HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Colin Dömer

Entwicklung einer digitalen Brettspielumsetzung von „Five Tribes“

Mittweida, 2018

Fakultät Angewandte Computer- und Biowissenschaften

BACHELORARBEIT

Entwicklung einer digitalen Brettspielumsetzung von „Five Tribes“

Autor:
Herr Colin Dömer

Studiengang:
Medieninformatik und Interaktives Entertainment

Seminargruppe:
MI14w3-B

Erstprüfer:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

Zweitprüfer:
M. Sc. Maik Benndorf

Einreichung:
Mittweida, 03.01.2018

Faculty Applied Computer Sciences & Biosciences

BACHELOR THESIS

Development of a Digital Board Game Version of “Five Tribes”

author:
Mr. Colin Dömer

course of studies:
Media Informatics and Interactive Entertainment

seminar group:
MI14w3-B

first examiner:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

second examiner:
M. Sc. Maik Benndorf

submission:
Mittweida, 03.01.2018

Bibliografische Angaben

Dömer, Colin:

Entwicklung einer digitalen Brettspielumsetzung von „Five Tribes“

Development of a Digital Board Game Version of “Five Tribes”

51 Seiten, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2018

Abstract

Diese Arbeit befasst sich mit der Entwicklung von digitalen Brettspielumsetzungen analoger Brettspiele. Dabei werden allgemeine Probleme erörtert und Lösungsansätze geboten, sowie am Beispiel von „Five Tribes“ eine eigene Implementierung genauer beschrieben. Erkenntnisse, die aus dem Entwicklungsprozess hervorgegangen sind, werden in allgemeiner Form festgehalten, sodass sie sich auf andere Projekte dieser Art übertragen ließen.

Inhaltsverzeichnis

Abbildungsverzeichnis IV

Abkürzungsverzeichnis VI

1 Digitale Umsetzung eines Brettspiels 1

1.1 Ausgangssituation und Problemstellung 1

1.2 Abgrenzung der Themenstellung 1

1.3 Aufbau der Arbeit 2

2 Begriffliche und theoretische Grundlagen 3

2.1 Klassifizierung von Gesellschaftsspielen 3

2.1.1 Unterscheidung nach Spielmaterial 3

2.1.2 Unterscheidung nach Spielcharakter 4

2.1.2.1 Eurogames 4

2.1.2.2 Amerigames 5

2.2 Digitale Brettspielumsetzungen 6

2.2.1 Onlineportale und Browsergames 6

2.2.2 Sandbox-Simulatoren 6

2.2.3 Mobile Apps 7

3 Mediale Unterschiede bei Brettspielen 9

3.1 Digitaler Mehrwert 9

3.2 Probleme bei der Digitalisierung 10

3.3 Digitale Umsetzungen von „Carcassonne“ 12

4 Das Brettspiel „Five Tribes“ 14

4.1 Spielmaterial und Spielaufbau 14

4.2 Ziel des Spiels 15

4.3 Spielablauf 15

4.4 Zugregeln 16

4.5 Aktionen 17

4.6 Sonderregeln 18

5 Technologien und Tools 19

5.1 Entwicklung mit Java 19

5.2 libGDX als Framework 19

5.3 Versionskontrolle mit Git 19

5.4 Zusätzliche Tools 20

5.4.1 TexturePacker 20

5.4.2 Hiero 20

6 Softwarearchitektur 21

6.1 Drei-Schichten-Architektur 21

6.1.1 Grafik / Präsentationsschicht 22

6.1.2 Logik / Steuerungsschicht 22

6.1.3 Daten / Domänenschicht 22

6.2 MVVM 22

6.3 Domänenmodell 23

6.3.1 Spielmaterial 24

6.3.2 Spielstruktur 24

6.3.3 Spielverlauf 24

6.4 Client-Server-Architektur 25

7 Implementierung 27

7.1 Asset-Beschaffung und Bearbeitung 27

7.2 Generische Klassen 28

7.2.1 Observer-Pattern 28

7.2.2 Factories 28

7.2.3 IndexedList und SyncedList 29

7.3 Event-System 31

7.3.1 GameController 31

7.3.2 Baiting 32

7.4 Implementierung der Spielregeln 33

7.4.1 Meeple-Zug-Logik 33

7.4.2 Regellücken 38

7.5 List- und Pile-ViewModel 39

7.6 Spieler-UI-Design 41

8 Erkenntnisse 42

8.1 Trennung von Spielmaterial und Spiellogik 42

8.2 Entwicklung im MVVM-Pattern 43

9 Ausblick 44

9.1 Implementierung von Sonderregeln 44

9.1.1 Dschinnkarten-Fähigkeiten 44

9.1.2 Rohstoffkarten Interaktionen 44

9.1.3 Offizielle Erweiterungen 45

9.2 Künstlicher Mitspieler 45

9.3 Verbesserungen im Design 48

9.3.1 Animation von Spielzügen 48

9.3.2 Simulierte Goldmünzen 49

9.3.3 Informationsfilter für den Spieler 49

9.3.4 Rückgängig machen von Aktionen 49

9.3.5 3D-Ansicht 50

10 Fazit 51

Literatur IX

Anhang XI

Selbstständigkeitserklärung XIX

Abbildungsverzeichnis

Abbildung 1: Meeple Finder Feature (Quelle: Asmodee Digital (2017)) 13

Abbildung 2: Screenshot vom Scoreboard in der Anwendung (Quelle: Days of Wonder (2014)) ... 15

Abbildung 3: Siegpunktrechnung aus der Spielregel (Quelle: Days of Wonder (2014)) 15

Abbildung 4: Zugregeln aus der Spielregel (Quelle: Days of Wonder (2014)) 16

Abbildung 5: Aktionen der Sippe aus der Spielübersicht (Quelle: Days of Wonder (2014)) 17

Abbildung 6: Aktionen der Plättchen aus der Spielübersicht (Quelle: Days of Wonder (2014)) 17

Abbildung 7: Punkte für Sets von Gütern (Quelle: Days of Wonder (2014)) 18

Abbildung 8: Drei-Schichten-Architektur (Quelle: B. Martin (2012) S. 6) 21

Abbildung 9: MVVM pattern (Quelle: Microsoft Corporation (o.J.)) 23

Abbildung 10: Netzwerkkommunikation (eigene Darstellung) 25

Abbildung 11: Moiré-Effekt Filterung (eigene Darstellung) 27

Abbildung 12: generische Factory-Klasse (eigene Darstellung) 29

Abbildung 13: Event-System mit MVVM (eigene Darstellung) 31

Abbildung 14: Parameter für "LastMeepleSameColor"-Algorithmus (eigene Darstellung) 34

Abbildung 15: "LastTile"-Check (eigene Darstellung) 34

Abbildung 16: Loop-Check (eigene Darstellung) 35

Abbildung 17: "LastTile in range"-Check (eigene Darstellung) 36

Abbildung 18: rautenförmiges Schachbrettmuster (eigene Darstellung) 37

Abbildung 19: ListViewModel-Konstruktor (eigene Darstellung) 39

Abbildung 20: ListViewModel-Layout (eigene Darstellung) 39

Abbildung 21: viele Meeples auf einem Plättchen (eigene Darstellung) 40

Abbildung 22: verdeckter Kartenstapel (eigene Darstellung) 40

Abbildung 23: Spielerinventar (eigene Darstellung) 41

Abbildung 24: Zugmöglichkeiten zu Spielbeginn (eigene Darstellung) 46

Abbildung 25: zwei Meeple-Bewegungen mit denselben Aktionen (eigene Darstellung) 47

Abbildung 26: Carcassonne im Browser (Quelle: Brettspielwelt GmbH (o.J.)) XI

Abbildung 27: alte Carcassonne App (Quelle: Exozet Potsdam GmbH (2016)) XI

Abbildung 28: neue Carcassonne App (Quelle: Asmodee Digital (2017)) XII

Abbildung 29: Carcassonne im Tabletop Simulator (Quelle: Berserk Games (2017)) XII

Abbildung 30: vereinfachtes Model-Klassendiagramm (eigene Darstellung) XIII

Abbildung 31: vereinfachtes Course-Klassendiagramm (eigene Darstellung) XIV

Abbildung 32: IndexedList-Klasse (eigene Darstellung) XV

Abbildung 33: SyncedList-Klasse (eigene Darstellung) XVI

Abbildung 34: canLastMeepleBeSameColor-Methode (eigene Darstellung) XVII

Abbildung 35: Screenshot vom Spiel (eigene Darstellung) XVIII

Abkürzungsverzeichnis

DoW	Days of Wonder
GUI	grafische Benutzeroberfläche (graphical user interface)
KI	künstliche Intelligenz
MVC	Model View Controller
MVVM	Model View ViewModel
UI	Benutzeroberfläche (user interface)

1 Digitale Umsetzung eines Brettspiels

Das Ziel dieser Bachelorarbeit ist die digitale Umsetzung eines analogen Brettspiels. Das Brettspiel, das dafür gewählt wurde, ist *Five Tribes* von Bruno Cathala, veröffentlicht 2014 vom Verlag *Days of Wonder*.

1.1 Ausgangssituation und Problemstellung

Five Tribes besitzt – zum Zeitpunkt dieser Arbeit im Dezember 2017 – keine alleinstehende digitale Umsetzung. Es gibt jedoch eine inoffizielle digitale *Five Tribes* Version für den *Tabletop Simulator*[1]. Diese besteht jedoch lediglich aus einer Ansammlung von Assets in einer virtuellen Umgebung, und umfasst keinerlei Regellogik. Scans der originalen Grafiken des Spiels ließen sich dieser Umsetzung entnehmen und konnten für die eigene Umsetzung verwendet werden. Der Verlag *Asmodee Editions*, welcher mit *Days of Wonder* zusammenarbeitet und für die digitalen Brettspieleversionen der DoW[2]-Spiele verantwortlich ist, hat sein Einverständnis gegeben, dass diese Grafiken im Rahmen der Bachelorarbeit verwendet werden dürfen.

1.2 Abgrenzung der Themenstellung

Es gibt verschiedene Anforderungen, die die Umsetzung erfüllen soll. Die Anwendung soll auf einem Windows-PC ausführbar sein und alle relevanten Spielkomponenten des analogen Brettspiels umfassen. Die Oberfläche soll den aktuellen Spielzustand zweidimensional darstellen und Benutzerinteraktion mittels der Maus entgegennehmen. Die Anwendung soll nur Spielzüge zulassen, die in den Spielregeln des Originals erlaubt sind. Diese möglichen Spielzüge sollen dem Spieler visuell hervorgehoben dargestellt werden. Das Spiel soll, über

[1] Siehe Kapitel 2.2.2

[2] Days of Wonder

ein Netzwerk verteilt, mit bis zu vier Spielern gespielt werden können. Im Code sollen Spieldaten und Spiellogik von der Darstellung getrennt sein und unabhängig davon funktionieren.

1.3 Aufbau der Arbeit

Zu Beginn der Arbeit wird ein Einblick in den Bereich der analogen und digitalen Brettspiele gegeben und diese in verschiedene Kategorien untergliedert. Anschließend werden verschiedene mediale Unterschiede und Probleme bei der Digitalisierung aufgezeigt und Lösungsansätze verschiedener Umsetzungen, am Fallbeispiel von *Carcassonne*, kurz analysiert.

Im Hauptteil der Arbeit wird zunächst ein Einblick in *Five Tribes* gegeben, und es werden die Regeln des Spiels erläutert. Danach wird kurz auf die Technologien und Tools eingegangen, die für die Realisierung des Projekts verwendet wurden. Im Kapitel Softwarearchitektur werden dann die verschiedenen Strukturen und Entwurfsmuster beschrieben, auf denen das Projekt basiert. Anschließend wird ein ausführlicherer Einblick in die Implementierung der Anwendung gegeben, in der auf verschiedene Klassen eingegangen wird, die auch für die Implementierung anderer Brettspiele anwendbar sind und sich somit auch auf andere Projekte übertragen ließen. Speziell für *Five Tribes* wird dann auch noch auf den Algorithmus eingegangen, der bestimmen soll, ob ein möglicher Teil-Zug[3] noch zulässige Folge-Züge hat.

Der Schlussteil der Arbeit befasst sich dann mit den verschiedenen Erkenntnissen, die bei der Entwicklung dieser Brettspielumsetzung gewonnen wurden und sich auf andere Projekte dieser Art übertragen ließen. Ebenso wird ein Ausblick gegeben, in welche Richtung die digitale Brettspielumsetzung von *Five Tribes* noch ausgebaut und verbessert werden kann, sowie potentielle Zusatzfeatures theoretisch erläutert. Abschließend wird ein Fazit über das Projekt gegeben, und festgestellt, in wieweit die Problemstellung der Arbeit gelöst wurde.

[3] In *Five Tribes* bestehen manche Spielzüge aus mehreren Teil-Zügen, die erst in der Summe einen gültigen Zug ausmachen. Daher muss vorrausschauen geprüft werden, ob ein Teil-Zug noch mögliche Folge-Züge hat, die dann zusammen einen regelkonformen Spielzug bilden.

2 Begriffliche und theoretische Grundlagen

Sowohl digitale Spiele als auch analoge Spiele lassen sich in verschiedene Kategorien unterteilen. Im Folgenden werden die wichtigsten Kategorien genannt und deren Besonderheiten hervorgehoben.

2.1 Klassifizierung von Gesellschaftsspielen

Brettspiele lassen sich anhand verschiedener Aspekte klassifizieren. Im Folgenden wird genauer auf eine Differenzierung anhand des Spielmaterials bzw. des Spielcharakters eingegangen.

2.1.1 Unterscheidung nach Spielmaterial

Ursprünglich umfasst der Begriff „Gesellschaftsspiele“ verschiedenen Unterkategorien, wie Brett-, Karten- oder Würfelspiele. Ebenso fallen unter diesen Begriff auch verschiedene Glücks-, Party- oder Geschicklichkeitsspiele. Für strategischen Gesellschaftsspiele, welche meist in Spielzüge unterteilt sind und einem klaren Ablauf folgen, wird im Englischen oft der Begriff *Tabletop Games* verwendet.[4] Das führt leicht zu einer gewissen Begriffsverwirrung, da im Deutschen *Tabletop* meist eine andere Bedeutung hat; der Begriff bezieht sich dort speziell auf Konfliktsimulationen mit realistischen Miniaturfiguren auf einer großen Landschaftsplatte. Diese Art Spiele unterscheiden sich meist deutlich von den klassischen, eher abstrakten Gesellschaftsspielen.

Diese klassischen abstrakten Gesellschaftsspiele, wie *Mensch ärger Dich nicht* oder *Skat*, lassen sich noch, anhand des Spielmaterials, in Kategorien wie Brett- und Kartenspiele unterteilen. In den letzten Jahrzehnten haben sich diese verschiedenen Elemente jedoch immer weiter vermischt, und es gibt immer mehr Spiele, die unterschiedliche Arten von Spielmaterial miteinander kombinieren. Unter anderem entstand in dieser Zeit *Carcassonne*[5], in dem die Hauptkomponente das Auslegen von Plättchen ist, und welches eine

[4] Vgl. Amelia Con (o.J.)

[5] Entwickelt von Klaus-Jürgen Wrede, veröffentlicht vom Hans-Im-Glück-Verlag, München, im Jahre 2000 und 2001 als „Spiel des Jahres“ ausgezeichnet

neue Art der Legespiele begründete. Neuere Spiele, wie *Five Tribes*, nutzen hingegen völlig verschiedene Elemente, wie ein modulares Spielfeld aus Plättchen und dazu noch verschiedene Karten unterschiedlicher Funktion, wodurch eine Kategorisierung nach Spielmaterial nicht mehr sinnvoll erscheint. Aus diesem Grund wird in dieser Arbeit der Begriff *Brettspiel* als Überbegriff für die verschiedenen Kategorien von Gesellschaftsspielen verwendet, und es wird nicht weiter nach Karten-, Würfel- oder Legespielen differenziert. Diese Verallgemeinerung gibt es oft auch im Englischen, unter dem Begriff „board game".

2.1.2 Unterscheidung nach Spielcharakter

In der einschlägigen Spieleszene wird meist eine Unterteilung in Euro- und Amerigames getroffen.[6] Diese Unterscheidung orientiert sich eher am Spielcharakter.

2.1.2.1 Eurogames

Eurogames, manchmal auch als Autorenspiele oder Designerspiele bezeichnet, sind moderne Gesellschaftsspiele, die von Autoren bzw. Spieledesignern entworfen und anschließend von einem Verlag verlegt wurden. Der Name des Spieleentwicklers steht dabei im Vordergrund und wird auch immer auf der Spieleverpackung angegeben. Bei älteren Spielen hingegen stand der eigentliche Erfinder meist im Hintergrund und das Spiel wurde stattdessen über den Namen des Verlags vermarktet. Spiele waren meist vom Verlag beauftragt und in dessen Namen entwickelt worden, während dieser Prozess bei modernen Eurogames von den Spieleautoren selbst ausgeht. Dadurch sind die Autoren deutlich mehr in den Fokus gerückt und die Spiele verkaufen sich meist über die Bekanntheit des Autors in der Spieleszene. Einer der bekanntesten frühen Spieleautoren ist Klaus Teuber, welcher mit *Die Siedler von Catan* schon 1995 eines der bekanntesten Eurogames entwickelte. Da die frühe Phase der Eurogames seinen Start in Deutschland hatte, werden diese Spiele im internationalen Raum auch oft als *German-style Games* bezeichnet.

Es gibt verschiedene spielerische Eigenschaften, durch die sich Eurogames von den Amerigames unterscheiden. Im Fokus des Spiels steht meist die Spielmechanik, welche von der Thematik und der Grafik des Spiels unterstützt wird. Das Spielmaterial ist hochwertig pro-

[6] Die Spieledatenbank *BoardGameGeek* führt diese auch unter den Begriffen „german game" und „Amerit-rash" in ihrem Glossar auf (vgl. BoardGameGeek, LLC. (o.J.)) – Internetseiten wie diese sind mangels wissenschaftlicher Quellen hier unentbehrlich.

duziert und es wird oft abstraktes Holzmaterial verwendet, was aus einfachen und wiederverwendbaren Formen besteht und erst durch die Spielregel seine Bedeutung erhält. Beispiel dafür sind die Meeples, abstrakte Holzfiguren, welche in unterschiedlichsten Spielen Verwendung finden. Außerdem sammeln die Spieler im Verlauf des Spiels häufig auf vielerlei Weise Punkte, die ihnen am Ende zum Sieg verhelfen. Alle Spieler nehmen dabei am gesamten Spiel teil und können nicht vorzeitig ausscheiden.

Five Tribes fällt in den Bereich der Eurogames. Es wurde von Bruno Cathala, einem etablierten französischen Spieleautor, entwickelt und besitzt verschiedene Merkmale eines Eurogames. Dazu zählt unter anderem die zentrale Spielmechanik für die Bewegung von Meeples über das Spielfeld, welche zwar aus nur wenigen Regeln besteht, aber trotzdem eine große Spieltiefe erzeugt.

2.1.2.2 Amerigames

Amerigames, auch als thematische Spiele oder „Ameritrash“ bezeichnet, sind Gesellschaftsspiele, bei denen der Konflikt zwischen den Spielern im Vordergrund steht und die Thematik die Spielmechanik bestimmt. Die Bezeichnung „Ameritrash“, welche meist in Spieleforen verwendet wird, ist offensichtlich abwertend gegenüber diesem Bereich der Spiele, und wird meist von Vertretern der Eurogames verwendet, die Amerigames nicht als richtige Brettspiele ansehen. Da diese Art Spiele ursprünglich aus den USA stammte, werden diese Spiele auch manchmal als *American-style Games* bezeichnet.

Im Gegensatz zu Eurogames sind Amerigames meist an ein bestimmtes Thema gebunden und simulieren einen historischen oder imaginären Konflikt, wobei die Spieler die verschiedenen Konfliktparteien vertreten. Die Spieldauer kann sich dabei oft auf mehrere Stunden belaufen und das vorzeitige Ausscheiden von Spielern ist möglich und teilweise auch für den Sieg des letzten verbleibenden Spielers erforderlich. Außerdem gibt es in solchen Spielen meist einen hohen Glücksfaktor. Beispiele dafür sind die analogen Spiele aus dem *Warhammer* Universum, welche einen starken thematischen Fokus haben und deren Spielmaterial aus einer großen Zahl an individuellen und detailliert gestalteten Plastikfiguren besteht.

Da *Five Tribes* eindeutiges Eurogame ist, wird mit dem Begriff „Brettspiel“ quasi als Synonym für Eurogames verwendet, da sich viele Erkenntnisse vermutlich nicht allgemein auf die Amerigames übertragen lassen.

2.2 Digitale Brettspielumsetzungen

Digitale Brettspielumsetzungen lassen sich in verschiedene Kategorien unterteilen. Neben eigenständigen Anwendungen, die auf den PC heruntergeladen und, unabhängig von anderer Software, ausgeführt werden können, gibt es noch drei weitere Arten von Implementierungen. Diese unterscheiden sich meist durch das Medium, in dem das Spiel präsentiert wird. Ein weiteres Merkmal der verschiedenen Umsetzungsarten ist die vorhandene oder fehlende Regelgebundenheit der Spiele.

2.2.1 Onlineportale und Browsergames

Es gibt verschiedenen Umsetzungen von Brettspielen, die sich im Browser spielen lassen. Dazu zählen die Onlineportale *Brettspielwelt* und *Yucata*, auf denen man eine Sammlung von verschiedensten Brettspielen, kosten- und werbefrei, online miteinander spielen kann. Die Brettspiele sind dort in einer zweidimensionalen Oberfläche realisiert und verwenden die originalen Grafiken der Spiele. Das Interface ist sehr schlicht gehalten und auf das Wesentliche reduziert.[7] Spielablauf und Logik sind dabei fest ins Spiel integriert und der Spieler kann nur Züge durchführen, die regeltechnisch auch erlaubt sind. Die Spielzüge sind simpel bis gar nicht animiert, und es gibt kaum visuelle Effekte. Aufgrund der optischen Abstriche ist es den Betreibern möglich, ein relativ großes Angebot an Spielen kostenlos zur Verfügung zu stellen. Lange Zeit gab es dort auch das erfolgreiche Kartenspiel *Dominion*, welches jedoch seit 2015 eine alleinstehende und kommerzielle Umsetzung für den Browser erhalten hat, in Form von *Dominion Online*.

2.2.2 Sandbox-Simulatoren

Eine weitere Möglichkeit für Brettspielumsetzungen sind Sandbox-Simulatoren. Dabei wird das Spiel in einer dreidimensionalen, teilweise physikbasierten, virtuellen Umgebung erstellt, einer sogenannten Sandbox.[8] Beispiele für solche Simulatoren sind der *Tabletop Simulator* sowie *Tabletopia*. Beide werden unter anderem über die Onlinevertriebsplattform *Steam* kostenpflichtig angeboten, und umfassen ebenfalls eine Ansammlung verschiedener

[7] Siehe *Abbildung 26: Carcassonne im Browser* im Anhang

[8] Siehe *Abbildung 29: Carcassonne im Tabletop Simulator* im Anhang

Spiele. Eine Besonderheit ist dabei jedoch, dass diese Spiele lediglich aus dem Spielmaterial bestehen und keinerlei Regeln implementiert sind. Um das Spiel zu spielen, müssen also alle Spieler bereits mit den Spielregeln vertraut sein und diese selbst befolgen. Das Spielmaterial muss manuell mit der Maus über das Spielfeld bewegt werden, und Verwaltungsaktionen, wie das Anpassen von Punkteleisten und das Nachfüllen von Vorräten, muss ebenfalls manuell vorgenommen werden. Durch den Verzicht auf eine automatisierte Spiellogik können auch Nutzer ohne Programmierkenntnisse selbstentwickelte oder bereits existierende Brettspiele in der Sandbox-Umgebung umsetzen. Universelle Brettspielelemente, wie zum Beispiel zweiseitige Spielkarten, Holzwürfel oder Meeples, werden dabei als Modell mit ein paar Standardfunktionen zur Verfügung gestellt. Diese ermöglichen es beispielsweise, Spielkarten zu einem Stapel zu gruppieren, diesen zu mischen, und Karten an die verdeckte Kartenhand eines Spielers auszuteilen. Um ein neues Brettspiel umzusetzen, müssen also lediglich die Grafiken für das Spielmaterial hochladen werden, und den entsprechenden Standard-Spielmaterial-Modellen in der Sandbox zugeordnet werden. Der *Tabletop Simulator* ermöglicht es allen Spielern, ihre Brettspielprojekte über den *Steam Workshop* mit der Community zu teilen. Beim Hochladen von originalen Brettspielen handeln die Nutzer dabei unter Umständen in einer rechtlichen Grauzone, wenn die Erlaubnis von den Verlagen für die Nutzung des Spiels und dessen Grafiken meist nicht eingeholt wurde.

2.2.3 Mobile Apps

Für den mobilen Markt gibt es verschiedene offizielle Brettspielumsetzungen. Ein Beispiel dafür ist *Carcassonne*, auf das im nächsten Kapitel noch genauer eingegangen wird.[9] Diese Brettspiel-Apps werden über den *Google Play Store* und den *App Store* kostenpflichtig vertrieben. Da diese Apps professionell produziert werden, gibt es selten eine kostenfreie Version und auch Erweiterungen müssen noch separat dazugekauft werden. Die Umsetzung ist dafür sehr hochwertig und professionell, und die Spielregeln sind ebenfalls fest integriert. Oft haben diese Apps auch deutlich mehr visuelle Effekte als die der Onlineportale und eine modernere Oberfläche.[10] In vielen Fällen gibt es neben einem Einzelspielemodus gegen

[9] Siehe Kapitel 3.3

[10] Siehe *Abbildung 27: alte Carcassonne App* und *Abbildung 28: neue Carcassonne App* im Anhang

eine KI[11] auch einen Mehrspielermodus in dem, lokal auf einem Gerät oder über das Internet, miteinander gespielt werden kann. Probleme mancher App-Umsetzungen, sind die zu kleine Darstellung auf dem Mobilgerät, weshalb sich manche Spiele nur auf Tablets mit einem größeren Bildschirm angenehm spielen lassen. Die eigene Brettspielumsetzung von *Five Tribes* wurde nicht auf Mobilgeräte portiert, da bei diesem Spiel die Übersicht über das Spielfeld essentiell ist, und diese Übersichtlichkeit nicht für das kleine Display eines Smartphones gewährleistet werden kann.

[11] künstliche Intelligenz

3 Mediale Unterschiede bei Brettspielen

Es gibt verschiedene, medial bedingte Unterschiede zwischen analogen und digitalen Brettspielumsetzungen. Der offensichtlichste Unterschied ist die fehlende Haptik bei digitalen Umsetzungen. Während das analoge Spielmaterial greifbar ist, ist das Spielmaterial im digitalen lediglich als eine Ansammlung von farbigen Pixeln dargestellt und Interaktionen finden meist über die Maus oder den Touchscreen statt. Durch das Herunterbrechen des dreidimensionalen Brettspiels auf eine zweidimensionale Oberfläche geht auch die optische Tiefe verloren, die bei analogen Brettspiel vorhanden ist. Dadurch lassen sich Objekte schwieriger voneinander unterscheiden, während bei den analogen Brettspielen meist schon durch die Wahl des unterschiedlichen Spielmaterials bestimmte Elemente hervorgehoben werden können. Dies ist beispielsweise bei den Meeples aus Holz der Fall, welche sich deutlich von flachen Plättchen unterscheiden und dadurch gut zu erkennen sind.

Auch eine gewisse zwischenmenschliche Komponente geht bei digitalen Brettspiele meist verloren, da die Spieler sich gegenseitig nicht sehen und auch oft nicht hören können, und der einzige Kommunikationsweg meist ein Text-Chat ist. Während bei den analogen Brettspielen für viele Spieler auch das Miteinander im Vordergrund steht, und die Spieler gegenseitig Reaktionen auf das Spielgeschehen in Mimik und Ausdruck wahrnehmen können, fällt dieser Aspekt bei den digitalen Umsetzungen vollkommen weg. Besonders verstärkt ist dies beim Spielen mit zufälligen Mitspielern, die man persönlich nicht kennt. Statt der sozialen Komponenten steht dann das Spiel mehr im Vordergrund, und es geht dabei dann auch meist mehr um das Gewinnen als um den gemeinsamen Spaß. Durch die fehlende Bindung zu den anderen Spielern, werden Partien, in denen die eigene Niederlage schon absehbar ist, deutlich frustrierender, da man sich kaum für den anderen Spieler freuen wird, zu dem man keine persönliche Bindung hat, wie es bei analogen Brettspielen sonst der Fall ist.

3.1 Digitaler Mehrwert

Digitalen Brettspielumsetzung besitzen auch verschiedene Vorteile gegenüber der analogen Brettspiele. Der Spielaufbau, der in manchen Spielen ein erheblichen Zeitaufwand mit sich bringt, wird komplett automatisiert vom Programm übernommen. Auch jeglicher Verwaltungsaufwand im Spiel, wie das Ziehen von Karten oder Auffüllen von Auslagen, kann

in einer digitalen Umsetzung vollautomatisch durchgeführt werden. Die Spieldauer einer Partie kann dadurch deutlich reduziert werden, und besteht nur noch aus der tatsächlichen Bedenkzeit der Spieler. Auch brauchen die Spieler die Auswertungen von Aktionen nicht mehr gründlich durchzugehen und auf Sonder- und Kleinregeln zu prüfen, da dies alles vom Programm übernommen wird und nicht regelkonforme Spielzüge gar nicht erst möglich sind. Die Spieler brauchen also nicht mehr die Züge der Mitspieler im Auge zu behalten, um sicher zu gehen, dass diese keine Spielregeln brechen.

Ein weiterer großer Vorteil von digitalen Brettspielumsetzungen ist die Ortsunabhängigkeit. Die meisten digitalen Spiele können über das Internet gespielt werden, und erfordern nicht die Anwesenheit aller Spieler an einem Ort. Auch ist es dadurch möglich mit völlig fremden Mitspielern zu spielen, und sich dadurch mit verschiedenen Spielertypen zu messen. Oft können diese Spiele auch für einen längeren Zeitraum unterbrochen und zu einem späteren Zeitpunkt weitergespielt werden. Während bei analogen Brettspielen der Tisch dann belegt ist, kann in einer digitalen Umsetzung auch jederzeit eine weitere Partie gestartet werden und es können auch mehrere Partien parallel gespielt werden.

3.2 Probleme bei der Digitalisierung

Verschiedene Aspekte der analogen Brettspiele lassen sich nur schwer digitalisieren. Um eine digitale Brettspielumsetzung mit festen Regeln zu realisieren, muss zunächst die Spielregel analysiert und für die Ablaufsteuerung abstrahiert werden. Dabei kann es vorkommen, dass die Spielregel nicht absolut eindeutig ist und sehr unwahrscheinliche Sonderfälle nicht klar definiert sind. Für die Implementierung ist es jedoch notwendig, dass jeder mögliche Fall, so unwahrscheinlich dessen Eintreten auch sein mag, von den Regeln abgedeckt ist. Es kann dann erforderlich sein, über die Originalregel hinaus, eigene Ergänzungen für Sonderfälle vorzunehmen, wie es auch bei *Five Tribes* der Fall war.[12]

Da digitale Spiele meist auch mit zufälligen Mitspielern gespielt werden können, fehlt die Bindung zwischen den Spielern, und dadurch die Verantwortung das Spiel zügig zu Ende zu spielen. Um sicher zu gehen, dass eine Partie zwischen Spielern, die sich nicht kennen, vorankommt, muss ein gewisses Zeitlimit gesetzt werden, innerhalb dessen jeder Spieler seinen Zug ausführen muss. Entsprechend muss auch bestimmt werden, was passiert,

[12] Siehe Kapitel 7.4.2

wenn dieses Zeitlimit überschritten wird. Es gibt verschiedene Möglichkeiten dieses Problem zu lösen. Am einfachsten ist es in einem solchen Fall, das gesamte Spiel abzubrechen, was bei mehr als zwei Spielern jedoch eventuell ein gutes Spiel zwischen zwei unbeteiligten Spielern zerstören könnte. Eine weitere Möglichkeit ist es, den Spieler, der seine Zugzeit überschritten hat, durch einen künstlichen Mitspieler zu ersetzen. Dieser sollte jedoch nicht zu stark oder zu schwach für die verbleibenden Mitspieler sein, um das Spiel nicht plötzlich zum Vorteil eines Spielers zu entscheiden. Noch eine Möglichkeit, dieses Problem zu umgehen, ist das Festlegen von Standard-Zügen in den entsprechenden Entscheidungssituationen, was in den meisten Fällen bedeuten würde zu passen. Ist ein Passen in einer Situation den Regeln nach jedoch nicht erlaubt, muss ein anderer Zug gefunden werden, der regelkonform ist, und dem Spieler, der sein Zeitlimit überschritten hat, möglichst keinen Vorteil gibt. Ein angemessenes Vorgehen ist von Spiel zu Spiel unterschiedlich und muss dem jeweiligen Projekt entsprechend entschieden werden.

Eine Besonderheit der analogen Spiele ist das nicht perfekte Mischen von Karten. Wenn Karten von Hand gemischt werden, handelt es sich dabei meist nur um ein blockweises Umsortieren der Karten, wobei die Blockgröße vom Mischstil der Person abhängt. In Spielen wie *Dominion*, wo der eigene Kartenstapel viele Male in einer Partie gemischt wird, kann es einen bedeutsamen Unterschied ausmachen, ob die Karten per Hand gemischt werden, oder digital in eine zufällige Reihenfolge gebracht werden. Beim digitalen Mischen werden die Elemente zufällig in eine komplett neue Reihenfolge gebracht, wobei jedes Mischergebnis – in der Theorie – die gleiche Wahrscheinlichkeit hat. Beim analogen Mischen hingegen gibt es häufig kleine Blöcke an Karten, die ihre Reihenfolge untereinander nicht ändern und lediglich als einheitlicher Block im Stapel ihre Position ändern. Es wird dann sehr häufig vorkommen, dass der Spieler mehrfach dieselbe Kombination an Karten zieht, wie vor dem Mischen. Es gilt zu entscheiden ob diese Besonderheit der analogen Brettspiele erstrebenswert ist und auch in einer digitalen Umsetzung simuliert werden sollte.

3.3 Digitale Umsetzungen von „Carcassonne“

Carcassonne[13] ist ein sehr erfolgreiches Legespiel aus dem Jahre 2000, welches 2001 zum Spiel des Jahres gewählt wurde. Auf Grund des großen Erfolgs gab es, neben vielen verschiedenen Erweiterungen und unterschiedlichen Neuauflagen des Spiels, auch mehrere digitale Umsetzungen von *Carcassonne*. Außer einer schlichten Browserumsetzung in der *Brettspielwelt* und mehreren Umsetzungen für den *Tabletop Simulator*, gibt es auch zwei professionelle Umsetzungen als App für den mobilen Markt.[14] Die ältere App, von *Exozet*, ist jedoch nicht mehr im *Google Play Store* verfügbar, da die Rechte für eine App-Umsetzung von Carcassonne, im November 2017, an die Firma *Asmodee Digital* übergeben wurden, welche daraufhin die neue offizielle *Carcassonne* App veröffentlicht hat.[15]

Neben den grafischen Unterschieden zwischen den verschiedenen Umsetzungen, gibt es auch unterschiedliche Herangehensweisen an die Lösung der Probleme, die mit der Digitalisierung eines Brettspiels einhergehen. Beispielsweise wurde das Problem des sich schlecht voneinander abhebenden Spielmaterials von *Asmodee Digital* erkannt, weshalb ein optionales *Meeple Finder Feature* hinzugefügt wurde, mit dem die Meeples durch einen Sprungeffekt hervorgehoben werden können.[16] Außerdem bietet die neue *Carcassonne* App, im Gegensatz zum Vorgänger, neben der zweidimensionalen Ansicht auch eine dreidimensionale Perspektive, in der die Meeples optisch stärker hervorstechen.

---

[13] Folgender Text fasst das Spiel prägnant zusammen: „Carcassonne is a tile-placement game in which the players draw and place a tile with a piece of southern French landscape on it. The tile might feature a city, a road, a cloister, grassland or some combination thereof, and it must be placed adjacent to tiles that have already been played, in such a way that cities are connected to cities, roads to roads, etcetera. Having placed a tile, the player can then decide to place one of his meeples on one of the areas on it: on the city as a knight, on the road as a robber, on a cloister as a monk, or on the grass as a farmer. When that area is complete, that meeple scores points for its owner.“ (BoardGameGeek, LLC. (o.J.))

[14] Siehe *Abbildung 26* bis *Abbildung 29* im Anhang

[15] Die Entwicklungszeit dieser neuen *Carcassonne* App von Asmodee Digital belief sich laut offiziellen Angaben auf über ein Jahr. Es wurden in der App 73 Mitwirkende aufgeführt, zu denen auch 9 Programmierer und 13 Designer/Artists zählen. Bei der alten *Carcassonne* App von Exozet wurden ebenfalls 9 Programmierer angegeben.

[16] Siehe *Abbildung 1: Meeple Finder Feature*

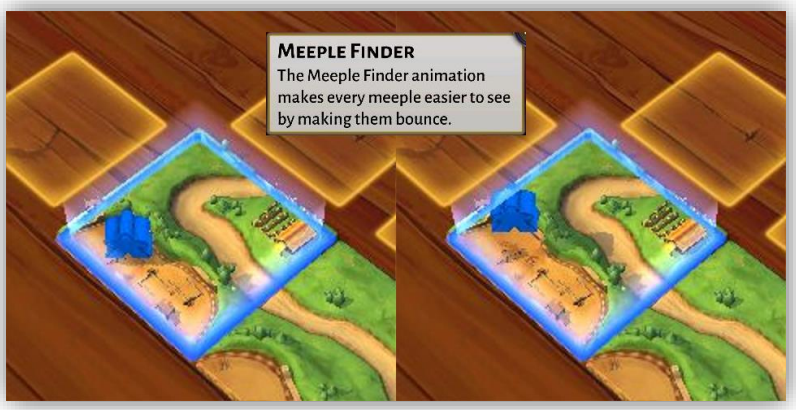

Abbildung 1: Meeple Finder Feature (Quelle: Asmodee Digital (2017))

Auch die Möglichkeiten, die die digitale Welt bietet, wurden in der neuen *Carcassonne* App vorteilhaft verwendet. So wurde beispielsweise ein optionaler Filter hinzugefügt, der zusammenhängende Wiesen hervorgehoben darstellt, da diese meist sehr verworren sein können.[17] Wie bei der Vorgänger-App auch schon, kann der Spieler Einsicht auf die noch verbleibenden Plättchen haben; der Ausgangsvorrat an Plättchen ist in jeder Partie identisch. Somit wird Wissen, das jedem Spieler theoretisch auch beim analogen Brettspiel gleichermaßen zur Verfügung stehen würde, direkt abrufbar und es muss nicht, auf Kosten der Spielzeit, von jedem Spieler selbst ermittelt werden, welche Plättchen denn noch fehlen und somit noch übrig sein müssen. Außerdem wird in der App gegebenenfalls angezeigt, wenn es kein passendes Plättchen mehr für eine bestimmte Anlegestelle gibt. Diese beiden Features sind jedoch optional und können auch deaktiviert werden, wenn die Spieler meinen, dass ihnen dadurch das Spielgefühl des Originals verloren geht.

[17] Siehe *Abbildung 28: neue Carcassonne App* im Anhang

4 Das Brettspiel „Five Tribes“

Five Tribes ist ein Brettspiel von Bruno Cathala und wurde 2014 vom Verlag *Days of Wonder* veröffentlicht. Das Spiel ist für 2 bis 4 Spieler ausgelegt und befindet sich, Stand Dezember 2017, in den Top 50 der bestbewerteten Spiele auf der Spieledatenbank *BoardGameGeek*.[18]

Five Tribes wurde für die digitale Umsetzung im Rahmen dieser Arbeit gewählt, da es zum einen ein komplexes Spiel mit einer interessanten Mechanik ist, und es außerdem bisher noch keine digitale Umsetzung für dieses Spiel gibt.

4.1 Spielmaterial und Spielaufbau

Das Spielmaterial setzt sich aus verschiedenen Elementen zusammen. 30 Plättchen bilden zusammen ein modulares Spielfeld, das Sultanat. Sie werden zu Beginn des Spiels als Rechteck ausgelegt und anschließend nicht mehr bewegt. Auf diesen befindet sich eine Punktzahl für den letztendlichen Besitzer und eine von 5 verschiedenen Plättchen-Aktionen. Die 90 Meeples, in 5 verschiedenen Farben, welche die 5 Sippen (Tribes) im Spiel repräsentieren, werden ebenfalls zu Spielbeginn auf die Plättchen im Sultanat verteilt. Außerdem gibt es in *Five Tribes* noch 2 verschiedene Arten von Karten, die Dschinnkarten und die Rohstoffkarten. Die 22 Dschinnkarten haben jeweils eigene Sonderfunktionen, die die Spielregeln erweitern oder verändern können. Ebenso haben sie eine Punktzahl für die Wertung am Ende des Spiels. Die Rohstoffkarten hingegen unterteilen sich in Güter und Sklaven bzw. Fakire[19]. Sklaven bzw. Fakire können genutzt werden, um Aktionen zu verstärken, während Güter pro verschiedener Art Punkte bringen. Aus den Dschinn- und Rohstoffkarten wird jeweils eine offene Auslage mit einem verdeckten Nachziehstapel gebildet. Die Spieler erhalten jeweils Kamele und einen Zugmarker (bei zwei Spielern zwei). Mit den Kamelen kann der Spieler den Besitz eins Plättchens im Sultanat markieren, während mit

[18] Vgl. BoardGameGeek, LLC. (o.J.)

[19] In der Erstauflage des Spiels wurden Sklavenkarten genutzt, welche zusammen mit den Gütern die Rohstoffkarten bildeten. Dies wurde jedoch von manchen Spielern stark kritisiert und führte dazu, dass diese durch die neuen Fakirkarten ausgetauscht wurden. Da Sklavenkarten jedoch, wie auch bereits vom Verlag beschrieben, besser in das thematische Setting des Spiels passen, wurde im Programm weiterhin mit dem Begriff der Sklaven gearbeitet, wobei diese in der Oberfläche jedoch als Fakire dargestellt wurden. (vgl. W. E. Martin (2015))

dem Marker die Biet- bzw. Zugreihenfolge auf einer entsprechenden Leiste angezeigt wird. Außerdem erhalten die Spieler zu Beginn 50 Gold in Form von Münzen, die der Spieler für verschiedene Aktionen bezahlt und bekommet, und die am Spielende als Punkte angerechnet werden. Außerdem gibt es noch Palmen und Paläste, die zu Beginn des Spiels beiseitegelegt werden und im Laufe des Spiels auf die Plättchen gestellt werden, um zusätzliche Siegpunkte zu bringen.

4.2 Ziel des Spiels

Das Ziel des Spiels ist es, bei Spielende die meisten Punkte zu haben. Punkte erhalten die Spieler durch verschiedene Bereiche:


- **1 SP** je 1 Gold in deinem Besitz.
- **1 SP** für jeden Wesir (gelber Meeple) in deinem Besitz + 10 SP für jeden Mitspieler, der weniger Wesire hat als du.
- **2 SP** für jeden Ältesten (weißer Meeple) in deinem Besitz.
- **Die Summe der SP** aller deiner Dschinn.
- **Die Summe der SP** aller deiner Plättchen (also die, auf denen du Kamele stehen hast).
- **3 SP** für jede Palme auf deinen Plättchen.
- **5 SP** für jeden Palast auf deinen Plättchen.
- **Die Summe der SP** jedes Sets unterschiedlicher Güter (ohne Sklaven) in deinem Besitz.

Abbildung 3: Siegpunktrechnung aus der Spielregel (Quelle: Days of Wonder (2014))

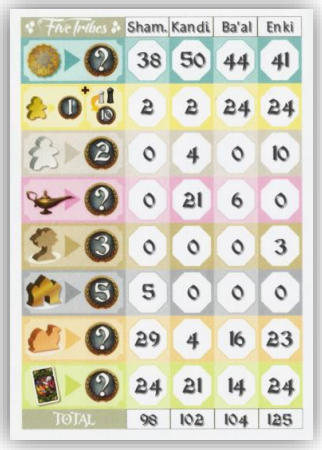

Abbildung 2: Screenshot vom Scoreboard in der Anwendung (Quelle: Days of Wonder (2014))

Das Spiel endet, wenn ein Spieler sein letztes Kamel auf ein Plättchen gesetzt hat, oder wenn keine zulässige Meeple-Bewegung mehr möglich ist.[20]

4.3 Spielablauf

Das Spiel untergliedert sich in Runden, welche jeweils aus einer Biet- und einer Zugphase bestehen, in denen jeder Spieler einmal an der Reihe ist (bei zwei Spielern zweimal). In der Bietphase bieten die Spieler jeweils um die Zugreihenfolge in der Zugphase. Dabei stellt der Spieler, beginnend mit dem Spieler, der in der letzten Runde am meisten geboten hat,

[20] Vgl. Spielregel von *Five Tribes* (Days of Wonder (2014))

seinen Zugmarker auf ein Feld der Zugreihenfolge-Leiste und zahlt den entsprechenden Betrag. Setzen mehrere Spieler beim Bieten auf 0 Gold (dieses Feld gibt es drei Mal), werden jeweils die bereits dort befindlichen Marker nach hinten geschoben.

Anschließend führen die Spieler ihren Zug aus, beginnend mit dem Spieler, der am meisten geboten hat. Der Zug des Spielers besteht dabei aus der Meeple-Bewegung und dem Ausführen der Sippen- und Plättchen-Aktion. Nachdem alle Spieler ihren Zug durchgeführt haben, wird die Auslage an Dschinn- und Rohstoffkarten wieder aufgefüllt und die nächste Runde beginnt.

4.4 Zugregeln

Um eine Meeple-Bewegung durchzuführen wählt der Spieler ein Plättchen aus, und nimmt die darauf befindlichen Meeples in die Hand. Anschließend bewegt er diese Meeples über das Spielfeld, wobei er auf jedem Plättchen, das er überquert, einen Meeple absetzen muss. Er kann also so viele Plättchen weit ziehen, wie sich Meeples auf dem Startplättchen befunden haben. Für das Bewegen der Meeples gibt es jedoch drei entscheidende Regeln:

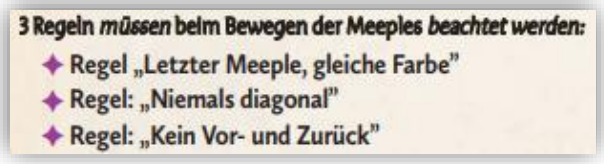

Abbildung 4: Zugregeln aus der Spielregel (Quelle: Days of Wonder (2014))

Der Spieler kann die Meeples nur vertikal und horizontal über das Spielfeld bewegen, niemals diagonal. Der Spieler darf nicht direkt auf ein Plättchen zurückziehen, von dem er gerade gekommen ist. Sich in einem Kreis zu bewegen ist jedoch zulässig, sofern der Spieler genug Meeples dafür hat. Die wichtigste Regel ist jedoch, dass der Spieler auf einem Plättchen enden muss, auf dem sich bereits ein Meeple befindet, der die selbe Farbe hat, wie der letzte Meeple, den der Spieler auf diesem Feld absetzt. Somit kann ein Zug niemals auf einem leeren Feld enden. Es kann jedoch über ein leeres Feld hinweggezogen werden, worauf dann ein Meeple abgesetzt wird.

Der Spieler nimmt anschließend alle Meeples von dem letzten Plättchen, die die selbe Farbe wie der zuletzt gesetzte Meeple haben, auf die Hand (inklusive des gerade drauf abgesetzten Meeple). Befinden sich nun keine Meeples mehr auf dem Plättchen, kann der

Spieler das Plättchen in Besitz nehmen und stellt eines seiner Kamele darauf. Sollten im späteren Verlauf des Spiels wieder Meeples auf diesem Plättchen landen, kann zwar jeder Spieler auch auf diesem Plättchen seinen Zug beenden, es kann aber nicht mehr erneut in Besitz genommen werden. Unabhängig davon führt der Spieler nun die Sippen-Aktion aus, abhängig von der Farbe der aufgenommenen Meeples, und anschließend die Plättchen-Aktion, die auf der linkten unteren Ecke des letzten Plättchens abgebildet ist.

4.5 Aktionen

Es gibt im Spiel jeweils fünf verschiedene Sippen- und Plättchen-Aktionen, die hier der Einfachheit halber aus der originalen Spielübersicht entnommen wurden:

Abbildung 5: Aktionen der Sippe aus der Spielübersicht (Quelle: Days of Wonder (2014))

Abbildung 6: Aktionen der Plättchen aus der Spielübersicht (Quelle: Days of Wonder (2014))

4.6 Sonderregeln

Abbildung 7: Punkte für Sets von Gütern (Quelle: Days of Wonder (2014))

Neben den bisher genannten Regeln gibt es auch noch ein paar Sonderregeln. So kann der Spieler am Ende seines Zuges vorzeitig Güter verkaufen, um sich für die nächste Runde die Goldressourcen für ein höheres Gebot zu verschaffen. Er erhält dabei, wie bei der Endwertung, Gold für unterschiedliche Arten von Gütern.[21] Sklaven lassen sich jedoch nicht verkaufen. Sie dienen stattdessen dazu, Aktionen wie dem Baumeister oder dem Meuchelmörder durch einen einmaligen Bonus zu verstärken, oder sie für das Erhalten von Dschinn am Heiligen Ort abzugeben. Die Fähigkeiten von Dschinn müssen teilweise ebenfalls mit Sklaven aktiviert werden. Diese Dschinn-Fähigkeiten haben dann verschiedene Fähigkeiten, die die bestehenden Regeln anpassen oder um neue ergänzen. Manche Dschinn habe auch einen Effekt auf die Berechnung der Siegpunkte am Spielende.

[21] Siehe *Abbildung 7: Punkte für Sets von Gütern* (Quelle:)

5 Technologien und Tools

Für die digitale Umsetzung dieses Brettspiels wurden verschiedene Technologien und Tools verwendet, auf welche im Folgenden eingegangen wird.

5.1 Entwicklung mit Java

Als Programmiersprache wurde Java 8 gewählt, da die Anwendung dadurch auch relativ einfach auf Mobilgeräte portiert werden könnte. Entwicklungsumgebung war IntelliJ IDEA, welches den Entwicklungsprozess mit verschiedenen Tools vereinfacht. Dazu zählen eine effiziente und übersichtliche Oberfläche für die Versionskontrolle mit Git, verschiedene Suchfunktionen, die die Navigation im Code vereinfachen, sowie effiziente Refactoring-Möglichkeiten, die das Umbenennen von Variablen auch auf den Rest des Codes übertragen. Alle benutzerdefinierten Bezeichnungen wurden auf Englisch gehalten, da Probleme mit Umlauten oder andere Eigenarten der deutschen Sprache vermieden werden sollten.

5.2 libGDX als Framework

Als Framework für das Projekt wurde libGDX verwendet, was auf Java basiert und sich ebenfalls auf verschiedene Plattformen portieren lässt. LibGDX bietet ein performantes und dennoch einfach zu implementierendes Rendering mit OpenGL im Core Profile. Für das Eventhandling gibt es das *Scene2d* Paket, was die Kollisionskontrolle von Mausinputs übernimmt und Events an die entsprechenden *Actors* sendet, welche Objekte in einer *Scene* sind. Dieses Eventhandling wurde für die *Views* der eigenen Anwendung verwendet und mittels Texturen um eine pixelgenaue Kollisionserkennung erweitert. Außerdem erleichtert libGDX das Laden der verschiedenen Assets und stellt diese anschließend zentral zur Verfügung.

5.3 Versionskontrolle mit Git

Für die Versionskontrolle des Projekts wurde Git verwendet. Als Git-Host und Client wurde Bitbucket von Atlassian genutzt, da dies, im Gegensatz zu anderen Diensten wie GitHub, kostenlose private Repositories anbietet.

5.4 Zusätzliche Tools

Für das Erstellen und Bearbeiten von Texturen wurden neben GIMP und InkScape noch zwei weitere Tools verwendet.

5.4.1 TexturePacker

TexturePacker ist ein Programm, dass aus verschiedenen Einzeltexturen eine große Textur (Textur-Atlas) erzeugt, mit einer Textdatei dazu, die die Positionen der einzelnen Ursprungstexturen in der großen Textur enthält. Außerdem bietet der TexturePacker eine Padding-Funktion, mit der ein Abstand zwischen allen Texturen hinzugefügt werden kann, da sonst beim Interpolieren die Pixel von benachbarten Texturen miteinbezogen werden würden.

5.4.2 Hiero

Hiero ist ein Programm, das für eine bestimmte Schriftart eine Textur mit den verschiedenen Schriftzeichen erzeugt und eine Datei mit deren entsprechenden Koordinaten. Diese kann dann von libGDX geladen werden und für ein performantes Text-Rendering verwendet werden. Ebenso lässt sich ein Textschatten hinzufügen, der die Schrift auf hellem Hintergrund besser lesbar macht.

6 Softwarearchitektur

Zu Beginn des Entwicklungsprozesses galt es Strukturen für die Architektur der verschiedenen Softwarebereiche zu definieren. Das zentrale genutzte Strukturprinzip ist dabei die Schichtenarchitektur, die das Softwaresystem in verschiedene Schichten unterteilt. Um diese verschiedenen Schichten miteinander zu verbinden, wurde als Entwurfsmuster das *Model View ViewModel* genutzt, kurz MVVM. Um die Daten und Logik eines Brettspiels zu realisieren, musste ebenfalls ein fachliches Domänenmodell aufgestellt werden, welches die allgemeinen Komponenten eines Brettspiels strukturiert.

6.1 Drei-Schichten-Architektur

Für den Anwendungsfall einer digitalen Brettspielumsetzung wurde eine herkömmliche Drei-Schichten-Architektur[22] verwendet, die die Anwendung in drei zentrale Schichten unterteilt. Die oberste dieser Schichten ist die Präsentationsschicht, welche die grafische Benutzeroberfläche (GUI) umfasst, mit welcher der Benutzer direkt interagiert. Darunter liegt die Steuerungsschicht, in der sich die Logik der Anwendung befindet. Diese regelt die Verarbeitung von Benutzereingaben aus der Präsentationsschicht und greift auf die darunterliegende Domänenschicht zu. In dieser dritten Schicht findet die Datenhaltung aller anwendungsrelevanten Daten statt. Da die Präsentationsschicht in diesem Fall auch einen direkten Zugriff auf die Domänenschicht hat, um Daten direkt auf der Oberfläche darzustellen, ohne diese durch die Steuerungsschicht zu geben, handelt es sich hierbei um eine offene Schichtenarchitektur[23].

Abbildung 8: Drei-Schichten-Architektur (Quelle: B. Martin (2012) S. 6)

[22] Vgl. B. Martin (2012) S. 5

[23] Vgl. F. Buschmann, et al. (1996)

6.1.1 Grafik / Präsentationsschicht

In der Präsentationsschicht befinden sich alle Komponenten, die zur Darstellung der Anwendung für den Benutzer benötigt werden. In dieser Schicht werden die Daten aus der Domänenschicht für den Benutzer zugänglich dargestellt. Alle Interaktionen vom Benutzer werden an die Steuerungsschicht übergeben. In diese Schicht gehören auch die *Views* und *ViewModels* des MVVM Musters.

6.1.2 Logik / Steuerungsschicht

In der Steuerungsschicht wird die Geschäftslogik, im Falle eine Brettspielumsetzung die Spielregeln, definiert. Alle Benutzerinteraktionen müssen den Regeln entsprechend verarbeitet werden. Der Spielverlauf wird außerdem in einem hierarchischen *Course* Modell gespeichert.[24] Dieses bestimmt den Ablauf des Programms und nimmt Änderungen in der Domänenschicht vor.

6.1.3 Daten / Domänenschicht

In diesem Anwendungsfall werden die Daten nicht in einer separaten Datenbank gespeichert, sondern nur direkt im Programm selbst. Dazu zählen auch alle Models, die für die Repräsentation des Spielmaterials relevant sind. Alle darstellungsrelevanten Daten werden hingegen über das MVVM in der Präsentationsschicht gespeichert.

6.2 MVVM

Als Schnittstelle zwischen GUI[25] und Benutzeroberfläche wurde das MVVM[26] Entwurfsmuster verwendet, welches eine Variante des MVC[27] Musters ist. Dabei werden die Darstellung, Darstellungslogik und die darzustellenden Daten mit deren Geschäftslogik voneinander getrennt. Das Model umfasst die Daten oder greift direkt auf diese zu. Es kennt dabei nur das Domänenmodell und enthält keinerlei grafische Objekte. Die View soll diese Daten dann in

[24] Siehe *Abbildung 31: vereinfachtes Course-Klassendiagramm* im Anhang

[25] grafische Benutzeroberfläche

[26] Model View ViewModel

[27] Model View Controller

der Oberfläche darstellen, hat jedoch keinen direkten Zugriff auf das Model. Als Bindeglied zwischen View und Model wird dann das ViewModel verwendet, welches die Darstellungslogik enthält und der View die Daten für die Darstellung des Models zur Verfügung stellt.

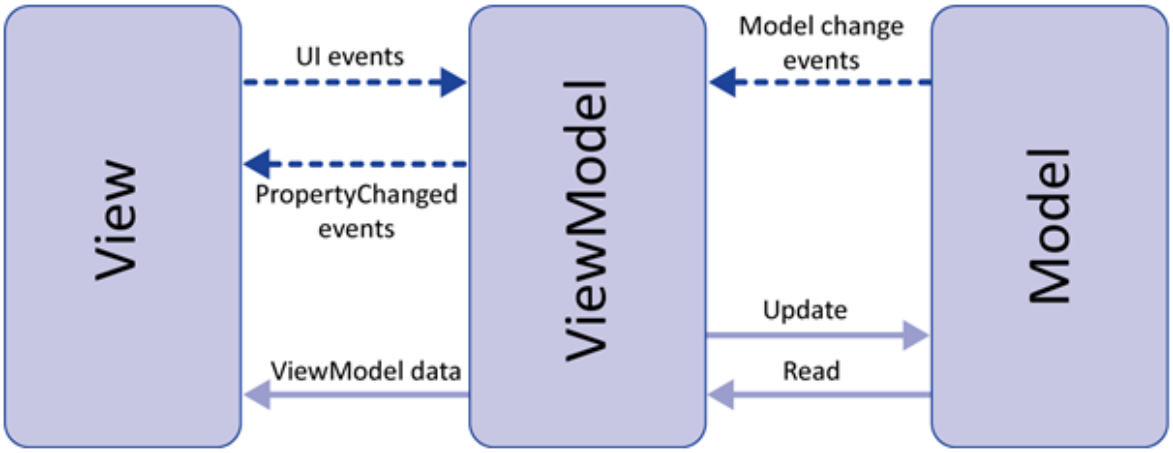

Abbildung 9: MVVM pattern (Quelle: Microsoft Corporation (o.J.))

Wie in Abbildung 9 zu sehen ist, interagiert die View nicht direkt mit dem Model. Stattdessen werden Events in der Oberfläche direkt an das ViewModel weitergegeben, welches anschließend das Model updated. Auf diese Änderung hin benachrichtigt das Model alle relevanten ViewModels, dass sich dessen Daten geändert haben. Diese lesen nun die geänderten Daten aus dem Model und passen die eigenen Daten entsprechend an. Anschließend wird auch die View benachrichtigt, dass sich die Daten des ViewModels geändert haben, und passt die eigene Darstellung den geänderten Daten der ViewModels an.

6.3 Domänenmodell

Die Datenstruktur einer Brettspielumsetzung lässt sich allgemein in drei zentrale Teilbereiche unterteilen: das Spielmaterial, den Spielverlauf und die Spielstruktur. Alle drei Komponenten sind miteinander verbunden und können so das gesamte Spielgeschehen abbilden.[28]

[28] Siehe *Abbildung 30: vereinfachtes Model-Klassendiagramm* im Anhang

6.3.1 Spielmaterial

Das Spielmaterial umfasst alle Elemente des Spiels, mit denen der Spieler interagieren kann und welche feste Eigenschaften besitzen. Dazu zählen unter anderem Spielfiguren, die über den Plan bewegt werden können, Karten, die gezogen, auf der Hand gehalten oder ausgespielt werden können, und Plättchen, die ausgelegt werden können. Dabei beschreibt jedes Element seinen eigenen Zustand, welcher nicht änderbare Eigenschaften und dynamische Relationen zu anderen Elementen umfasst. Außerdem ist ein weiterer Teil des Zustands eine Referenz auf das Eltern-Model, welches sich über den Spielverlauf jedoch ändern kann. Jede Spielmaterial-Model-Klasse besitzt außerdem eine ViewModel- und View-Klasse, die für die Darstellung des Elements zuständig sind.

6.3.2 Spielstruktur

Die Spielstruktur dient der strukturellen Repräsentation von abstrakten Zusammenhängen des Spielmaterials. Teile der Spielstruktur sind unter anderem die Kartenhand eines Spielers, die Plättchenauslage in der Spielfeldmitte oder auch das gesamte Spiel an sich, bestehend aus all seinen Elementen. Dies dient dazu, die Model-Hierarchie besser zu strukturieren und Spielmaterial-Models unter einem Spielstruktur-Model als Eltern-Model zu sammeln. Somit lässt sich zum Beispiel auch bestimmen, ob sich eine Karte in einer Auslage oder einem bestimmten Stapel befindet, was für die Einhaltung der Spielregeln relevant ist.

6.3.3 Spielverlauf

Der Spielverlauf ist eine Datenstruktur, die alle Interaktionen mit dem Spielmaterial wiederspiegelt. Dabei sind die verschiedenen Phasen des Spiels so verschachtelt, dass ein eindeutiger Spielablauf entsteht. Die Phasen, später auch als *Steps* bezeichnet, bestehen aus verschiedenen Zustandsänderungen am Model, oder auch weiteren Unter-*Steps*, die ausgeführt werden müssen, um die Phase abzuschließen. Diese Unter-*Steps* können dann auch wieder weitere Zustandsänderungen und *Steps* umfassen. Die Logik der einzelnen *Steps* wird dabei immer sofort ausgeführt, und es wird lediglich für das Warten auf Benutzereingaben pausiert. Sollte ein *Step* mehrere Unter-*Steps* umfassen, werden diese nacheinander ausgeführt, und jeweils auf deren Abschluss gewartet, bis die Phase fortgesetzt werden kann. Alle Zustandsänderungen sind dabei als Objekt zu speichern, sodass es theoretisch möglich ist, diese rückgängig zu machen und einen vorherigen Spielzustand wiederherzustellen.

6.4 Client-Server-Architektur

Da die Brettspielumsetzung netzwerkfähig sein soll, damit die Spieler auch an verschiedenen Geräten und Orten miteinander spielen können, bedarf es eines Modells für die Netzwerkkommunikation. Dies wurde über das Client-Server-Modell gelöst, bei dem ein zentraler Server die verschiedenen Clients verbindet. Dieser Server erhält alle Benutzereingaben der verschiedenen Clients und gibt diese an alle weiter.

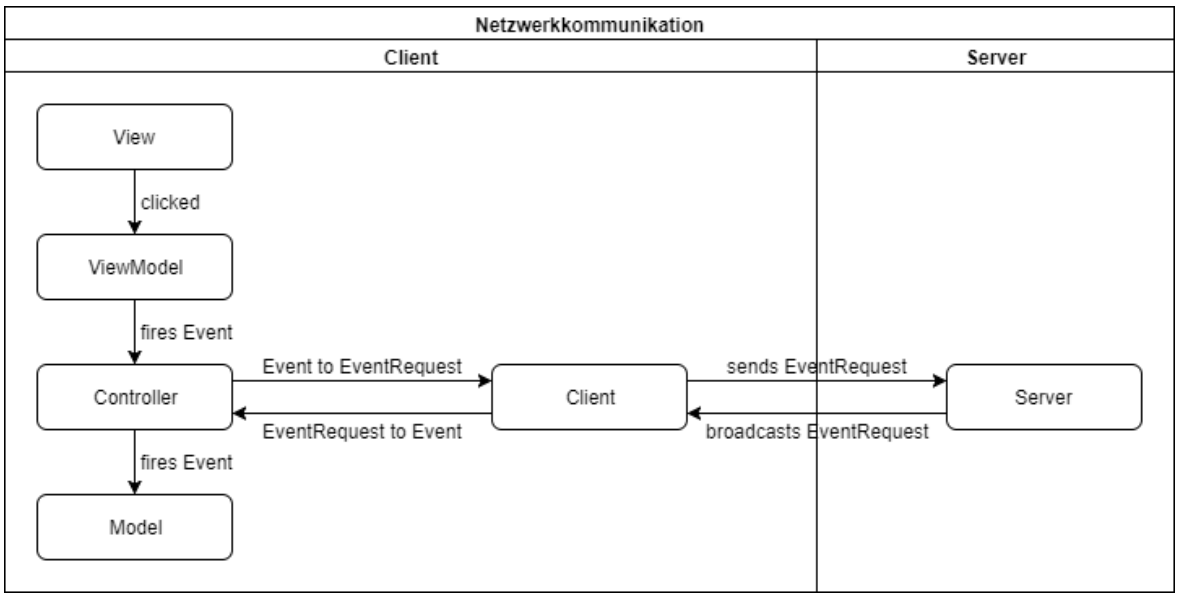

Abbildung 10: Netzwerkkommunikation (eigene Darstellung)

Damit jede Benutzerinteraktion abgefangen werden kann, wird eine allgemeine *Controller*-Klasse als Schnittstelle zwischen Benutzeroberfläche und Spiellogik genutzt, über die alle spielrelevanten *Events* laufen. Im offline Modus werden diese *Events* direkt an das *Model* weitergegeben. Im online Modus hingegen wird diese Schnittstelle durch den *ClientController* erweitert. Dieser macht aus allen *Events,* die er erhält, zunächst ein *EventRequest*-Objekt und sendet dieses über den *Client* an den *Server*. Dieser broadcastet anschließend dieses *EventRequest*-Objekt an alle verbundenen *Clients*. Erhält der *Client* dann seine Antwort vom *Server*, wird aus dem *EventRequest* wieder ein *Event* erzeugt und vom *ClientController* an das *Model* weitergegeben. Das *EventRequest*-Objekt ist dazu da, die Abhängigkeiten des *Events* vom *Model*, durch Adressierung über Indices aufzulösen.[29]

[29] Siehe *Abbildung 10: Netzwerkkommunikation*

Obwohl der Server selbst keine Instanz des Spiels speichert, ist es möglich, sich mit dem Server in einem bereits gestarteten Spiel zu verbinden. Dazu werden alle *EventRequests* geloggt und anschließend an den neu verbundenen Client gesendet, welcher daraus dann den gesamten Spielverlauf wiederherstellt. Der Spielaufbau wird dabei über ein *SetSeedEvent* synchronisiert, welches den genutzten Seed für das Setup beinhaltet.

7 Implementierung

Nachdem die Softwarearchitektur definiert und das Domänenmodell aufgestellt war, ging es an die Implementierung des Projekts. Aus zeitlichen Gründen war es leider nicht möglich, die Sonderregeln aus *4.6* vollständig umzusetzen.

7.1 Asset-Beschaffung und Bearbeitung

Einer der ersten Schritte bestand darin, die grafischen Assets des analogen Brettspiel-Originals zu beschaffen. Da der Verlag leider nicht berechtigt war, diese Grafiken selbst herauszugeben, jedoch zugestimmt hat, dass diese an sich für die Umsetzung im Rahmen der Bachelorarbeit genutzt werden dürfen, wurden Scans der Grafiken dem *Steam Workshop* entnommen, wo der Nutzer *Damian M.* diese, im Rahmen einer *Five Tribes* Umsetzung für den *Tabletop Simulator*, eingescannt hochgeladen hatte.[30]

Wie jedoch in Abbildung 11 zu sehen ist, weisen die Scans noch erkennbare Artefakte auf, welche auch als *Moiré-Effekt* bezeichnet werden, und durch das Scannen einer gedruckten Vorlage entstehen.[31] Diese können beim Verkleinern der Grafik ohne lineare Filterung zu starkem Flackern führen. Um das zu verhindern, wurden alle Scans mit einem entsprechenden „Flecken entfernen“-Filter in *GIMP* bearbeitet.

Abbildung 11: Moiré-Effekt Filterung (eigene Darstellung)

Zur einfachen und relativ performanten Darstellung von Schatten wurden jeweils eigene Texturen angelegt, die die Silhouette des Objekts leicht vergrößert und verschwommen darstellen sollen. Dieselbe Textur konnte dann ebenfalls für eine farbige Hervorhebung des Objekts bei Interaktion genutzt werden. Zusammen mit den anderen Texturen wurden diese, mit Hilfe vom *TexturePacker*, in einen Textur-Altas gepackt, um das häufige Wechseln zwischen Texturen beim Rendern zu vermeiden. Davon ausgenommen waren große

[30] Quelle der gescannten Grafiken: D. M. (2016)

[31] Vgl. GIMP Team (o.J.)

Texturen wie die Spielübersicht und das Scoreboard, welche in separate Texturen geladen wurden.

7.2 Generische Klassen

Um die Redundanz von logisch zusammenhängendem Code zu verringern, wurden an vielen Stellen Gebrauch von generischen Klassen gemacht.

7.2.1 Observer-Pattern

Damit im MVVM-Pattern jeweils das *ViewModel* und die *View* über geänderte Daten in Kenntnis gesetzt werden können, müssen bei Änderung des *Models* und *ViewModels* jeweils *Change*-Objekte an alle von ihnen abhängigen Objekte weitergegeben werden. Diese *Change*-Objekte der *ViewModels* enthalten *booleans*, die die verschiedenen Änderungen unter anderem an der Position, Größe und Sichtbarkeit des darzustellenden Objekts widerspiegeln. Die *View* muss sich dem Observer-Pattern entsprechend bei dem *ViewModel* anmelden um über jegliche Änderungen in Kenntnis gesetzt zu werden. Die *View* erhält dann das *Change*-Objekt des *ViewModels* und passt seine Darstellung entsprechend der geänderten Werte an.

Dadurch, dass nur bei Änderungen der Daten Updates erfolgen, werden regelmäßige Zustand-Checks, die in dieser Häufigkeit unnötig sind, vermieden und die Performance der Anwendung optimiert, während keine Zustandsänderungen stattfinden. Dies kann besonders nützlich bei mobilen Geräten sein, welche dadurch im Leerlauf eine deutlich geringere Prozessorauslastung haben und den Akku des Geräts schonen. Dies ließe sich auch auf das Rendering übertragen, sodass nur noch das Bild neu gerendert wird, wenn eine tatsächliche Änderung in der Darstellung stattgefunden hat.

7.2.2 Factories

Das Factory-Pattern ist ein Entwurfsmuster, welches ermöglicht, Objekte zentral zu erstellen und zu verwalten.[32] In einer Factory-Klasse befindet sich eine Factory-Methode, die ein bestimmtes Objekt erzeugt und zurückgibt. Der Konstruktor dieses Objekts muss dadurch

[32] Vgl. R. Friesen, et al. (2007)

nicht mehr von außerhalb aufgerufen werden, und es kann stattdessen eine Instanz der Factory-Klasse das entsprechende Objekt erzeugen.

Wie in Abbildung 12 zu sehen ist, gibt es eine *create*-Methode, welche für ein bestimmtes Input-Objekt ein entsprechendes Output-Objekt erzeugt. Diese Methode muss in den *Factory*-Klassen implementiert werden und den entsprechenden Konstruktor des zu erzeugenden Objekts aufrufen. Die *destroy*-Methode hingegen ist optional und wird aufgerufen, wenn ein bestimmtes Output-Objekt zerstört werden soll.

```
public abstract class Factory<O, I> {

    public abstract O create(I input);

    public void destroy(O output) {}
}
```

Abbildung 12: generische Factory-Klasse (eigene Darstellung)

Das *Factory*-Pattern wird verwendet, um für ein bestimmtes *Model*-Objekt ein entsprechendes *ViewModel*- und *View*-Objekt zu erzeugen. Da beim Erzeugen einer *View* diese der Eltern-*View* hinzugefügt wird, muss sie beim Zerstören auch wieder entfernt werden, was dann in der *ViewFactory* über die *destroy*-Methode geschieht.

7.2.3 IndexedList und SyncedList

Alle Elemente des Spielmaterials werden in Listen gespeichert und häufig auch entsprechend dargestellt. Dies ist beispielweise bei den Meeples auf einem Plättchen oder den Karten in einer Auslage der Fall. Da der Index des Objekts in dieser Liste meist relevant für die Spiellogik ist, beispielsweise für das Nehmen von Karten vom Markt, und auch für die Darstellung in der Oberfläche, wo der Index den Offset des Objekts bestimmt, soll dieser auch direkt im Element selbst abgelegt werden.

Zu diesem Zweck wird eine eigene allgemeine Implementierung einer *IndexedList* genutzt, welche die *ModifiableObservableListBase* aus JavaFX erweitert. *ModifiableObservableListBase* ist eine Liste, die sowohl hinzugefügte *ListChangeListener* über eine Änderung in der Liste, mittels eines *Change*-Objekts, benachrichtigt, als auch selbst doAdd-, doSet- und doRemove-Methoden besitzt, die bei jeder Listenmanipulation aufgerufen werden. *IndexedList* nutzt diese Methoden, um jeweils alle geänderten Indices der Elemente anzupassen.

Dazu müssen Elemente der *IndexedList* das Interface *Indexed* implementieren, welches Methoden für das Setzen eines Indexes in einer bestimmten Liste beinhaltet.[33]

Um verschiedene Listen miteinander zu synchronisieren, wurde eine generische *SyncedList* Klasse geschrieben. Diese wird beispielsweise benötigt, um Listen von *Models* mittels Listen von *ViewModels* und *Views* für die Oberfläche darzustellen. Um für die entsprechenden *Models* in der Quellenliste entsprechende *ViewModels* anzulegen, wird im Konstruktor der *SyncedList*-Klasse ein *Factory*-Objekt übergeben, welches eine *create*-Methode besitzt, die für ein bestimmtes *Model* ein entsprechendes *ViewModel* erzeugt. Um die Liste zu updaten, wird von außerhalb die *onChanged*-Methode aufgerufen und ein entsprechendes *Change*-Objekt der Quellenliste übergeben. Das *Change*-Objekt beinhaltet unter anderem Informationen über entfernte und hinzugefügte Objekte in der Quellenliste. Diese Änderungen werden nun ebenfalls auf die Element-Liste in der SyncedList übertragen, indem die *ElementFactory* aufgerufen wird, welche für die Objekte der Quellenliste entsprechende Vertreter in der Element-Liste erzeugt.[34]

[33] Siehe *Abbildung 32: IndexedList-Klasse* im Anhang

[34] Siehe *Abbildung 33: SyncedList-Klasse* im Anhang

7.3 Event-System

Für die netzwerkfähige Annahme, Weitergabe und Verarbeitung von Benutzereingaben wurde ein eigenes Event-System implementiert.

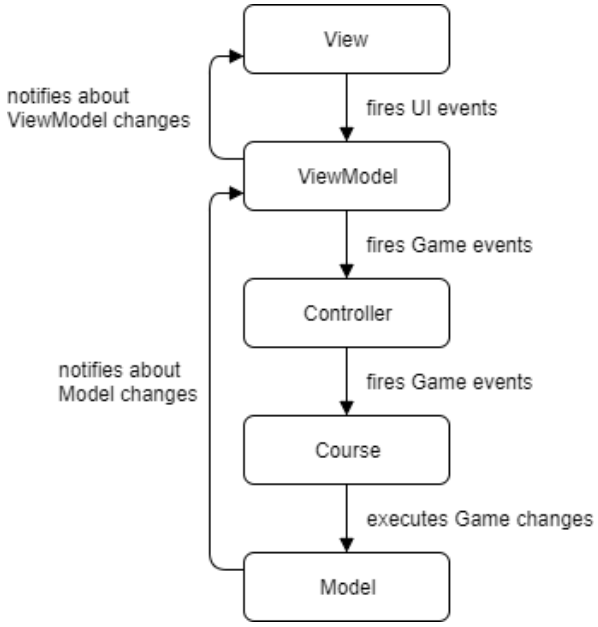

Abbildung 13: Event-System mit MVVM (eigene Darstellung)

Dieses besteht aus den *Views*, die Maus-Events in der Oberfläche annehmen und gegebenenfalls die *click*-Methode ihrer *ViewModels* aufrufen. Die *ViewModels* erzeugen dann ein *SelectElementEvent*, welches das geklickte *Model* beinhaltet, und senden es an den *GameController*, von dem aus es anschließend an das *FiveTribesModel* geht. Dieses schickt das Event dann an alle sogenannten *Steps*[35] des Spielverlaufs, die noch nicht abgeschlossen sind. Die *Steps* entscheiden dann jeweils, ob sie das Event regelkonform verarbeiten können und nehmen dann gegebenenfalls Änderungen am *Model* vor.[36]

7.3.1 GameController

Als Schnittstelle zwischen Frontend und Backend der Anwendung steht der *GameController*, welcher für die Weitergabe von *GameEvents* der ViewModels, an das *Course*-Model

[35] Als *Steps* werden die einzelnen Schritte im Spielverlaufs bezeichnet. Dazu zählen unter anderem die Runden, welche wiederum in Gebote und Züge unterteilt sind, und alle die *Steps* Klasse erweitern.

[36] Siehe *Abbildung 13: Event-System mit MVVM* und *Abbildung 31: vereinfachtes Course-Klassendiagramm* im Anhang

zuständig ist. Dieser wird gegebenenfalls vom *GameClientController* erweitert, welcher alle vom Spieler ausgelösten Events zunächst an den Server sendet und dann von diesem die *GameEvents* aller Spieler erhält, welche schließlich an das *Model* weitergegeben werden.[37]

7.3.2 Baiting

Da dem Spieler in der Oberfläche alle Elemente, mit denen er interagieren kann, hervorgehoben angezeigt werden sollen, wurde eine eigens erdachte Technik verwendet, die im Folgenden als *„Baiting"*[38] bezeichnet wird. Für das Event-*Baiting* besitzt jedes Event eine boolesche Klassenvariable *baiting*. Um nun zu testen, ob ein Plättchen oder eine Karte anwählbar ist und von der Spiellogik als gültiger Zug akzeptiert wird, wird ein Event für die Auswahl dieses Elements erzeugt, wie es beim tatsächlichen Anwählen dieses Elements der Fall wäre, jedoch wird die *baiting*-Variable auf *true* gesetzt. Anschließend wird dieses Event genauso an das *Model* gesendet wie ein tatsächlich ausgelöstes Event. Dort durchläuft es ebenfalls die Hierarchie des Spielverlaufs und wird auf die Zulässigkeit seiner Ausführung geprüft. Ist die Annahme des Events regeltechnisch zulässig, wird zunächst geprüft, ob das Event ein *Bait* ist, bevor eine Änderung am Model vorgenommen wird. Ist dies der Fall, werden keine Änderungen vorgenommen und lediglich *true* an das *ViewModel* zurückgegeben. Wird das Event nicht vom Spielverlaufsmodell angenommen, wird hingegen *false* zurückgegeben.

Da das *Baiting* von Events keine Auswirkungen auf den Spielzustand hat, brauchen *Baiting*-Events nicht an den Server übermittelt werden. Der Vorteil dieser Vorgehensweise ist, dass die Prüfung der Anwählbarkeit eines Elements durch die selbe Spiellogik läuft, wie das tatsächliche Event, was beim Anwählen erzeugt wird. Regeln bezüglich der verschiedenen Interaktionsmöglichkeiten des Spielers müssen somit nur einmal in der Spielverlaufslogik implementiert werden.

---

[37] Siehe *Abbildung 10: Netzwerkkommunikation* in Kapitel 6.4

[38] *Baiting:* Englisch für „Ködern"

7.4 Implementierung der Spielregeln

Die Spielregeln sind in einer eigenen Datenstruktur implementiert, dem *Course*[39]. Diese *Course*-Datenstruktur speichert den Spielverlauf und nimmt die Änderungen am *Model* vor. Der *Course* spiegelt dabei hierarchisch die verschiedenen Schritte und Unterschritte des Spielverlaufs wider. Dabei handelt es sich um ein eigenständiges System, was automatisch die verschiedenen Phasen startet und durchführt, und lediglich für das Warten auf Benutzereingaben pausiert. Die Änderungen am *Model* werden dabei über *Change*-Objekte ausgeführt und gespeichert. Diese *Change*-Objekte speichern, neben der Änderung am *Model*, ebenfalls den Ursprungszustand, sodass diese Änderung auch rückgängig gemacht werden kann. Das *Model* des Spiels enthält hingegen keinerlei Spiellogik und umfasst lediglich die Daten, die den aktuellen Spielzustand des Spielmaterials widerspiegeln.

7.4.1 Meeple-Zug-Logik

Ein zentraler Bestandteil von *Five Tribes* ist die Zug-Logik der Meeple-Bewegung, welche an mehrere Regeln gekoppelt ist.[40]. Dabei muss bestimmt werden, ob ein Teil-Zug zulässig ist, was der Fall ist, wenn dieser noch zulässige Folge-Züge hat. Dies ist nötig, damit der Spieler sich nicht in eine Sackgasse navigiert, von der aus kein Zug mehr zulässig ist und das Spiel somit nicht mehr fortgesetzt werden könnte. Zu diesem Zweck wurde ein Algorithmus aufgestellt, der das Problem möglichst performant lösen soll. Dieser Algorithmus soll bestimmen, ob noch ein regelkonformer Folge-Zug möglich ist, wenn der Spieler ein bestimmtes Plättchen auswählen oder einen bestimmten Meeple auf diesem absetzen würde. Zu prüfen ist dabei, ob es nach Auswahl dieses Plättchens bzw. Meeples noch möglich ist, seinen Zug auf einem Plättchen zu beenden, auf dem sich mindestens ein Meeple befindet, der die selbe Farbe hat, wie der darauf abgestellten Meeple („Letzter Meeple, gleiche Farbe“-Regel). Im Folgenden wird auf diesen Algorithmus und dessen Implementierung genauer eingegangen.[41]

---

[39] *Course:* Englisch für „Verlauf“

[40] Siehe Kapitel 4.4

[41] Siehe *Abbildung 34: canLastMeepleBeSameColor-Methode* im Anhang

```
// check if it is possible to land on a last Tile with a Meeple of a matching Tribe color on it
// tile: suggested Tile that is supposed to be moved to next
// meeple: suggested Meeple that is supposed to be dropped on the Tile
// tribes: suggested Tribes that will be left on Hand
public boolean canLastMeepleBeSameColor(TileModel tile, MeepleModel meeple, Set<MeepleModel.Tribe> tribes) {
    // number of Meeples left after dropping 1 Meeple on the suggested Tile
    int range = getGame().getHand().getMeeples().size() - 1;
    // direction that the Tile is moved to from
    SultanateModel.Direction comingFrom = null;
    if (currentTile != null) {
        comingFrom = SultanateModel.Direction.getByOffset( rowOffset: currentTile.getRow() - tile.getRow(),
                columnOffset: currentTile.getColumn() - tile.getColumn());
    }
```

Abbildung 14: Parameter für "LastMeepleSameColor"-Algorithmus (eigene Darstellung)

Für den Algorithmus sind verschiedene Parameter relevant. Dazu zählen das Plättchen (tile), auf das der Spieler ziehen will, und der Meeple (meeple), den der Spieler auf diesem Plättchen absetzen will, welcher auch *null* sein kann, wenn es sich nur um die Prüfung des Plättchens handelt. Ebenso ist relevant, welche Farbe bzw. Sippe die noch auf der Hand verbleibenden Meeples (tribes) haben, und von welchem Plättchen aus der Spieler auf das neue Plättchen zieht (currentTile). Daraus wird dann bestimmt, aus welcher Richtung sich der Spieler auf das neue Plättchen bewegt (comingFrom) und die Reichweite (range), die der Spieler noch für seinen restlichen Zug hat, welche der Anzahl an verbleibenden Meeples auf der Hand entspricht.

```
// suggested Tile is the last Tile
if (range == 0) {
    for (MeepleModel potentialMeeple : tile.getMeeples()) {
        // check if Tribe colors match
        if (tribes.contains(potentialMeeple.getTribe())) {
            return true;
        }
    }
    return false;
}
```

Abbildung 15: "LastTile"-Check (eigene Darstellung)

Wenn sich keine Meeples mehr auf der Hand befinden, ist das ausgewählte Plättchen das letzte Plättchen auf das der Spieler zieht. In diesem Fall muss geprüft werden, ob die Farbe bzw. Sippe des letzten Meeple, den der Spieler auf dieses Plättchen absetzt, mit einer der Meeples auf dem Plättchen übereinstimmt. Ist dies der Fall, so ist es ein regulärer Zug und es wird *true* zurückgegeben. Ist dies nicht der Fall, wird entsprechend *false* zurückgegeben.[42] In beiden Fällen muss nicht mehr geprüft werden, ob es noch einen möglichen Folge-Zug gibt, da dies der letzte Zug der Meeple-Bewegung ist.

[42] Siehe *Abbildung 15: "LastTile"-Check*

```
// check if full a circle is possible onto the selected tile matching the Tribe of the dropped Meeple
if (range >= 4 && range % 2 == 0) {
    if (meeple == null) {
        if (range > tribes.size()) {
            return true;
        }
    } else {
        if (tribes.contains(meeple.getTribe())) {
            return true;
        }
    }
}
// check if full a circle onto another Tile is possible
// (then both matching meeples haven't been placed yet, requires duplicate Tribe)
if (range >= 5 && range > tribes.size()) {
    return true;
}
```

Abbildung 16: Loop-Check (eigene Darstellung)

Da es von den Spielregeln her erlaubt ist, im Kreis zu ziehen und auf einem Feld zu landen, auf dem man zuvor bereits gewesen ist, muss diese Möglichkeit auch vom Algorithmus berücksichtigt werden. Dabei wird unterschieden zwischen einem Kreis-Zug, der auf dem Plättchen endet, auf das der Spieler gerade zieht, und einem Kreis-Zug, der im späteren Verlauf des Zuges durchgeführt werden kann.

Im ersten Fall muss der Spieler noch mindestens 4 Meeples auf der Hand haben um in einem kleinen Kreis wieder auf diesem Feld zu landen. Außerdem muss er noch einen Meeple auf der Hand haben, dessen Farbe bereits auf dem Plättchen vertreten ist. Hat er mehr Meeples, muss die Anzahl durch 2 teilbar sein, damit er einen entsprechenden Kreis machen kann.[43]

Im zweiten Fall sind beide Meeples, deren Farben am Ende übereinstimmen sollen, noch nicht auf einem Plättchen platziert worden. Somit muss der Spieler noch 5 oder mehr Meeples auf der Hand haben, von denen mindestens 2 dieselbe Farbe haben, damit diese beiden zusammen auf dem letzten Plättchen landen können. Der eine Meeple am Anfang des Kreises und der andere am Ende. Dies ist der Fall, wenn sich auf der Hand noch mehr Meeples als Farben befinden.[44]

Sollte der Spieler in einem Kreis ziehen wollen, dessen letzter Meeple eine übereinstimmende Farbe mit einem Meeple hat, der sich bereits auf dem Plättchen befindet, wird dies vom folgenden Teil des Algorithmus bereits berücksichtigt.

---

[43] Siehe *Abbildung 16: Loop-Check* erster if-Block

[44] siehe *Abbildung 16: Loop-Check* zweiter if-Block

```
// diamond shaped checkerboard pattern with all Tiles in range
for (int rowOffset = -range; rowOffset <= range; rowOffset++) {
    // corners of the direction the move is coming from are unreachable and get cut of
    if (rowOffset == -range && comingFrom != null && comingFrom == SultanateModel.Direction.NORTH) {
        continue;
    }
    if (rowOffset == range && comingFrom != null && comingFrom == SultanateModel.Direction.SOUTH) {
        continue;
    }
    for (int columnOffset = abs(rowOffset) - range; columnOffset <= range - abs(rowOffset); columnOffset+=2) {
        if (rowOffset == 0 && columnOffset == abs(rowOffset) - range && comingFrom != null && comingFrom == SultanateModel.Direction.WEST) {
            continue;
        }
        if (rowOffset == 0 && columnOffset == range - abs(rowOffset) && comingFrom != null && comingFrom == SultanateModel.Direction.EAST) {
            continue;
        }
        // no immediate backtracking
        if (rowOffset == 0 && columnOffset == 0 && range == 2) {
            continue;
        }
        // potential last tile that the move might end on
        TileModel potentialTile = getGame().getSultanate().getTile( row: tile.getRow() + rowOffset,  column: tile.getColumn() + columnOffset);
        if (potentialTile == null) {
            continue;
        }
        // check if the last move was towards a border of the Sultanate then the orthogonal Tiles can not be reached within 3 more moves
        if (comingFrom != null && getGame().getSultanate().neighbor(tile, comingFrom.opposite()) == null && range == 3) {
            if (getGame().getSultanate().neighbor(tile, comingFrom.switchedOffsets()) != null
                    && getGame().getSultanate().neighbor(tile, comingFrom.switchedOffsets()).equals(potentialTile)) {
                continue;
            }
            if (getGame().getSultanate().neighbor(tile, comingFrom.opposite().switchedOffsets()) != null
                    && getGame().getSultanate().neighbor(tile, comingFrom.opposite().switchedOffsets()).equals(potentialTile)) {
                continue;
            }
        }
        // check if Tribe colors of the Meeples from Hand and the Meeples on the potential last Tile match
        for (MeepleModel potentialMeeple : potentialTile.getMeeples()) {
            if (tribes.contains(potentialMeeple.getTribe())) {
                return true;
            }
        }
    }
}
return false;
```

Abbildung 17: "LastTile in range"-Check (eigene Darstellung)

Dieser letzte Teil des Algorithmus prüft, ob es ein Plättchen in Reichweite gibt, auf dem der Spieler seinen Zug beenden kann und welches dann einen Meeple hat, dessen Farbe der des letzten abgesetzten Meeples entspricht. Das Problem hat dabei Ähnlichkeiten mit dem „Mutilated Chessboard Problem“[45] von Max Black. Wenn man sich die Plättchenauslage von *Five Tribes* als Schachbrett vorstellt, wird dies deutlich. Da das Bewegen von Meeples nur horizontal und vertikal möglich ist, muss jeder Zug von einem Feld einer bestimmten Farbe, zu einem Feld der jeweils anderen Farbe gehen. Dadurch ergibt sich ein alternierendes Muster an Schachbrettfarben. Wenn ein Zug nun auf einem weißen Feld startet und eine gerade Anzahl an Feldern gezogen wird, kann der Zug nur auf einem weißen Feld enden. Bei einer ungeraden Zahl entsprechend nur auf einem schwarzen Feld. Beachtet

[45] Das „Mutilated Chessboard Problem“ beschreibt ein Problem, bei dem auf einem Schachbrett, von dem 2 gegenüberliegende Ecken abgeschnitten wurden, 31 Dominosteine platziert werden sollen (vgl. M. Black (1946)). Diese dürfen horizontal und vertikal platziert werden, dürfen sich aber nicht überlappen. Dieses Problem ist jedoch unlösbar, da ein Dominostein auf einem Schachbrett immer ein weißes und ein schwarzes Feld überdecken muss und somit die Anzahl überdeckter schwarzer und weißer Felder gleich groß sein muss. Dies ist jedoch nicht gegeben, wenn 2 gegenüberliegende Ecken entfernt sind, da diese die selbe Farbe haben und dadurch Schwarz und Weiß nicht mehr gleich oft vertreten sind.

man diese Regel und zieht außerdem noch die Bewegungs-Reichweite in Betracht, ergibt sich das folgende, rautenförmige Schachbrett-Muster an möglichen End-Plättchen für einen Zug mit 3 Meeples.[46]

Abbildung 18: rautenförmiges Schachbrettmuster (eigene Darstellung)

Eine weitere Regel ist, dass der Spieler nicht direkt auf das Feld zurückgehen kann, von dem er gerade gekommen ist. Somit ist das äußerste Feld in die Richtung, von der aus der Spieler auf das Plättchen zieht, nicht mehr erreichbar und kann übersprungen werden.[47]

Wenn sich nur noch 2 Meeples auf der Hand befinden, kann das angewählte Plättchen selbst auch nicht mehr das Ende eines vollständigen Kreises sein und kann somit übersprungen werden. Ein weiterer Sonderfall ist, wenn der Spieler sich im letzten Teil-Zug, orthogonal zum Spielfeldrand, auf den Spielfeldrand zu bewegt hat. Das heißt also, der Spieler kann nicht nochmal in dieselbe Richtung ziehen wie im Zug davor, da er sonst den Rand der Plättchenauslage überschreiten würde. Das führt dann dazu, dass die Felder, die benachbart zum ausgewählten Feld sind und sich ebenfalls am Rand des Spielfelds befinden, nicht mehr in 3 Schritten erreicht werden können. Das liegt daran, dass der Spieler

[46] Siehe *Abbildung 18: rautenförmiges Schachbrettmuster*

[47] Siehe *Abbildung 17: "LastTile in range"-Check*

zwar noch genug Meeples hat um den Bogen zu gehen, aber er müsste dabei unweigerlich direkt zurück auf das Plättchen ziehen, von dem er gerade gekommen ist.[48]

Sollte ein Plättchen alle vorherigen Zug-Checks bestanden haben, ist es ein Plättchen, auf dem der Zug potenziell enden könnte. Das letzte Kriterium, dem das Plättchen entsprechen muss ist nun die „Letzter Meeple, gleiche Farbe“-Regel. Dazu wird geprüft, ob sich auf dem Plättchen noch ein Meeple befindet, der einer Farbe entspricht, die sich noch auf der Hand befindet. Ist dies der Fall, gibt es für das ursprünglich vorgeschlagene Plättchen (tile) mindestens 1 Plättchen, auf dem der Zug regelkonform beendet werden kann, und es wird somit *true* zurückgegeben. Wenn keines dieser Plättchen diese Anforderungen erfüllt, wird *false* zurückgegeben und der Spieler kann nicht auf das vorgeschlagene Plättchen (tile) ziehen.[49]

7.4.2 Regellücken

Im Gegensatz zu analogen Brettspielen, wo man auch mal improvisieren oder kurzerhand seine eigenen Regeln festlegen kann, ist ein solches Vorgehen bei einer digitalen Brettspielumsetzung nicht möglich. Alle Regeln müssen absolut eindeutig definiert sein und auch für die unwahrscheinlichsten Situationen braucht es eine klare Vorgehensweise.

Eine diese unwahrscheinlichen Fälle in *Five Tribes* ist die Situation, dass ein Spieler kein Gold mehr hat, aber ein Mindestgebot von 1 Gold abgeben muss. In der Spielregel ist klar definiert, dass nur 3 Spieler 0 Gold bieten können und der vierte Spieler dann automatisch mindestens 1 Gold bieten muss. Dieses Problem galt es, mit so wenig wie möglich Änderungen in der Regel zu lösen. Daher fiel das Aussetzen des Spielers weg, da dies weitreichende Folgen haben und weitere Regelprobleme mit sich ziehen würde. Auch die Möglichkeit, ein viertes Mal auf die 0 zu gehen, würde klar der oben genannten Festlegung in der offiziellen Spielregel widersprechen, welche dies explizit verbietet. Die einfachste und beste Lösung wäre es, den Spieler in solch einem Fall dazu zu zwingen, genau 1 Ware zu verkaufen um dafür das nötige Gold zu erhalten. Für den Fall, dass auch dies nicht möglich sein sollte (wenn der Spieler keine Waren hat), wurde eine „Gnaden“-Regel eingeführt. Diese besagt Folgendes: Wenn ein Spieler sein Mindestgebot von 1 Gold nicht bezahlen kann, und es für ihn keine andere Möglichkeit gibt an das Gold zu kommen, darf er ein

[48] Siehe *Abbildung 17: "LastTile in range"-Check* ab potentialTile Deklaration

[49] Siehe *Abbildung 17: "LastTile in range"-Check* letzter for-Loop

Gebot von 1 Gold tätigen – und damit auch Startspieler sein – ohne etwas dafür zu bezahlen.

7.5 List- und Pile-ViewModel

Da es im Spiel mehrere *Element*-Listen gibt, die auch entsprechend in der Oberfläche als Auslage oder Stapel dargestellt werden sollen, wurden allgemeine *List*- und *Pile*-Klassen geschrieben, die die Darstellung einer Liste generisch realisiert. Listen und Stapel brauchen jedoch keine unterschiedlichen *Model*-Klassen, da sie lediglich eine unterschiedliche Repräsentation einer allgemeinen Liste von Elementen sind, und sich in der Datenhaltung nicht unterscheiden. Aus Sicht des *Models* ist ein Stapel ebenfalls eine Liste und benötigt im Fall von Five Tribes auch keine weiteren Attribute.

```
public ListViewModel(ListModel<EM, ?> model, ViewModelFactory<EVM, EM> factory,
                     int rows, int columns, float elementWidth, float elementHeight,
                     float widthPadding, float heightPadding, float xOffset, float yOffset,
                     float xOffsetPerColumn, float yOffsetPerRow, int elementAlign, boolean resize) {
```

Abbildung 19: ListViewModel-Konstruktor (eigene Darstellung)

Wie in Abbildung 19 zu sehen ist, umfasst das *ListViewModel* verschiedene Attribute für die Darstellung der *ListView*. Das *model* ist das *ListModel* mit den Elementen, die in der Liste dargestellt werden sollen. Die *factory* ist ein Factory-Objekt mit einer Factory-Methode, die aus den *Element-Models Element-ViewModels* erzeugt. Die anderen Parameter sind für die Darstellung der Liste und deren Elemente relevant.

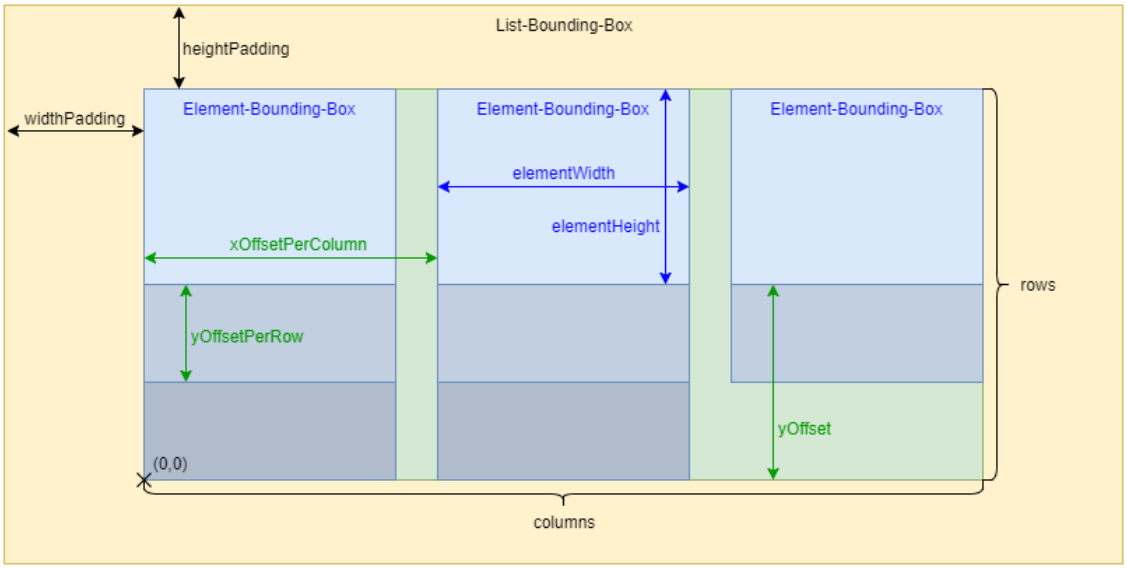

Abbildung 20: ListViewModel-Layout (eigene Darstellung)

Wie in Abbildung 20 dargestellt ist, werden die Elemente der eindimensionalen Liste in Spalten und Zeilen eingeteilt. Dies kann nützlich sein, wenn der Platz in die Breite begrenzt ist, und die Elemente daher in eine neue Zeile verschoben werden müssen. Ist die Anzahl der Zeilen unbekannt, kann die Liste *resizeable* gemacht werden, und passt dann seine eigene Größe so an, dass alle Elemente hineinpassen. Die gesamte Liste und auch Elemente in der untersten, noch nicht vollständigen, Zeile können dann automatisch zentriert werden, wie es bei den Meeples auf einem Plättchen der Fall ist.[50]

Abbildung 21: viele Meeples auf einem Plättchen (eigene Darstellung)

Im Gegensatz zu Listen sollen Stapel lediglich das oberste Element darstellen und alle darunterliegenden Elemente durch einen Höheneffekt repräsentieren. Für diesen Höheneffekt wird eine Stapel-Textur erstellt, welche wie gestapelte Karten aussehen soll und außerdem die Größe einer Karte dieses Stapels hat. Abhängig von der Anzahl an Karten in diesem Stapel wird das obenliegende Element, leicht nach oben verschoben, über die Stapel-Textur gezeichnet. Um auch einen verdeckten Ablagestapel zeichnen zu können, müssen die *ViewModels* der Elemente im Stapel die *CardViewModel*-Klasse erweitern, welche das Attribut *open* besitzt. Abhängig von diesem wird in der *CardView*-Klasse dann die Kartenvorder- oder Rückseite gezeichnet.[51]

Abbildung 22: verdeckter Kartenstapel (eigene Darstellung)

[50] Siehe *Abbildung 21: viele Meeples auf einem Plättchen*

[51] Siehe *Abbildung 22: verdeckter Kartenstapel*

7.6 Spieler-UI-Design

Für die digitale Repräsentation des Spielers und der Spielelemente, die er im analogen Brettspiel vor sich liegen hat, wurde eine Spielerinventar entworfen, dass alle diese Komponenten übersichtlich und kompakt darstellt.

Abbildung 23: Spielerinventar (eigene Darstellung)

Wie man in Abbildung 23 sehen kann, befindet dich am oberen Ende des Inventars der Name des Spielers und ein passendes Charakterbild. Der Marker rechts vom Namen zeigt an, dass der Spieler gerade am Zug ist. Das gelbe Leuchten hingegen bedeutet, dass der Spieler aktuell mit dem Spiel interagieren kann und das Spiel auch einen entsprechenden Input vom Spieler erwartet. In den meisten Fällen ist dies der Fall, wenn er auch selbst am Zug ist. Ausnahmen sind jedoch Entscheidungen, die der Spieler im Zug eines Mitspielers treffen muss.

Direkt unter dem Namen des Spielers befindet sich eine Übersicht über die Anzahl an Kamelen, die der Spieler noch zur Verfügung hat. Die Liste der Dschinn, die sich mittig im Spielerinventar befindet, ist die größte Komponente, da mit dieser theoretisch noch im Verlauf des Spiels interagiert werden kann. Damit die Effekte der Dschinn nicht überdeckt werden, ist die Liste von rechts nach links sortiert, wodurch der zuletzt erworbene Dschinn auch immer, gut erkennbar, am linken Ende der Liste steht. Unter den Dschinn befindet sich dann noch der Stapel an Rohstoffkarten, die Meeples und das Gold des Spielers. Auch mit den Meeples kann im Spielverlauf noch interagiert werden, da sie für die Meuchelmörder-Aktion als Ziel gewählt werden können.

8 Erkenntnisse

Im Laufe der Entwicklung wurden verschiedene Erkenntnisse gewonnen, die bei weiteren Projekten dieser Art von Nutzen sein könnten.

8.1 Trennung von Spielmaterial und Spiellogik

Die wichtigste Erkenntnis war die Trennung von Spielmaterial und Spiellogik. Der anfängliche Gedanke, die Spiellogik in den einzelnen Komponenten des Spielmaterials zu implementieren, brachte einige Probleme mit sich. So war es beispielsweise ursprünglich angedacht gewesen, die Logik-Methoden des *Models* direkt aufzurufen, wodurch das Rückgängigmachen von Aktionen nicht möglich gewesen wäre, da der Ursprungszustand nicht in einer Datenstruktur gespeichert werden würde. Außerdem stellte sich eine Ablaufsteuerung in den einzelnen Komponenten als schwierig dar, da sich der Zustand meist nicht mit nur einer Zustandsvariable beschreiben lassen würde, und auch Ereignisse, die im bisherigen Spielverlauf ausgelöst wurden, für den weiteren Spielverlauf weiter relevant sind. So muss beispielsweise bei der Meeple-Bewegung das letzte Plättchen gespeichert werden, auf das der Spieler gezogen ist, was vom *Model* jedoch nicht festgehalten wird.

Durch die Trennung dieser beiden zentralen Komponenten ist es möglich, auf der einen Seite ein „dummes“ *Model* zu haben, dass keinerlei Logik beinhaltet und lediglich als Datenstruktur dient, und auf der anderen Seite die Spiellogik in einem gesonderten, hierarchischen System zu haben, in dem die Änderungen am Spielzustand auch gespeichert und rückgängig gemacht werden können. Der Spieler interagiert somit nicht direkt mit dem Spielmaterial, sondern mit dem Spielverlauf, welcher dann den Zustand des Spielmaterials ändert.[52]

[52] Siehe *Abbildung 13: Event-System mit MVVM* in Kapitel 7.3 und *Abbildung 31: vereinfachtes Course-Klassendiagramm* im Anhang

8.2 Entwicklung im MVVM-Pattern

Eine weitere wichtige Erkenntnis waren die Erfahrungen, die mit der Entwicklung im MVVM-Pattern gemacht wurden. Anfänglich wurden noch in jeder *ViewModel*-Klasse die *ViewModel*-Listen manuell mit den *Model*-Listen synchronisiert, was zu einer hohen Coderedundanz geführt hat. Abhilfe hat dann die generische *SyncedList* und später dann noch die *ListViewModel*-Klasse geschafft, die diese Synchronisierung automatisch vornimmt. Ebenso konnte der Code auch durch die generische Darstellung von Listen, durch die *ListView*-Klasse, nochmal deutlich übersichtlicher gemacht werden.

Optimal wäre es, wenn man dieses Muster auch noch auf alle *Models* und *ViewModels* übertragen würde, die sich nicht in einer Liste befinden. Dies ließe sich durch eine Art *Model*-Slot umsetzen, der das *Model* in sich kapselt. Dies wäre vergleichbar mit einer *Model*-Liste mit genau einem Element. Dieser *Model*-Slot könnte dann allgemein mit einem *ViewModel*-Slot verbunden sein, welcher für das darin enthaltene *Model* mittels einer *Factory* ein entsprechendes *ViewModel* dazu erzeugt. Durch diese Kapselung in einem Slot kann die Verbindung zwischen *Model* und *ViewModel* bereits hergestellt werden, bevor das *Model* überhaupt initialisiert wurde.

9 Ausblick

Neben den bereits umgesetzten Features gibt es noch einige Features, deren Umsetzung zwar angedacht wurde, im zeitlichen Rahmen der Bachelorarbeit aber nicht realisierbar war. Jene Features werden im Folgenden theoretisch behandelt.

9.1 Implementierung von Sonderregeln

Wie bereits zuvor erwähnt, wurden die Sonderregeln aus Kapitel 4.6 noch nicht umgesetzt. Dazu zählt auch der Verzicht auf Aktionen, wie das Nehmen von Rohstoffen vom Markt. Diese Funktion ließe sich relativ einfach mit zwei extra Buttons umsetzen, die den Spieler entscheiden lassen, ob er bezahlen und die Aktion ausführen will oder nicht. Erst danach wird dann die Aktion durchgeführt oder komplett übersprungen.

9.1.1 Dschinnkarten-Fähigkeiten

Ein weiterer Kernaspekt des Spiels, der noch fehlt, sind die Dschinnfähigkeiten. Diese lassen sich alle in Auslöser, Kosten und Effekt unterteilen. Auslöser sind dabei die verschiedenen Spielsituationen, in denen die Fähigkeit aktiviert werden kann. Im *Course* würde dann bei allen Situationen, die eine Dschinnaktivierung erlauben würden, geprüft werden, ob ein Spieler einen Dschinn mit einem entsprechenden Auslöser besitzt. Ist dies der Fall, wird ein neuer Dschinn-*Step* erzeugt, der den aktuellen *Step* pausiert. Der Spieler erhält nun die Wahl, die Kosten für die Aktivierung zu bezahlen oder darauf zu verzichten. Entscheidet er sich die Kosten zu bezahlen, wird der Effekt des Dschinns wieder als neuer *Step* des Dschinn-*Steps* ausgelöst. Durch diese Trennung von Auslöser, Kosten und Effekt lassen sich die verschiedenen Dschinn beliebig aus diesen drei Teilen zusammensetzen. Auslöser und Kosten, die mehrfach vorkommen, müssen dann nur einmal in der Spiellogik berücksichtigt werden, und es muss nicht jeweils auf jeden Dschinn einzeln getestet werden.

9.1.2 Rohstoffkarten Interaktionen

Während die Punkteberechnung der Rohstoffkarten bereits implementiert ist, fehlen noch Features wie das Ansehen der eigenen Rohstoffkarten und der vorzeitige Verkauf von Gütern, für den eine solche Ansicht benötigt werden würde. Dies ließe sich mit einem eigenen

UI[53]-Fenster umsetzen, in dem das Spielerinventar vergrößert angezeigt wird. Beim eigenen Inventar wären dann die Rohstoffkarten sichtbar und könnten einzeln ausgewählt werden. Ebenso könnten darüber Sklavenkarten ausgewählt werden, die verwendet werden können, um einen Aktion zu aktivieren oder einen Bonus zu erhalten.

9.1.3 Offizielle Erweiterungen

Neben dem Grundspiel von *Five Tribes* gibt es noch drei große Erweiterungen und ein paar kleinere. Diese Erweiterungen bringen unter anderem völlig neue Spielkomponenten ins Spiel und erweitern auch bereits vorhandene Spielmechaniken. So gibt es neben neuen Dschinn auch neue Plättchen und Hindernisse, die die Bewegung der Meeples einschränken, und dadurch auch einen komplexeren Meeple-Bewegung-Algorithmus erfordern. Außerdem gibt es auch eine offizielle Solo-Variante von *Five Tribes*, bei der der Spielaufbau und Ablauf auf einen Spieler angepasst wurden.

9.2 Künstlicher Mitspieler

Ein weiterer Bereich, in dem die digitale Umsetzung erweitert werden könnte, ist die Realisierung eines künstlichen Mitspielers. Dabei gibt es jeweils die Möglichkeit, diesen manuell zu programmieren, oder eine KI[54] mittels maschinellem Lernen zu erschaffen. Für die manuelle Programmierung wurde bereits ein grobes Konzept entwickelt, in dessen Kern die Evaluierung der verschiedenen Zugmöglichkeiten stehen.

Um den heuristisch besten Zug zu finden, werden zu Spielbeginn zunächst alle möglichen und verschiedenen Meeple-Bewegungen gesammelt. Dabei wird von jedem Plättchen einzeln ausgegangen, es werden alle möglichen Routen für eine Meeple-Bewegung bestimmt, und diese Routen in die verschiedenen Möglichkeiten, die Meeples mit unterschiedlichen Farben abzusetzen, unterteilt.

Anschließend wird für jeden dieser Züge bestimmt, wie viele Punkte der Spieler für das Einnehmen des Plättchens und für die Sippen- und Plättchenaktion erhält. Um dabei Rechenzeit zu sparen, werden jeweils die Inputparameter, die für die Punkteberechnung der

---

[53] Benutzeroberfläche (user interface)

[54] künstliche Intelligenz

einzelnen Punktequellen relevant sind, gehasht und zusammen mit dem Punkte-Output in einer Map abgelegt. Alle Punkteberechnungen prüfen nun zuerst in der Map, ob die Berechnung bereits durchgeführt wurde, und entnehmen dann den entsprechenden Output-Punktewert, sofern dies der Fall ist.

Abbildung 24: Zugmöglichkeiten zu Spielbeginn (eigene Darstellung)

Beispiel: Der in Abbildung 24 dargestellte Spielaufbau hat 1.243 verschiedene mögliche Meeple-Bewegungen (manuell berechnet). Um jedoch nicht 1.243-mal die Berechnung für die Punkte, die durch die Sippen- und Plättchenaktion erzielt werden, für jede dieser einzelnen Zugmöglichkeiten, zu berechnen, werden diese gecached.

Abbildung 25: zwei Meeple-Bewegungen mit denselben Aktionen (eigene Darstellung)

Beispiel: Wie in Abbildung 25 zu sehen ist, kann der Spieler auf verschiedenen Wegen eine Händleraktion mit 2 Meeples ausführen. Dabei sind die Punkte, die der Spieler durch das Ausführen der Händleraktion mit 2 Meeples erhält, immer gleich, da sich die Auslage, trotz unterschiedlicher Meeple-Bewegungen, nicht geändert hat. Das Gleiche gilt in diesem Fall für die darauffolgende Plättchenaktion. Es ist dabei irrelevant, wie die Meeples über den Plan bewegt wurden, solange der Spieler dadurch keine weißen Meeples erhalten hat, die es ihm ermöglichen würden, am heiligen Ort einen Dschinn zu kaufen. Sollte der Spieler im Zug jedoch weiße Meeples erhalten haben, wären diese Meeples ein weiterer Inputfaktor, der bei der Berechnung des Inputhashs berücksichtigt werden muss.

Der Zug, der dem Spieler den höchsten, heuristisch bestimmten Punktewert gibt, wird ausgeführt. Sollten mehrere Züge den gleichen Punktewert geben, wäre es möglich zu bestimmen, welcher dieser Züge danach den geringsten Punktewert für den nächsten Spieler bringt. In diesem Fall stände die eigene Gewinnmaximierung vor der Gewinnminimierung für die Mitspieler.

Um zu bestimmen, wie viel der künstliche Mitspieler bieten sollte, wird angenommen, dass jeder andere Spieler nach derselben heuristischen Punkteberechnung handelt. Somit wird dann berechnet, wie groß die Punktedifferenz zwischen dem besten Zug des ersten Spielers und dem daraus resultierenden besten Zug des zweiten Spielers ist. Je höher diese Differenz ist, desto höher das Gebot.

9.3 Verbesserungen im Design

Auch im Design gibt es noch einige Verbesserungsmöglichkeiten, um die Anwendung benutzerfreundlicher zu machen. Neben verschiedenen Menüs, in denen Spieleinstellungen geändert werden können und ein Spielserver konfiguriert, gestartet und diesem beigetreten werden kann, gibt es außerdem noch Anpassungen, die das Spiel selbst betreffen und sich größtenteils auch auf digitale Brettspiele im Allgemeinen übertragen lassen.

9.3.1 Animation von Spielzügen

Benutzereingaben werden vom Programm direkt ausgewertet und der neue Spielzustand wird in der Oberfläche direkt dargestellt. Dadurch kann es schwer sein, die Züge der Mitspieler nachzuvollziehen, da man als Beobachter erst einmal feststellen muss, was von wo nach wo über den Plan bewegt wurde. Dies ließe sich durch klarere Bewegungsanimationen des Spielmaterials in der Oberfläche verbessern. Dabei sollte diese Animation sich nur auf die Benutzeroberfläche auswirken und keine Änderungen im *Model* erfordern. Während Zustandsänderungen im *Model* noch direkt stattfinden, stellt die Oberfläche einen Übergang zwischen den beiden Zuständen dynamisch dar. Die dafür benötigten Informationen über den alten Zustand werden dann jeweils dem *Course* und den *ViewModels* entnommen.

Um das Spielgefühl weiter zu verbessern, sollten außerdem die Aktionen des Spielers lokal direkt verarbeitet werden und nicht erst auf die Antwort des Servers gewartet werden. Bei einer schlechten Verbindung zum Server würde die Verzögerung zwischen Aktion und Reaktion sonst als Eingabeverzögerung wahrgenommen werden. Stattdessen werden die Benutzereingaben des lokalen Spielers dann direkt an das *Model* weitergegeben, und unter Vorbehalt verarbeitet. Sollte der Server danach entscheiden, dass die Benutzereingabe nicht zulässig war, wird diese wieder rückgängig gemacht und der Spielzustand, wie er vor der Benutzereingabe war, wiederhergestellt. Dies kann beispielsweise vorkommen, wenn der Spieler eine begrenzte Zeit für seinen Zug hat, die, zum Zeitpunkt des Interagierens, auf dem Server jedoch schon abgelaufen ist. Um in Fällen wie diesen die Benutzereingaben auf Richtigkeit zu überprüfen, sollte der Server selbst auch eine eigene Instanz des Spiel-*Model* besitzen, und alle Eingaben, die er von den Clients erhält, an dieser zunächst geprüft werden.

9.3.2 Simulierte Goldmünzen

Im analogen Brettspiel wird das Vermögen der Spieler mit 1er und 5er Goldmünzen repräsentiert. Diese haben die Spieler als verdeckten Stapel vor sich. Die Rückseite der Münzen ist dabei identisch, sodass jeder Spieler nur die Anzahl, aber nicht die Wertigkeit der Münzen seiner Mitspieler kennt. Dies kann eine relevante strategische Komponente sein, da die Spieler dadurch beim Bieten grob abschätzen können, wie viel die Mitspieler jeweils ungefähr bieten können. Um dies in der digitalen Umsetzung ebenfalls zu simulieren, sollte das Vermögen der Spieler als verdeckter Stapel aus Goldmünzen angezeigt werden. Die Spieler sollten dann nur noch ihr eigenes Vermögen als genauen Wert sehen, während das der anderen nur geschätzt werden kann.

9.3.3 Informationsfilter für den Spieler

Eine gute Übersicht über das Spielfeld ist ein wichtiger Faktor für das schnelle Finden eines guten Zuges. Daher sollte der Vorteil von digitalen Anwendungen genutzt werden, und dem Spieler Filter zur Verfügung gestellt werden, die die Übersicht über das Spielgeschehen verbessern können. Solche Filter könnten gerade relevante Spielelemente hervorheben, indem irrelevante Objekte entsättigt dargestellt werden. Beispielsweise könnten, wenn der Spieler zu Beginn der Meeple-Bewegung mit der Maus über ein Plättchen geht, alle Plättchen hervorgehoben werden, auf denen der Spieler seinen Zug beenden kann, wenn er seinen Zug von dem angewählten Plättchen aus startet.

9.3.4 Rückgängig machen von Aktionen

Ein relativ wichtiges Feature, was zwar in der Spielablaufstruktur bereits angedacht wurde, aus zeitlichen Gründen aber nicht umgesetzt werden konnte, ist das Rückgängigmachen von Aktionen. Dieses Feature ist besonders bei der Bewegung der Meeples relevant, wenn der Spieler bereits seinen Zug begonnen hat und dann feststellt, dass er einen Fehler in seiner Zugplanung gemacht hat. Für solche Fälle sollte dem Spieler ein „Rückgängig“-Button zur Verfügung gestellt werden, mit dem er seine letzte Entscheidung zurücknehmen kann. Wichtig ist dabei, dass der Spieler durch die letzte Aktion kein neues Wissen erlangt hat, dass ihm, auch noch nach dem Rückgängigmachen, einen Vorteil verleihen würde. So darf beispielsweise das Ziehen einer verdeckten Karte nicht wieder Rückgängig gemacht werden, da der Spieler durch diese Aktion neues Wissen erlangt hat. Wenn der Spieler eine Aktion ausführt, die er nicht mehr rückgängig machen kann, muss er diese Aktion nochmal explizit bestätigen. Erst bei Bestätigung sollten die Interaktionen dann an die anderen Spieler verschickt werden. Solch eine Bestätigung ist immer nötig, wenn danach ein anderer

Spieler aktiv werden würde, wie es nach der Abgabe eines Gebots oder dem Ende des eigenen Zugs der Fall ist.

9.3.5 3D-Ansicht

Ein relevanter Faktor für die Übersichtlichkeit beim analogen Brettspiel kann die Tiefe von verschiedenen Objekten sein. Im Falle von *Five Tribes* befinden sich auf dem Spielfeld bis zu 90 Meeples, welche sich jedoch beim analogen Brettspiel, durch die Dreidimensionalität als Holzteile, von den Plättchen absetzen. In der digitalen Umsetzung wurde versucht, das Spielmaterial durch einen dunklen Schattenumriss vom Hintergrund abzuheben. Es wäre jedoch von Vorteil, dem Spieler, neben der zweidimensionalen Darstellung des Spielfeldes, auch noch eine dreidimensionale Ansicht zu bieten. In dieser würde sich dann das hohe Holzmaterial, wie die Meeples, Paläste, Palmen oder Kamele, deutlich von den Plättchen abheben. Für solch eine dreidimensionale Darstellung sollte, der Schichtenarchitektur entsprechend, keine Änderung am *Model* oder der Spiellogik nötig sein, da letztere komplett unabhängig von der Visualisierung in der Benutzeroberfläche realisiert wurden.

10 Fazit

Das zentrale Ziel des Bachelorprojekts war die digitale Umsetzung des Brettspiels *Five Tribes* in einer eigenen, netzwerkfähigen Anwendung für den PC. Dabei sollte in der Arbeit auch ein allgemeiner Einblick in den Bereich der Brettspiele und deren digitale Umsetzung gegeben werden, was in den ersten drei Kapiteln geschah. Anschließend wurde genauer auf Architektur und Implementierung der Anwendung eingegangen, was den Hauptteil der Arbeit gebildet hat. Die Erkenntnisse aus dieser Arbeit wurden bereits in einem eigenen Kapitel ausführlich behandelt und es wurde auch ein Ausblick für das Projekt gegeben und ausbaubare Bereiche der Anwendung erläutert.

Das Ziel des Projekts wurde grundsätzlich erreicht, wenn auch mit ein paar Einschränkungen. Der Programmieraufwand eines solchen Projekts wurde zu Beginn deutlich unterschätzt. Dies hat dazu geführt, dass einige geplante Features nicht rechtzeitig implementiert werden konnten und auch Grundfunktionen des originalen Spiels, wie die Fähigkeiten der Dschinn, leider nicht zeitgerecht umgesetzt werden konnten. Dennoch wurde deren Umsetzung in der Arbeit zumindest theoretisch behandelt. Nichtsdestotrotz wurde jedoch die Kernmechanik des Spiels erfolgreich implementiert und in einer grafischen Oberfläche dargestellt.[55] Die Zug-Mechanik der Meeples funktioniert einwandfrei und auch die verschiedenen Aktionen wurden der Spielregel entsprechend umgesetzt. Es wurde ein allgemeines Muster für die Implementierung des Spielverlaufs erstellt, was sich auch auf andere Brettspielprojekte übertragen lässt.

Abschließend lässt sich sagen, dass, den zeitlichen Vorgaben entsprechend, ein Softwareprodukt entstanden ist, welches die Kernmechanik des analogen *Five Tribes* korrekt widerspiegelt. Die Spielregeln wurden erfolgreich abstrahiert und es wurden nützliche Erkenntnisse in der Ablaufsteuerung und bei der Entwicklung im MVVM-Pattern gewonnen. Diese Erkenntnisse, und der umfangreiche theoretische Ausblick, können bei der Strukturierung und Implementierung von zukünftigen Projekten dieser Art behilflich sein und den Entwicklungsprozess beschleunigen.

[55] Siehe *Abbildung 35: Screenshot vom Spiel* im Anhang

Literatur

Amelia Con. o.J.. Amelia Con. *What is Tabletop?* [Online] o.J. [Zitat vom: 14. 12 2017.] http://www.ameliacon.com/special-events/tabletop-gaming/what-is-tabletop/.

Asmodee Digital. 2017. *Carcassonne.* [App] 2017.

Berserk Games. 2017. *Tabletop Simulator.* [Videospiel] 2017.

Black, Max. 1946. *Critical Thinking.* s.l. : Prentice-Hall, Incorporated, 1946.

BoardGameGeek, LLC. o.J.. BoardGameGeek. *Glossary.* [Online] o.J. [Zitat vom: 15. 12 2017.] https://boardgamegeek.com/wiki/page/glossary.

—. o.J.. BoardGameGeek. *Five Tribes.* [Online] o.J. [Zitat vom: 11. 12 2017.] https://boardgamegeek.com/boardgame/157354/five-tribes.

—. o.J.. BoardGameGeek. *Carcassonne.* [Online] o.J. [Zitat vom: 16. 12 2017.] https://boardgamegeek.com/boardgame/822/carcassonne.

Brettspielwelt GmbH. o.J.. BrettspielWelt. *Carcassonne online spielen.* [Online] o.J. [Zitat vom: 17. 12 2017.] http://www.brettspielwelt.de/Spiele/Carcassonne/.

Buschmann, Frank, et al. 1996. *Pattern-Oriented Software Architecture: A System of Patterns.* s.l. : John Wiley & Sons, 1996.

Days of Wonder. 2014. Days of Wonder. *Five Tribes Spielregel.* [Online] 2014. [Zitat vom: 11. 12 2017.] http://cdn1.daysofwonder.com/five-tribes/en/img/ft_rules_de.pdf.

Exozet Potsdam GmbH. 2016. *Carcassonne.* [App] 2016.

Friesen, Rainer, Stollenwerk, Markus und Valentin, Daniel. 2007. Hochschule Trier. *Factory Pattern.* [Online] 06. 03 2007. [Zitat vom: 07. 12 2017.] https://public.hochschule-trier.de/~rudolph/gdv/cg/node49.html.

GIMP Team. o.J.. GIMP Documentation. *Flecken entfernen.* [Online] o.J. [Zitat vom: 05. 12 2017.] https://docs.gimp.org/2.6/de/plug-in-despeckle.html.

M., Damian. 2016. Steam Workshop: Tabletop Simulator. *Five Tribes.* [Online] 2016. [Zitat vom: 17. 09 2017.] http://steamcommunity.com/sharedfiles/filedetails/?id=790627499.

Martin, Bill. 2012. DocPlayer. *Konzeption und Analyse einer skalierbaren Architektur für Social Media Streams.* [Online] 2012. http://docplayer.org/1308783-Konzeption-und-analyse-einer-skalierbaren-architektur-fuer-social-media-streams.html.

Martin, W. Eric. 2015. BoardGameGeek. *Five Tribes Revised: Slaves Are Out, Fakirs In.* [Online] 25. 03 2015. [Zitat vom: 11. 12 2017.] https://boardgamegeek.com/blogpost/40219/five-tribes-revised-slaves-are-out-fakirs.

Microsoft Corporation. o.J.. Microsoft. *Implementing the Model-View-ViewModel Pattern.* [Online] o.J. [Zitat vom: 03. 12 2017.] https://msdn.microsoft.com/en-us/library/ff798384.aspx.

Anhang

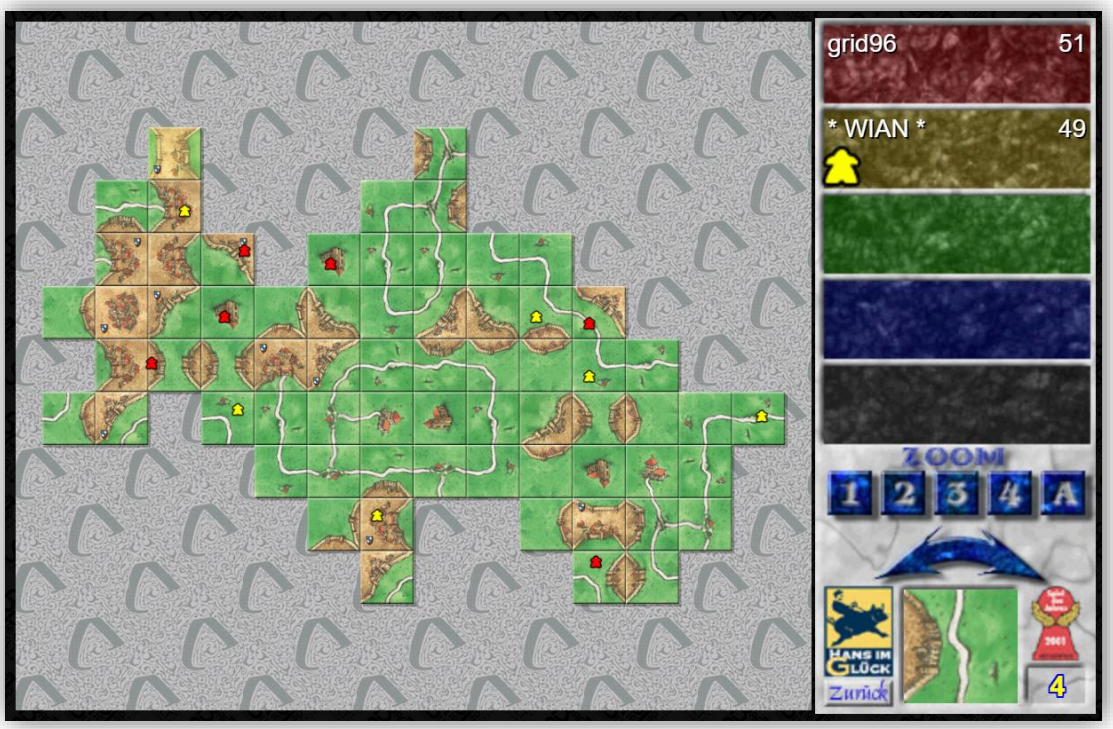

Abbildung 26: Carcassonne im Browser (Quelle: Brettspielwelt GmbH (o.J.))

Abbildung 27: alte Carcassonne App (Quelle: Exozet Potsdam GmbH (2016))

Abbildung 28: neue Carcassonne App (Quelle: Asmodee Digital (2017))

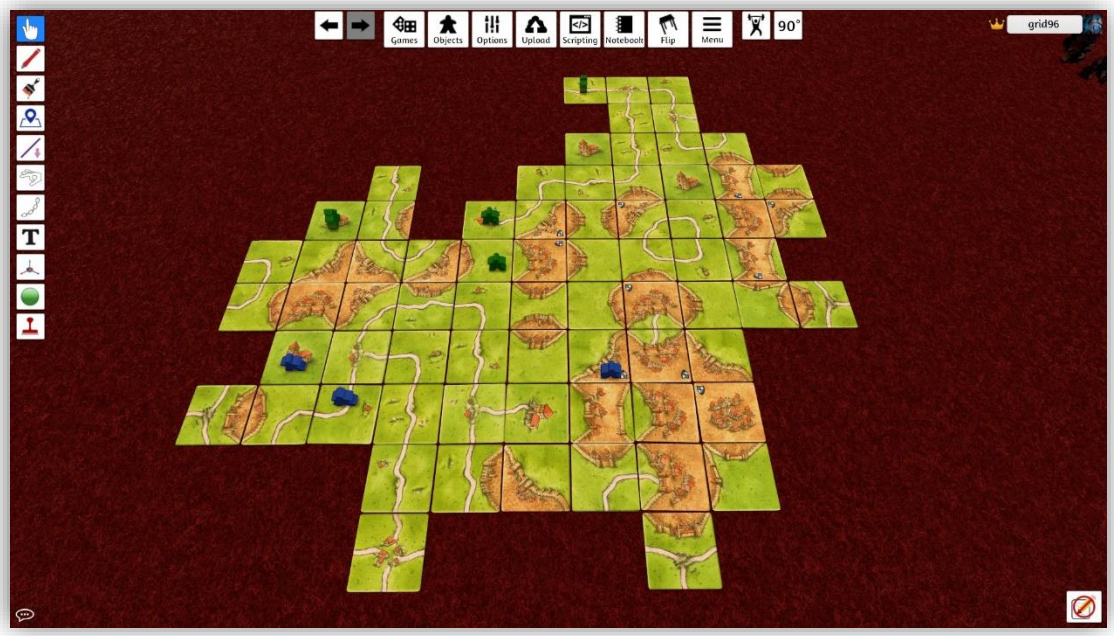

Abbildung 29: Carcassonne im Tabletop Simulator (Quelle: Berserk Games (2017))

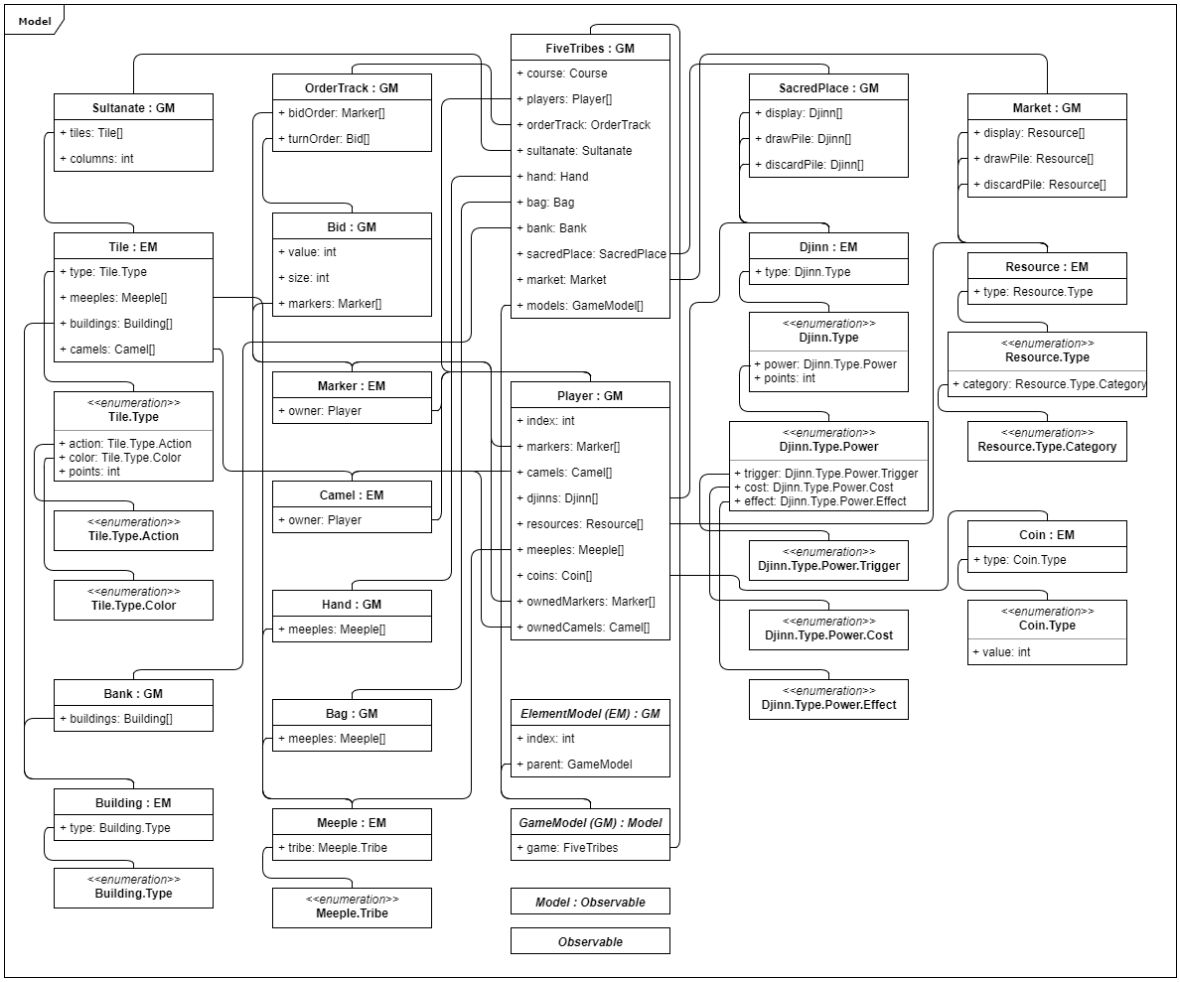

Abbildung 30: vereinfachtes Model-Klassendiagramm (eigene Darstellung)

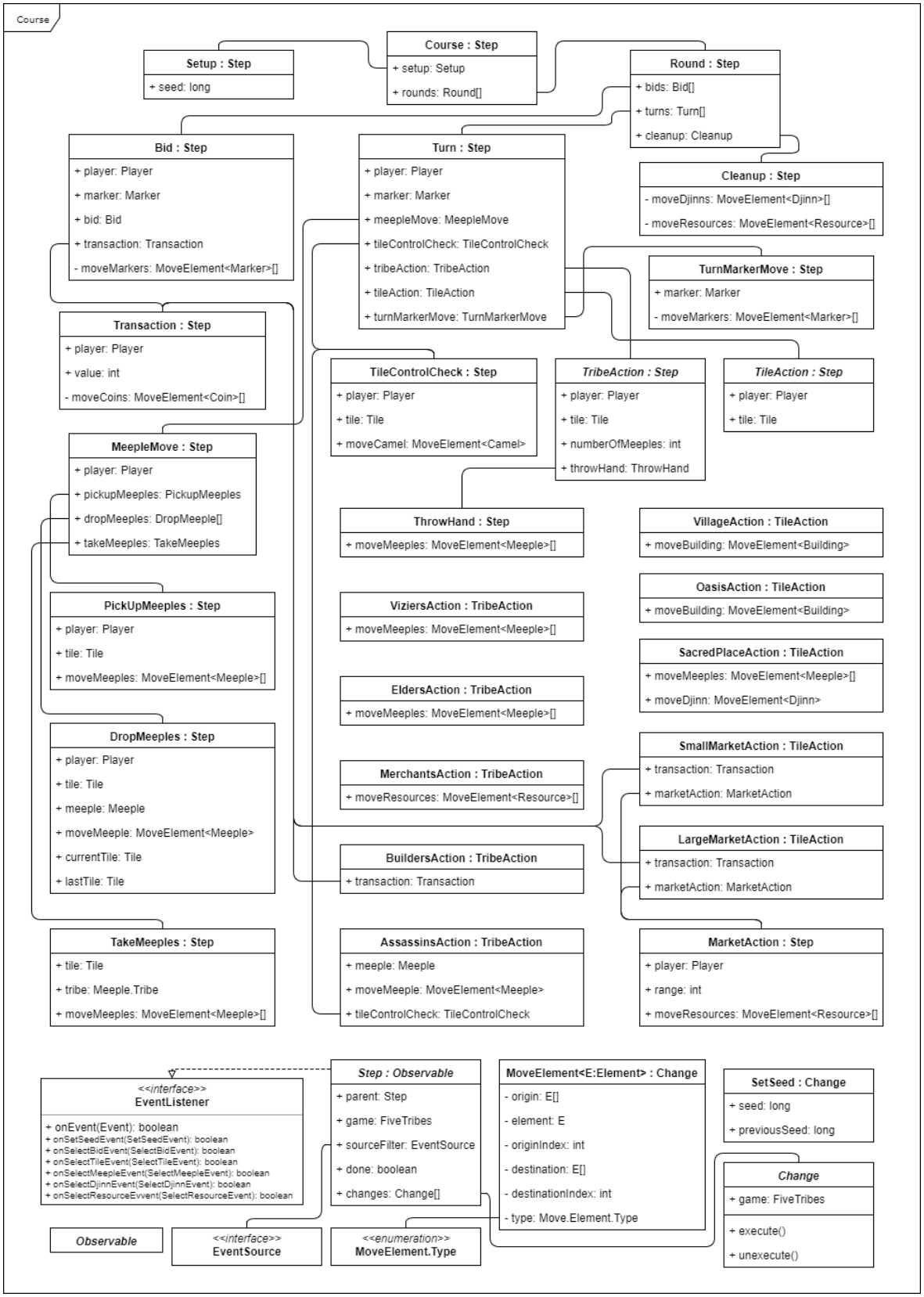

Abbildung 31: vereinfachtes Course-Klassendiagramm (eigene Darstellung)

```java
public class IndexedList<E extends IndexedList.Indexed> extends ModifiableObservableListBase<E> {

    private Vector<E> elements;

    public IndexedList() {
        elements = new Vector<>();
    }

    @Override
    public E get(int index) {
        return elements.get(index);
    }

    @Override
    public int size() {
        return elements.size();
    }

    @Override
    protected void doAdd(int index, E element) {
        elements.add(index, element);
        if (element != null) element.setIndex(index, list: this);
        for (int i = index; i < size(); i++) {
            if (elements.get(i) != null) elements.get(i).setIndex(i, list: this);
        }
    }

    @Override
    protected E doSet(int index, E element) {
        E removed = elements.set(index, element);
        if (removed != null) removed.removeFromList(this);
        if (element != null) element.setIndex(index, list: this);
        return removed;
    }

    @Override
    protected E doRemove(int index) {
        E removed = elements.remove(index);
        if (removed != null) removed.removeFromList(this);
        for (int i = index; i < size(); i++) {
            if (elements.get(i) != null) elements.get(i).setIndex(i, list: this);
        }
        return removed;
    }

    public interface Indexed<L extends IndexedList> {
        int getIndex(L list);
        void setIndex(int index, L list);
        void removeFromList(L list);
    }
}
```

Abbildung 32: IndexedList-Klasse (eigene Darstellung)

```java
public class SyncedList<E, S> extends ModifiableObservableListBase<E> implements ListChangeListener<S> {

    private List<E> elements;
    private List<S> source;
    private Factory<E, S> elementFactory;

    public SyncedList(List<S> source, Factory<E, S> elementFactory) {
        this.source = source;
        this.elementFactory = elementFactory;
        elements = new Vector<>();
    }

    public void init() {
        for (S sourceElement : source) {
            add(elementFactory.create(sourceElement));
        }
    }

    @Override
    public E get(int index) {
        return elements.get(index);
    }

    @Override
    public int size() {
        return elements.size();
    }

    @Override
    protected void doAdd(int index, E element) {
        elements.add(index, element);
    }

    @Override
    protected E doSet(int index, E element) {
        return elements.set(index, element);
    }

    @Override
    protected E doRemove(int index) {
        return elements.remove(index);
    }

    @Override
    public void onChanged(Change<? extends S> c) {
        while (c.next()) {
            if (c.wasPermutated()) {
                System.out.println("permutated");
            } else if (c.wasUpdated()) {
                System.out.println("updated");
            } else {
                for (int i = 0; i < c.getRemovedSize(); i++) {
                    elementFactory.destroy(remove(c.getFrom()));
                }
                for (int i = c.getFrom(); i < c.getTo(); i++) {
                    add(i, elementFactory.create(c.getList().get(i)));
                }
            }
        }
    }
}
```

Abbildung 33: SyncedList-Klasse (eigene Darstellung)

```java
// check if it is possible to land on a last Tile with a Meeple of a matching Tribe color on it
// tile: suggested Tile that is supposed to be moved to next
// meeple: suggested Meeple that is supposed to be dropped on the Tile
// tribes: suggested Tribes that will be left on Hand
public boolean canLastMeepleBeSameColor(TileModel tile, MeepleModel meeple, Set<MeepleModel.Tribe> tribes) {
    // number of Meeples left after dropping 1 Meeple on the suggested Tile
    int range = getGame().getHand().getMeeples().size() - 1;
    // direction that the Tile is moved to from
    SultanateModel.Direction comingFrom = null;
    if (currentTile != null) {
        comingFrom = SultanateModel.Direction.getByOffset( rowOffset: currentTile.getRow() - tile.getRow(),
                columnOffset: currentTile.getColumn() - tile.getColumn());
    }
    // suggested Tile is the last Tile
    if (range == 0) {
        for (MeepleModel potentialMeeple : tile.getMeeples()) {
            // check if Tribe colors match
            if (tribes.contains(potentialMeeple.getTribe())) {
                return true;
            }
        }
        return false;
    }
    // check if full a circle is possible onto the selected tile matching the Tribe of the dropped Meeple
    if (range >= 4 && range % 2 == 0) {
        if (meeple == null) {
            if (range > tribes.size()) {
                return true;
            }
        } else {
            if (tribes.contains(meeple.getTribe())) {
                return true;
            }
        }
    }
    // check if full a circle onto another Tile is possible (then both matching meeples haven't been placed yet, requires duplicate Tribe)
    if (range >= 5 && range > tribes.size()) {
        return true;
    }
    // diamond shaped checkerboard pattern with all Tiles in range
    for (int rowOffset = -range; rowOffset <= range; rowOffset++) {
        // corners of the direction the move is coming from are unreachable and get cut of
        if (rowOffset == -range && comingFrom != null && comingFrom == SultanateModel.Direction.NORTH) {
            continue;
        }
        if (rowOffset == range && comingFrom != null && comingFrom == SultanateModel.Direction.SOUTH) {
            continue;
        }
        for (int columnOffset = abs(rowOffset) - range; columnOffset <= range - abs(rowOffset); columnOffset+=2) {
            if (rowOffset == 0 && columnOffset == abs(rowOffset) - range && comingFrom != null && comingFrom == SultanateModel.Direction.WEST) {
                continue;
            }
            if (rowOffset == 0 && columnOffset == range - abs(rowOffset) && comingFrom != null && comingFrom == SultanateModel.Direction.EAST) {
                continue;
            }
            // no immediate backtracking
            if (rowOffset == 0 && columnOffset == 0 && range == 2) {
                continue;
            }
            // potential last tile that the move might end on
            TileModel potentialTile = getGame().getSultanate().getTile( row: tile.getRow() + rowOffset, column: tile.getColumn() + columnOffset);
            if (potentialTile == null) {
                continue;
            }
            // check if the last move was towards a border of the Sultanate then the orthogonal Tiles can not be reached within 3 more moves
            if (comingFrom != null && getGame().getSultanate().neighbor(tile, comingFrom.opposite()) == null && range == 3) {
                if (getGame().getSultanate().neighbor(tile, comingFrom.switchedOffsets()) != null
                        && getGame().getSultanate().neighbor(tile, comingFrom.switchedOffsets()).equals(potentialTile)) {
                    continue;
                }
                if (getGame().getSultanate().neighbor(tile, comingFrom.opposite().switchedOffsets()) != null
                        && getGame().getSultanate().neighbor(tile, comingFrom.opposite().switchedOffsets()).equals(potentialTile)) {
                    continue;
                }
            }
            // check if Tribe colors of the Meeples from Hand and the Meeples on the potential last Tile match
            for (MeepleModel potentialMeeple : potentialTile.getMeeples()) {
                if (tribes.contains(potentialMeeple.getTribe())) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Abbildung 34: canLastMeepleBeSameColor-Methode (eigene Darstellung)

Abbildung 35: Screenshot vom Spiel (eigene Darstellung)

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Langenfeld, den 30.12.2017

_____

Colin Dömer