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

5. Oktober 2015

Inhaltsverzeichnis

Abstract

coming soon

Zusammenfassung

kommt noch

1 Einleitung

1.1 Motivation

Das Unternehmen nodapo Software GmbH wurde 2015 von Alexander Poschmann gegründet. Das unternehmen besteht mittlerweile aus 4 Mitarbeitern. 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. Worauf das Spiel **Drawix** entstand. Das Ziel mit dem Spiel ist es, einen Einstieg in den Spiele-Markt zu bekommen und in Zukunft weitere Spiele zu produzieren.
Das Besondere an an Drawix ist, dass das Spiel eine gute Basis für Erweiterungen und Verbesserungen liefert. Aus einem Spiel was nur ein Level hat, kann später ein großes Projekt werden. Des weiteren könnte man Erfahrung sammeln, die in anderen, in Zukunft kommenden Spielen, hilfreich sein könnte.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung des Spieles in Apples neuer Programmiersprache Swift. Da diese Sprache relativ Jung ist, kann es stets zu Problemen kommen. Jedoch bietet das Projekt eine gute Übung bei der Arbeit mit Swift. Des weiteren ist es gewünscht ein Entwurfsmuster zu verwenden, welches im Unternehmen nicht in einem iOS-Projekt zum Einsatz kam.
Folgende Schwerpunkte werden für diese Arbeit definiert.

1. Anwendung verschiedener Entwurfsmuster um eine übersichtliche Architektur zu schaffen. Diese soll stark modular sein und ohne Probleme in Zukunft erweitert werden.

2. Möglichst viel von der Programmierung soll nicht auf vorhandenen Frameworks o.ä. basieren. Kleinere Skripte, die Zeit sparen, sollen ruhig ihre Verwendung finden.

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. In diesem sollen Steine durch Malgesten gesetzt werden, Reihen/Zeilen wenn gefüllt gelöscht werden und erkannt werden, dass wenn ein Stein nicht mehr rein passt, dass das Spiel vorbei ist.

1.3 Aufbau

Im ersten Kapitel dieser Arbeit 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 ein besseres Verständnis von dem Spiele-Markt liefern. Das Ziel ist, ein grobes Bild von der Gestaltung der Konkurrenten zu erhalten. Außerdem könnte die Spielweise der Konkurrenz Aufschluss geben, welche Dinge in Drawix beachtet werden sollten.

Im dritten Kapitel werden die Technologien die für das Spiel gewählt wurden näher erläutert. Dabei wird Swift im groben vorgestellt und die wesentlichen Punkte, die im Verlauf der Arbeit erwähnt werden erklärt. Einen Vergleich zu Objective-C[1] zeigen. Hinterher wird auf SpriteKit eingegangen und der Grund für diese Entscheidung erläutert. Des weiteren die Snippets, die für die Arbeit genommen wurden näher gezeigt und erklärt.

Im vierten Kapitel Design-Pattern vorgestellt die ihre Anwendung in Drawix finden. Dabei werden Observer-Pattern, Singleton-Pattern, MVC als Basis für das MVCS-Entwurfsmuster genommen. Außerdem sollen die Kommunikationswege von den einzelnen Modulen näher gebracht und veranschaulicht werden.

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

Im sechsten Kapitel wird die Umsetzung des Spiels vorgestellt. Dabei wird abstrakt erläutert wie das Programm funktioniert. Nur die 3 Algorithmen, die für das Spiel extra konzipiert wurden, werden in diesem Kapitel detailliert erklärt.

[1] Erweiterte Programmiersprache C um objektorientierte Programmierung anzubieten.

Im siebten Kapitel wird der Prototyp vorgestellt. Dabei werden die Punkte aus 1.2 abgeglichen und auf deren Vollständigkeit geprüft. Des weiteren werden die Wünsche von dem Unternehmen 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. Dabei wird die Erfahrung und die Erkenntnis weitergegeben.

2 Analyse

2.1 Marktanalyse

Der Markt für Puzzle-Spiele ist groß und unübersichtlich. Dies könnte an ihrer Einfachheit und dadurch auch ihrer Beliebtheit liegen. Häufiger sind diese Spiele reine Denkspiele. In solchen fällen muss der Spieler keine große Aktion machen, sondern über jeden Zug nachdenken. Manche Spiele fügen für die gewisse Spannung und Herausforderung Zeitdruck hinzu. Hierbei hat der Spieler pro Zug wenig Zeit und muss sich schnell entscheiden, ansonsten wird dieser mit einem Punktabzug o.ä. bestraft.

Zusätzlich müssen Puzzle-Spiele nicht ununterbrochen gespielt werden. Meistens entscheidet der Spieler ob und wann er das Spiel spielen möchte. Dieser Luxus ist im Genre von Simulationen nicht immer enthalten. Durch die klein gehaltenen Level in Puzzle-Spielen benötigt der Spieler nicht unbedingt viel Zeit um ein Level zu meisten. 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 sollten erstmal Kriterien festgelegt werden. Zum einen ist es wichtig zu verstehen, warum die ausgewählten Puzzle-Spiele so erfolgreich sind. Zusätzlich können gestalterische Aspekte viel über das Spiel und seine Zielgruppe aussagen.
Nach einigen Gesprächen mit dem Geschäftsführer wurden folgende Kriterien aufgestellt:

- Welche Spiele könnten ein ähnliches Spielprinzip, wie Drawix haben?
- Was ist das Ziel von den verglichenen Spielen?
- Besitzen die Spiele besondere Eigenschaften, die das Spiel im zweifel interessant für den Spieler machen?
- Wie und warum ist die Gestaltung so, wie es ist?

2.1.1 Candy Crush Saga

Candy Crush Sage [1] von King.com Limeted einer Firma aus England wurde am 14. November 2012 für mobile Endgeräte entwickelt 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 - Kategory: Spiele - Puzzle. [2]

Das Spielprinzip ist zufällig angeordnete Süßigkeiten durch Wechselgesten so anzuordnen, dass dreier (oder mehr) Paare von der selben Süßigkeit entstehen. Sobald ein Paar gefunden ist, wird dieses aufgelöst und von oben kommen weitere Süßigkeiten nach. Bei einer Wechselgeste kann nur eine Süßigkeit horizontal oder vertikal mit einer anderen Süßigkeit 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 Wechsellungen. Eine Aufgabe kann daraus bestehen, dass eine andere Süßigkeit bis zum Rand des Spielfeldes zu treiben und diese dadurch aufzulösen.
Beim erstmaligen Start des Spiels bekommt der Spieler eine Anleitung. Bei der Anleitung und dem weiteren Spielgeschehen begleitet ein Charakter (oder Maskottchen) den Spieler. Sobald der Spieler nicht weiter 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 [3] ausgeben oder ein Leben zu opfern. Hat der Spieler sein letztes Leben ausgegeben, kann nicht weiter gespielt werden. Die Leben regenerieren sich alle halbe Stunde, d.h. jede halbe Stunde kommt ein Leben hinzu, bis es fünf sind. Außerdem können Leben über das soziale Netzwerk Facebook[4] verschenkt werden.

Das Spiel ist sehr bunt gestaltet, jede Süßigkeit hat dabei seine eigene Farbe. Im Hintergrund ist immer eine Grafik die zu der jeweiligen Welt 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 die anhand der Punktzahl eine Bewertung in Sternen anzeigt. Grundsätzlich wirkt das Spiel sehr kindlich und knallig. Dies könnte daran liegen, dass das Thema die Süßigkeiten selber sind. Zusätzlich erinnert das Spiel eher an ein Süßigkeitengeschäft.

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

2.1.2 1010!

Das Spiel mit dem Titel „1010!“ ist einer der interessantesten Konkurrenten von Drawix. Entwickelt wurde das Spiel von http://www.grams.gs, welcher seinen Sitz in Istanbul - Türkei hat.
Das Spielprinzip ähnelt dem Spielprinzip von Tetris. Mit dem Unterschied, dass der Spieler aus drei vorgegebenen Steinen einen auswählt und diesen in das Spielfeld reinzieht. Wenn alle drei Steine im Spielfeld eingesetzt wurden, kommen drei neue. Wie auch in Tetris werden vollständige Reihen und zusätzlich Spalten entfernt und hierfür Punkte vergeben. Das Auflösen mehrerer Reihen und Spalten wird durch Bonuspunkte ergänzt. Die Formel hierfür könnte wie folgt aussehen:

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

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, das reinziehen vom Stein an eine beliebige Stelle im Spielfeld. Das Spiel wirkt bunt, jedoch ordentlich. Das Spielfeld besteht aus grauen Steinen und die einzelnen Steine verfügen über eine bestimmte Farbe. Alle Spielelemente sind auf einem weißen Hintergrund platziert. Des weiteren existiert ein Nachtmodus. Wenn dieser aktiviert ist, wird der Hintergrund schwarz.

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 einem endlosen Spielerlebnis, 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 wie in 1010! müssen Reihen/Spalten befüllt werden. Zusätzlich müssen gewisse Aufgaben erledigt werden. Dabei hat der Spieler jedoch nur eine bestimmte Anzahl an Spielzügen.

Im Gegensatz zu 1010! wirkt 1010! World bunt und stark verniedlicht. Außerdem wurde ein Maskottchen hinzugefügt, welches einen über den 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 ordentlich und übersichtlich. Der Nachtmodus, wie in 1010! existiert nicht.

Tabelle 2.1: Eigenschaftstabelle

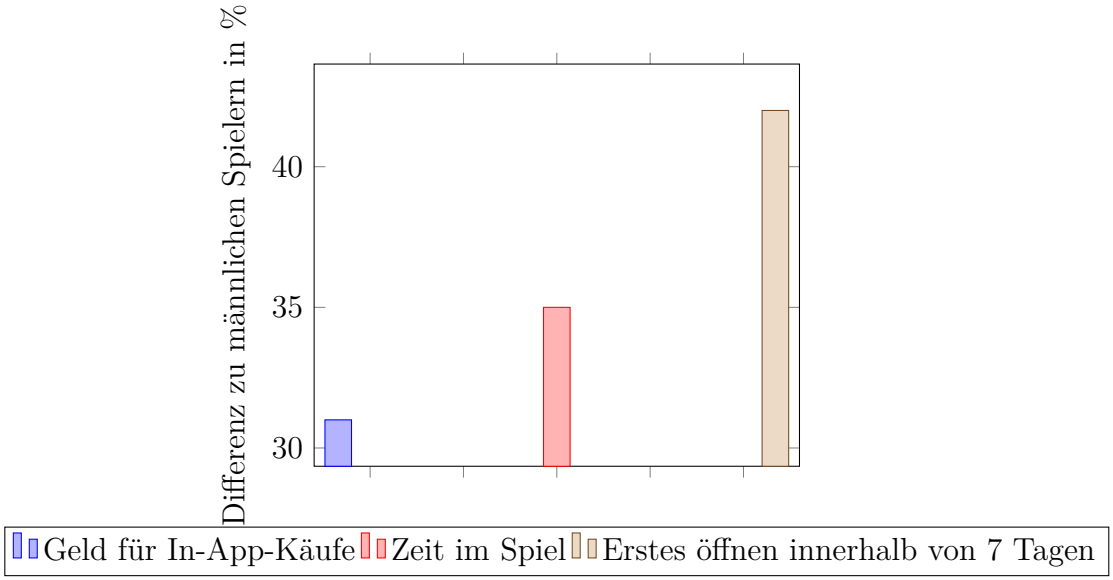
	Soziale Medien	Charakter	Level	Spielwelt	Anleitung
Candy Crush Saga	ja	ja	ja	ja	ja
1010!	begrenzt	nein	nein	nein	nein
1010! World	begrenzt	ja	ja	ja	ja

2.1.4 Zusammenfassung

Diese Tabelle dient dazu, eine Übersicht über die Eigenschaften zu schaffen. Alle drei Spiele sind bunt gestalten. Des weiteren haben alle drei Spiele eine Anbindung zu sozialen Medien, wobei bei 1010! und 1010! World nur das veröffentlichen vom aktuellen Spielstand ermöglichen. Bei Candy Crush können Freunde von sozialen Plattformen den Spieler im Spiel unterstützen. Bis auf 1010! verfügen alle anderen Spiele über einen Charakter, ein Level-System, eine Spielwelt und eine Anleitung. Die Anleitung dient als Beschreibung über die Steuerung des Spieles.

2.2 Zielgruppe

Um eine Zielgruppe zu ermitteln, wird erstmal auf die Studie von (Ingo Kamps 2014) verwiesen. Laut dieser Studie sind knapp 31% der Frauen eher bereit für In-App-Käufe [5] Geld auszugeben und verbringen 35% mehr Zeit als männliche Spieler auf mobilen Endgeräten. Selbst das erste öffnen der Spiele, in der ersten Woche, werden mit einer 42% höherer Wahrscheinlichkeit getätigt.



[5] Softwaregüter oder Features, die durch Kauf in der App/Spiel freigeschaltet werden.

Die Nachfrage bei Puzzle-Spielen ist nicht hoch. Diese liegt bei ca. 30% und hat kein führendes Geschlecht. Sowohl Männer, als auch Frauen spielen Puzzle-Spiele im gleichen Verhältnis. Männliche Spieler spielen eher Karten-, Rollenspiele oder „Tower-Defense[6]“ Spiele. Die weiblichen Spieler bevorzugen eher Spiele vom Genre Management bzw. Simulation[(Ingo Kamps 2014)].
Aus dem Grund ist eine Geschlechtertrennung bei einem Spiel wie Drawix nicht zwingend nötig. Von daher 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 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+ 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. Im Endeffekt, kann zum Zeitpunkt der Konzeption keine feste Zielgruppe angegeben werden.

[6] Tower-Defense oder zu Deutsch Turm Verteidigung sind Spiele wo der Spieler Türme aufstellen muss um seine Basis von Angreifern zu verteidigen.

3 Technologiewahl

3.1 Swift

Drawix wird komplett in Swift geschrieben. Swift ist eine recht junge Sprache und wurde 2014 auf der WWDC[1] vorgestellt. Grund für die neue Programmiersprache ist, dass Objektive-C aus den 80er Jahren stammt und eine Grunderneuerung nötig war[Stefan Popp & Ralf Peters (2015)]. Der Versuch bei der Entwicklung von Swift ist das Beste aus C und Objective-C zu bekommen, 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 der LLVM 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.

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)]. Jede Aufzählung enthält in Swift Fälle oder sogenannte Cases. Jedem Case kann ein Wert von den vorhin 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 wurden, in Variablen vom Typ der im Case deklariert wurde, übertragen werden. Ein wirklich interessanter Punkt ist, dass Enumeratioren in Swift sich selbst rekursive aufrufen können. In Drawix werden Enumeratoren ihren Einsatz in den Farben[5.3]

[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

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)].

- 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.
- Beide können Protokollen entsprechen.

Im Gegensatz dazu können Strukturen folgende Dinge nicht:

- Klassen können sowohl Properties als auch Methoden vererben
- Klassen können deinitialisiert werden, sobald das Objekt nicht mehr benötigt wird.
- Bei Klassen kann ein und die selbe Instanz an mehren Stellen gleichzeitig 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.

3.2 Frameworks & Snippets

Alle Frameworks und Snippets die für das Spiel benötigt wurden, werden in diesem Kapitel erklärt. Einige Funktionen sind in Swift nicht vorhanden oder schwer zu implementieren. Um keine Zeit zu verlieren wurden die hier vorgestellten Snippets und Framesworks genauer gezeigt.
Die hier verwendet Snippets sollten einen überschaubaren Code besitzen. Unter anderem bedeutet es, dass auch die Implementierung ohne Schwierigkeiten ablaufen sollte. SpriteKit ist von Apple selber 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. Trotz des Risikos hat sich das Unternehmen für diese zwei Snippets entschieden.

3.2.1 SpriteKit

SpriteKit ist eine Spiele Engine für 2D Spiele. Es bietet eine große Infrastruktur für Grafiken[5] 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[6]:

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

Im Gesamtumfang dient dient diese Schleife dazu, dass in jedem Frame eine gewünschte Änderung durchgeführt wird.
Sobald ein Frame geladen werden soll, führt die Schleife erstmal die Update-Methode aus. In dieser können Simulationen ausgeführt werden, Spiellogik angewendet werden oder Aktionen die benötigt werden definiert. Im nächsten Schritt werden alle Aktionen[7], die erstellt wurden, 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 nochmal 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 von einander haben sollen. Auch hier kann nach der Anwendung der Bedingungen 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 immer aufgerufen. Aus dem Grund sollte bei

[5] Eine Grafik wird im SpriteKit als Sprite bezeichnet.
[6] Wurde auf die Anforderung für diese Arbeit verkleinert.
[7] Animationen, Soundeffekte o.ä.

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 was 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.

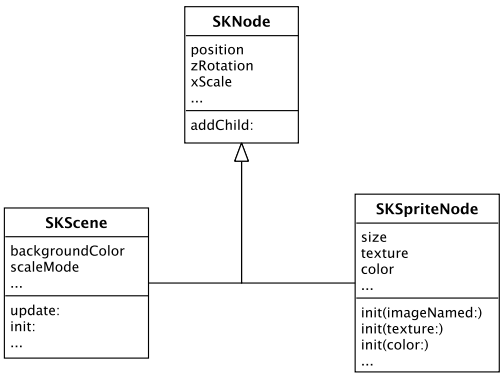


Abbildung 3.1: Erbverhalten von SpriteKit-Klassen

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 müsen Kind-Elemente nicht nur in der Szene vorhanden sein. Wenn mehrere **SKSpriteNodes** zusammen gehören, kann man die in ein Container setzen. Der Container kann sowohl ein **SKSpriteNode** als auch ein **SKNode** Objekt sein.
Jede Szene gibt vier Methoden vor mit den 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 Anruf kommt, 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, wo sich der Benutzer mit dem Finger befinden, abgefragt werden. Durch die Positionen können alle Objekte, die durch die Geste berührt wurden mit Aktionen versehen werden, beispielsweise das Entfernen der Objekte.

3.2.2 Array2D

Das Spielfeld[5.2] wird ein zweidimensionaler Array. Swift bietet zwar zweidimensionale Arrays an, jedoch mit der Einschränkung, dass es sich hierbei um ein sogenanntes Dictionary[8] handelt. Für Drawix wäre es von Vorteil 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.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Abbildung 3.2: Eindimensionaler Array von Array2D mit 4*4

Eine Funktion sorgt bei der Abfrage von Zeile und Spalte dafür, welcher Index im eindimensionalen Array dies entspricht.

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

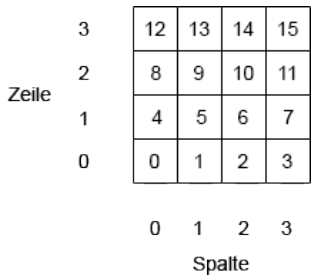


Abbildung 3.3: Zweidimensionaler 4*4 Array2D

[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.

In Abb 3.4 wird der theoretischer zweidimensionaler Array von dem eindimensionalen veranschaulicht. Angenommen es wird der Wert aus Zeile 2 und Spalte 2 benötigt, ergibt dies laut der Formel 6.3 den Index 10.

3.2.3 Observable

Da in Swift ein Observer von der Klasse NSObject abhängen muss und die Umsetzung relativ aufwendig ist, wurde der Code von Niels de Hoog (2014) gewählt.
Mit dieser generischen Klasse wird beim Aufruf ein Objekt vom Typ Obserable erzeugt. Diese trägt das 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.

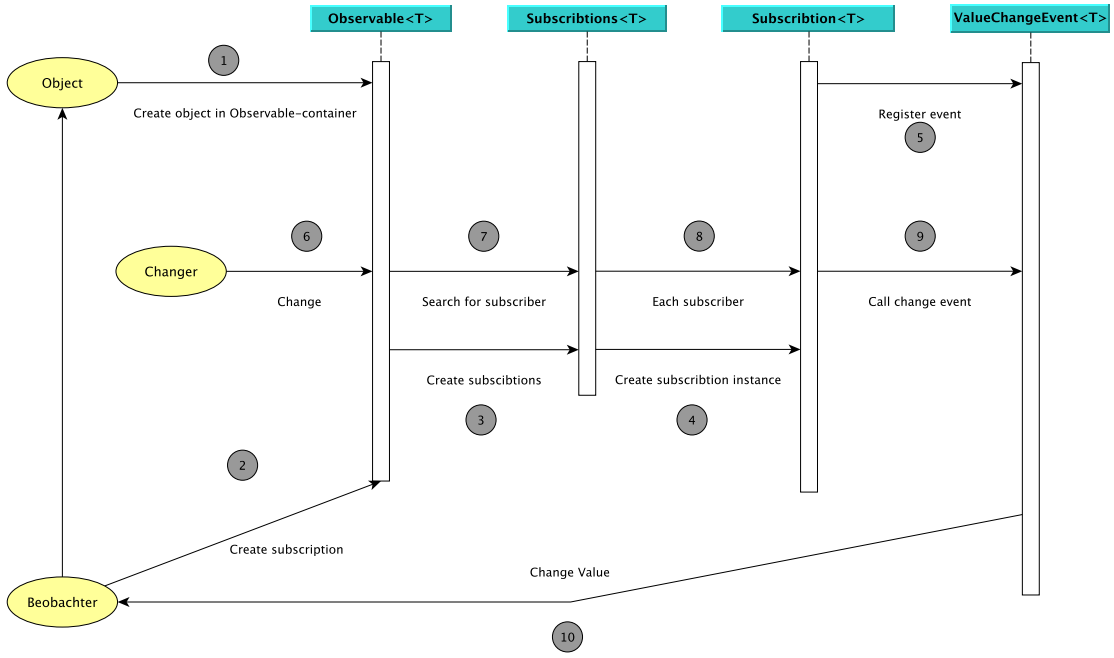


Abbildung 3.4: Zweidimensionaler 4*4 Array2D

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 getan. Dieser enthält alle Beobachter für dieses Objekt.

4. Für jeden Beobachter wird im Container eine Instanz für den Beobachter erstellt.
5. Jeder Beobachter weißt der Instanz eine Funktion zu, die sobald sich das Objekt ändert aufgerufen wird.
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 Boabachter.
8. Bei jedem Beobachter wird dazugehörige Funktion rausgesucht.
9. Die Referenz zur Funktion 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 ist, 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 Objekte an ihrer Beobachter zu übermitteln.
Um ein Verständnis für dieses Pattern zu bekommen sollte man ein reales Beispiel betrachten:

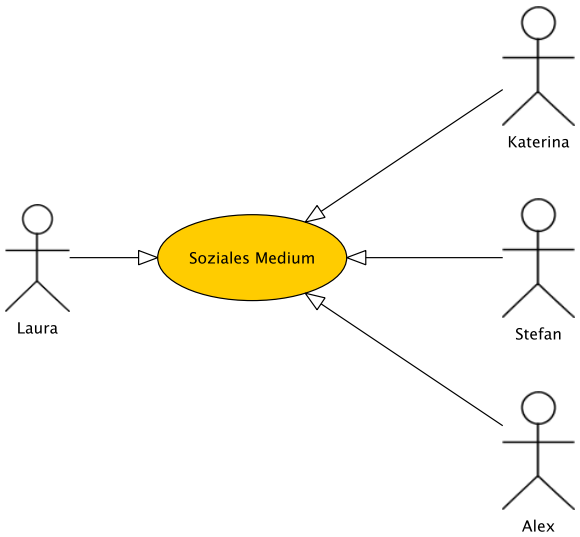


Abbildung 4.1: Beispiel ohne Observer Pattern anhand eines sozialen Mediums

Angenommen vier Freunde haben einen Account auf einem sozialen Medium [Abb. 4.1], in dem Fall einem fiktiven. Jede Person hat die Möglichkeit eine Mitteilung zu veröffentlichen und erwartet, dass die anderen 3 Freunde diese lesen. In dem Fall werden fiktive 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 könnte einige Mitteilung untergehen oder niemals gelesen werden.

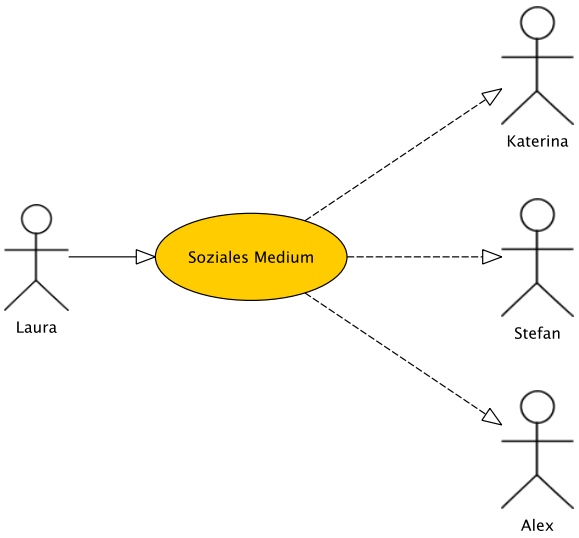


Abbildung 4.2: Beispiel mit Observer Pattern anhand eines sozialen Mediums

Im Fall eines mit Observer-Pattern ausgestaltetem 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.

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 Objekte die beobachtet werden, müssen nicht an jeden Beobachter angepasst werden, denn was mit den Daten geschieht, entscheidet der Beobachter. Dies sind nur einige Vorteile,

4.1.2 Nachteile

Wenn mehrere Beobachter ein Objekt beobachten kann es 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 ein Integer-Wert benötigt, kann es zu Laufzeitfehler kommen.
Ein anderes Szenario wäre, wenn der Beobachter auch gleichzeitig bei 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 könnte das Observer-Pattern hilfreich sein.

4.2 Singleton Pattern

Ein Singleton Pattern (zu deutsch: Einzelstück Muster) ist eine Art von Entwurfsmuster wo ein Objekt in der selben Instanz bleibt, in der es beim ersten mal initialisiert wurde [Wikipedia Singleton (Entwurfsmuster)]. Wenn ein Objekt im Normalfall instanziiert wird, erhält dieses immer eine eigene Instanz. Das kann zu Folge haben, dass Änderungen in Instanz A kein Einfluss für Instanz B nach sich ziehen, obwohl es im zweifel gewünscht oder erwartet wird. Dies wird interessant wenn mehrere Objekte gleichzeitig das Objekt bearbeiten sollen und dabei Unabhängig von einander 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)].

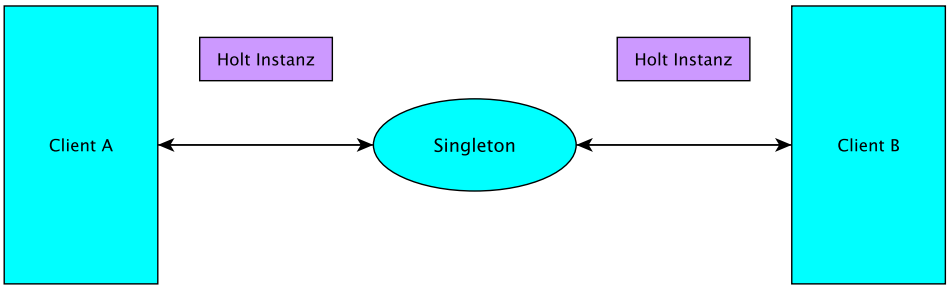


Abbildung 4.3: Veranschaulichung von einem globalen Objekt mit der selben Instanz

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 instanziieren, sondern eine Methode aufrufen, welche die das Objekt mit der vorhanden Instanz zurück gibt. 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 einsetzten, da die Gefahr könnte 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. Es hängt stark davon ab wie die Implementierung stattgefunden ist und wie das Singleton genutzt 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. @todo: mvcs zeigen

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, die für die Arbeit relevant sind zu nennen. 1978/79 wurde von Trygve Reenskaug, einem norwegischen Forscher der Informatik, das Konzept des Entwurfsmusters entwickelt. Die erste Anwendung fand in der Programmiersprache Smalltalk[2] statt. Die

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

[2] Smalltalk ist eine dynamische Programmiersprache. Auf diese Sprache wird in der Arbeit nicht weiter eingegangen.

Idee hinter MVC ist, eine saubere Trennung zwischen den vorhandenen Daten und der Interaktion vom 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, kann jedoch auch in Anwendungen mit einer View angewendet werden. MVC ist variable 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 Änderung 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. Dabei kann es sich sehr wohl um die gleichen Daten handeln [Abb. 4.4].

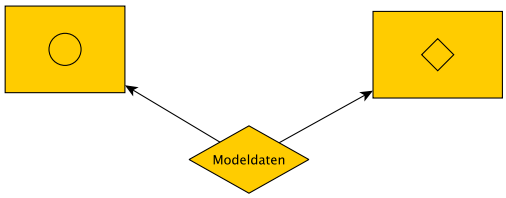


Abbildung 4.4: Selbe Model-Daten unterschiedliche Darstellung

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 vom Model bedienen. Je nach Kommunikation 4.5 gibt der Controller die neuen Daten, falls nötig an die View weiter. Andersrum kann der Controller auch direkt die View manipulieren. Ein Controller kann sowohl mehrere Views als Models steuern, hierbei gibt es keine Beschränkung.

4.3.4 Zwei Kommunikationsarten von MVC-Komponenten

Wie vorher erwähnt, kann ein MVC den Anforderungen angepasst werden. In dem Fall werden zwei Konstrukte vorgestellt. Beide verletzen demnach nicht das MVC-Entwurfsmuster. Das erste Entwurfsmuster bedient sich dem Observer-Pattern und das zweite einem typischen MVC.

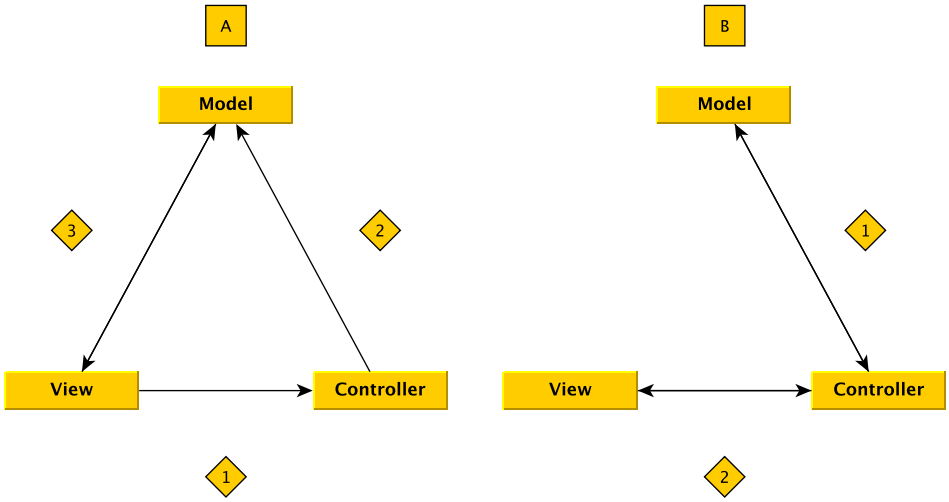


Abbildung 4.5: Zwei Varianten von MVC, die in dieser Arbeit behandelt werden

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) - Seite 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: zeigt eine andere Variante des MVC Entwurfsmusters. 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 über die Existenz des anderen.

4.4 MVCS Paradigma

Das MVCS Paradigma entspricht im Grundgedanken dem MVC 4.3. In diesem Entwurfsmuster gibt es, im Gegensatz von MVC eine Service Ebene. Auch bei MVCS gibt es viele mögliche Kommunkationswege 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). Der Versuch für diese Arbeit ist, ein MVCS zu entwickeln, welches wie das MVC-Entwurfsmuster Moduler und in Schichten eingeteilt ist.

4.4.1 Serviceorientierte Architektur

Bevor die einzelnen Module erklärt werden, sollte erstmal klar sein 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. Ein Beispiel:

Abbildung 4.6: Beispiel Service

Angenommen Software A und Software B sind von verschiedenen Herstellern und die Kommunikation zwischen den sind auf dem direkten Wege nicht möglich, da Software A nicht verstehen kann was Software B übermitteln möchte. Um die Kommunikation zwischen den Applikationen herzustellen könnten die serviceorientierte Architektur eine Lösung bieten.
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 andersrum, falls gewünscht und in den Services möglich. Jegliche Kommunikation zwischen der Software A und B findet über die Services statt. Auch jede Anfrage von der Software muss über den Service geschehen. 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 Service geregelt werden, jedoch ist für das hier vorgestellte MVCS eher die Kommunikation zwischen zwei Services oder mehr relevant.

4.4.2 VCSM

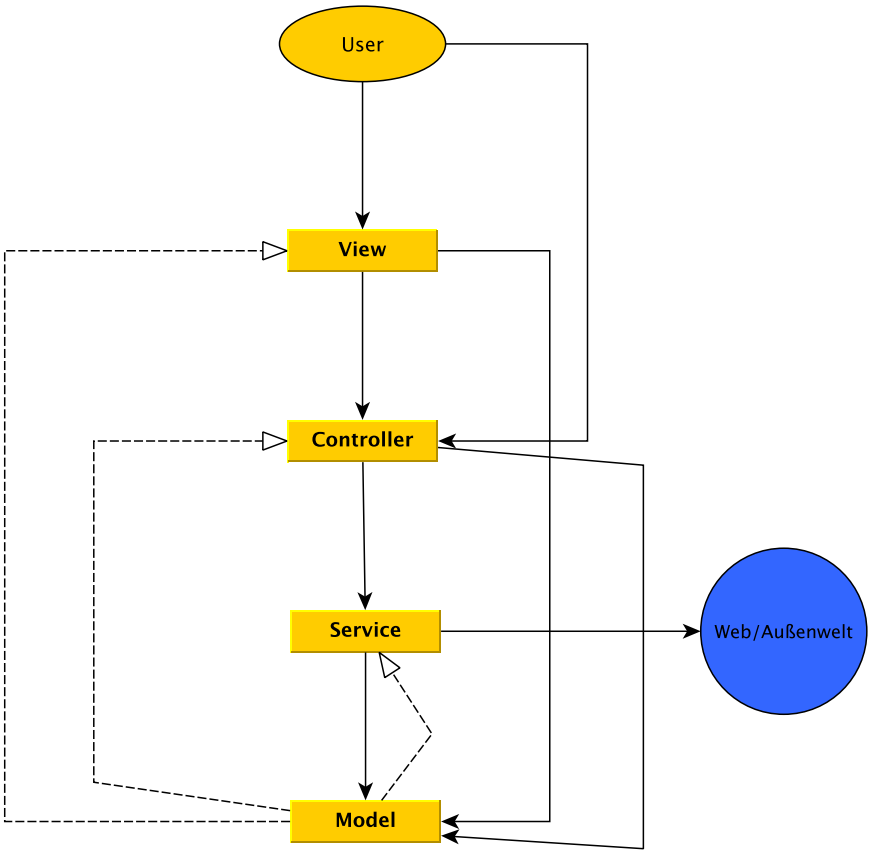


Abbildung 4.7: Beispiel für MVCS als Schichtenarchitektur[Glossar Hochschule Augsburg (2014)]

Der Aufbau in dieser Variante des MVCS-Entwurfsmusters (in diesem Fall VCSM wegen der Anordnung) entspricht verschiedener Schichten. Bei diesen Schichten können die höheren von den darunter liegenden Schichten zwar Anfragen stellen, jedoch keine Anfragen erhalten. Um trotz alldem eine rückwirkende Kommunikation zu bieten, bedient man sich hier dem Observer Pattern.
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 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 liegendem Modul Bescheid wissen.

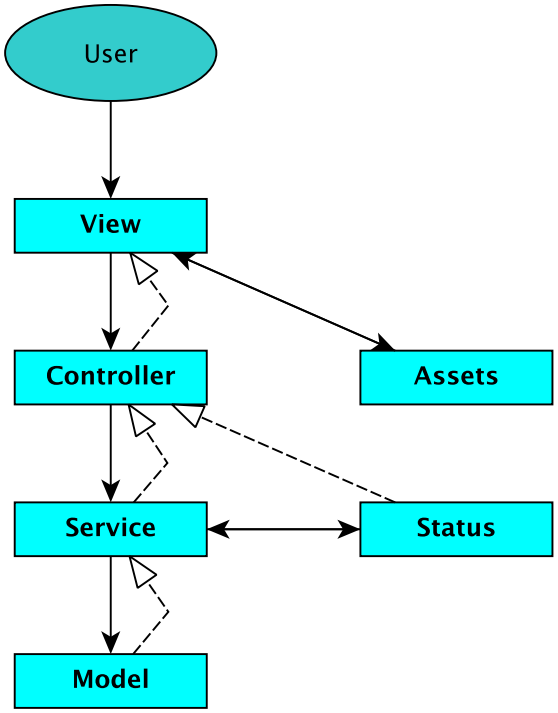


Abbildung 4.8: Aufbau vom MVCS-Entwurfsmuster für Drawix

Auf 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.

Tabelle 4.1: Diese Tabelle zeigt 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

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 aus 4.2, dass ein Äquivalent zu globalen 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.
Die Status-Klasse könnte als Vorteil vielleicht für eine bessere Überschaubarkeit sein, da es dazu kommen kann, dass die Übersicht über die globalen Variablen verloren gehen kann.

4.4.4 Assets

Hinter den Assets hängt in iOS ein Asset-Katalog. Dieser enthält alle Bilder die in einem iOS Projekt dargestellt werden müssen. Das hat den Vorteil, dass unabhängig davon welche Auflösung das Gerät hat, immer das richtige Bild darstellt. Im Code kümmert man sich hinterher nur darum das gewünschte Bild auszugeben. Dabei handelt es sich um ein Gruppe an Bildern. In dieser Gruppe müssen die Bilder in jeder Auflösung[3].

In diesem Management müssen folgende Bilder hinterlegt werden:

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

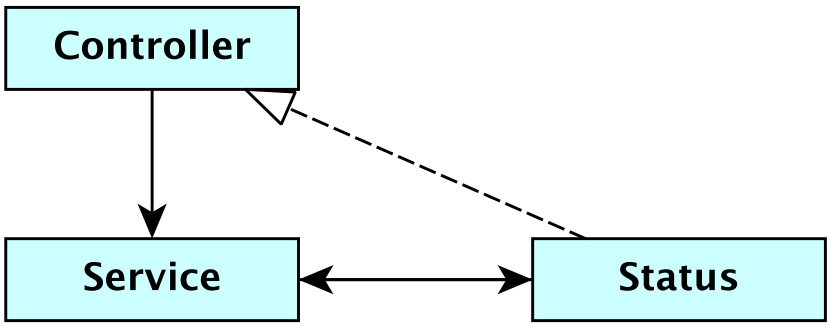
Auf dieses Management soll im MVCS nur die View Zugriff haben.

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

4.4.5 Service

Beim Service besteht die Möglichkeit, dass auf Anfrage vom Controller eine Manipulation den 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. In diesem Fall würde der Controller der Beobachter der von der View dargestellten Daten sein. Wie in MVC [4.5 - B] würden Aktualisierungen der View vom Controller übernommen werden. Aus dem Grund müsste der Service dem Controller nicht antworten, sondern die gewünschte Änderung in der Status-Klasse umsetzen.

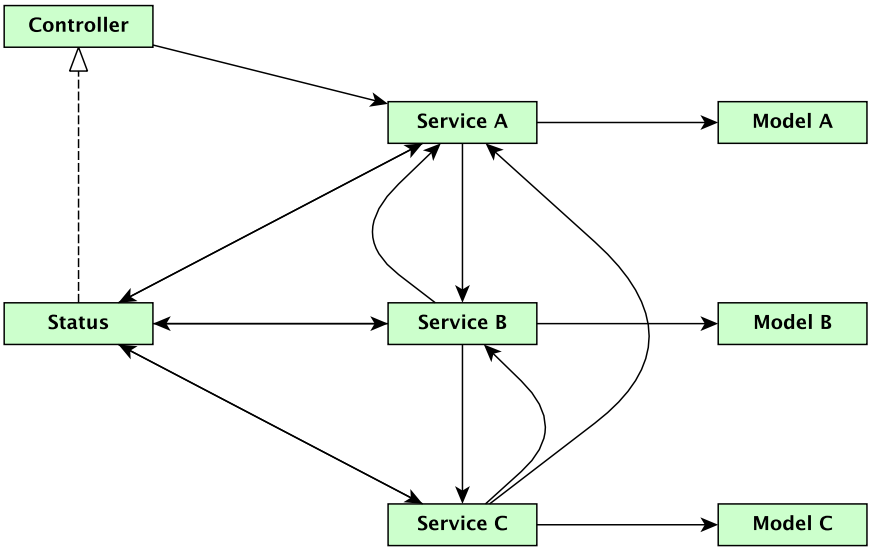
Abbildung 4.9: Controller - Service - Status-Klasse Kommunikation



In diesem Konzept gibt es für jedes Model ein 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 anpasst wird.

4.4.6 Unterschied zu MVC

- Bei der **View** gibt es im keinen 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 Statusklasse. Nur bei bedacht kann der Service einen Rückgabewert übermitteln, welchen der Controller benötigt. Ein Bolean um eine bestimmte Bedingung zu erfüllen.
- Im Gegensatz zu 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 von einem Datenobjekt. Vergleichbar ist das Model mit einer Blaupause und keinen Eigenschaften, die das Objekt selbst ändern kann.

5 Konzeption

5.1 Spielprinzip

Das Spielprinzip orientiert sich an Tetris und anderen Puzzle-Spielen auf mobilen Endgeräten (siehe: 2.1).
Dabei wird 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 befüllt ist, werden diese Reihen und/oder Spalten[3] zurück gesetzt. Das rein malen eines Steines verursacht ein Event, dass der eingezeichnete von den drei vorgegebenen Steinen verschwindet und durch einen neuen ersetzt wird. Jedes rein malen eines Steines als auch das verschwinden einer Reihe und/oder Spalte gibt dem Spieler Punkte.

Im Prototypen sind zwei Varianten geplant. Im einem kann der Spieler solange spielen bis keiner der drei vorgegebenen Steine reinpasst, was einem Modus ohne Zeitbeschränkung entspricht. In dem anderen wird ein Countdown runter zählen. Das Spiel ist vorbei, sobald kein vorgegebener Stein passt oder wenn der Countdown bei Null angelangt ist. Das Hauptziel in dem Spiel, jedenfalls im Prototypen, ist eine möglichst hohe Punktzahl zu erreichen.

5.1.1 Tetromino

Im Prototypen werden die Steine den Tetrominos entsprechen. Steine die aus Quadraten zusammen gebaut werden, heißen Polyminos. Je nach Anzahl der Steine besitzen diese andere Namen. Bei den Tetrominos besteht jeder Stein genau aus vier Quadraten. Wenn man die vier Steine auf einem Feld von 2*4 anordnet und keine Lücke hinterlässt 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.

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 2*4 Anordnung. Bei dem grünen Stein hingegen, wurde eine richtige Konstruktion angewendet. Demnach müssen die Steine an einander hängen. Wenn

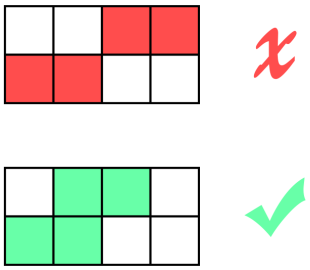


Abbildung 5.1: Falsche und richtige Anordnung

man dieser Regel folgt, ergeben sich folgende Steine:

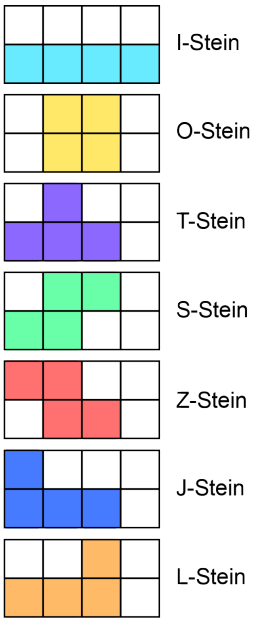


Abbildung 5.2: Alle möglichen Tetrominos

Diese können jedoch in jeder möglichen Drehung gemalt werden. Das ergibt eine Anzahl an 19 verschiedene Variationen. Angenommen es werden alle Steine in eine 4*4 Matrix überführt. Hierbei werden alle 19 Variationen von den Steinen in diese Matrix reinpassen. In der Umsetzung könnte dies vielleicht hilfreich sein. Wenn

herausgefunden werden muss, um welchen Stein es sich handelt den der Spieler rein gemalt hat.

5.2 Gestaltung

Bei der Gestaltung von Drawix wurde darauf geachtet, dass es relativ sauber und übersichtlich ist. Dieses Design ist kein Finales Ergebnis, es dient hauptsächlich nur für die Entwicklung. Es ist Sinnvoll schon mal die Elemente zu implementieren und bei einem anderen Design diese einzubinden. Hierbei wurden keine künstlerischen Regeln befolgt, da es auch nicht Bestand der Arbeit ist. Insgesamt hat das Unternehmen nodapo zur Veröffentlichung vier unterschiedliche Screens geplant. Im weiteren Verlauf werden alle Screens vom Design 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. Einen 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

Des weiteren befinden sich reine Anzeigeobjekte wie das Logo und der aktuell beste Punktestand. Von allen aufgelisteten Punkten ist für den Prototypen Punkt eins und zwei relevant.

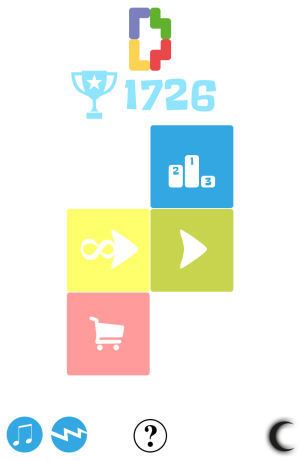


Abbildung 5.3: Startbildschirm in Drawix

Im eigentlichen Spiel befinden sich die vorgegebenen Steine, das Spielfeld, die übrige Zeit, der Pause-Button, der aktuelle Punktestand und die Werbung[4]. Um einen direkten Blick auf die Zeit und den aktuellen Punktestand sowie die vorgegeben Stein 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 durch verdecken mit der Hand nicht schnell den nächsten Stein auswählen kann. Unter den Anzeige-Elementen wird sich das eigentliche Spielfeld befinden. Dies soll möglichst den gesamten Bildschirm füllen.

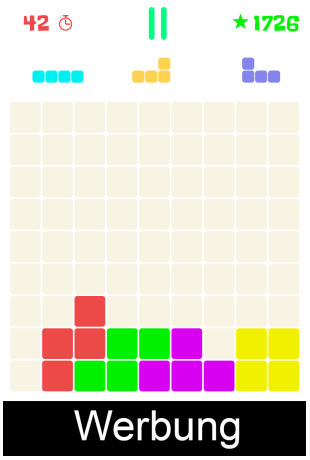


Abbildung 5.4: Eigentliche Spiel in Drawix

Wenn der Spieler auf den Pause-Button drückt, soll ein Bereich erscheinen der das gesamte Spiel verdeckt, sonst könnte die Pause zum schummeln genutzt werden. In der Pause kann der Spieler das Spiel neu starten, fortführen auf den Startbildschirm gehen oder in den Shop[5] gelangen. Außerdem soll von hier aus der Sound ein- und ausgeschaltet werden.

[4]Werbung soll zukünftig zur Finanzierung des Spiels sein.
[5]Später für In-App-Käufe, z.B. von Werbung freikaufen.

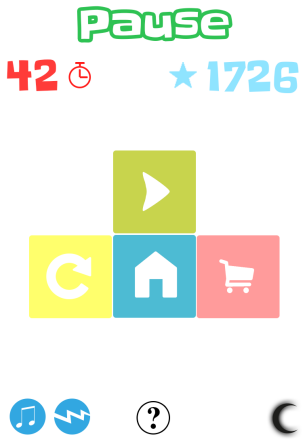


Abbildung 5.5: Pause Bildschirm in Drawix

Beim verlieren soll ein *Game Over*-Bildschirm angezeigt werden. Dieser enthält sowohl den Button für den Neustart des Spieles, ein Button für die Highscore-Liste und 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.

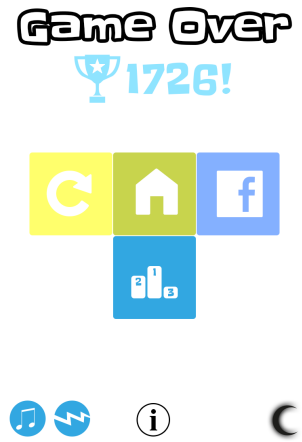


Abbildung 5.6: Game Over in Drawix

5.3 Anwendungsarchitektur

5.3.1 Diagramm Erklärung

Um ein grobes Bild von der Architektur zu erhalten, wird ein UML-Diagramm benötigt. Dabei wird auf dem MVCS 4.8 aufgebaut. Wird mit *(-)* nicht markiert, handelt

es sich hierbei um eine public Methode/Propertie.

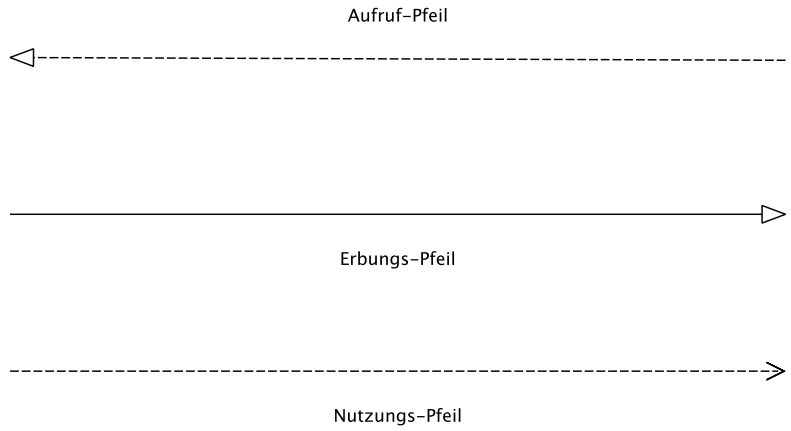


Abbildung 5.7: Pfeillegende für die verwendeten Pfeile.

Um zu verstehen wofür welche Pfeil im Detail steht, soll diese Grafik verdeutlichen.

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

5.3.2 Model Entwurf

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

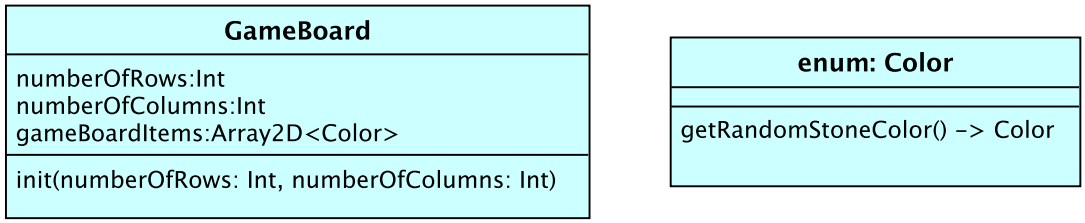


Abbildung 5.8: Drawix Model Entwurf - Farbe und Spielfeld.

die Anzahl der Zeilen und Spalten in sich trägt. Außerdem erstellt das Spielfeld ein Array2D mit den Inhalt Farbe. Dies ist das eigentliche Spielfeld. Die Farben sind in einem Enumerator @todo->swift enum erklären. In diesem werden alle Farben für die

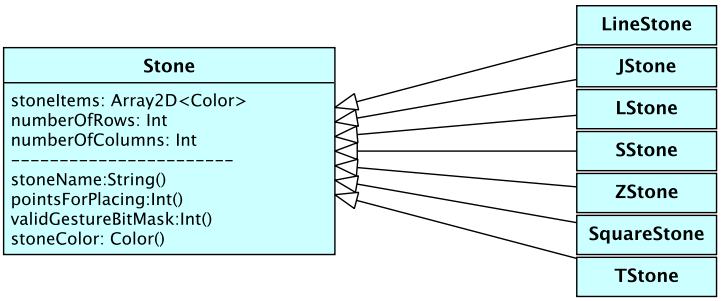


Abbildung 5.9: Drawix Model Entwurf - Steine.

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 vom von dieser Superklasse. Dabei werden sieben Steine benötigt [5.2]. Alle Properties unter dem gestrichelten sind diese, welche von jedem Stein überschrieben werden müssen. Das **Color**-Model ist in dem Sinne kein Model, sondern eine reine Auflistung von Farbvarianten. Diese Farben werden als Wert genommen, damit der dazugehörige Service das richtige Bild für die View raus sucht.

5.3.3 Controller Entwurf

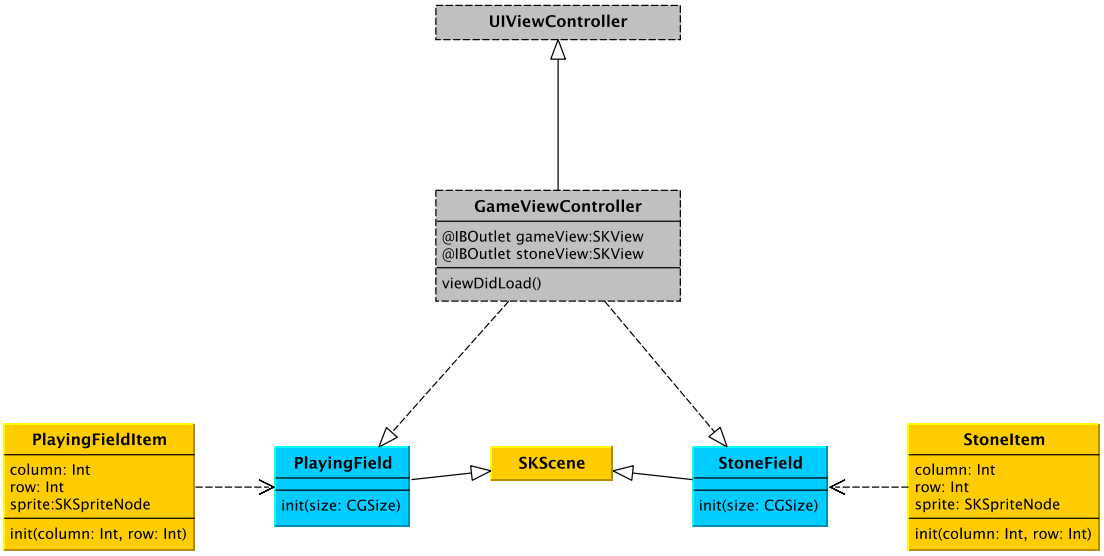


Abbildung 5.10: Drawix Controller Entwurf.

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 ist. 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 vom Aufbau aus 3.1 Szenen deklariert werden, welche von der Klasse SKScene erben. Wie in 3.1 kümmern die sich um Teilbereiche der View. Präsentiert werden diese von dem **GameViewController**. In den Szenen werden sowohl die dargestellten Texturen als auch Animationen implementiert. Da nur der **GameViewController** mit der View kommuniziert geben die Szenen dem GameViewController bescheid, was auf der View sich ändern soll. Damit das **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 selbe für das Spielfeld. Dabei zieht sich die Szene die Informationen aus dem **GameBoard**-Model.@todo: vielleicht lieber in die umsetzung.
StoneItem ist die Model-Klasse die sich um die Darstellung der Steine kümmert [5.2] 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 war eine Schichtentrennung wie in 4.8 beschrieben.

5.3.4 Service Entwurf

Die Services sind alle unabhängig von einander und komplett eigenständig. Jedes Model gehört zu einem Services. 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 Services 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 bei einem Spielstart die Initialisierung vom Spielfeld 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, sollen der Service über die Informationen vom Stein die Spielfeldsteine im Spielfeld anpassen. Hierbei soll nur die Farbe von den einzelnen **PlayingFieldItems** geändert werden (Color Aufzählung). In der Methode *checkGameBoard* soll das Spielfeld auf volle Reihen bzw. Spalten geprüft werden. Wenn welche gefunden sind, 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. Darum soll sich die *resetGesture* Methode kümmern.
StoneService erstellt beim Spielstart drei initiale Steine. Dabei soll die initial-Methode drei mal 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 die **getRandomStoneVariant()** aufgerufen. Diese soll eine zufällige Zahl ($\mathbf{Z}_7$) ermitteln und in einem eindimensionalen Array mit allen Steinvarianten, diese Zahl als Index verwenden. Gespeichert werden die Steine in einem Array, wo 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[6] vorerst im Service gespeichert wurden. Die grundlegende Aufgabe von diesem Service 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 zurück gegeben.
GameBoardValidatorService dient dazu das Spielfeld zu kontrollieren ob Zeilen und/oder Spalten gefüllt sind und ob kein vorgebender Stein mehr reinpasst. Zusätzlich soll dieser Service nach jedem Zug aufgerufen werden und Prüfen ob ein vorgegebener Stein reinpasst oder nicht. Ist dies nicht der Fall soll das Spiel zu ende gehen.
ScoreService hat die Aufgabe Punkte für die Reihen bzw. Spalten zu vergeben und hierfür den gewünschten Multiplikand zu ermitteln. Denn je nachdem wie viele Reihen und/oder Spalten gefüllt waren, bekommt der Spieler eine gewisse Punktzahl.Formel für volle Reihen/Spalten und bei multiplen Reihen/Spalten: $p = p + \sum_{n=1}^{a} p_2 * 1,5 | p, p_2 \in \mathbf{Z}, a \in \mathbf{Z}_{16}$
$a => AnzahlderdoppeltenZeilen/SpaltenalsSumme$
$p => VorhandenePunktzahl$
$p_2 => BekommenePunktzahl$[7]

Zudem soll der Service auch Punkte das Legen eines Steines berechnen. In Zukunft kommen Bonuspunkte hinzu, die z.B. durch schnelles legen dazu gerechnet werden. Wie viel man für das Legen eines Steines erhält, liegt in den Subklassen vom Stein-Model.

[6] Namen der Bilder
[7] Zeile oder Spalte liefert 150 Punkte, Stein liefert derzeit 1 Punkt

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.

6 Umsetzung

6.1 Überblick

Bei der Umsetzung wurde mit dem Projektmanagement-Verfahren Scrum gearbeitet. Bei Scrum werden erstmal 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 2, in seltenen Fällen auf 3 Wochen festgelegt. Für jeden Sprint werden Anforderungen, die für diesen Sprint benötigt werden, aus dem Produkt-Backlog[1] in den Sprint-Backlog getan. Die Anforderungen werden in dem Fall als User-Stories[2] abgelegt. Bei einem Sprint werden die Anforderungen für den jeweiligen Sprint nicht geändert und das Team von Entwicklern kann ungestört daran arbeiten. Am ende eines Sprints wurden stets die umgesetzten Anforderungen vorgestellt. Zudem wurde der Sprint im Ablauf und seinen Inhalten wiedergegeben. Dies ist eine Möglichkeit ein Ablauf im Sprint zu verbessern oder an die Teamgröße anzupassen. Alle stattgefunden Sprints wurden in folgender Reihenfolge abgearbeitet:

1. Spielfeld erzeugen und abbilden
2. Steine erzeugen und abbilden
3. Touche-Events ermitteln
4. Geste erkennen und Zeilen und Spalten löschen, wenn voll
5. Game Over Status ermitteln und Punkte Vergabe und Zeit beschränktes Spiel

Nachdem die Implementierung von allen Klassen aus 5.3 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.

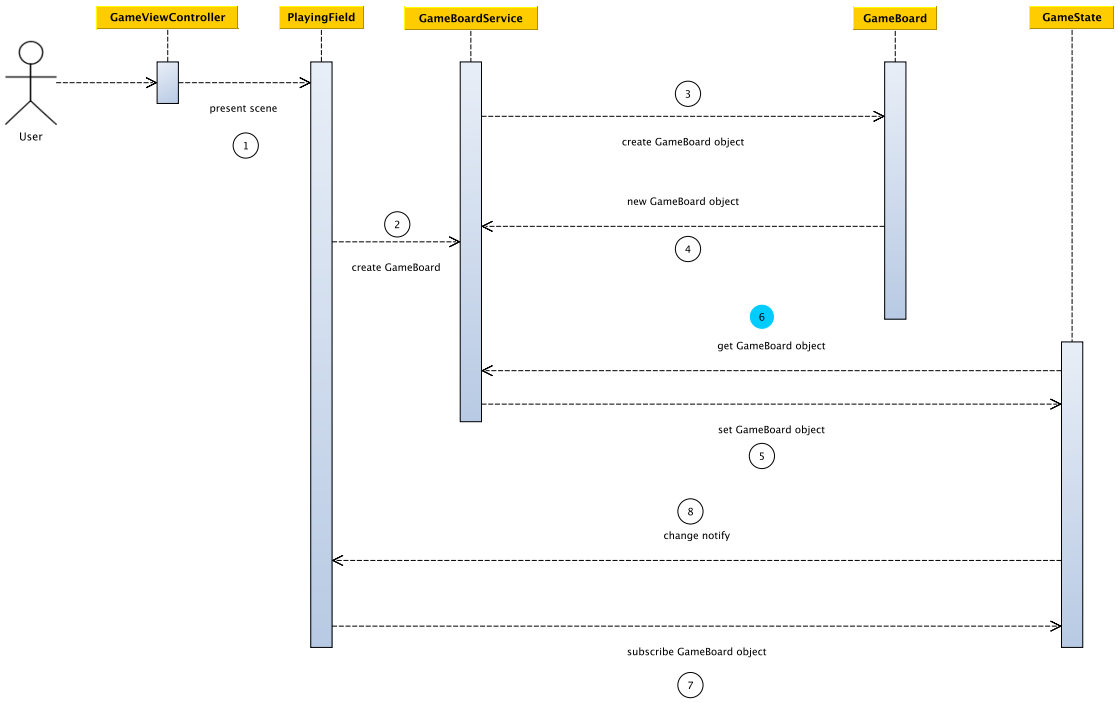


Abbildung 6.1: Sequenz für das Spielfeld.

Wenn das Programm geöffnet ist, wird von iOS der GameViewController initialisiert. 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*). Die erstellt das Spielfeld anhand von dem Model **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 in anderen Methoden vom Spielfeld benötigt, wird der Punkt (*6*) aufgerufen.

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

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

Durch diese Methode werden alle Objekte nebeneinander gesetzt. @todo: Geste anhand dieser Methode.

6.3 Steine

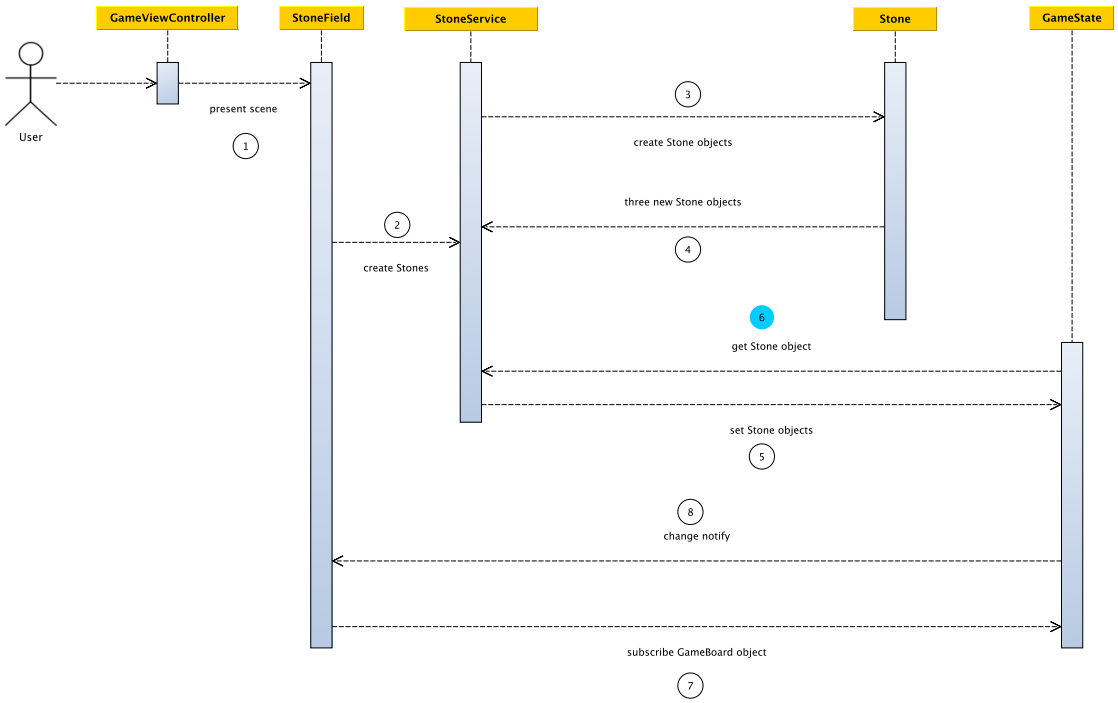


Abbildung 6.2: Sequenz für die Steine.

Die Kommunikation beim erstellen der Steine, entspricht im groben dem vom Spielfeld. Nachdem das Spielfeld kreiert wurde, werden die Steine erstellt. Dafür präsentiert der **GameViewController** das StoneField(*1*). Diese ruft den **StoneService**(*2*) auf und erstellt initial drei Steine und speichert diese in einem eindimensionalen Array im **GameState** ab(*3,4,5*). Hinterher registriert das **StoneField** sich 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 vorhanden 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 vorhanden Steine im Array erwünscht ist.

Auch hier wird die Methode vom Spielfeld in zur Positionierung der Bilder verwendet. Mit dem Unterschied, dass jeder Stein in einem Wrapper-Element[3] sich befindet. Im

[3]Ein Wrapper verpackt Elemente in sich.

Wrapper wird die Funktion aus dem Spielfeld benutzt und für das Positionieren der Wrapper-Elemente wurde eine Extra Funktion entwickelt. Diese ermittelt anhand des Indexes vom Array Element die Position.

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

Der Wert für den Wrapper vom Stein wird anhand von 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

Da nun das Spielfeld und die Steine vorhanden sind, 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** aufzurufen wird. Dabei werden alle berührten Koordinaten in einen Array im **GameState** gespeichert, jedoch nur von Beginn 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 von den Koordinaten, durch Subtraktion
3. Füllt einen leeren 4x4 Array
4. Bilde binären Wert (16 Bit)
5. Erstelle Bitmask
6. Vergleiche Bitmask mit vorhandenen Steinen

[4]Dieser Wert wurde durchs Probieren ermittelt.

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. Hinterher von den original Werten abgezogen. Mit diesen Koordinaten kann nun ein 4x4 Array gefüllt werden. Hierbei wird für jede Koordinate im 4x4 Array eine 1 gesetzt und alle anderen auf 0.

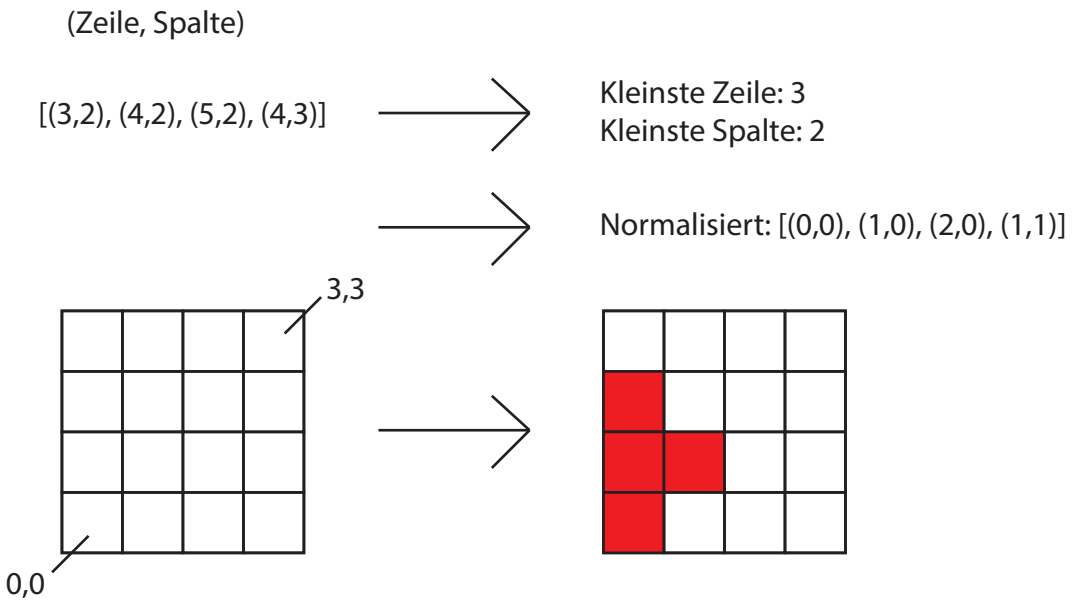


Abbildung 6.3: Stein normalisieren anhand einer Geste für einen T-Stein.

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) geendet. Zudem wird der Binärwert von hinten gefüllt. Der Grafik 6.3 ergibt es einen Binärwert von: 0000000100110001 und dezimal 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 vom Stein 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 gefüllt, muss diese geleert werden. Ein Algorithmus prüft nach jedem Zug das Spielfeld, ob eine

[5]Ein 4x4 Array wurde gewählt, da jeder Stein in jeder Ausrichtung in diesen Array reinpasst.

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 was für einen 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 von der Spalte bzw. von der Zeile.
Bei jedem Schleifendurchgang wird geprüft ob das Element im Array an dem Index der Zeile bzw. Spalte dem Wert der Zeilen- bzw. Spaltenanzahl entspricht (im Prototypen 9). Ist es der Fall, so ist es klar, dass diese Zeile und/oder Spalte komplett gefüllt ist. Da der Index wo der Wert 9 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 vorgegeben Steine mehr reinpasst das Spiel vorbei sein. Für den Algorithmus wurde auf der Abtastmethode aufgebaut. Dabei wird das gesamte Spielfeld nach jedem Zug abgetastet und 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) rüber 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 über das Spielfeld bewegt. Sobald beim **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 erste binäre Wert vom Stein mit dem aus dem Spielfeld

[6]Increment, immer Wert+1

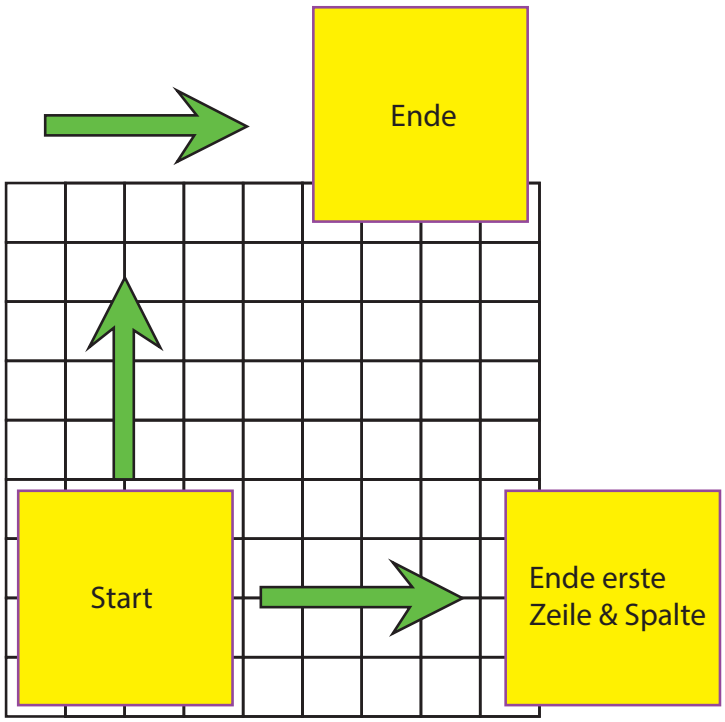


Abbildung 6.4: Abtastart über dem Spielfeld.

durch eine UND-Verknüpfung umgesetzt. Wenn das Ergebnis dem Ausgangswert vom Stein entspricht, handelt es sich hier um eine Möglichkeit den Stein an diese Stelle zu setzen. Andererseits läuft der Algorithmus weiter. Wenn bei diesem Ablauf das Ergebnis aus 3 nicht mit Stein übereinstimmt, zieht das Feld eine Spalte weiter. Läuft der Algorithmus durch ohne abzubrechen, wird daraus resultiert, dass das Spiel vorbei ist.

6.6.1 Pro

Durch diesen Ablauf kann garantiert werden, dass jeder Stein in jeder Variante geprüft werden kann 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 eine maximale Anzahl der Variationen, d.h. 12 Drehungen, für die

vorgebenden Steine. Dies geschieht jedoch nur, wenn der T-Stein 3 mal vorgegeben ist und es kein Platz für diesen Stein im Spielfeld gibt.

6.7 Spielmodus

Beim 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, wir beim Spielbeginn ein Timer gestartet. Dieser führt bei jeder Sekunde eine Methode aus, welche die vorgegebene Zeit von 120 Sekunden runter rechnet. Der **GameViewController** [5.10] beobachtet dabei den Wert und übermittelt der View den Stand. Je nach Stand des Countdowns wird ein Label[7] 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.

[7] iOS Element für ein Text auf dem Bildschirm.

7 Prototyp

7.1 Erfüllte Aufgaben

Im Prototypen wurden alle Ziele erreicht.

1. Das Spiel verfügt über eine simple und übersichtliche Architektur
2. Ein Spielfeld und die Steine werden vorgegeben.
3. Der Spieler kann einen beliebigen vorgebenden Stein, ohne diesen speziell zu markieren, rein malen.
4. Sind Reihen und/oder Spalten gefüllt, werden diese entfernt.
5. Der Benutzer bekommt Punkte fürs legen der Steine bzw. für gefüllte Zeilen/-Spalten.
6. Wenn die vorgegebenen Steine nicht reinpassen, 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.
- In allen Rotationen und allen Geräten von Apple können das Spiel nutzen.
- Sound-Effekte werden abgespielt, sowohl beim legen des Steines, als auch entfernen einer Reihe/Spalte.
- In-App-Käufe werden nun als Kauf angeboten.
- Der aktuelle Highscore wird im Apples-GameCenter gespeichert.
- Wenn innerhalb von 2 Spielzügen jeweils eine Reihe/Spalte gefüllt war, bekommt der Spieler Bonuspunkte.

7.2 Verbesserungen

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

- Design einbinden
- Schönere Animationen
- Multiplayer
- Levelsystem

Des weiteren könnte das Spiel durch Zeitdruck interessanter werden. Zum Beispiel durch, dass durch längeres nicht legen vom Stein Punktabzüge stattfinden. Alle Algorithmen funktioniert tadellos. Beim Untersuchen auf der Hardware kamen jedoch Probleme auf. 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 steht nicht außer acht.

8 Fazit

8.1 Swift

Swift ist eine schöne und saubere Sprache, jedoch zu jung um native 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.
- Manche Dokumentation wurde für Swift zum Zeitpunkt der Entwicklung nicht aktualisiert.
- 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 nochmal verbessert und eine Abhängigkeit von den Objektive-C Objekten wurde mehr entfernt. Zudem wurde die Dokumentation verbessert und aktualisiert. Auch das beobachten von einem Enumeration-Wert wurde realisiert. Ob nun die Sprache wirklich besser 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 ein Strich durch die Rechnung 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 Geld vom Unternehmen 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 klare Zugriffskontrolle einbauen müssen. Dies wurde jedoch zum Zeitpunkt des Prototypen ignoriert. Die folge davon ist, dass ein große Überarbeitung vom gesamten Code stattfinden müsste.

8.3 SpriteKit

SpriteKit bietet einem Spiele-Entwickler viele grundlegende Funktionen an, aus Zeitmangel wurden diese jedoch in dieser Arbeit wenig eingesetzt. Das Spiel wie es als Prototyp ist, hätte auch unabhängig von SpriteKit funktionieren können. Jedoch werden die Funktionen in Zukunft vom Unternehmen mehr 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 möglich für den Entwickler an jedem Punkt der Schleife Einfluss auf das Spielgeschehen zu nehmen. Zudem ist die Physik in SpriteKit für solche 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 alldem ist SpriteKit eine übersichtliche und gute 2D-Engine. Da diese von Apple weiter entwickelt und nativ unterstützt wird, kann das Spiel immer schnell von den Entwicklern auf das neue System angepasst und/oder erweitert werden.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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