einen Container setzen. Der Container

kann sowohl ein **SKSpriteNode** als auch ein **SKNode**-Objekt sein.

Jede Szene gibt vier Methoden vor, mit deren Hilfe ein Event vom Benutzer auf dem Bildschirm erfasst werden kann. Bei jedem Berühren des Bildschirms wird die **touchesBegan**-Methode ausgeführt; das bedeutet, dass eine Geste begonnen wurde. Jede Bewegung nach dem Beginn wird über die **touchesMoved**-Methode abgedeckt und das Abheben des Fingers vom Bildschirm über die **touchesEnded**. Falls bei einer Geste ein Telefonanruf eingeht, wird die **touchesCancelled**-Methode aufgerufen. Mit diesen vier Methoden können so gut wie alle Fälle abgedeckt und eine Geste ermittelt werden. Um eine Geste zu ermitteln, können in den Methoden die genauen Positionen, an denen sich der Benutzer mit dem Finger befindet, abgefragt werden. Durch die Positionen können alle Objekte, die durch die Geste berührt wurden, mit Aktionen versehen werden, beispielsweise dem Entfernen der Objekte.

3.2.2 Array2D

Das Spielfeld [5.3] wird als zweidimensionaler Array abgebildet. Swift bietet zwar zweidimensionale Arrays an, jedoch mit der Einschränkung, dass es sich hierbei um ein sogenanntes Dictionary[8] handelt. Für Drawix ist es notwendig, Koordinaten wie Zeile und Spalte für die Objekte zu definieren. Hierfür gibt es eine generische Klasse[9], die genau dieser Anforderung entspricht [Matthijs Hollemans (2014)].

Ein Array2D wird beim Erzeugen in einen eindimensionalen Array umgewandelt. Wenn ein Array von vier Zeilen und vier Spalten benötigt wird, erzeugt die generische Klasse einen eindimensionalen Array mit der Länge 16.

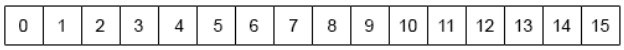

Abbildung 3.2: Eindimensionaler Array von Array2D mit 4x4

Die folgende Formel bestimmt bei der Abfrage von Zeile und Spalte, welchem Index dies im eindimensionalen Array entspricht.

$$Zeile \cdot Spaltenanzahl + Spalte | Zeile, Spalte, Spaltenanzahl \in \mathbf{Z}_{16}$$

In Abb 3.3 wird der theoretische zweidimensionale Array von dem eindimensionalen veranschaulicht. Angenommen es wird der Wert aus Zeile 2 und Spalte 2 benötigt, ergibt dies laut der Formel 3.2.2 den Index 10.

[8] Ein Dictionary entspricht einem Array mit einem Key-Value Prinzip.
[9] Eine generische Klasse ist in dem Fall ein struct mit einem Datentypen in sich.

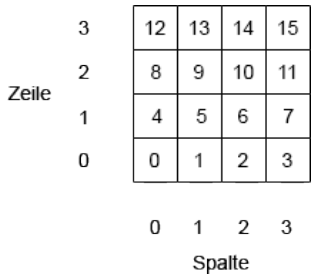

Abbildung 3.3: Zweidimensionaler 4x4 Array2D

3.2.3 Observable

Da in Swift ein Observer von der Klasse NSObject erben muss und die Umsetzung aufwendig ist, wurde der Algorithmus von Niels de Hoog (2014) gewählt.

Mit dieser generischen Klasse wird beim Aufruf ein Objekt vom Typ **Observable** erzeugt. Diese trägt das zu observierende Objekt/Variable o.ä. in sich. Sobald eine andere Klasse dieses Objekt beobachten soll, muss diese das Objekt *abonnieren*. Wenn nun eine Änderung im beobachteten Objekt stattfindet, wird die gewünschte Methode, welche ausgewählt wurde, aufgerufen.

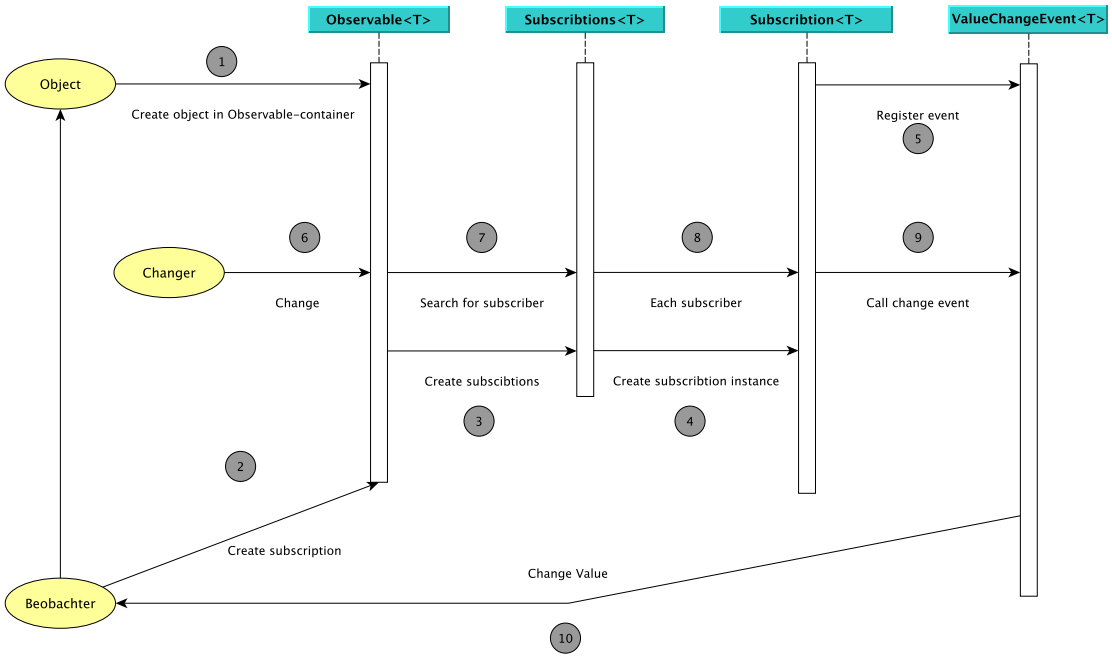

Abbildung 3.4: Observable Sequenz

1. Im ersten Schritt erstellt der Beobachter ein Objekt, welches im Observable struct gehalten wird.

2. Als Nächstes kann der Beobachter eine Funktion aufrufen, um das Objekt zu beobachten.
3. Da mehrere Beobachter das Objekt beobachten können, werden diese in einen Container eingefügt. Dieser enthält alle Beobachter für dieses Objekt.
4. Für jeden Beobachter wird im Container eine Instanz für den Beobachter erstellt.
5. Jeder Beobachter weist der Instanz eine Funktion zu, die aufgerufen wird, sobald sich das Objekt ändert.
6. Die Funktion wird als Referenz im **ValueChangeEvent** abgespeichert.
7. Ändert eine Funktion nun das Objekt, muss der Schritt über das Observable laufen. Diese durchsucht alle Beobachter.
8. Bei jedem Beobachter wird die dazugehörige Funktion herausgesucht.
9. Das referenzierte Event zum Beobachter wird nun ausgeführt.
10. Im letzten Schritt ruft die Referenz die eigentliche Funktion vom Beobachter auf, um auf die Änderung zu reagieren.

Dabei achtet das Script darauf, dass wirklich eine Änderung stattgefunden hat, d.h. bevor eine Benachrichtigung stattfindet, prüft es, ob der Wert sich verändert hat. Wenn dies stattfindet, übermittelt das Script den alten und neuen Wert an die referenzierte Methode, die bei Änderungen reagieren soll.

4 Entwurfsmuster

4.1 Observer Pattern

Das Observer Pattern (zu Deutsch: Beobachter-Muster) ist ein Entwurfsmuster in der Kategorie der Verhaltensmuster. Das sind Muster, die auf eine flexible Weise Änderungen austauschen [Wikipedia Beobachter (Entwurfsmuster)]. Das Prinzip der Observer-Pattern besteht darin, Objektinformationen von beobachteten Objekten an ihre Beobachter zu übermitteln.

Um ein Verständnis für dieses Pattern zu bekommen, sollte man ein reales Beispiel betrachten:

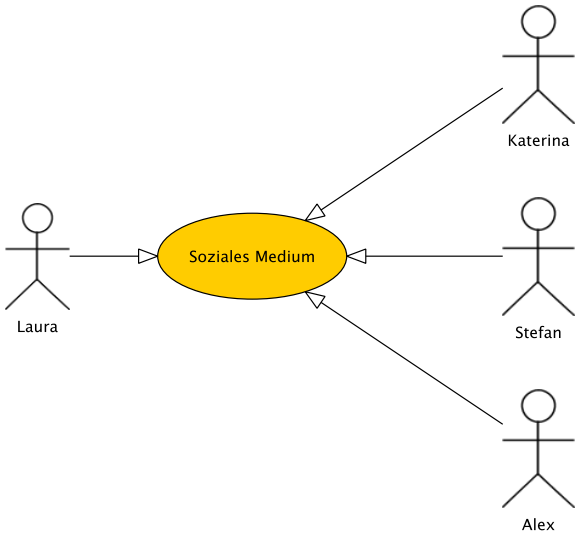

Abbildung 4.1: Beispiel ohne Observer Pattern anhand eines sozialen Mediums

Angenommen vier Freunde haben einen Account auf einem sozialen Medium [Abb. 4.1]. Jede Person hat die Möglichkeit eine Mitteilung zu veröffentlichen und erwartet, dass die anderen drei Freunde diese lesen. In dem Fall werden die fiktiven Personen Laura, Katerina, Stefan, Alex genannt. Nun hat Laura eine Nachricht veröffentlicht und erwartet Feedback von ihren Freunden. Ohne Observer-Pattern müsste jeder Freund von Laura ihr Profil besuchen und nachschauen, ob es Neuigkeiten bei ihr gibt. Dadurch könnten einige Mitteilungen untergehen oder niemals gelesen werden.

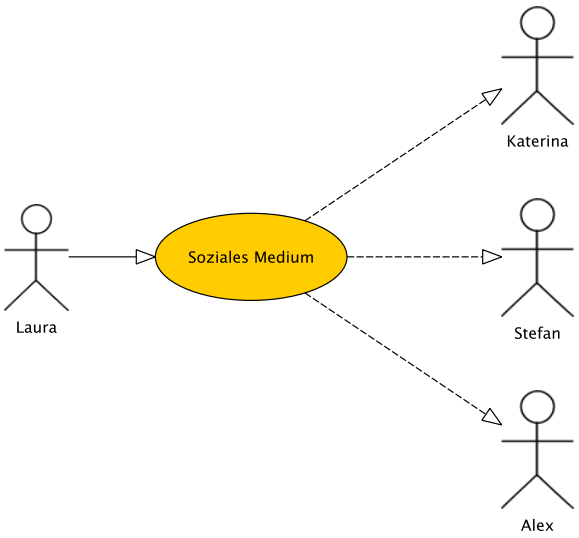

Abbildung 4.2: Beispiel mit Observer Pattern anhand eines sozialen Mediums

Im Fall eines mit Observer-Pattern ausgestattetem sozialem Medium, müssten die Freunde das Profil von Laura nicht besuchen. Wenn die Freunde die Nachrichten von Laura abonniert haben, werden die Freunde von Laura nicht gezwungen sein, ihr Profil zu besuchen. Denn das Observer-Pattern würde die Freunde über jede neue Mitteilung von Laura informieren oder anzeigen und das ohne Interaktion der Freunde von Laura. Hierbei sind demnach Lauras Nachrichten die **Observable** und die Freunde **Observer**.

4.1.1 Vorteile

Jedes beobachtete Objekt ist unabhängig von allen Beobachtern. Das hat den Vorteil, dass eine Änderung in einem beobachteten Objekt von mehreren Beobachtern übernommen werden kann, ohne dass diese angefragt werden müssen. Auch ein Beobachter kann mehrere Objekte beobachten und über deren Änderung informiert werden.

Alle beobachteten Objekte, müssen nicht an jeden Beobachter angepasst werden, denn was mit den Daten geschieht, entscheidet der Beobachter. Im Beispiel 4.2 können Lauras Freunde selbst entscheiden, ob sie die Mitteilungen von Laura kommentieren oder nicht. Dies sind nur einige Vorteile von dem Observer-Pattern.

4.1.2 Nachteile

Wenn mehrere Beobachter ein Objekt beobachten, kann dies zu Folge haben, dass Änderungen übergeben werden, die von einigen Beobachtern nicht benötigt werden. Des Weiteren werden die Beobachter über Änderungen informiert, jedoch nicht um

welche Änderungen es sich handelt. Wenn ein Beobachter A ein Objekt A mit einem String-Wert beobachtet und Beobachter B das selbe Objekt beobachtet, mit dem Unterschied, dass dieser einen Integer-Wert benötigt, kann es zu Laufzeitfehlern kommen.

Ein anderes Szenario wäre, dass der Beobachter zum Zeitpunkt einer Benachrichtigung das selbe Objekt ändert. Durch das Ändern würde der Beobachter über seine eigene Änderung informiert werden. So ein Aufruf kann eine Endlosschleife erzeugen[Wikipedia Beobachter (Entwurfsmuster)].

In Drawix hat das Observer-Pattern eine bestimmte Aufgabe. Da das Spielfeld aus mehreren Teilen bestehen wird, die von dem Controller beobachtet werden und eine Änderung in einem Teil des Spielfeldes stattfindet, könnte der Controller darauf reagieren und die View updaten. Das könnte den Vorteil haben, dass nicht nach jedem Zug (Berühren des Spielfeldes) das gesamte Spielfeld durchgeschaut werden muss. Aber auch in anderen Teilen des Programms ist das Observer-Pattern hilfreich.

4.2 Singleton Pattern

Ein Singleton-Pattern (zu Deutsch: Einzelstück-Muster) ist eine Art von Entwurfsmuster wo sichergestellt wird, dass von einer Klasse eine Instanz existiert, d.h. es gibt nur ein Objekt dieser Klasse [Wikipedia Singleton (Entwurfsmuster)]. Wenn ein Objekt im Normalfall instanziiert wird, erhält dieses immer eine eigene Instanz. Das kann zur Folge haben, dass Änderungen in Instanz A keinen Einfluss auf Instanz B haben, obwohl es im Zweifel gewünscht oder erwartet wird. Dies wird relevant wenn mehrere Objekte das Objekt gleichzeitig bearbeiten sollen und dabei unabhängig voneinander agieren. Wenn es erforderlich ist, dass ein Objekt immer in der selben Instanz bleiben soll, weil andere Objekte oder Services mit den Werten arbeiten, so kann ein Singleton Objekt hierfür benutzt werden [Philipp Hauer (2009)].

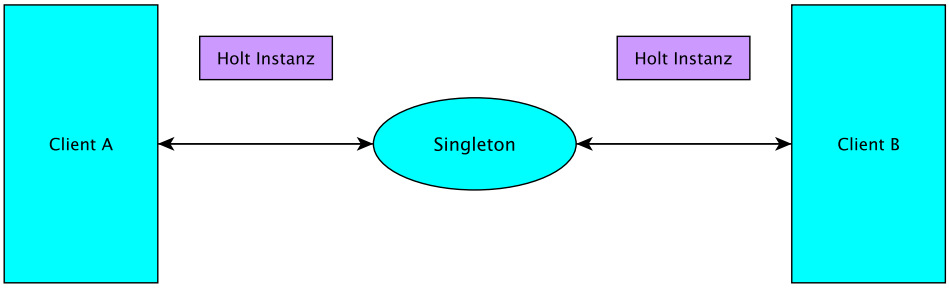

Abbildung 4.3: Veranschaulichung von einem globalen Objekt mit der selben Instanz

In Abb. 4.3 ist zu sehen, dass zwei Objekte (Client A und B) auf ein Singleton zugreifen. Jedes Objekt, welches als Singleton definiert werden soll, enthält eine konstante

Variable, in der die Instanz gespeichert wird. Wenn ein anderes Objekt die Instanz des Singleton benötigt, darf es nicht nochmal instanziiert werden, sondern muss eine Methode aufrufen, welche das Objekt mit der vorhandenen Instanz zurückgibt. Dadurch können Client A und B an den selben Werten arbeiten.

Vorteile

- Ein Objekt kann von einem Singleton erben und als Unterklasse anders aufgebaut werden. Welche Unterklasse hierbei Verwendung findet, kann während des Betriebs definiert werden.
- Ob und wie mit den Werten von einem Singleton gearbeitet werden soll, kann in diesem definiert werden. Diese Zugriffskontrolle unterscheidet ein Singleton von einer globalen Variable [Wikipedia Singleton (Entwurfsmuster)].

Nachteile

- Dieses Entwurfsmuster darf man nur mit großer Vorsicht einsetzen, da die Gefahr besteht, dass ein Äquivalent zu globalen Variablen implementiert wird [Wikipedia Singleton (Entwurfsmuster)].
- Ob ein Singleton immer in der selben Instanz genutzt wird, kann in Swift derzeit nicht sicher gestellt werden. Daher muss die Implementierung sicherstellen, dass nur eine Instanz erzeugt wird.

Anwendung in Drawix

In Drawix findet das Singleton Pattern nur an einer Stelle Anwendung. Als Singleton wird eine Klasse erzeugt, welche als Träger für verschiedene Objekte dient. In dieser Klasse werden nur Objekte und Variablen gespeichert, welche für eine spätere Auswertung benötigt werden.

4.3 MVC

In diesem Kapitel wird das Model-View-Controller[1] Entwurfsmuster (kurz: MVC), vorgestellt. Dabei geht es darum, nur die wesentlichen Aspekte dieses Entwurfsmusters zu nennen, die für die Arbeit relevant sind. 1978/79 wurde von Trygve Reenskaug, einem norwegischen Forscher der Informatik, das Konzept des Entwurfsmusters entwickelt. Die erste Anwendung fand in der Programmiersprache Smalltalk statt [Wikipedia MVC (Entwurfsmuster)]. Die Idee hinter MVC ist, eine saubere Trennung

[1] Fachbegriffe wie „Model“, „View“ und „Controller“ ersetzen in dieser Arbeit die Begriffe aus dem deutschen Modell, Ansicht und Regler.

zwischen den vorhandenen Daten und der Interaktion mit dem Benutzer zu erreichen. Heute kann fast jede Programmiersprache dieses Entwurfsmuster und es ist eines der meist verbreiteten Entwurfsmuster [Bernhard Lahres & Gregor Rayman (2009)]. Wirklich sinnvoll ist die Anwendung von MVC bei mit vielen Views versehener Software, es kann jedoch auch in Anwendungen mit einer View angewendet werden. MVC ist variabel im Aufbau, dadurch kann man das Entwurfsmuster an die gewünschte Anforderung anpassen. In dieser Arbeit werden zwei Varianten vorgestellt, jedoch nur eine bearbeitet. Für weitere Ansätze und Varianten wird die Literatur Joachim Goll (2013) empfohlen.

4.3.1 Model

Das Model ist ein Datenobjekt, welches darstellbar sein muss. Die Verwaltung und das Auslesen der Daten geschieht über die Geschäftslogik. Mit dieser Geschäftslogik können die enthaltenen Daten im Model manipuliert und ausgelesen werden. Meistens werden die Änderungen von einem Model über das Observer-Pattern[4.1] übermittelt.

4.3.2 View

Die View ist die eigentliche Ansicht, welche der Benutzer betrachtet. Dabei wird eine Darstellung der Bedienelemente und Informationselemente realisiert. Die Daten vom Model können auf der View in unterschiedlichen Ansichten angezeigt werden [(Joachim Goll 2013: S. 382)]. Dabei kann es sich sehr wohl um die gleichen Daten handeln [Abb. 4.4].

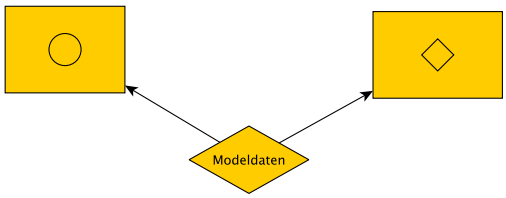

Abbildung 4.4: Selbe Model-Daten in unterschiedlicher Darstellung

4.3.3 Controller

Jegliche mögliche Interaktion in der View wird vom Controller abgefangen. In der Regel übernimmt ein Controller auch die Kommunikation zwischen View und Model. Wenn Logik ausgeführt werden soll, kann sich der Controller an der Geschäftslogik des Models bedienen. Je nach Kommunikation [Abb. 4.5] gibt der Controller, falls nötig, die neuen Daten an die View weiter. Andersherum kann der Controller auch direkt die View manipulieren. Ein Controller kann sowohl mehrere Views als auch Models steuern. Hierbei gibt es keine Beschränkung.

4.3.4 Zwei Kommunikationsarten von MVC-Komponenten

Wie zuvor erwähnt, kann MVC den Anforderungen angepasst werden. In dem Fall werden zwei Konstrukte vorgestellt. Beide verletzen demnach nicht das MVC-Entwurfsmuster. Das erste Entwurfsmuster bedient sich des Observer-Patterns und das zweite eines typischen MVCs.

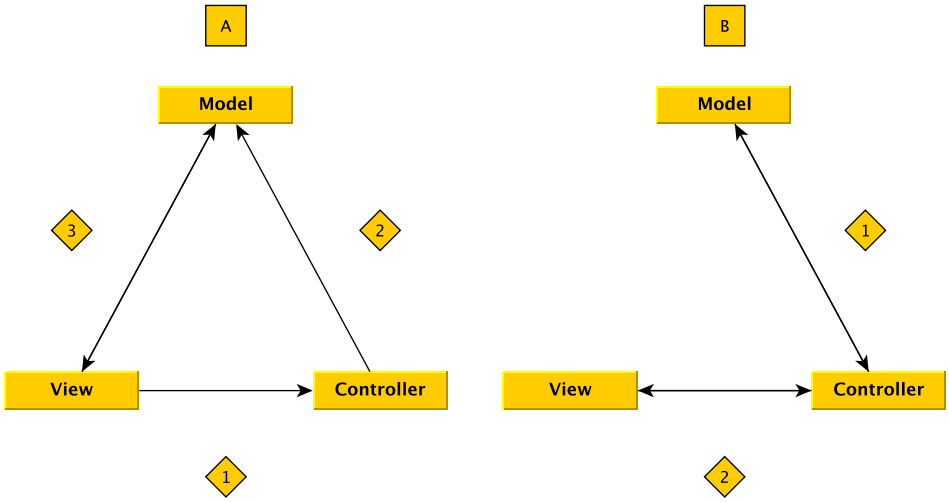

Abbildung 4.5: Zwei Varianten von MVC

A: Wenn der Benutzer eine Aktion auf der View durchführt, sendet die View die Interaktion an den Controller - *Abschnitt: A - 1*. Dieser wendet die Geschäftslogik vom Model an und ändert ggf. die Daten in dem Model - *Abschnitt: A - 2*. Die geänderten Daten werden an die View weitergeleitet und aktualisiert - *Abschnitt: A - 3*. Dabei handelt es sich um ein Model, das aktiv kommunizieren kann, d.h. das Model kommuniziert über einen direkten Weg mit der View [(Joachim Goll 2013: S. 381)]. Die Kommunikation beschränkt sich hierbei nur auf eine Aktualisierung der Daten, die von der View abgebildet werden. In dieser Arbeit wird nicht weiter auf diese Variante eingegangen.

B: Wenn der Benutzer eine Aktion auf der View durchführt, sendet die View die Interaktion an den Controller - *Abschnitt: B- 2*, wie auch in **A**. Dieser wendet die Geschäftslogik vom Model an, um die Daten vom Model zu manipulieren. Das Model übersendet die neuen Daten an den Controller - *Abschnitt: BA - 1*. Wenn eine Änderung stattgefunden hat, übermittelt der Controller die Daten an die View und diese aktualisiert die Ansicht - *Abschnitt: B - 2*. Eine direkte Kommunikation zwischen dem Model und der View existiert nicht. Beide Komponenten wissen nicht von der Existenz des anderen.

4.4 MVCS Paradigma

Das MVCS Paradigma entspricht im Grundgedanken dem MVC [4.3]. In diesem Entwurfsmuster gibt es, im Gegensatz zu MVC, eine zusätzliche Service Ebene. Auch bei MVCS gibt es viele mögliche Kommunikationswege zwischen den einzelnen Komponenten. In dieser Arbeit wird nur eine Komponente behandelt. In einem MVCS-Entwurfsmuster werden die einzelnen MVCS-Komponenen in einem Schichtenparadigma angeordnet [Glossar Hochschule Augsburg (2014)]. Im Rahmen der Arbeit wird versucht, ein MVCS zu entwickeln, welches wie das MVC-Entwurfsmuster modular und in Schichten eingeteilt ist.

4.4.1 Serviceorientierte Architektur

Bevor die einzelnen Module erklärt werden, wird zunächst geklärt, woher die Service-Schicht herkommt. Abgeleitet wird diese von der serviceorientierten Architektur. Hier geht es rein um den Grundgedanken und deren Kommunikationswege. Bei der serviceorientierten Architektur kann ein Service als Zwischenschicht eingebaut werden, um die Kommunikation zwischen verschiedener Software zu übernehmen [Service-Oriented Architecture (SOA)]. Ein Beispiel:

Abbildung 4.6: Beispiel Service

Angenommen Software A und Software B stammen von verschiedenen Herstellern und die Kommunikation zwischen diesen ist auf dem direkten Wege nicht möglich, da Software A nicht verstehen kann, was Software B übermitteln möchte.

Software A übergibt Daten an Service A und dieser übermittelt die Daten an Service B. Daraufhin übergibt Service B die Daten in übersetzter Form an Software B. Das gleiche funktioniert auch andersherum, falls gewünscht. Jegliche Kommunikation zwischen der Software A und B findet über die Services statt. Auch jede Anfrage von der Software muss über den Service erfolgen. Dabei stellt Service A dem Service B, oder umgekehrt, die Anfrage welche Informationen benötigt werden und diese werden von Service B angefragt und an Service A übermittelt. Theoretisch könnte diese

Kommunikation auch über einen einzigen Service geregelt werden, jedoch ist für das hier vorgestellte MVCS eher die Kommunikation zwischen zwei Services oder mehr relevant.

4.4.2 VCSM

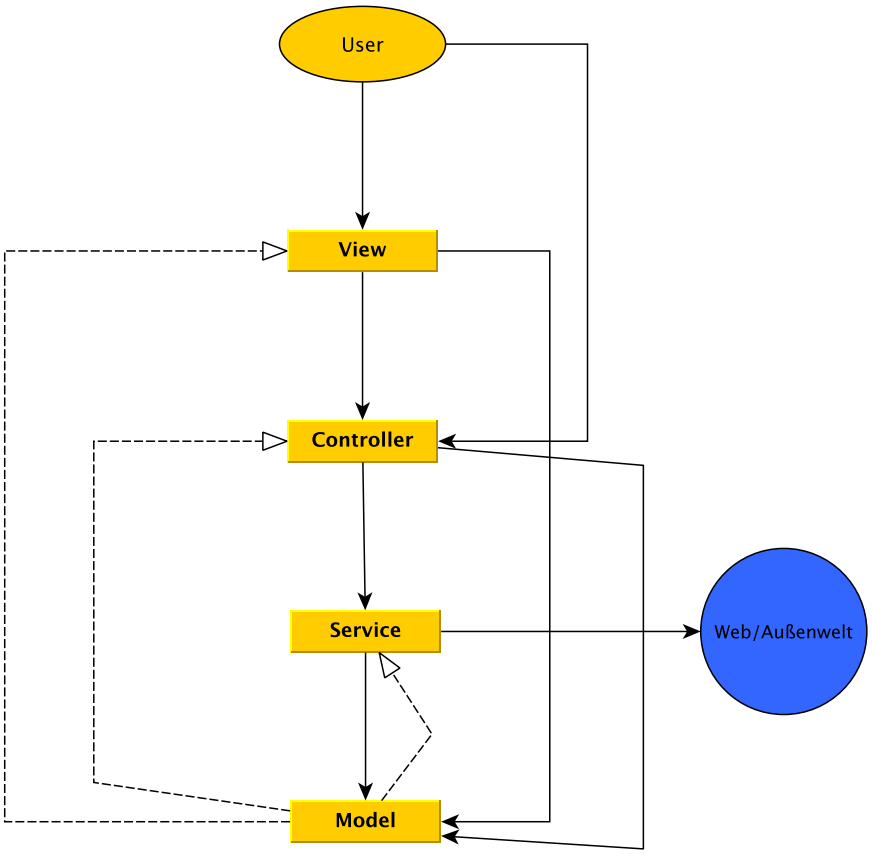

Abbildung 4.7: Beispiel für MVCS als Schichtenarchitektur

Der Aufbau in dieser Variante des MVCS-Entwurfsmusters (in diesem Fall VCSM wegen der Anordnung) entspricht verschiedener Schichten. Bei diesen Schichten können die höheren an die darunter liegenden Schichten zwar Anfragen stellen, jedoch keine Anfragen erhalten. Um trotz alledem eine rückwirkende Kommunikation zu bieten, bedient man sich hier des Observer-Patterns.

Das Konzept vom VCSM von Glossar Hochschule Augsburg (2014) [Abb. 4.7] entspricht einem Entwurfsmuster, welches in unterschiedliche Module unterteilt ist. Dabei können nur die höheren mit den darunter liegenden Schichten kommunizieren und von den unteren Schichten Antworten erhalten. Des Weiteren können die höheren Ebenen mit mehreren der darunter liegenden kommunizieren. Das Service Modul

dient als Kommunikationspunkt zwischen Datenbanken, Webservices etc. Die Logik könnte in dem Fall weiterhin im Model bzw. im Controller sein, wie im Beispiel 4.3.

Das Ziel für Drawix ist jedoch ein Konzept zu entwickeln, bei dem die höheren Schichten nur mit einem direkt darunter liegendem Modul kommunizieren können. Das würde bedeuten, dass die Module nur über die Existenz des darunter liegenden Moduls Bescheid wissen. Die darüber liegenden Module werden nur für den Moment gemerkt.

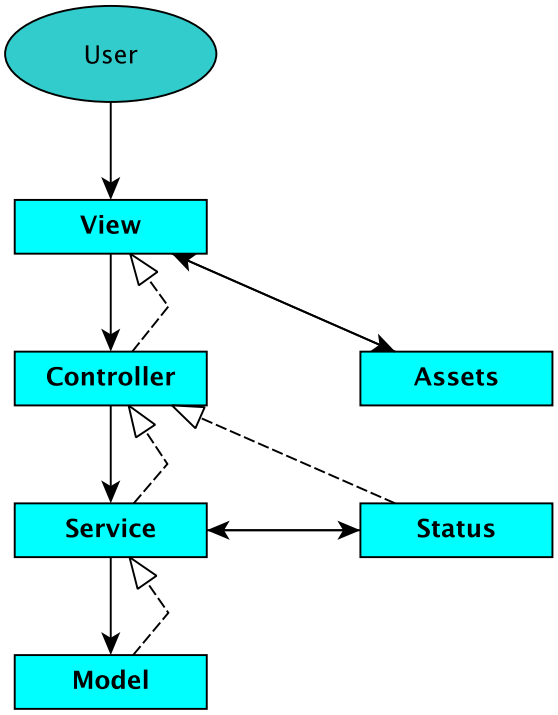

Abbildung 4.8: Aufbau vom MVCS-Entwurfsmuster für Drawix

In der Abb. 4.8 ist im Gegensatz zur Abb. 4.7 ein Kommunikationsweg von oben nach unten dargestellt. Der Benutzer kann nur mit der View interagieren. Sobald eine gültige Interaktion stattfindet, entscheidet der Controller, welcher Service nötig ist und initialisiert diesen. Ein Kommunikationsweg zu den oberen Schichten existiert zwar, jedoch nur für die Antwort an den Anfragenden. Eine Antwort vom Service zum Controller beschränkt sich auf einen Rückgabewert, z.B. einen Boolean, vom Service.

4.4.3 Status-Klasse

Die Status-Klasse kann eine Klasse für globale Objekte sein. Diese wird als Singleton [4.2] implementiert und enthält alle Objekte, die global sein könnten oder zum späteren Zeitpunkt benötigt werden. Trotz des Nachteils, dass ein Äquivalent zu globalen

Tabelle 4.1: Diese Tabelle liefert einen genauen Überblick, welche Module in welchem Entwurfsmuster mit wem kommunizieren können.

Kommunikationsmöglichkeiten	VCSM	VCSM für Drawix
User	View, Controller	View, Assets
View	Model, Controller	Controller
Controller	Service, Model	Service
Service	Web etc, Model	Model, Status-Klasse, Controller

Variablen entstehen kann, wenn dieses Entwurfsmuster stark genutzt wird, könnte die Nutzung einer solchen Klasse für eine bessere Überschaubarkeit des Programmcodes sorgen bzw. einer besseren Zugriffskontrolle.

4.4.4 Assets

Hinter den Assets hängt in iOS ein Asset-Katalog. Dieser enthält alle Bilder die in einem iOS-Projekt dargestellt werden können. Das hat den Vorteil, dass unabhängig davon welche Auflösung das Gerät hat, immer das richtige Bild dargestellt wird. Im Code muss lediglich das gewünschte Bild bestimmt werden. Dabei handelt es sich um eine Gruppe von Bildern. In dieser Gruppe müssen die Bilder in jeder Auflösung[2] vorhanden sein.

In diesem Verwaltungssystem müssen folgende Bilder hinterlegt werden:

- Bilder, die in der App genutzt werden
- Programm-Icon
- Startbild (optional)
- Sprite-Atlas, z.B. animierte Figuren (optional)

Auf dieses Verwaltungssystem soll im MVCS nur die View Zugriff haben.

4.4.5 Service

Beim Service besteht die Möglichkeit, dass auf Anfrage vom Controller eine Manipulation der Daten stattfindet. Hierbei sind die Daten in der Status-Klasse gespeichert. Der Controller benötigt hierbei keine Verbindung zu der Status-Klasse. Daten die von der View dargestellt werden, kann man mit Hilfe des Observer-Pattern [4.1] aktualisieren. Der Controller würde in diesem Fall der Beobachter der Daten sein, die von

[2] Je nachdem welche Geräte das Programm unterstützen soll, einfacher, zweifacher und dreifacher Auflösung.

der View dargestellt werden. Wie in MVC [Abb. 4.5 - B] würden Aktualisierungen der View vom Controller übernommen werden. Daher müsste der Service dem Controller nicht antworten, sondern die gewünschte Änderung in der Status-Klasse umsetzen.

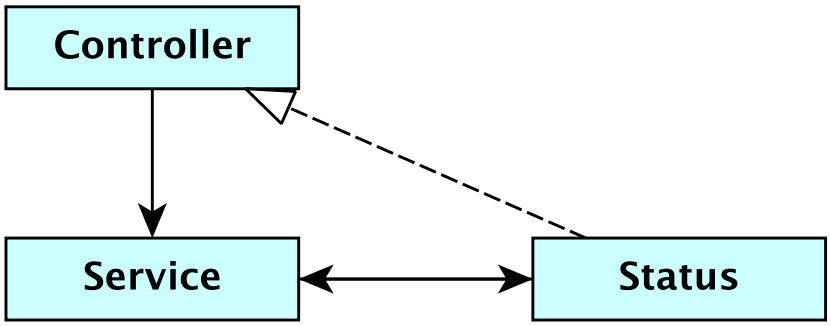

Abbildung 4.9: Controller - Service - Status-Klasse Kommunikation

In diesem Konzept gibt es für jedes Model einen zuständigen Service. Vergleichbar mit Abb. 4.6 würde jedes Model (Software in Abb. 4.6) über einen zuständigen Service verfügen. Der Unterschied zu der serviceorientierten Architektur 4.4.1 ist, dass das Model keine Anfragen an den zuständigen Service übersendet.

Bei vielen Services, kann die Kommunikation untereinander unabhängig vom Controller stattfinden. Jeder Service hat die Möglichkeit, einen anderen zu kontaktieren. Dabei muss nicht unbedingt eine Rückmeldung von Nöten sein. Wie auch in Abb. 4.9 sollten die Services Daten in der Status-Klasse ändern. Bei so einem Konzept kann eine Anfrage vom Controller zu einer Kette von Aufrufen führen [Abb. 4.10].

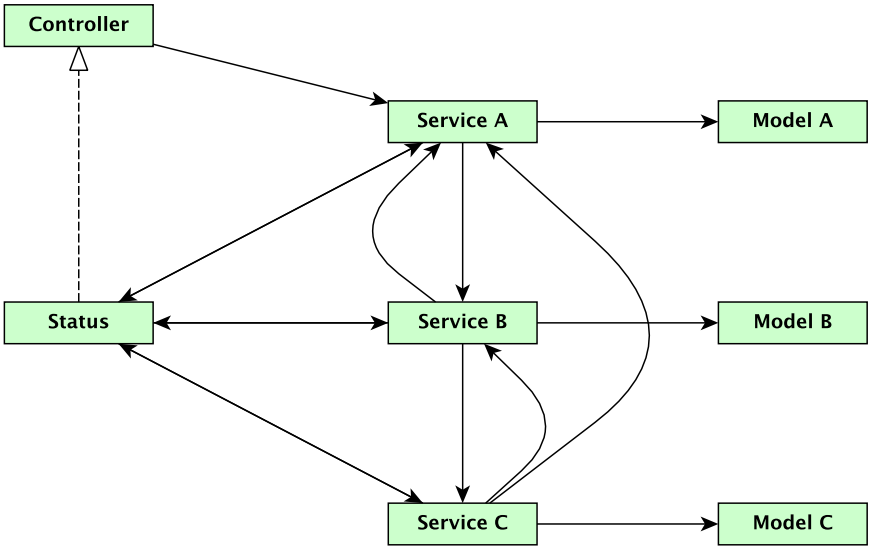

Abbildung 4.10: Kommunikation zwischen Services

Andererseits kann es sein, dass der Service andere Services nicht benötigt und die Änderung direkt in der Status-Klasse angepasst wird.

4.4.6 Unterschied zu MVC

- Bei der **View** gibt es keinen großen Unterschied zu MVC [Abb. 4.5 - B], außer dass die View die Assets holt.
- Der **Controller** hat im Gegensatz zum MVC nur den Kommunikationsweg zum Service. Jegliche Datenänderung übernimmt der Controller über die Status-Klasse. Nur bei Bedarf kann der Service einen Rückgabewert übermitteln, welchen der Controller benötigt, z.B. ein Boolean um eine bestimmte Bedingung zu erfüllen.
- Im Gegensatz zu MVC [4.3] enthält der **Service** die Geschäftslogik vom **Model**, da das Model in diesem Konzept ein reines Datenobjekt sein soll.
- Das Model enthält keine Logik und bildet im MVCS nur ein Konstrukt eines Datenobjekts. Vergleichbar ist das Model mit einer Blaupause.

5 Konzeption

5.1 Spielprinzip

Das Spielprinzip orientiert sich an Tetris und anderen Puzzle-Spielen auf mobilen Endgeräten (siehe: 2.1).

Dabei soll der Aufbau ähnlich wie bei 1010![1] sein. Es werden drei Steine vorgegeben, die auf einem Spielfeld von 9 x 9 Feldern[2] durch das Malen platziert werden. Die Steine werden „frei“ per Berührungsgeste eingezeichnet und verfügen über verschiedene Farben. Sobald eine Reihe oder Spalte auf dem Spielfeld vollständig befüllt ist, werden diese Reihen und/oder Spalten[3] zurückgesetzt. Das Hineinmalen eines Steines verursacht ein Event, bei dem der eingezeichnete von den drei vorgegebenen Steinen verschwindet und durch einen neuen ersetzt wird. Jedes Hineinmalen eines Steines als auch das Verschwinden einer Reihe und/oder Spalte erhöht die Punktzahl des Spielers.

Im Prototyp sind zwei Varianten geplant. In einem kann der Spieler solange spielen bis keiner der drei vorgegebenen Steine hineinpasst, was einem Modus ohne Zeitbeschränkung entspricht. In dem anderen wird ein Countdown herunter gezählt. Das Spiel ist vorbei, sobald kein vorgegebener Stein passt oder wenn der Countdown bei null angelangt ist. Das Hauptziel in dem Spiel ist das Erreichen einer möglichst hohen Punktzahl.

5.2 Tetromino

Im Prototypen werden die Steine den Tetrominos[4] entsprechen. Steine, die aus Quadraten zusammengebaut werden, heißen Polyminos. Je nach Anzahl der Steine besitzen diese andere Namen. Bei den Tetrominos besteht jeder Stein aus genau vier Quadraten. Wenn man die vier Steine zusammenhängend auf einem Feld von 2x4 anordnet können nur sieben Variationen entstehen [Polyminos (2014)]. Diese Tetrominos wurden für das Spiel Drawix hauptsächlich wegen ihrer Bekanntheit und Beliebtheit gewählt.

[1] Siehe : 2.1.2
[2] Im Prototypen sind nur 9 x 9 Felder definiert
[3] Es kann sowohl eine Reihe als auch eine Spalte gleichzeitig befüllt sein.
[4] Name kommt von der Anzahl der Steine (tetra = vier römisch).

Ein Beispiel, wie ein Tetromino aussehen soll und welchen Regeln dieser entsprechen soll sieht man in dieser Grafik Abb. 5.1. Der rote Stein entspricht nicht den Regeln von den Tetromino. Grund hierfür ist, dass eine Lücke zwischen dem oberen und dem unteren Teil der 2x4-Anordnung ist. Bei dem grünen Stein hingegen wurde eine richtige Konstruktion angewendet. Demnach müssen die Steine zusammenhängend sein.

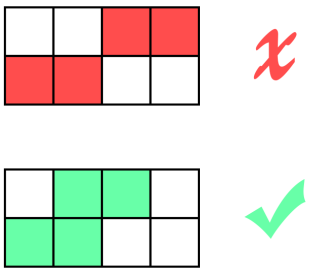

Abbildung 5.1: Falsche und richtige Anordnung

Wenn man dieser Regel folgt, ergeben sich folgende Steine:

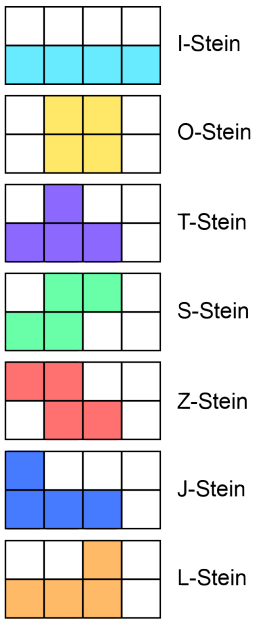

Abbildung 5.2: Alle möglichen Tetrominos

Diese können jedoch in jeder möglichen Drehung gezeichnet werden. Das ergibt eine Anzahl an 19 verschiedenen Variationen. Angenommen es werden alle Steine in eine

4*4-Matrix überführt. Hierbei werden alle 19 Variationen der Steine in diese Matrix hineinpassen. Dies wird benötigt, wenn ermittelt werden muss, um welchen Stein es sich handelt, den der Spieler hinein gemalt hat.

5.3 Gestaltung

Bei der Gestaltung von Drawix wurde darauf geachtet, dass es schlicht und übersichtlich ist. Dieses Mock-Up ist kein finales Ergebnis, es dient hauptsächlich nur zur Orientierung für die Entwicklungsphase. Es ist sinnvoll die Funktionen zu implementieren und diese bei einem endgültigen Design mit Grafiken zu versehen. Insgesamt hat das Unternehmen nodapo zur Veröffentlichung vier unterschiedliche Screens geplant. Im weiteren Verlauf werden alle Screens des Designs gezeigt, jedoch werden nicht alle in der Arbeit implementiert.

Der Startbildschirm soll dem Benutzer folgende Möglichkeiten zur Auswahl bieten:

1. Spielstart für den Modus mit einer Zeitbeschränkung
2. Spielstart für den Modus ohne Zeitbeschränkung
3. Eine Verknüpfung zu der Rangliste
4. Eine Verknüpfung zum Shop
5. Einen Button zum ein- und ausstellen von der Musik bzw. Geräuscheffekten
6. Einen Button für eine Erklärung vom Spiel
7. Einen Button für den Nachtmodus[5].

Des Weiteren befinden sich dort reine Anzeigeobjekte wie das Logo und der aktuell beste Punktestand. Von allen aufgelisteten Punkten sind für den Prototypen Punkt eins und zwei relevant.
Im eigentlichen Spiel befinden sich die vorgegebenen Steine, das Spielfeld, die übrige Zeit, der Pause-Button, der aktuelle Punktestand und die Werbung[6]. Um einen direkten Blick auf die Zeit und den aktuellen Punktestand, sowie die vorgegeben Steine zu haben, werden diese oben ausgerichtet. Wenn diese unter dem Spielfeld angeordnet ist, könnte es dazu führen, dass diese Objekte von der Hand des Spielers verdeckt werden. Das könnte dazu führen, dass eine schnelle Auswahl des nächsten Steins verhindert wird. Unterhalb der Anzeige-Elemente wird sich das eigentliche Spielfeld befinden. Dies soll möglichst den gesamten Bildschirm füllen.

[5]Färbt den Hintergrund schwarz, was bei Dunkelheit angenehmer wirkt.
[6]Werbung soll zukünftig der Finanzierung des Spiels dienen.

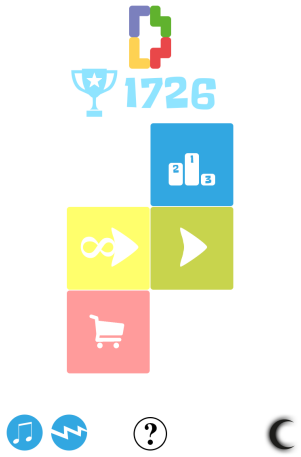

Abbildung 5.3: Startbildschirm in Drawix

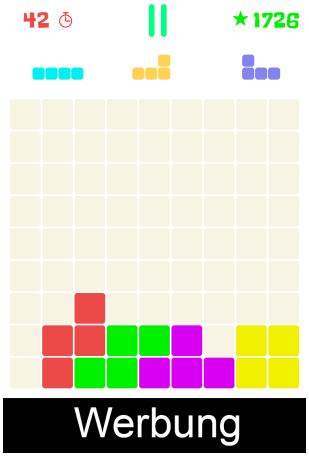

Abbildung 5.4: Eigentliches Spiel in Drawix

Wenn der Spieler auf den Pause-Button drückt, soll ein Overlay[7] erscheinen, der das gesamte Spiel verdeckt, da die Pause andernfalls zum Schummeln genutzt werden könnte. In der Pause kann der Spieler das Spiel neu starten, fortführen, auf den Startbildschirm wechseln oder in den Shop[8] gelangen. Außerdem soll von hier aus der Sound ein- und ausgeschaltet werden können.

[7] Ein darüber gelegte Ebene.
[8] Später für In-App-Käufe, z.B. das Freikaufen von Werbung.

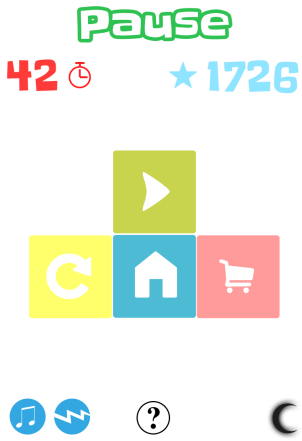

Abbildung 5.5: Pause Bildschirm in Drawix

Beim Verlieren soll ein *Game Over*-Bildschirm angezeigt werden. Dieser enthält sowohl den Button für den Neustart des Spieles, einen Button für die Highscore-Liste sowie einen Facebook-Button. Des Weiteren sieht der Benutzer seine letzte Punktzahl. Wenn diese einem neuem Rekord entspricht, wird der Pokal angezeigt, ansonsten wie im Pause-Bildschirm ein Stern.

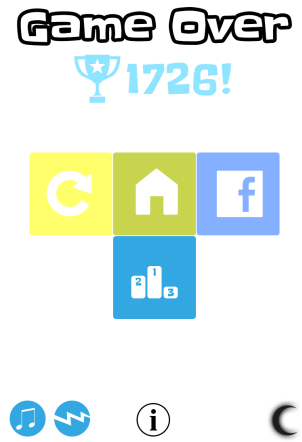

Abbildung 5.6: Game Over in Drawix

5.4 Anwendungsarchitektur

5.4.1 Erklärung des Diagramms

Um ein grobes Bild von der Architektur zu erhalten, wird ein UML-Diagramm benötigt. Dabei wird auf dem MVCS 4.8 aufgebaut. Wird eine Methode/Property mit

(-) nicht markiert, handelt es sich hierbei um eine public Methode/Property.

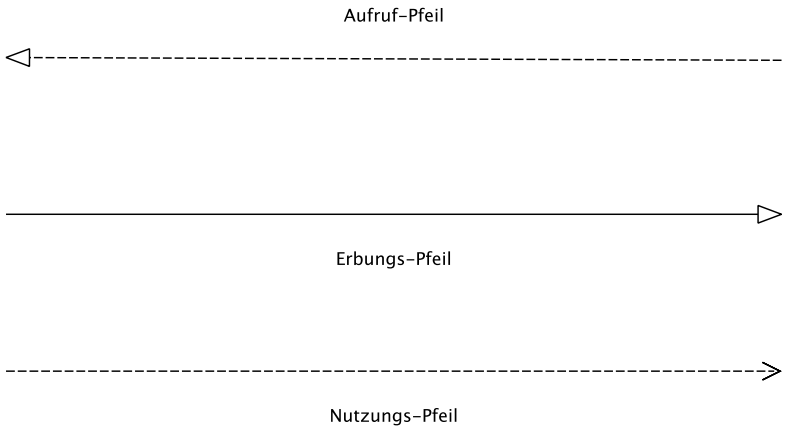

Abbildung 5.7: Pfeillegende für die verwendeten Pfeile.

Wofür welcher Pfeil im Detail steht, soll diese Grafik verdeutlichen.

- Aufruf-Pfeil: Bedeutet, dass die Klasse, auf welche der Pfeil zeigt, initialisiert wird.
- Erbungs-Pfeil: Soll erklären, dass von der Klasse, auf welche der Pfeil zeigt geerbt wird.
- Nutzungs-Pfeil: Soll verdeutlichen, dass diese Klasse von der darauf zeigenden Klasse genutzt wird.

5.4.2 Model Entwurf

Im Prototypen sind nur drei Model verfügbar. Das Spielfeld ist ein Model, welches

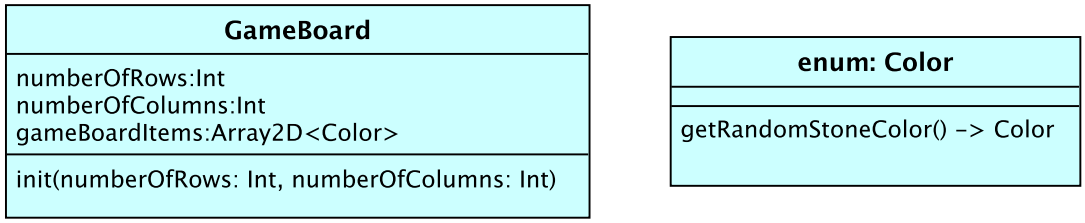

Abbildung 5.8: Drawix Model Entwurf - Farbe und Spielfeld.

die Anzahl der Zeilen und Spalten in sich trägt. Außerdem erstellt das Spielfeld ein Array2D mit dem Inhalt Farbe. Dies ist das eigentliche Spielfeld. Die Farben sind in

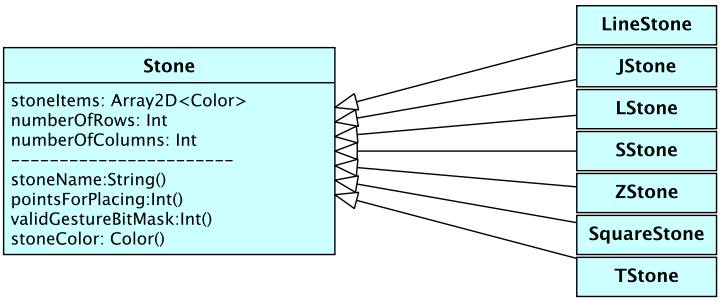

Abbildung 5.9: Drawix Model Entwurf - Steine.

einem Enumerator. In diesem werden alle Farben für die Steine festgelegt. Mit einer Funktion können zufällige Farben für die Steine ermittelt werden. Das **Stein**-Model besteht aus einer abstrakten Klasse. Jeder andere Stein erbt von dieser Superklasse. Dabei werden sieben Steine benötigt [5.2]. Die Properties unterhalb der gestrichelten Linie sind jene, welche von jedem Stein überschrieben werden müssen. Das **Color**-Model hält eine reine Auflistung von Farbvarianten. Diese Farben werden als Wert genommen, damit der dazugehörige Service das richtige Bild für die View heraussucht.

5.4.3 Controller Entwurf

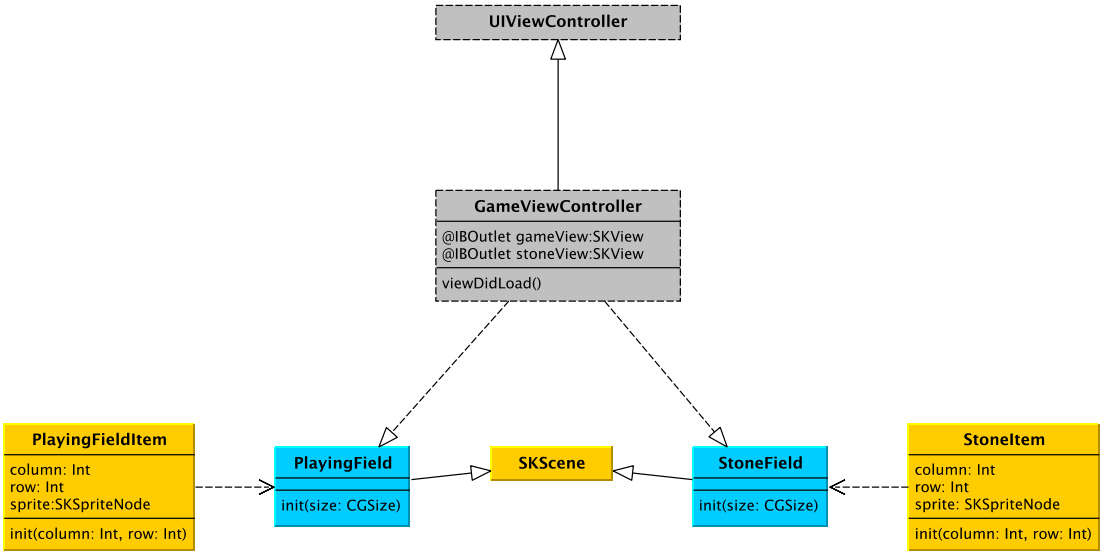

Abbildung 5.10: Drawix Controller Entwurf.

Jedes iOS Projekt wird mit einem ViewController definiert. Dieser ist für die Steuerung der View zuständig. Außerdem entspricht dieser dem Controller aus 4.3. Der

GameViewController erbt von der Klasse UIViewController. **UIViewController** ist eine Klasse, die das Fundament für die View-Steuerung bildet. Sobald von dieser Klasse geerbt wird, ermöglicht es dem Entwickler, eigene Verhaltensweisen zu deklarieren. Damit eine View gesteuert wird, muss in einem iOS-Projekt in dem UIViewController definiert werden, welche View gesteuert werden soll.

Für Drawix müssen aufgrund des Aufbaus aus 3.1 Szenen deklariert werden, welche von der Klasse SKScene erben. Wie in 3.1 kümmern diese sich um Teilbereiche der View. Präsentiert werden sie von dem **GameViewController**. In den Szenen werden sowohl die dargestellten Texturen als auch Animationen implementiert. Da nur der **GameViewController** mit der View kommuniziert, informieren die Szenen den GameViewController darüber, was sich auf der View ändern soll. Damit auf dem **StoneField** alle drei Steine abgebildet werden, holt sich die Szene die vorgebenen Steine aus einem Array und fügt die notwendige Textur hinzu. Das **PlayingField** macht das gleiche für das Spielfeld. Dabei zieht sich die Szene die Informationen aus dem **GameBoard**-Model.

StoneItem ist die Model-Klasse, die sich um die Darstellung der Steine kümmert [5.3] und **PlayingFieldItem** um die Darstellung im Spielfeld. Beide Klassen sind nur einzelne Bestandteile vom Spielfeld. Um diese richtig zu platzieren, werden die Eigenschaften *column* und *row* vom Typ Integer erstellt. Grund für diese zwei Models ist eine Schichtentrennung wie in 4.8 beschrieben.

5.4.4 Service-Entwurf

Die Services sind alle unabhängig voneinander und komplett eigenständig. Jedes Model ist einem Service zugeordnet. Um die Schichten wie in 4.1 zu trennen, haben die Services auch nur Zugang zu dem zugewiesenen Model. Das hat den Vorteil, dass jeder Service einen bestimmten Aufgabenbereich ausfüllt. In diesem Kapitel werden alle in Drawix vorhandenen Services aufgelistet.

GameBoardService ist für die Änderung am Spielfeld zuständig. Zu den Aufgaben gehören zum einen die Initialisierung des Spielfelds beim Spielstart als auch die Veränderung im Spielgeschehen. Dabei soll der Service das Spielfeld in gewünschter Form von 9x9 Feldern erstellen und das Objekt mit dem *Leer* Status(Color Aufzählung) füllen. Des Weiteren bietet der Service für andere Methoden/Klassen die Getter- und Setter-Methoden an, um Zugriff auf das **GameBoard**-Model zu erhalten. Diese ermöglichen einen Status, sowohl von einem Spielfeldstein abzufragen, als auch zu verändern. Wenn ein Stein gesetzt wird, soll der mithilfe der Informationen über den Stein die Spielfeldsteine im Spielfeld anpassen. Hierbei soll nur die Farbe der einzelnen **PlayingFieldItems** geändert werden (Color Aufzählung). In der Methode *checkGameBoard* soll das Spielfeld auf volle Reihen bzw. Spalten geprüft werden. Wenn welche gefunden worden, werden diese auf den Status *leer* aus dem

Color-Model gesetzt.

GestureService soll die Verwaltung der Geste übernehmen. Dabei bekommt dieser in der Methode *startGesture* die nötigen Koordinaten. Jegliche Weiterführung einer Geste wird mit der *continueGesture* abgefangen. Beim Beenden einer Geste soll *endOfGesture* dafür sorgen, dass eine Prüfung auf eine gültige Geste stattfindet. Ist eine Geste ungültig, muss der Status von jedem betätigten Spielfeldstein zurückgesetzt werden. Hierum soll sich die *resetGesture*-Methode kümmern.

StoneService erstellt beim Spielstart drei initiale Steine. Dabei soll die Initial-Methode dreimal die *createStone()*-Methode aufrufen. Im späteren Verlauf des Spieles wird nach dem Legen eines Steines ein neuer Stein durch die selbe Methode erstellt. Um jedoch immer einen zufälligen Stein zu liefern, wird *getRandomStoneVariant()* aufgerufen. Diese Methode soll eine zufällige Zahl ($\mathbf{Z}_7$) ermitteln und in einem eindimensionalen Array mit allen Steinvarianten als Index verwenden. Gespeichert werden die Steine in einem Array in dem alle drei Steine abgebildet werden. Dieser Array befindet sich in der Status-Klasse, damit die Szene diese anzeigt.

Der **AssetService** fällt zwar durch seine Eigenschaften nicht in die Service-Kategorie, jedoch liegt das nur daran, dass in dem Prototypen die notwendigen Werte[9] vorerst im Service gespeichert wurden. Die grundlegende Aufgabe dieses Services ist, den Namen für das Bild der View zur Verfügung zu stellen. Hierfür gibt es zwei Methoden: *getGameBoardItemSprite* und *getStoneItemSprite*. Beide Methoden nehmen als Parameter den Color-Datentypen. Anhand des Raw-Values, wird derzeit der Index in einem der jeweiligen Arrays, der benötigte String mit dem Namen für das Bild zurückgegeben.

GameBoardValidatorService dient dazu, das Spielfeld daraufhin zu kontrollieren, ob Zeilen und/oder Spalten gefüllt sind und ob kein vorgegebener Stein mehr hineinpasst. Zusätzlich soll dieser Service nach jedem Zug aufgerufen werden und prüfen, ob ein vorgegebener Stein hineinpasst oder nicht. Ist dies nicht der Fall soll das Spiel enden.

ScoreService hat die Aufgabe, Punkte für die Reihen bzw. Spalten zu vergeben und hierfür den gewünschten Faktor zu ermitteln. Denn je nachdem, wie viele Reihen und/oder Spalten gefüllt waren, bekommt der Spieler eine gewisse Punktzahl. Formel

[9] Namen der Bilder

für volle Reihen/Spalten und bei multiplen Reihen/Spalten:

$$p = p + \sum_{n=1}^{a} p_2 \cdot 1,5 | p, p_2 \in \mathbf{Z}, a \in \mathbf{Z}_{16}$$

$a := Anzahl\ der\ doppelten\ Zeilen/SpaltenalsSumme$
$p := Vorhandene\ Punktzahl$
$p_2 := Für\ den\ Zug\ erhaltene\ Punktzahl$[10]

Zudem soll der Service auch Punkte für das Legen eines Steines berechnen. In Zukunft kommen Bonuspunkte hinzu, die z.B. durch schnelles Legen dazugerechnet werden. Die Höhe der Punktzahl für das Legen eines Steines wird in den Subklassen vom Stein-Model definiert.

Im **GestureValidatorService** geht es darum, die getätigte Geste zu validieren. Hierbei werden verschiedene Methoden angewendet, worauf im Kapitel 6.4.1 näher eingegangen wird.

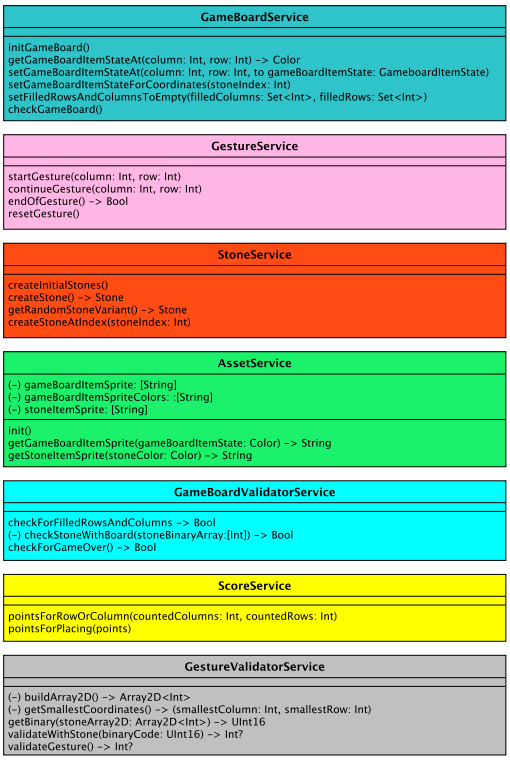

Abbildung 5.11: Drawix Services Entwurf.

[10] Zeile oder Spalte liefert 150 Punkte, Stein liefert derzeit 1 Punkt

6 Umsetzung

6.1 Überblick

Bei der Umsetzung wurde mit dem Projektmanagement-Verfahren Scrum gearbeitet. Bei Scrum werden zunächst alle Anforderungen für das Projekt gesammelt und festgehalten. Hinterher werden die einzelnen Sprints und deren Dauer festgelegt. Beim Projekt Drawix war die Dauer eines Sprints auf zwei, in seltenen Fällen auf drei Wochen festgelegt. Für jeden Sprint werden Anforderungen, die für diesen Sprint benötigt werden, aus dem Produkt-Backlog[1] in den Sprint-Backlog verschoben. Die Anforderungen werden als User-Stories[2] abgelegt. Während eines Sprints werden die Anforderungen für den jeweiligen Sprint nicht geändert, sodass das Team von Entwicklern ungestört daran arbeiten kann. Am Ende eines Sprints wurden stets die umgesetzten Anforderungen vorgestellt. Zudem wird der vergangene Sprint im Ablauf und in seinen Inhalten wiedergegeben. Ziel ist es den Ablauf künftiger Sprints zu verbessern oder die Teamgröße anzupassen. Alle stattgefundenen Sprints wurden in folgender Reihenfolge abgearbeitet:

1. Spielfeld erzeugen und abbilden
2. Steine erzeugen und abbilden
3. Touch-Events ermitteln
4. Geste erkennen und Zeilen und Spalten löschen, wenn voll
5. Game-Over-Status ermitteln und Punktevergabe und zeitbeschränktes Spiel

Nachdem die Implementierung aller Klassen aus 5.4 umgesetzt wurde, wird in diesem Kapitel der Ablauf des Programms erklärt. Hierbei werden die Abläufe vom Spielfeld, der Steine, die zwei Algorithmen für den *Game Over* und für die Gestenerkennung erläutert.

[1] Ein Produkt-Backlog enthält alle Anforderung für das Projekt.
[2] Anforderung aus Sicht des Benutzers.

6.2 Spielfeld

Das Spielfeld wird von den Klassen GameBoard, GameBoardService, PlayingField dem GameState und dem Controller erzeugt.

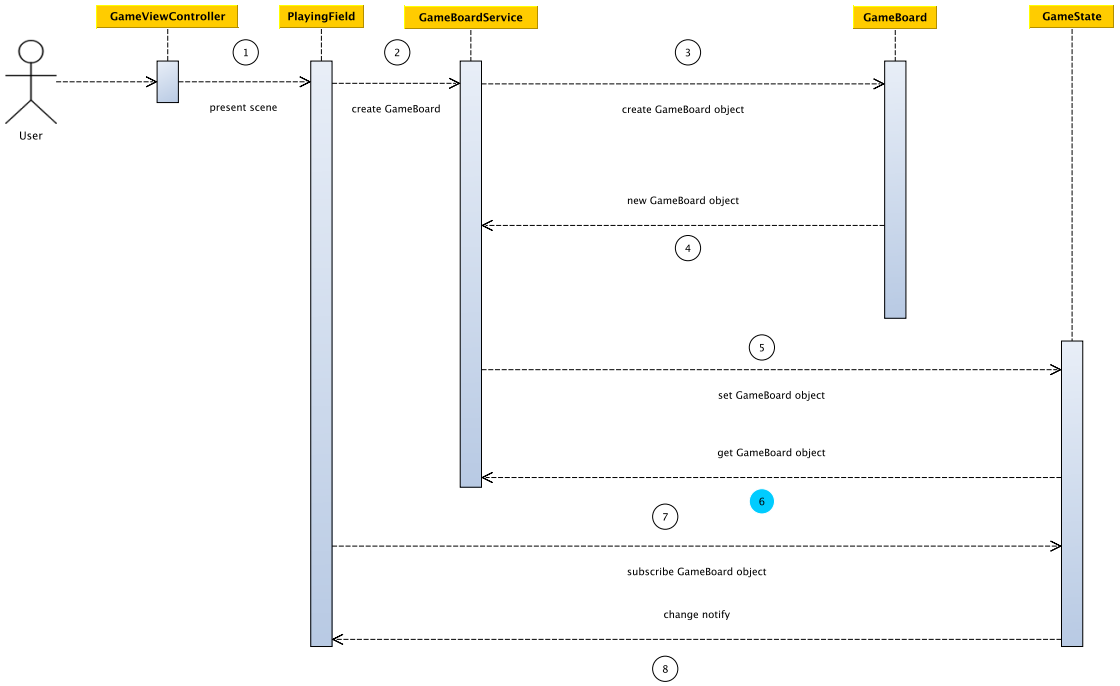

Abbildung 6.1: Sequenz für das Spielfeld.

Wenn das Programm geöffnet ist, initialisiert iOS den GameViewController. Dieser stellt die Szene **PlayingField** dar (*1*). Beim Erzeugen der Szene wird auch das Spielfeld dargestellt. Hierfür ruft die Szene den **GameBoardService** mit der initial Methode auf (*2*). Diese erstellt das Spielfeld anhand des Models **GameBoard** und speichert dieses im GameState (*3,4,5*). Hinterher beobachtet die Szene die einzelnen Elemente vom Array2D, damit bei einer Änderung der Daten auch die View automatisch angepasst wird[4.1](*7,8*). Werden Daten vom Spielfeld in anderen Methoden benötigt, wird der Punkt (*6*) aufgerufen.

Die Abbildung der einzelnen Spielfeld-Elemente erfolgt über eine Methode, die anschließend aufgerufen wird. Diese erstellt für jedes Element im Array2D ein SKSpriteNode-Objekt 3.2.1. Anhand der Zeile und Spalte des jeweiligen Spielfeld-Elements errechnet eine Methode die Position von jedem SKSpriteNode-Objekt.

$$x = column \cdot imageSize + imageSize/2 | column \in \mathbf{Z}_8, imageSize \in \mathbf{R}$$
$$y = row \cdot imageSize + imageSize/2 | column \in \mathbf{Z}_8, imageSize \in \mathbf{R}$$

Durch diese Methode werden alle Objekte nebeneinander gesetzt.

6.3 Steine

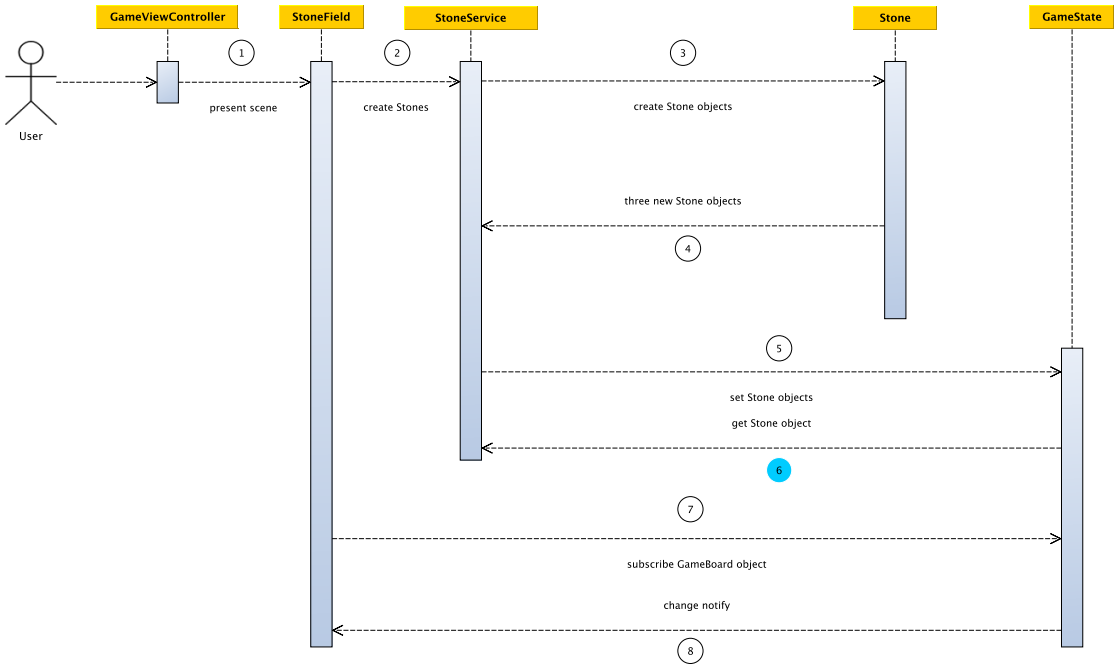

Abbildung 6.2: Sequenz für die Steine.

Die Kommunikation beim Erstellen der Steine entspricht im Groben der vom Spielfeld. Nachdem das Spielfeld erzeugt wurde, werden die Steine erstellt. Dafür präsentiert der **GameViewController** das StoneField(*1*). Diese ruft den **StoneService**(*2*) auf, erstellt initial drei Steine und speichert diese in einem eindimensionalen Array im **GameState** ab(*3,4,5*). Hinterher registriert sich das **StoneField** als Beobachter für die einzelnen Elemente vom Array, um der View die Änderungen zu übermitteln(*9*). Dies geschieht, wenn ein gültiger Stein gesetzt wird, weil der gültige Stein in den vorhandenen Steinen durch einen neuen ersetzt werden muss(*8*). Das Ersetzen eines Steines geschieht, sobald der Benutzer diesen ins Spielfeld gelegt hat. Der Punkt (*6*) wird wie beim Spielfeld nur benötigt, wenn die Information über die vorhandenen Steine im Array erwünscht ist.

Auch hier wird die Methode vom Spielfeld [6.2] zur Positionierung der Bilder verwendet; mit dem Unterschied, dass sich jeder Stein in einem Wrapper-Element[3] befindet. Im Wrapper wird die Funktion aus dem Spielfeld benutzt und für das Positionieren der Wrapper-Elemente wurde eigens eine Funktion entwickelt. Diese ermittelt anhand des Indexes vom Array-Element die Position.

$$x = arrayIndex \cdot imageSize \cdot 6 | arrayIndex \in \mathbf{Z}_2, imageSize \in \mathbf{R}\ y = 0$$

[3]Ein Wrapper verpackt Elemente in sich.

Der Wert für den Wrapper des Steins wird anhand der Bildgröße ermittelt. Dabei wird die Bildgröße mit sechs multipliziert, um einen leichten Abstand zwischen den einzelnen Wrappern zu schaffen[4]. Da eine vertikale Veränderung nicht nötig ist, bleibt der y-Wert immer 0.

6.4 Gesten

Nun sind das Spielfeld und die Steine vorhanden und kann das Spiel gespielt werden. Da der Benutzer die Steine ins Spielfeld einzeichnen soll, müssen die Gesten validiert werden. Wenn der Benutzer eine Geste beginnt, wird über die Szene **PlayingField** ein Event ausgelöst. Dieses Event kümmert sich darum, dass der **GestureService** aufgerufen wird. Dabei werden alle berührten Koordinaten in einem Array im **GameState** gespeichert, jedoch nur von Beginn bis zum Ende der Geste. Bei diesem Array ist eine Grenze von vier Koordinaten festgelegt. Wenn der Array am Ende der Geste keine vier Koordinaten besitzt, wird die Geste ignoriert. Enthält der Array genau vier Koordinaten wird geprüft, welchen Stein die Geste darstellen soll. Ob es sich um einen gültigen Stein handelt, weiß der Service zu dem Zeitpunkt noch nicht.

6.4.1 Gesten-Algorithmus

Sobald der Array vier Koordinaten gespeichert hat und die Geste beendet ist, ruft der **GestureService** den **GestureValidationService** auf. Dieser ermittelt in folgenden Schritten, ob es sich um einen gültigen Stein handelt:

1. Kleinste Spalte und Zeile ermitteln
2. Normalisieren der Koordinaten, durch Subtraktion
3. Füllt einen leeren 4x4-Array
4. Bilde binären Wert (16 Bit)
5. Erstelle Bitmask
6. Vergleiche Bitmask mit gültigen Steinen

Wenn die Koordinaten von der Geste feststehen, müssen diese normalisiert werden, damit ein 4x4-Array gefüllt werden kann[5].

Um die Koordinaten zu normalisieren, wird der kleinste Wert für die Spalte und für die Zeile ermittelt und hinterher von den Originalwerten abgezogen. Mit diesen Koordinaten kann nun ein 4x4-Array gefüllt werden. Hierbei wird für jede Koordinate im 4x4-Array eine 1 gesetzt und für alle anderen eine 0.

[4] Dieser Wert wurde durchs Probieren ermittelt.
[5] Ein 4x4-Array wurde gewählt, da jeder Stein in jeder Ausrichtung in diesen Array hineinpasst.

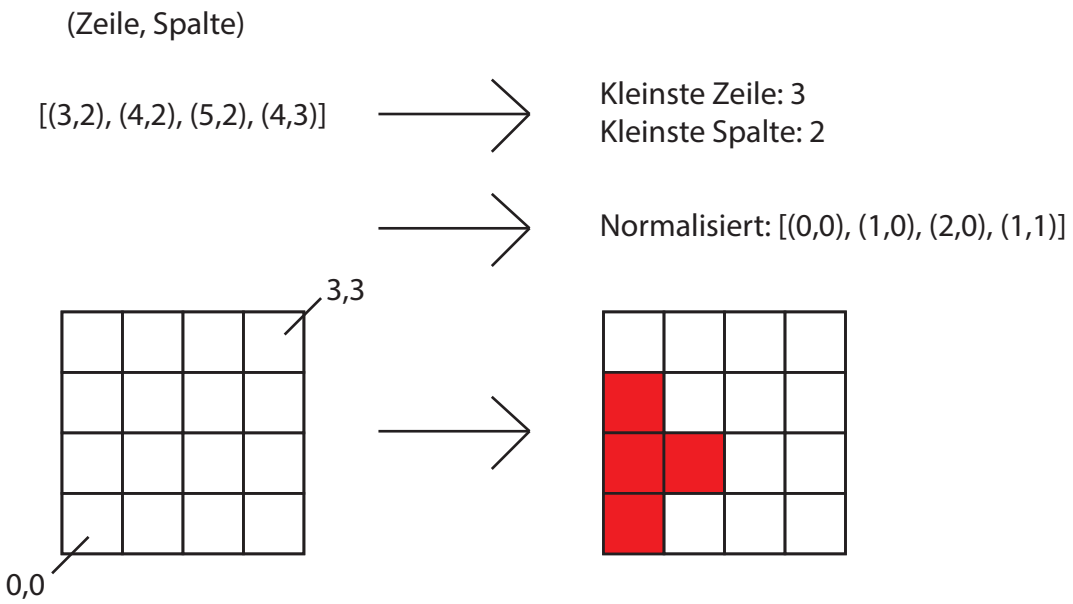

Abbildung 6.3: Stein normalisieren anhand einer Geste für einen T-Stein.

Aus dem fertigen 4x4-Array wird hinterher ein Binärwert gebildet. Dieser wird wiederum in einen Integer-Wert umgewandelt, dies entspricht einer Bitmask. Hierbei wird mit der Koordinate (0,0) begonnen und mit (3,3) beendet. Zudem wird der Binärwert von hinten gefüllt. In der Grafik 6.3 ergibt es einen Binärwert von 0000000100110001 und einen Dezimalwert von 305. Dann wird dieser Wert mit allen vorgegebenen Steinen verglichen. Handelt sich es dabei um einen gültigen Stein, werden die Koordinaten von der Geste mit der entsprechenden Farbe des Steins versehen und das Observer-Pattern kümmert sich darum, das StoneField anzupassen und einen neuen Stein zu generieren.

6.5 Vollständig gefüllte Reihen & Spalten

Hat der Benutzer durch das Legen der Steine eine Zeile oder Spalte vollständig gefüllt, muss diese geleert werden. Ein Algorithmus prüft nach jedem Zug das Spielfeld daraufhin, ob eine Reihe und/oder Spalte gefüllt ist. Dabei läuft eine Schleife einmal das gesamte Spielfeld durch und prüft jede Koordinate auf ihren Status. Enthält diese einen gefüllten Status, unabhängig davon welcher genau, wird in einem Array der Wert an dem Index hochgezählt[6]. Hierfür werden zwei Arrays verwendet, die genau so viele Elemente in sich tragen wie das Spielfeld Zeilen bzw. Spalten hat. Der Index entspricht der Koordinate der Spalte bzw. der Zeile.

[6]Increment, immer Wert+1

Bei jedem Schleifendurchgang wird geprüft, ob der Wert des Array-Elements mit dem jeweiligen Index der Zeilen- bzw. Spaltenanzahl entspricht (im Prototypen neun). Ist es der Fall, muss dass diese Zeile und/oder Spalte komplett gefüllt sein. Da der Index, in dem der Wert neun sich befindet, einer vollen Zeile/Spalte entspricht, wird der Index als Koordinate notiert. Dabei wird unterschieden zwischen den Koordinaten der Reihe und Spalte.

Hinterher kümmert sich eine Methode darum, die vorher ermittelte Spalte und/oder Zeile zu entfernen und dieser einen *leeren* Status zuzuweisen. Nun können diese Koordinaten neu mit Steinen befüllt werden.
Der Vorteil eines solchen Algorithmus ist, dass man mit einem Durchlauf über das Spielfeld herausfinden kann, ob eine Zeile oder Spalte voll ist, was im Zweifel Rechenleistung sparen könnte.

6.6 Game Over

Nun ist das Spiel spielbar, jedoch muss nachdem keiner der vorgegebenen Steine mehr hineinpasst das Spiel vorbei sein. Für den Algorithmus wurde auf der Abtastmethode aufgebaut. Dabei wird das gesamte Spielfeld nach jedem Zug abgetastet. Dieser Vorgang bricht ab, sobald ein Stein passt. Um konsequent jede Koordinate abzutasten, muss ein 4x4 Quadrat über das Spielfeld von Anfang (links unten im Spielfeld) bis zum Ende(rechts oben im Spielfeld) darüber gelegt werden.

Folgende Punkte werden beim auslesen vom Feld behandelt:

1. Aus dem 4x4-Feld eine Bitmask bilden (ähnlich wie beim Gestenalgorithmus)
2. Leeres Feld: 1, gefülltes Feld: 0
3. Bitmask vom Stein und vom Spielfeld über eine UND-Verknüpfung verknüpfen
4. Ergebnis mit der Bitmask vom Stein vergleichen

In Abb. 6.4 wird gezeigt, wie der Algorithmus sich über das Spielfeld bewegt. Sobald bei **Start** begonnen wird, werden die befüllten Werte im 4x4-Qudrat als 1 und die leeren als 0 abgespeichert. Mit diesen Werten wird wie beim Gestenalgorithmus ein Binärwert gebildet. Dann wird der binäre Wert vom Stein mit dem aus dem Spielfeld durch eine binäre UND-Verknüpfung verknüpft. Wenn das Ergebnis dem Ausgangswert des Steins entspricht, handelt es sich hier um eine Möglichkeit den Stein an diese Stelle zu setzen. Andernfalls läuft der Algorithmus weiter. Wenn bei diesem Ablauf das Ergebnis aus 3 [6.6] nicht mit einem Stein übereinstimmt, zieht das Feld eine Spalte weiter. Läuft der Algorithmus durch ohne abzubrechen, wird daraus resultiert, dass das Spiel vorbei ist.

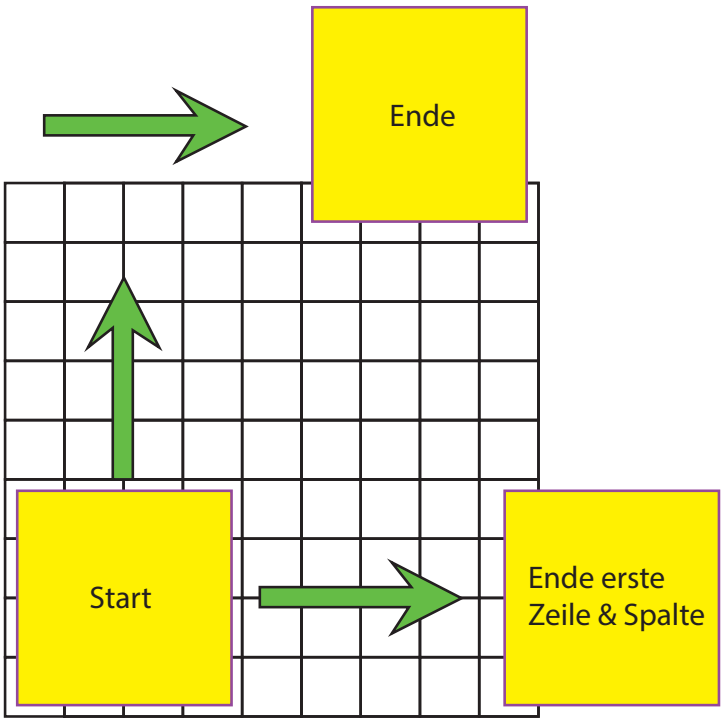

Abbildung 6.4: Abtastart über dem Spielfeld.

6.6.1 Pro

Durch diesen Ablauf kann garantiert werden, dass jeder Stein in jeder Variante geprüft wird und ermittelt werden kann, ob dieser auf das Spielfeld passt.

6.6.2 Contra

Bei diesem Algorithmus könnte das Problem bestehen, dass das Spiel langsam wird. Grund hierfür ist, dass im schlimmsten Fall bei einem Schleifendurchgang insgesamt ca. 972 if-Abfragen stattfinden können. Dies ergibt sich aus 81 Durchläufen (9x9 Spielfeld) und einer maximalen Anzahl der Variationen, d.h. zwölf Drehungen, für die vorgebenden Steine. Dieser Worst-Case tritt jedoch nur auf, wenn der T-Stein, J-Stein oder L-Stein[7] drei mal vorgegeben ist und es keinen Platz für diesen Stein im Spielfeld gibt.

[7] T-Stein, J-Stein und L-Stein besitzen mit allen Drehungen jeweils 4 Variationen.

6.7 Spielmodus

Bei den zwei Spielvariationen wird im Startbildschirm die Auswahl vorgegeben, ob der Spieler mit oder ohne Zeitbegrenzung spielen möchte. Wenn die Variante mit Zeitbegrenzung gewählt wurde, wird beim Spielbeginn ein Timer gestartet. Dieser führt bei jeder Sekunde eine Methode aus, welche die vorgegebene Zeit von 120 Sekunden heruntergerechnet. Der **GameViewController** [5.10] beobachtet dabei den Wert und übermittelt der View den Stand. Je nach Stand des Countdowns wird ein Label[8] mit dem Wert angepasst.

Je nach Modus wird ein Timer gestartet oder nicht. Wenn das Spiel ohne Zeitbegrenzung stattfindet, wird auch das Label für den Countdown entfernt.

[8] iOS-Element für einen Text auf dem Bildschirm.

7 Prototyp

7.1 Erfüllte Aufgaben

Im Prototypen wurden alle Ziele erreicht.

1. Das Spiel verfügt über eine modular und übersichtliche Architektur
2. Ein Spielfeld und die Steine werden vorgegeben.
3. Der Spieler kann einen beliebigen vorgegebenen Stein hinein malen, ohne diesen speziell zu markieren.
4. Sind Reihen und/oder Spalten gefüllt, werden diese entfernt.
5. Der Benutzer erhält Punkte fürs Legen der Steine bzw. für gefüllte Zeilen/Spalten.
6. Wenn die vorgegebenen Steine nicht hineinpassen, wird das Spiel beendet.
7. Zwei Spielvarianten wurden eingebunden, mit und ohne Zeit.
8. Im Spiel mit Zeitbegrenzung wird das Spiel beendet, wenn der Spieler keinen Stein mehr legen kann oder wenn die Zeit abgelaufen ist.

Darüber hinaus wurde an der Marktfähigkeit gearbeitet, hierauf wurde in dieser Arbeit nicht weiter eingegangen.

- Animationen beim Platzieren vom Stein.
- Alle Geräte die iOS 8 oder neuer haben, können das Spiel benutzen.
- Sound-Effekte werden abgespielt, sowohl beim Legen des Steines, als auch Entfernen einer Reihe/Spalte.
- In-App-Käufe werden nun angeboten.
- Der aktuelle Highscore wird im Apples-GameCenter gespeichert.
- Wenn innerhalb von zwei Spielzügen jeweils eine Reihe/Spalte gefüllt ist, bekommt der Spieler Bonuspunkte.

7.2 Erweiterung

Im weiteren Verlauf wird sich die Firma nodapo um folgende Punkte kümmern:

- Design einbinden
- Animationen
- Multiplayer
- Levelsystem

Des Weiteren könnte das Spiel durch Zeitdruck interessanter werden. Zum Beispiel dadurch, dass durch längeres Nichtlegen von Steinen Punktabzüge erfolgen. Bei den Algorithmen wurden bis zu diesem Zeitpunkt keine Fehler gefunden. Beim Untersuchen auf der Hardware kamen jedoch Probleme auf, da die vorherigen Tests auf dem Simulator stattfanden. Eines der Größten ist der extrem hohe Akkuverbrauch. Hierbei steht hauptsächlich der Algorithmus für die Überprüfung zum Game-Over in Verdacht. Aber auch der Algorithmus zum Prüfen auf vollständig gefüllte Reihen ist nicht außer Acht zu lassen.

8 Fazit

8.1 Swift

Swift ist eine klar strukturierte und moderne Sprache, jedoch zu jung um nativ und ohne Objekte von Objective-C Software zu entwickeln.

- Häufiger war man drauf angewiesen manche Pattern, wie das Oberserver-Pattern mit einem Script zu lösen. Grund war, dass Swift 1.2 zwar eine Observer-Variante hat, diese jedoch nur Objekte beobachten kann, die auf der Klasse NSObject von Objective-C erben.
- Die Dokumentation von Apple war zum Zeitpunkt der Entwicklung unvollständig.
- Häufiger wurden Fehler vom Compiler wiedergegeben, die weder hilfreich noch verständlich waren.
- Eine Enumeration konnte nur in einem Container beobachtet werden.

Kurz vor dem Vollenden des Prototypen hat Apple Swift 2.0 veröffentlicht (16. September 2015). Hierbei wurde die Sprache nochmals verbessert und eine Abhängigkeit von den Objektive-C Objekten wurde entfernt. Zudem wurde die Dokumentation verbessert und aktualisiert. Auch das Beobachten von Enumeration-Werten wurde realisiert, da dies nur über eine Container-Klasse bis dahin funktionierte. Auch das aktualisieren der Snippets Array2D und Observable war trotz der Sorgen einwandfrei. Ob nun die Sprache wirklich besser als vor dem Update ist, konnte zum Zeitpunkt der Arbeit nicht geklärt werden.

8.2 MVCS

Das MVCS-Entwurfsmuster ist gut in einem iOS-Projekt umsetzbar. Trotz der simplen Architektur kam es jedoch häufig zu Komplikationen. Zum einen hat Swift Probleme durch die unvollständige Dokumentation gemacht, zum anderen die Architekturvorstellung von Apple.

Bei der Implementierung des Game Centers und der In-App-Käufe war vorgesehen, dass die Services die Kommunikation zu den Apple-Servern tätigen und die Daten

von Apple in der Status-Klasse abspeichern. Daraufhin sollte der jeweilige Controller für die Anzeige bzw. die Aktualisierung der View sorgen. Hierbei war nicht sichergestellt, ob Apple diese Form des Codes akzeptiert. Aus dem Grund wurde, um Zeit und Kosten zu sparen, auf den Programmiervorgaben von Apple aufgebaut.

Die Status-Klasse hat ein Äquivalent zu globalen Variablen erreicht. In diesem Punkt hätte man vorzeitig reagieren sollen und eine klare Zugriffskontrolle einbauen müssen. Dies wurde jedoch zum Zeitpunkt des Prototypen ignoriert. Die Folge hiervon ist, dass eine umfangreiche Überarbeitung des gesamten Codes stattfinden müsste.

Durch das MVCS müssen bei einer Portierung auf ein anderes System nur die View und der Controller neu geschrieben werden. Die Models und Services müssen nur auf eine entsprechende Sprache übersetzt werden. Die Skripte Array2D und Observable, sollten übersetzbar sein, wobei das Observable z.B. in Java nativ abgedeckt wird.

8.3 SpriteKit

SpriteKit bietet einem Spiele-Entwickler viele grundlegende Funktionen an. Das Spiel als Prototyp, hätte auch unabhängig von SpriteKit funktionieren können. Jedoch werden die Funktionen in Zukunft vom Unternehmen verstärkt verwendet. Die Animationen und die Geräuscheffekte wurden schon umgesetzt und durch SpriteKit übersichtlich. Wenn es gewünscht ist, ein 2D-Spiel für die Plattform iOS zu entwickeln, ist diese Lösung optimal. Durch die Schleife, die nach jedem Frame stattfindet, ist es dem Entwickler möglich an jedem Punkt der Schleife Einfluss auf das Spielgeschehen zu nehmen. Zudem ist die Physik in SpriteKit für 2D-Spiele optimal. Nicht so schön war der Szenen-Builder, dieser ist derzeit stark beschränkt und bietet nicht die Möglichkeiten, die andere Spiele-Engines von Haus aus mitliefern.

Trotz allem ist SpriteKit eine übersichtliche und gute 2D-Engine. Da diese von Apple weiterentwickelt und nativ unterstützt wird, kann das Spiel von den Entwicklern immer schnell auf das neue System angepasst und/oder erweitert werden.

A Material

A.1 Gesamtarchitektur von Drawix

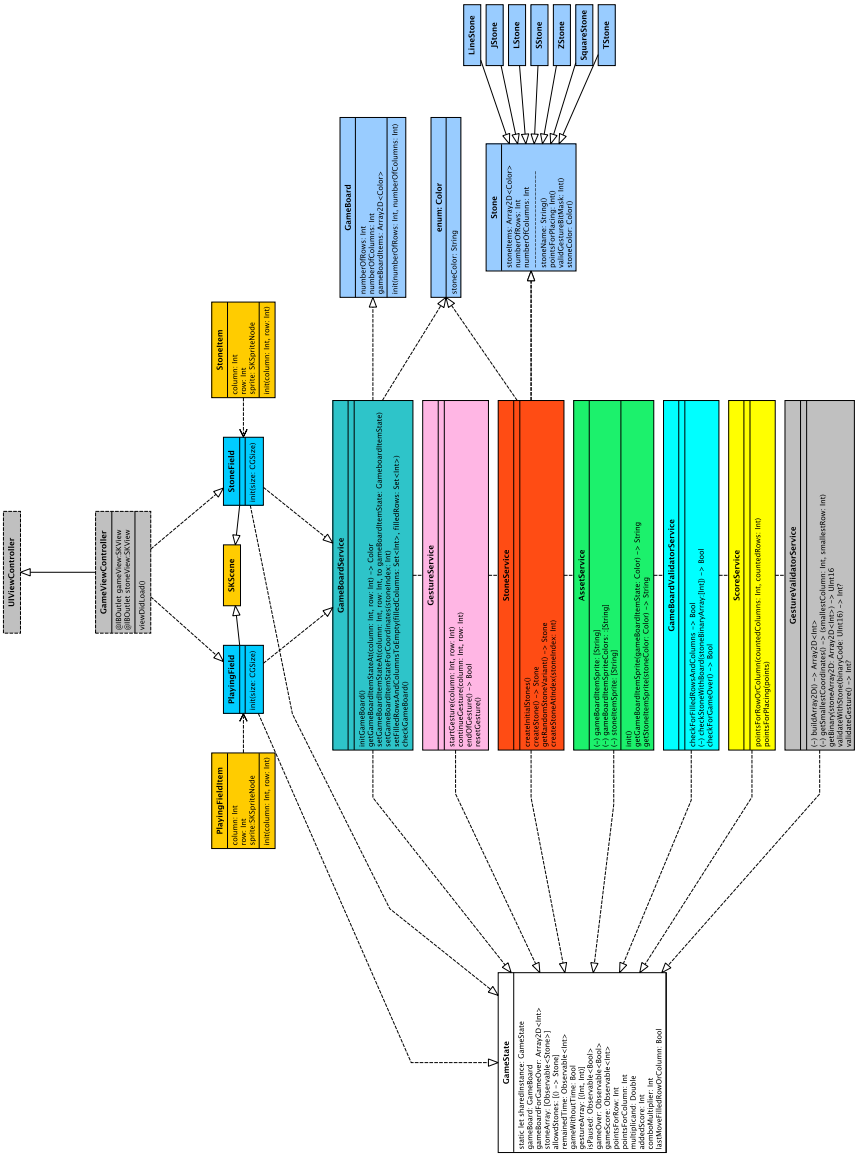

Abbildung A.1: Gesamtarchitektur von Drawix

A.2 Code Information

Folgende Dateien wurden vom Schreiber programmiert:
Services:

- AssetService.swift (zum Teil, siehe Datei)
- GameBoardService.swift
- GestureService.swift
- StoneService.swift
- GestureValidatorService.swift
- ScoreService.swift
- GameBoardValidatorService.swift

Models:

- GameState.swift
- Color.swift
- GameBoard.swift
- PlayingFieldItem.swift
- Stone..swift
- StoneItem.swift
- ProductCell.swift
- Alle Stein Subklassen (LineStone, JStone, LStone, ZStone, SquareStone, TStone)

Scenes:

- PlayingField.swift (zum Teil, siehe Datei)
- StoneField.swift

Controllers:

- IAPurchaseViewController.swift
- GameViewController.swift (zum Teil, siehe Datei)
- StartViewController.swift (zum Teil, siehe Datei)
- GameOverViewController.swift (zum Teil, siehe Datei)

Des Weiteren existiert eine HTML Dokumentation, die mit Jazzy[1] generiert wurde.

[1] https://github.com/Realm/jazzy

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

gamasutra.com: *Trends and next steps for mobile games industry in 2015*, http://gamasutra.com/blogs/IevgenLeonov/20141230/233356/Trends_and_next_steps_for_mobile_games_industry_in_2015.php, 2014, letzter Zugriff: 05.08.2015

Glossar Hochschule Augsburg: *Model View Controller Service Paradigma*, http://glossar.hs-augsburg.de/Model-View-Controller-Service-Paradigma, 2014, letzter Zugriff: 15.08.2015

Goll, Joachim: *Architektur- und Entwurfsmuster der Softwaretechnik*, 2. Aufl., Springer Fachmedien Wiesbaden 2013, 2014

iOS Developer Library - Apple: *About SpriteKit*, https://developer.apple.com/library/ios/documentation/GraphicsAnimation/Conceptual/SpriteKit_PG/Introduction/Introduction.html, 2014, letzter Zugriff: 12.08.2015

Java ist auch eine Insel: *Java ist auch eine Insel*, http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_19_016.htm#mja85628d88938f5fc859de8cfcc010a77, 2010, letzter Zugriff: 07.08.2015

medium.com: *The state of KVO in Swift*, https://medium.com/proto-venture-technology/the-state-of-kvo-in-swift-aa5cb1e05cba, 2014, letzter Zugriff: 27.09.2015

Mobile Insider: *Weibliche Mobile Gamer spielen länger, geben mehr Geld aus und sind loyaler*, http://www.mobileinsider.de/2014/08/11/weibliche-mobile-gamer-spielen-laenger-geben-mehr-geld-aus-und-sind-loyaler, 2014, letzter Zugriff: 31. 07. 2015

Objektorientierte Programmierung: *Objektorientierte Programmierung*, http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_08_002.htm, 2009, letzter Zugriff: 07.08.2015

philipphauer.de: *Das Singleton Design Pattern*, http://www.philipphauer.de/study/se/design-pattern/singleton.php, 2009, letzter Zugriff: 03.08.2015

raywenderlich.com: *How to Make a Game Like Candy Crush with Swift Tutorial: Part 1*, http://www.raywenderlich.com/75270/make-game-like-candy-crush-with-swift-tutorial-part-1, 2014, letzter Zugriff: 21.08.2015

Service-Oriented Architecture (SOA): *Service-Oriented Architecture (SOA) Definition*, http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html, 2015, letzter Zugriff: 10.09.2015

Stefan Popp & Ralf Peters: *Durchstarten mit Swift*, 1. Aufl., O'Reilly Verlag 2015, 2015

Wikipedia: *Beobachter (Entwurfsmuster)*, https://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster), 2015, letzter Zugriff: 05.08.2015

Wikipedia: *Candy Crush Saga Wikipedia*, https://de.wikipedia.org/wiki/Candy_Crush_Saga, 2015, letzter Zugriff: 03.08.2015

Wikipedia: *MVC (Entwurfsmuster)*, https://de.wikipedia.org/wiki/Model_View_Controller, 2015, letzter Zugriff: 05.08.2015

Wikipedia: *Polyminos*, https://de.wikipedia.org/wiki/Polyomino, 2014, letzter Zugriff: 31.08.2015

Wikipedia: *Singleton (Entwurfsmuster)*, https://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster), 2015, letzter Zugriff: 18.08.2015

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Eugen Waldschmidt