Autoren	Bodack Sandra, Hirsbrunner Daniel, Velickovic Maja, Widmer Michelle							
Version	1.0							
Bearbeitungs-stand	☐	Entwurf / in Bearbeitung	☒	zur Abnahme	☐	definitive Fassung		

Nicht Lustig

Technische Spezifikation

IT-Projekt 2016

Zusammenfassung

Im Rahmen des Moduls „IT-Projekt“ bei Bradley Richards und Lukas Frey wurde nachfolgende Technische Spezifikation als Grundlage für die Entwicklung des Spiels „Nicht Lustig“ erstellt. Die Dokumentation soll als Grundlage für die Entwicklung dienen und die Lösungsansätze der Umsetzung aufzeigen.

Dokumentengeschichte

Version	Datum	Überarbeitungsgrund	Ersteller
0.1	07.05.2016	Datei erstellt, Tools und Richtlinien beschrieben	Daniel Hirsbrunner
0.2	19.05.2016	Überarbeitung	Michelle Widmer
0.3	19.05.2016	Ergänzung der Kapitel Game, GameFinisher, BoardManager, Würfeln und Wertung	Maja Velickovic
0.4	21.05.2016	Überarbeitung und Erstellung Kapitel Server- / Client-GUI	Sandra Bodack
0.5	21.05.2016	Ergänzung GUI-Konzept	Michelle Widmer
0.6	23.05.2016	Diverse Klassen Diagramme, Kapitel Datenstruktur	Daniel Hirsbrunner
0.7	24.05.2016	Diverse Ergänzungen	Maja Velickovic
0.8	25.05.2016	Kapitel Server-Client Zusammenspiel	Daniel Hirsbrunner
1.0	27.05.2016	Diverse Korrekturen	Sandra Bodack

Dokumentenverwaltung

Datei	TechnischeSpezifikation_NichtLustig_V1.0.docx
Tool	Microsoft Office Word

Inhaltsverzeichnis

1 Referenzen und Verzeichnisse

1.1 Tabellenverzeichnis

1.2 Abbildungsverzeichnis

2 Entwicklungsrichtlinien

Bei der Umsetzung von unserem Projekt möchten wir uns an gewisse Programmierrichtlinien halten.

Wir wollen uns aus folgenden Gründen an diese Richtlinien halten:

- Einheitliche Code-Formatierung
- Einheitliche Namensgebung
- Erleichterung der Lesbarkeit unseres Codes
- Erleichterung der Wartung

Bei der Einhaltung der Richtlinien hilft uns einerseits Eclipse durch die automatische Formatierung des Codes. Ebenfalls wollen wir uns in gegenseitigen Code-Reviews auf Richtlinienverletzungen aufmerksam machen, falls sich ein Teammitglied an einem Ort nicht daran gehalten hat.

Unsere Vorgaben sehen folgendermassen aus:

- Die Namensgebung ist in Englisch
- Klassennamen werden gross geschrieben
- Packages sind klein geschrieben und folgendermassen strukturiert: teamamused.<Projekt Name>.<Package Name>.<Subpack. Name>.<Spezifizierung>
- Interfaces haben den Präfix I
- Abstrakte Klassen haben den Präfix Abstract
- Variablen Namen werden klein geschrieben
- Auf Variablen wird via getter und setter zugegriffen, ausser bei Objekten welche rein für die Datenhaltung sind.
- Konstante Werte werden in Grossbuchstaben geschrieben und sind statisch. Beispiel: „static final int PICTURE_HEIGHT = 125;“

Wir wollen möglichst die Logik, die zusammengehört in eigene Klassen auslagern, jedoch unnötige Abhängigkeiten zwischen den Klassen vermeiden.

Die Funktionalität von Klassen und Methoden werden mit JavaDoc beschreiben, in den Methoden sollen einzelne Kommentarzeilen den anderen Teammitgliedern helfen, einen raschen Überblick über die Funktion zu gewinnen.

3 Tools

3.1 Entwicklungsumgebung

Wir werden unser IT-Projekt mit Eclipse umsetzen, da dies eine der bewährtesten IDEs (Integrated Development Environment) für Java ist und wir auch im Unterricht mit Eclipse arbeiten.

Eclipse ist eine Open Source Software und selber auch in Java programmiert. Aus diesem Grund gibt es diverse Plug-ins und Erweiterungen, welche uns die Programmierung erleichtern werden.

Programm	Eclipse
Hersteller	Eclipse Foundation
Webseite	www.eclipse.org

Abbildung 1: Eclipse Logo

3.2 Source Code Verwaltung

Wir werden unseren Source Code mit dem Eclipse Plug-in EGit verwalten und haben dazu ein Repository auf www.github.com (https://github.com/teamamused/nichtlustig.git) erstellt.

Das EGit Plugin ist ab der Version 4.2.2 von Eclipse bereits Bestandteil des offiziellen Releases. Somit fiel die Entscheidung sehr schnell auf EGit, da es jeder bereits hatte, das Plug-in gut dokumentiert und einfach zu bedienen ist.

GitHub ist ein Online-Anbieter, bei welchem öffentliche Git Repositories gratis gehostet werden können. GitHub gibt es schon seit Februar 2008 und erfreut sich grosser Beliebtheit. Wir haben uns für GitHub entschieden, da wir aus anderen Projekten bereits mit der Plattform Erfahrung hatten.

Programm	EGit
Hersteller	Eclipse Foundation
Webseite	www.eclipse.org/egit

Abbildung 2: EGit Logo

Dienst	GitHub
Eigentümer	GitHub Inc.
Webseite	www.github.com

Abbildung 3: GitHub Logo

3.3 Build-Management-Tool

Das Build-Management-Tool sorgt dafür, dass unsere Java Klassen automatisch deployed werden.

Als Build-Management-Tool werden wir ANT (Another Neat Tool) einsetzen. ANT ist eine Open Source Software, welche von der Apache Software Foundation verwaltet wird. Wir haben uns für ANT entschieden, da es altbewährt, einfach zu bedienen und in Eclipse integriert ist. ANT kann zudem auch unsere Unit Tests ausführen und die Java Doc erstellen.

Programm	ANT
Hersteller	Apache Software Foundation
Webseite	ant.apache.org

Abbildung 4: Apache ANT Logo

3.4 Kontinuierliche Integration / automatisches Testing

Travis ist ein online Buildserver, welcher bei jedem Push in das GitHub Repository automatisch einen Build starten kann. Die Builds können anhand von ANT definiert werden, somit stellen wir sicher, dass die Unit Tests bei jedem Push durchführt werden. Falls nun jemand aus unserem Team ein nicht buildbares Changeset pushed oder einen Unit Test zum failen bringt, werden wir sofort per E-Mail informiert. So erhöhen wir die Codequalität in unserem GitHub Repository.

Unser Account hat folgende URL: travis-ci.org/teamamused/nichtlustig

Dienst	ANT
Eigentümer	Travis CI community
Webseite	www.travis-ci.org

Abbildung 5: Travis CI Logo

3.5 Dokumentation

Wir werden Java Doc zur Code Dokumentation verwenden. JavaDoc ist Teil des offiziellen Java Development Kits und daher automatisch bei allen vorhanden.

Wir haben uns für JavaDoc entschieden, da Eclipse die Java Doku während dem arbeiten direkt in einem PopUp Fenster anzeigt, so sehen wir während dem Arbeiten die Dokumentation der anderen Teammitglieder. Zusätzlich lässt sich die Dokumentation als HTML-Seiten exportieren, was zu einem schönen Gesamtüberblick führt. Java Doc wurde im Unterricht Java 2 von Lukas Frey behandelt und ist daher bereits jedem aus unserem Team bekannt.

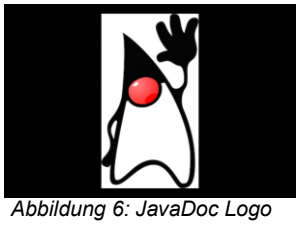

Abbildung 6: JavaDoc Logo

Die aktuelle Java Doc des Master Branches befindet sich jeweils auf unseren GitHub-Pages: teamamused.github.io/nichtlustig/docs/

Programm	JavaDoc
Eigentümer	Oracle (Teil des JDK)
Webseite	docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

Die Anforderungs- sowie die technische Dokumentation verwalten wir in einer zentralen Word-Datei, welche durch den Dropbox Dienst versioniert wird. Die Phasenergebnisse werden auf unserer GitHub Page (teamamused.github.io/nichtlustig) veröffentlicht und via Moodle zur Bewertung eingereicht.

3.6 Bug Tracking

Wir werden unsere Pendenzen in einer zentralen Excel-Datei auf unserem Dropbox Share verwalten. Falls die Excel Datei unseren Ansprüchen nicht mehr genügt, werden wir das Issues System von GitHub.com einsetzen.

4 Software Architektur

4.1 Gesamtübersicht

Hier sehen Sie einen gesamten Überblick über die Klassen in unserem Projekt. Hilfsklassen für die Tests oder die Dokumentation sind hier nicht aufgeführt, da sie nicht direkt mit der Programmlogik interagieren.

Wir setzen unser Projekt mit 3 Java Projekten um, aus welchen die 2 Programme „Client“ und „Server“ resultieren.

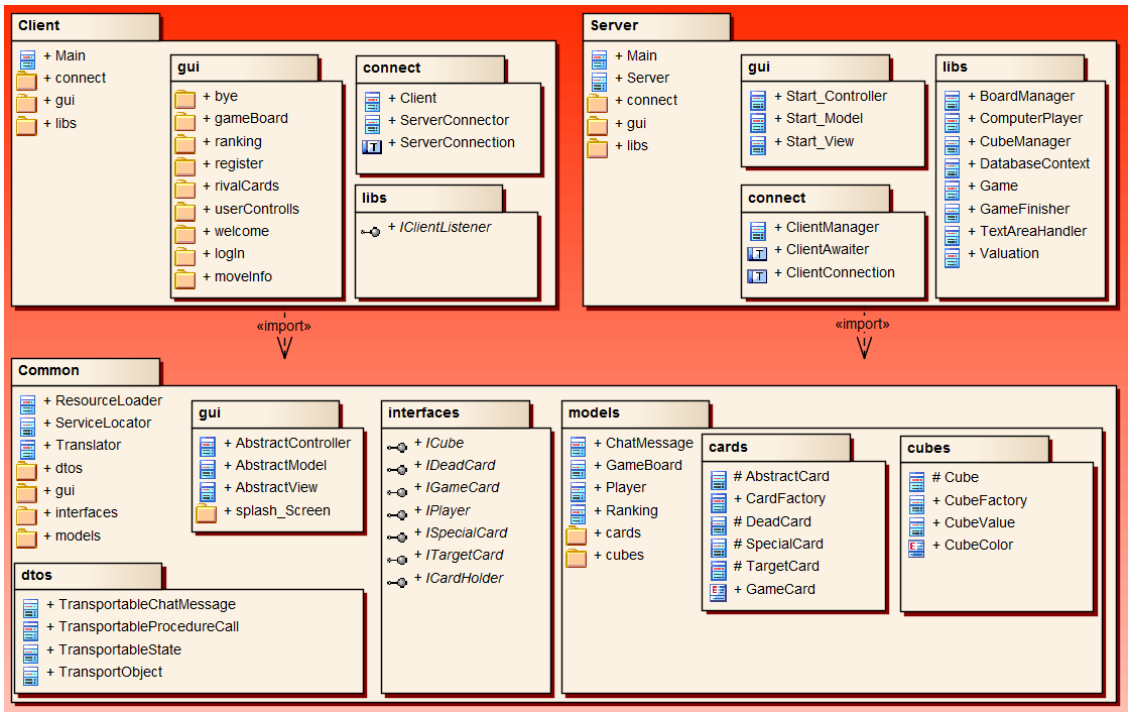

Abbildung 7: Klassen Gesamtübersicht

4.1.1 Projekt Common

Im Projekt Common möchten wir sämtlichen Code auslagern, welchen wir im Server sowie im Client benötigen. Dies beinhaltet grundsätzliche Standardfunktionalitäten, die Business Objekte sowie die Definition der Schnittstellen. Das Projekt kann nicht gestartet werden, es ist eine reine Klassenbibliothek zur Unterstützung der Projekte „Client“ und „Server“.

4.1.2 Projekt Client

Das Projekt „Client“ repräsentiert den Spieler. Es benötigt die Logik, um sich auf den Server zu verbinden, die Logik zur Darstellung sämtlicher vom Server gelieferten Daten, sowie die Eingabemasken für Daten, welche dem Server geliefert werden müssen. Spiellogik soll im Client keine vorhanden sein.

4.1.3 Projekt Server

Das Projekt „Server“ ist für die Administration des gesamten Spieles zuständig. Er verwaltet die Spieler, den Spielstand, prüft die Regeln, übernimmt die Wertung und die gesamte Spiellogik. Zusätzlich kümmert er sich um die persistente Datenhaltung, stellt einen Chatserver zur Verfügung und protokolliert alle Ereignisse.

5 Klassenübersicht

5.1 Projekt Common

5.1.1 Schnittstellen

Um bei den Würfeln, Karten und beim Spieler nicht mit den spezifischen Klassen arbeiten zu müssen, haben wir mit den Interfaces eine zusätzliche Abstraktionsschicht erstellt. Somit können wir die Logik für einen Spieler mit dem IPlayer Interface programmieren ohne uns darum kümmern zu müssen ob es sich um einen reellen oder ein Computer Spieler handelt.

5.1.2 Würfel

Wir haben grundsätzlich 7 verschiedene Würfel in 4 verschiedenen Farben. Jedoch haben die Würfel viele Gemeinsamkeiten, weshalb wir nur beim Spielinitialisieren die Würfel mit ihren verschiedenen Eigenschaften initialisieren und danach mit einem generell gültigen Interface arbeiten.

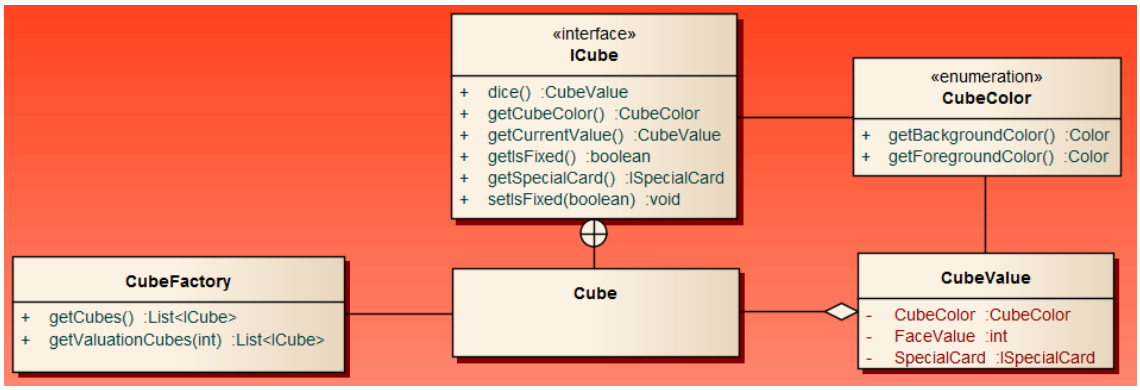

Abbildung 8: Würfel Klassen

Jeder Würfel hat einen Verweis auf eine Sonderkarte, ausgenommen der Todeskarten-Würfel.

Ein CubeValue ist ein gewürfelter Wert, er wird durch eine Farbe und eine Augenzahl definiert. Jeder Würfel hat genau 6 CubeValue's. Ist eine Runde fertig, kann anhand der aktuellen CubeValue geprüft werden, welche Zielkarten zur Verfügung stehen.

Objekt: Weisser Würfel 1	
Color	CubeColor.White (BackgroundColor = Color.Whilte, ForegroundColor = Color.Black)
CurrentCubeValue	CubeColor = CubeColor.White FaceValue = 1 SpecialCard = null
IsFixed	false
SpecialCard	Objektreferenz zur KillerVirus-Karte
CubeValues	(Weiss, 1) (Weiss, 2) (Weiss, 3) (Weiss, 4) (Weiss, 5) (Weiss, 0)

Tabelle 1: Würfel Objekt

5.1.3 Karten

Es gibt 3 verschiedene Arten von Karten: Spezial-, Todes- und Zielkarten. Sie haben alle gewisse Gemeinsamkeiten, nämlich haben sie eine Vorder-, eine Rückseite und eine Kennung. Jede Karte hat einen Eintrag in der Enumeration GameCard, dieser entspricht der Kennung mit der Sie im ganzen Programm eindeutig identifiziert werden kann.

Da die ganzen Kartendetails nur im Package sichtbar sind, kümmert sich die CardFactory, welche Public ist, um das Erstellen der Karten. Im Server sowie im Client soll rein mit den Interfaces und der GameCard Enumeration gearbeitet werden.

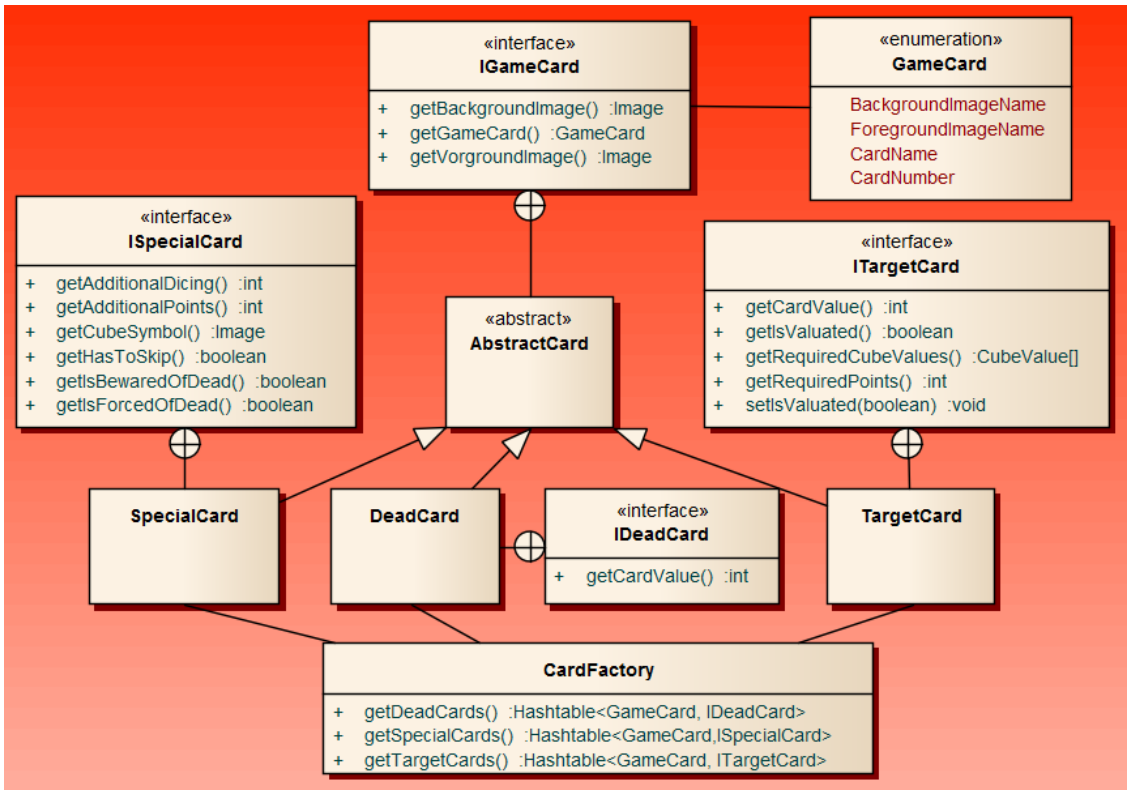

Abbildung 9: Karten Klassen

5.1.3.1 Sonderkarten

Die Sonderkarten haben jeweils einen positiven oder negativen Einfluss, welchen wir anhand verschiedener Attribute in der Klasse SpecialCard abbilden.

- Ente: Einen Extra-Wurf (additionalDicing = 1)
- UFO: eine Runde aussetzen (hasToSkip = true)
- Zeitmaschine: + Zwei Punkte beim Würfeln (additionalPoints = 2)
- Killervirus: Du musst einen Tod nehmen (isForcedOfDead = true)
- Clown: beschützt vom Tod (isBewaredOfDead = true)
- Roboter: nur zweimal Würfeln (additionalDicing = -1)

5.1.3.2 Zielkarten

Die Zielkarten haben einen Wert, sind entweder gewertet oder nicht und man muss entweder gewisse Würfelwerte haben (Bsp. weisses 3, rotes 2 und schwarzes 3) oder für die Dinos eine bestimmte Summe an benötigten Würfelaugen erreichen. Diese Anforderungen werden mit den Properties „requiredCubeValues“ für Riebmans, Yettis, Leminge und Professoren sowie „requiredPoints“ für Dinos abgedeckt. Bei den Dinos ist die Liste der „requiredCubeValues“ leer, bei den Restlichen sind die „requiredPoints“ leer. So kann für alle Karten die gleiche Prüfroutine verwendet werden. Entscheidend im Spiel ist zusätzlich, ob die Karten bereits gewertet wurden.

Dies wird über das Property „isValuated“ abgedeckt, der Wert der Karte wird für die Wertung benötigt und entspricht dem „getCardValue()“.

5.1.3.3 Todes Karten

Die Todeskarten haben rein die Punktzahl als zusätzliche Eigenschaft. Ist diese 0, handelt es sich um den Pudel des Todes.

5.1.4 Daten Transport Objekte

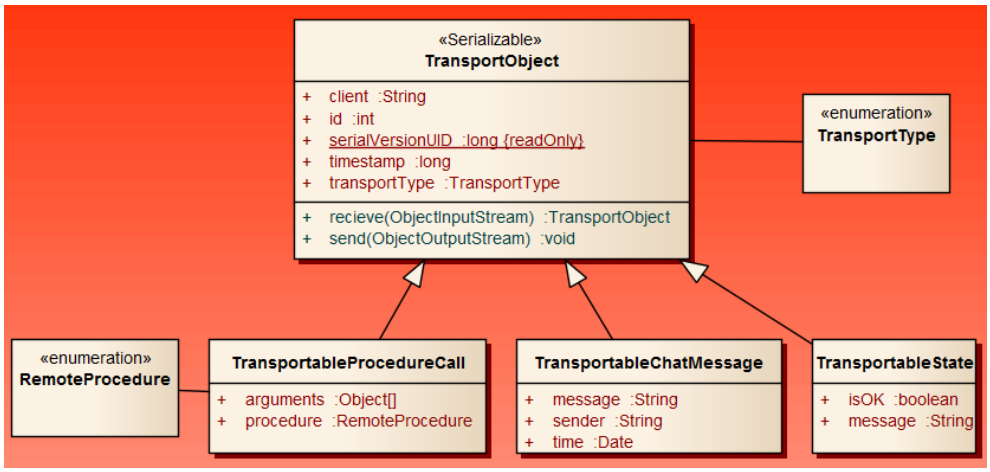

Abbildung 10: Daten Transport Objekte

Bei der Datenübertragung zwischen Server und Client wird immer ein TranportObject übertragen. Ein TransportObject ist die Basisklasse für die 3 Spezialisierungen TransportableProcedureCall, TransportableChatMessage und TransportableState. Sie beinhaltet das Feld TransportType, damit der Empfänger weiss, um welche Spezialisierung es sich handelt. Das Konzept lehnt sich an das Konzept aus der Lektion 10 XML Messaging von Brad an, bei welchem eine Message mit bestimmtem Typ serialisiert wurde.

TransportableProcedureCall

Das Transport Objekt ProcedureCall wird verwendet um eine Funktion beim Empfänger aufzurufen. Die möglichen Funktionen sind in der Enumeration RemoteProcedure definiert. Da jede RemoteProcedure verschiedene Parameter benötigt, können diese dem Transport Objekt als Object Array übergeben werden. Wichtig ist dabei, dass alle übergebenen Objekte auch serialisiert werden können.

TransportableChatMessage

Das Transport Objekt ChatMessage wird für den Nachrichtenaustausch im Chat benötigt. Es wird vom Server empfangen und Broadcast mässig an alle Empfänger weitergeleitet.

TransportableState

Das Transport Objekt State dient als Rückmeldung von Funktionsaufrufen und gibt zurück ob der Aufruf erfolgreich war und falls nicht eine Fehlermeldung.

5.1.5 Spielbrett

Das eigentliche Spielbrett wird durch die Klasse „GameBoard“ abgebildet. Sie implementiert das Interface „ICardHolder“. „CardHolder“ (Kartenhalter) können entweder Spieler oder das Spielbrett sein. Es dient dem Boardmanager, damit die Karten generell einem Kartenhalter zugeteilt und somit die Logik generell gehalten werden kann.

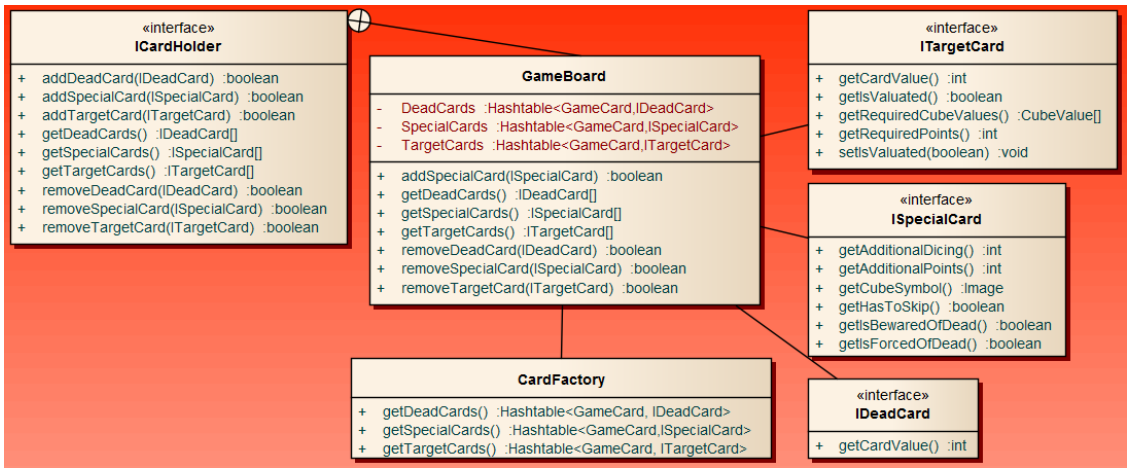

Abbildung 11: Gameboard und Beziehungen

Die Klasse erstellt beim Initialisieren via „CardFactory“ alle Spielkarten. Dies ist je Kartentyp eine lokale Hash-Tabelle. Die Karten, welche auf dem Spielbrett liegen, können mit den Methoden „getTargetCards“, „getSpecialCards“ sowie „getDeadCards“ abgefragt werden. Diese liefern jeweils ein Array mit den Karten zurück.

Über die „add()-“ und „remove()“-Methoden können Karten vom Tisch entfernt oder hinzugefügt werden. Diese Veränderungen finden während dem Spiel durch den Boardmanager des Gameservers statt.

In der folgenden Tabelle eine Übersicht zu den Karten:

KartenTyp	Anzahl	add()-Methode	remove()-Methode
DeadCards	6	Nein	Ja
SpecialCards	6	Ja	Ja
TargetCards	25	Nein	Ja

Tabelle 2: Spielbrett Eigenschaften

- DeadCards: 6 Todeskarten (1 davon Pudel des Todes). Todeskarten können entfernt aber nicht zurückgelegt werden.
- SpecialCards: 6 Spezialkarten. Sie können entfernt und wieder zurückgelegt werden.
- TargetCards: 25 Zielkarten. Sie können nur entfernt werden.

5.1.6 Player

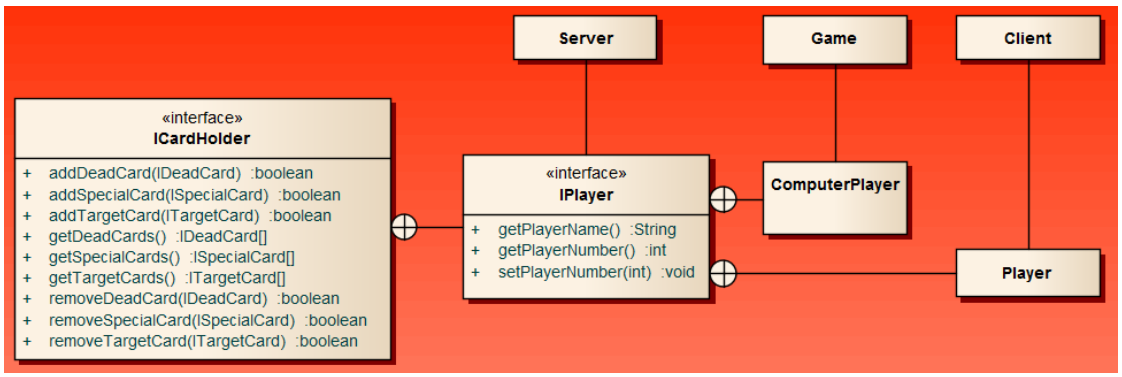

Abbildung 12: Player Klasse mit Kontext

Der Spieler selbst ist eine Spezialisierung eines Kartenhalters. Er ergänzt den Kartenhalter um einen Spielernamen und die Spielernummer.

Der Spieler selbst wird durch den Client erzeugt. Der Server arbeitet lediglich mit dem Interface „IPlayer“. Diese Abstraktion haben wir erstellt, damit wir gegebenenfalls einen „ComputerPlayer“ einbinden können. Der Computerspieler würde durch die Game-Klasse erstellt werden. Da der Computerspieler im Projektscope sehr tiefe Priorität hat, ist dessen Implementierung jedoch noch offen.

5.1.7 Chat Nachricht

Die Klasse ChatMessage bildet eine Chat Nachricht ab. Die Klasse ist sehr simpel und enthält eine ID einen Absender, die Nachricht und die Zeit zu welcher sie gesendet wurde.

5.1.8 Platzierung

Die Klasse Ranking bildet eine Platzierung ab. Die Klasse dient rein zur Datenhaltung und besteht aus der ID des Spieles, der ID des Spielers, der Punktzahl der Platzierung im Spiel und der Gesamtplatzierung.

5.1.9 ServiceLocator

Die Klasse „ServiceLocator“ verwaltet die zentralen Referenzen zu Objekten welche an mehreren Orten im Programm benötigt werden. Diese beinhalten zum Beispiel das „Logging“ und den „Translator“.

5.1.10 ResourceLoader

Die Klasse „ResourceLoader“ kümmert sich um das Laden externer Ressourcen. In erster Linie handelt es sich dabei um die Bilder, welche im Spiel angezeigt werden. Sie ist statisch und kann nicht initialisiert werden. Ihre Methoden werden direkt mit „ResourceLoader.getImage(„TestBild.png“)“ aufgerufen.

5.1.11 Translator

Die Translator Klasse kümmert sich um die Sprachtexte und liefert abhängig vom gesetzten Locale die benötigten Texte zurück. Bis auf kleiner Anpassungen, wird sie dem im Unterricht behandeltem Beispiel aus dem ApplicationFramework von Brad entsprechen.

5.2 Projekt Server

5.2.1 Main

Die Mainklasse ist der Einstiegspunkt um die Serveranwendung zu starten. Sie erstellt den ClientAwaiter Thread, den Server und startet das GUI.

5.2.2 Server

Die Server Klasse ist der zentrale Kommunikationsknoten welche bei Anfragen kontaktiert wird. Sie stellt alle Grundfunktionalitäten welche in der Anforderungsspezifikation ermittelt wurden zur Verfügung und leitet alle Anfragen gemäss ihrer Zuständigkeit an die Hilfsklassen weiter.

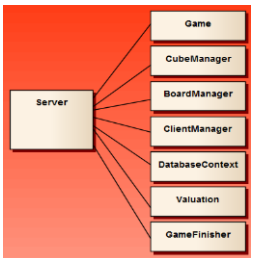

Abbildung 13: Server Klasse

5.2.3 Verbindungs Management

Die Serveranwendung hat mehrere Klassen welche sich um die Verbindungen mit den Clients kümmern. Wie diese miteinander interagieren können Sie im Kapitel „Server - Client Zusammenspiel“ nachlesen.

5.2.3.1 ClientAwaiter

Der ClientAwaiter Thread öffnet ein Server Socket und wartet auf Anfragen der Clients. Sobald sich ein Client verbindet erstellt er pro Client einen ClientConnection Thread und registriert diesen im ClientManager.

5.2.3.2 ClientConnection

Für jeden Client, welcher sich auf den Server verbindet, wird ein ClientConnection Thread erstellt. Wenn der Client eine Anfrage schickt leitet der ClientConnection Thread diese an den Server weiter. Aktualisierungen im Spiel leitet der Server via ClientManager an die entsprechenden Clients weiter.

5.2.3.3 ClientManager

Der „ClientManager“ verwaltet die verbundenen Clients, schickt ihnen Aktualisierungen weiter und macht die Zuordnung zwischen Client und Player. Er wird gemäss dem Singelton-Pattern umgesetzt.

5.2.4 Computer Spieler

Die Klasse ComputerPlayer soll einen Spieler simulieren für den Single Player Modus. Sie hat in unserem Projekt Scope die kleinste Priorität, wir werden uns, falls genügend Zeit vorhanden ist, später mit der technischen Umsetzung befassen.

5.2.5 TextAreaHandler

Der TextAreaHandler wird benötigt um die Log Ausgaben im GUI anzuzeigen. Unser zentraler Logger, welche vom ServiceLocator im Common Projekt verwaltet wird, hat grundsätzlich einen LogHandller, welcher die Log Datei führt. Um die Logging Ausgaben im Server anzuzeigen registrieren wir noch einen zweiten Loghandler, welcher die Ausgaben in einer TextArea darstellt.

Wir lehnen uns dabei an den im Unterricht bei Brad behandelten Stoff an und adaptieren den TextAreaHandler aus dem Beispiel Projekt der Lektion 10 Xml Messaging.

5.2.6 Game

Die Game Klasse steuert das Spiel gesamthaft und ist zuständig dafür das Spiel zu aktivieren, die aktiven Spieler zu ändern und das Spiel abzuschliessen, wenn nur noch 5 oder weniger Zielkarten auf dem Spielbrett vorhanden sind. Dazu erstellen wir folgende Klasse:

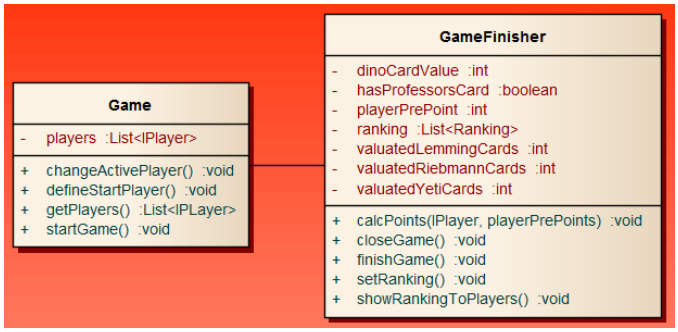

Abbildung 14: Klasse Game

Der Ablauf innerhalb der Klasse sieht wie folgt aus:

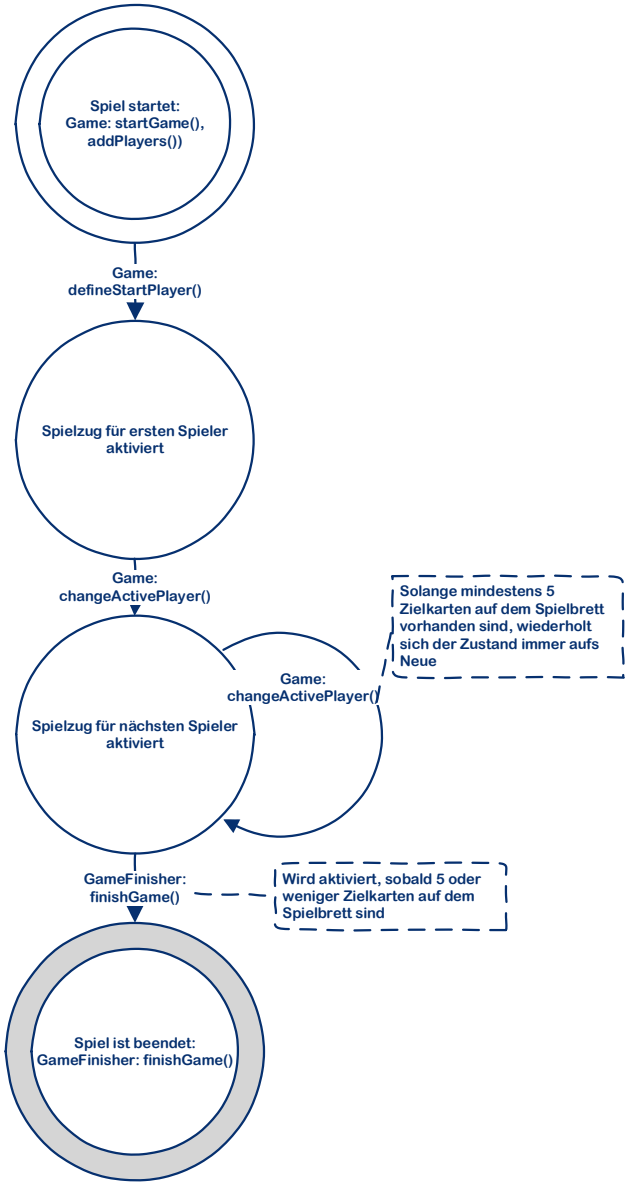

Abbildung 15: Ablauf Klasse Game

5.2.7 Boardmanager

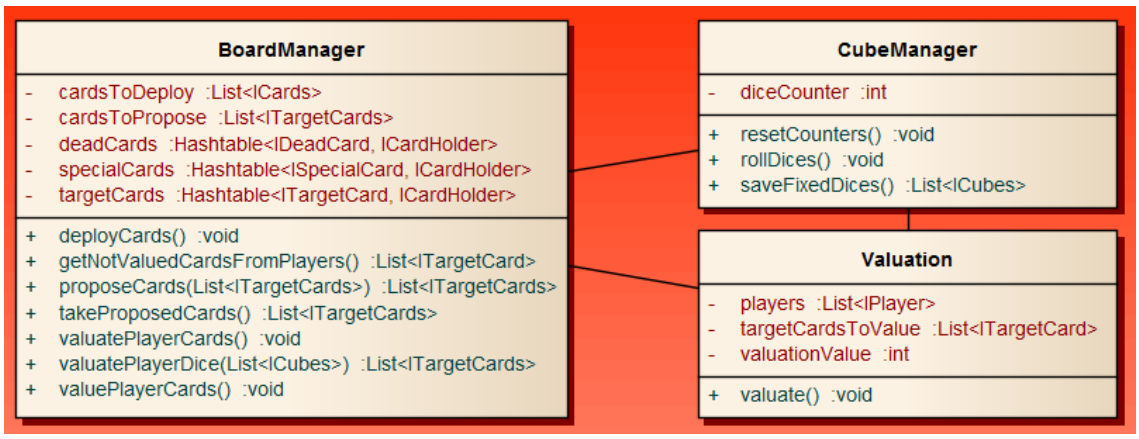

Abbildung 16: Klasse BoardManager

Mit den Variablen targetCards, specialCards und deadCards merkt sich der BoardManager jeweils, bei wem sich die verschiedenen Karten jeweils befinden (auf dem Board oder bei einem Spieler). Dazu wird bei den Zielkarten und den Todeskarten noch gespeichert, ob die Karten umgedreht (gewertet) sind oder nicht.

5.2.7.1 Methoden

CubeManager: rollDices

Ruft die Funktion vom CubeManager auf, um die Würfel des Spielers zu würfeln. Siehe dazu mehr im Kapitel „CubeManager“.

Valuation: valuate()

Die Karten aller Mitspieler werden gewertet gemäss pinkem Würfel.

getNotValuedCardsFromPlayers

Wird von der Klasse Valuation aufgerufen, um zu prüfen, welche nicht-gewerteten Karten bei den Mitspielern vorhanden sind, um diese entsprechend gemäss pinkem Würfel zu werden.

valuatePlayerCards

Wertet die Karten der Mitspieler aus, indem der Wert der nicht-gewerteten Karten mit dem Wert des pinken Würfel verglichen wird.

valuePlayerCards

Wertet die Karten der Mitspieler effektiv nach dem Wert des pinken Würfels anhand des Vergleichs aus der Methode „valuatePlayerCards“.

valuatePlayerDice

Die Methode wertet den Würfel-Wurf von dem aktiven Spieler aus, sobald der Spielzug abgeschlossen wurde und die Karten der Mitspieler über „Valuation: valuate“ gewertet wurden. Anschliessend wird die Methode „proposeCards“ aufgerufen, insofern der Spieler mehrere Zielkarten zur Auswahl hat. Hat der Spieler keine Auswahl, wird direkt die Methode „deployCards“ ausgeführt und dem Spieler die entsprechenden Ziel-, Sonder- oder/und Toderkarten zugewiesen.

proposeCards

Mit dieser Funktion wird dem Spieler eine Auswahl an Zielkarten vorgeschlagen, welche er auswählen kann, insofern er mehrere Karten zur Verfügung hat.

Die Voraussetzung für diese Methode ist die Methode „valuatePlayerDice()“.

takeProposedCards

Der Spieler ruft diese Funktion auf, indem er anhand der vorherigen Mehode „proposeCards“ wählt, welche Zielkarten er nehmen möchte. Die Methode speichert anschliessend die ausgewählten Karten und verteilt danach mit „deployCards“ die Karten.

deployCards

Die Karten werden dem Spieler zugewiesen, nachdem der Würfel-Wurf nach einem Spielzug über die Methode „valuatePlayerDice“ ausgewertet wurden (die Methode proposeCards könnte noch vorher zum Zug kommen, wenn der Spieler zwischen Zielkarten wählen darf).

Mit dieser Methode werden dem Spieler die erwürfelten / gewählten Zielkarten und Sonderkarten zugewiesen. Hat der Spieler keine Zielkarten erwürfelt, wird ihm gemäss pinkem Würfel die entsprechende Todeskarte zugewiesen.

Die Logik für die Verteilung der Todeskarte funktioniert dabei wie folgt:

- Todeskarte wird gemäss Wert von pinkem Würfel verteilt (egal, ob Karte auf Board oder bei anderem Spieler)
- Ist die Todeskarte bereits bei dem entsprechendem Spieler, wird die nächst höhere Todeskarte genommen und dem Spieler zugewiesen (z.B. würfelt er den Tod 4 und hat diesen schon, bekommt er den Tod 5… hat er den Tod 5 auch schon, bekommt er den Pudel des Todes (Wert 0) etc.)

5.2.7.2 Ablauf eines Spielzuges

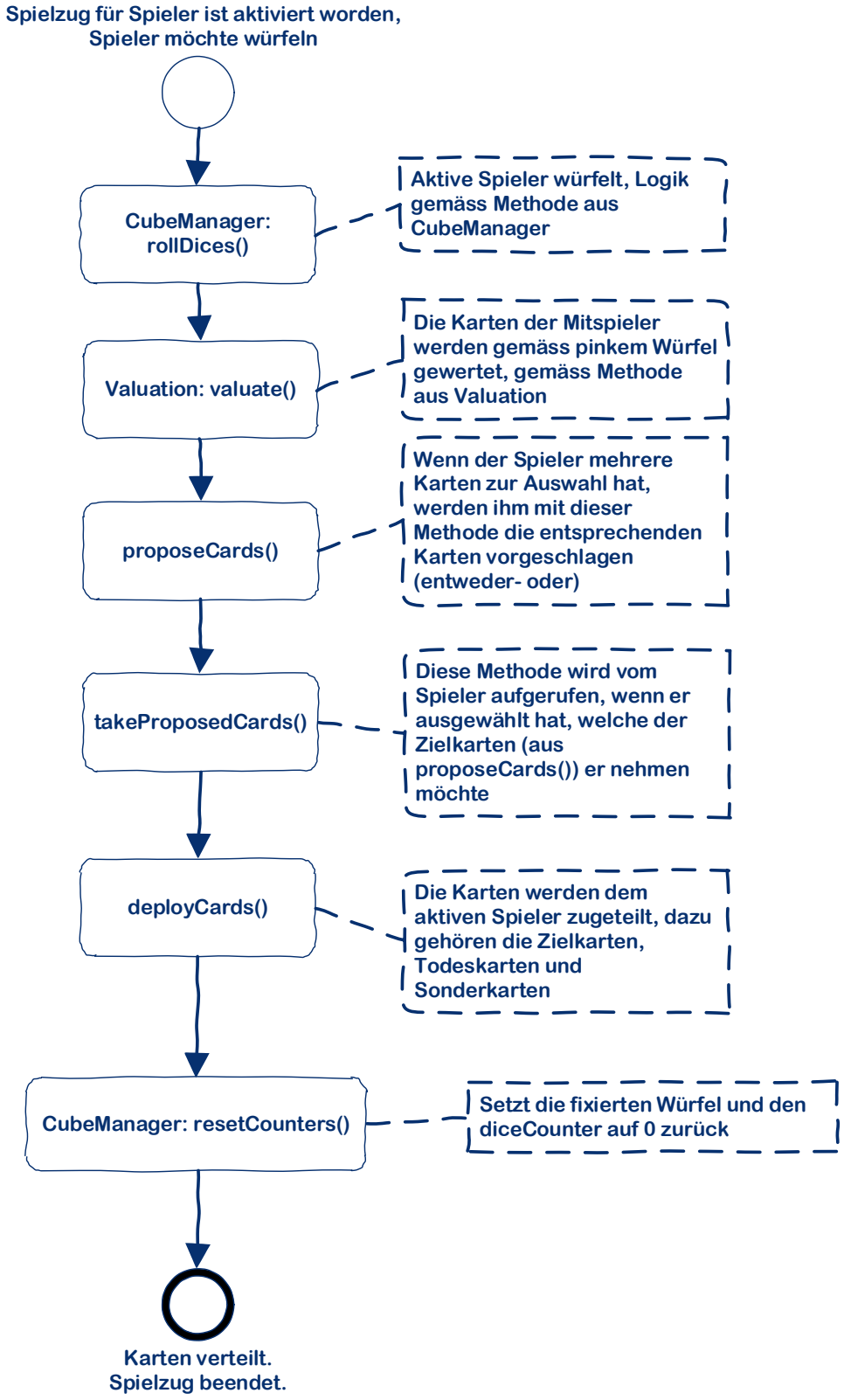

Abbildung 17: Ablauf Klasse BoardManager

5.2.8 GameFinisher

Die Klasse GameFinisher beendet das Spiel:

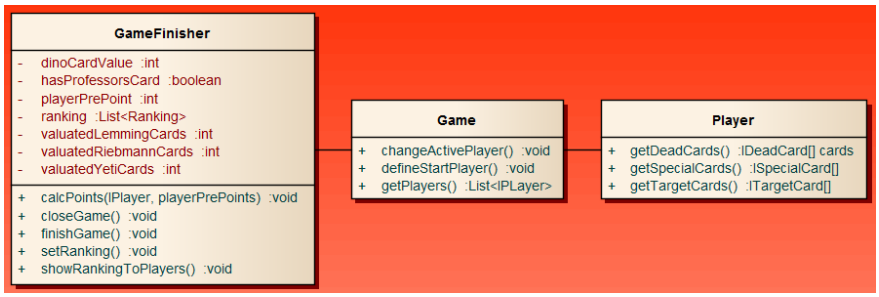

Abbildung 18: Klasse GameFinisher

Dazu werden die Punkte der Spieler kalkuliert und ausgegeben, die Rangliste wird im Hintergrund nachgeführt. Das Spiel wird anschliessend geschlossen.

Der Ablauf für die Funktion „finishGame“ sieht wie folgt aus:

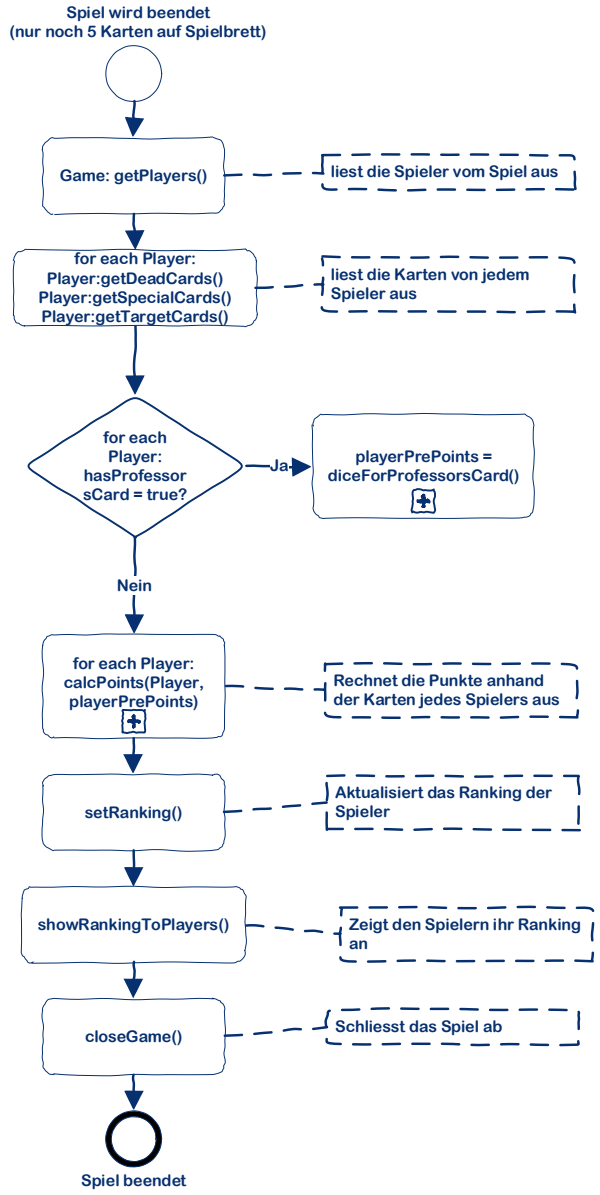

Abbildung 19: Ablauf Methode finishGame von GameFinisher

Der Ablauf für die Funktion „calcPoints“ läuft wie folgt:

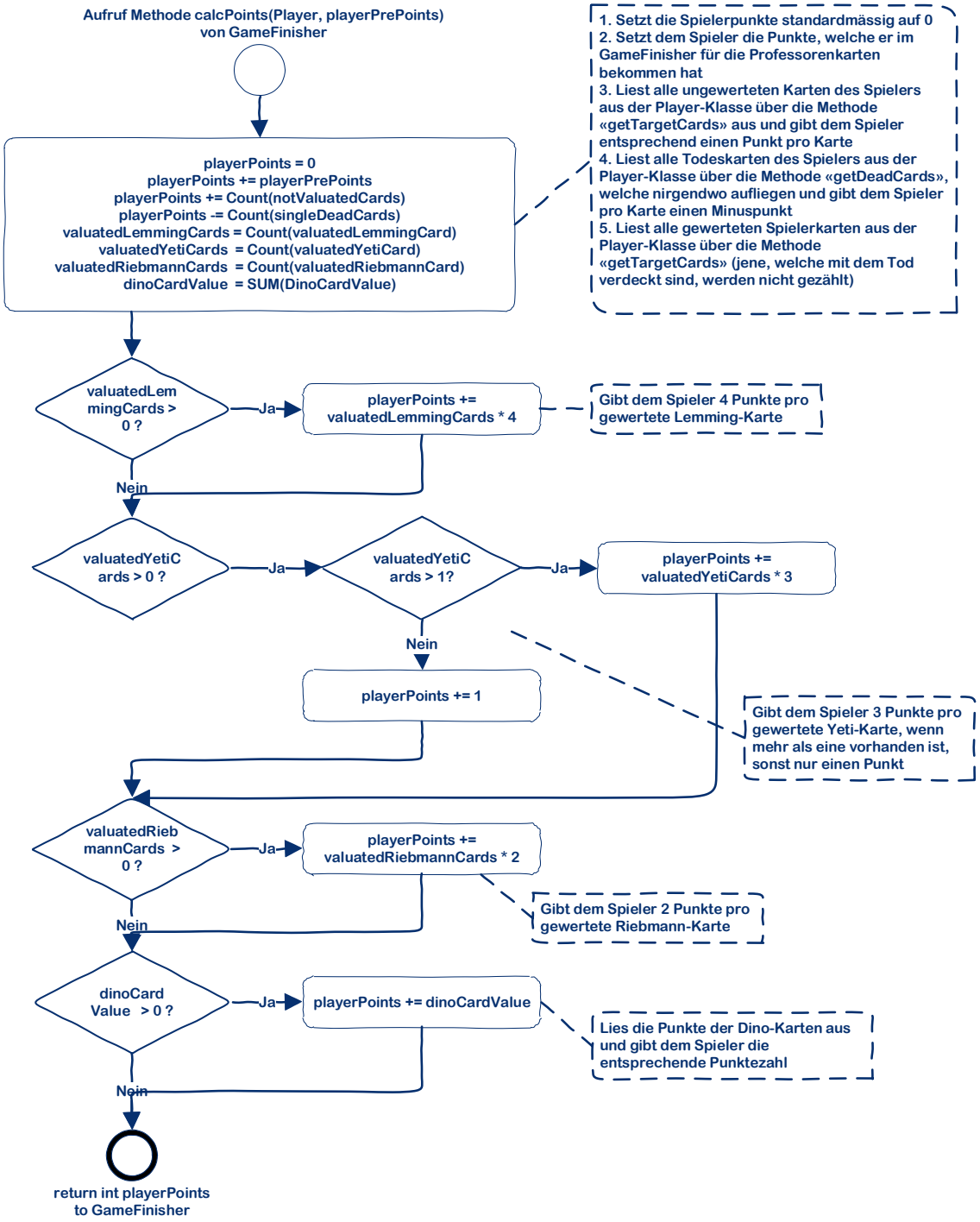

Abbildung 20: Ablauf Methode caldPoints von GameFinisher

5.2.9 Datenbank Kontext

Die Klasse DatabaseContext kümmert sich um das Speichern der Spieler und Spielergebnisse. Sie erstellt zur Verwaltung eine XML Datei im Verzeichnis des Servers. Näheres dazu finden Sie im Kapitel „Datenmodell“.

5.2.10 Wertung

Für die Wertung haben wir die Klasse Valuation erstellt, welche eine Liste von den Mitspielern, den Wert des pinken Würfelt und die Zielkarten der Mitspieler speichert, welche zu werten sind.

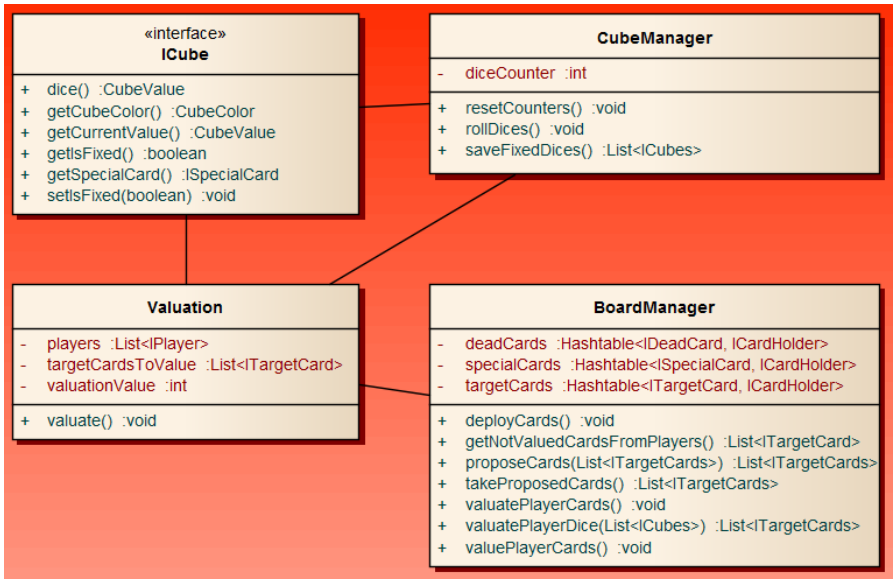

Abbildung 21: Klasse Valuation

Die Methode „valuate“ wird vom BoardManager aufgerufen, sobald der aktive Spieler die maximale Anzahl von Würfelzügen abgeschlossen hat. Der Ablauf sieht der Methode sieht wie folgt aus:

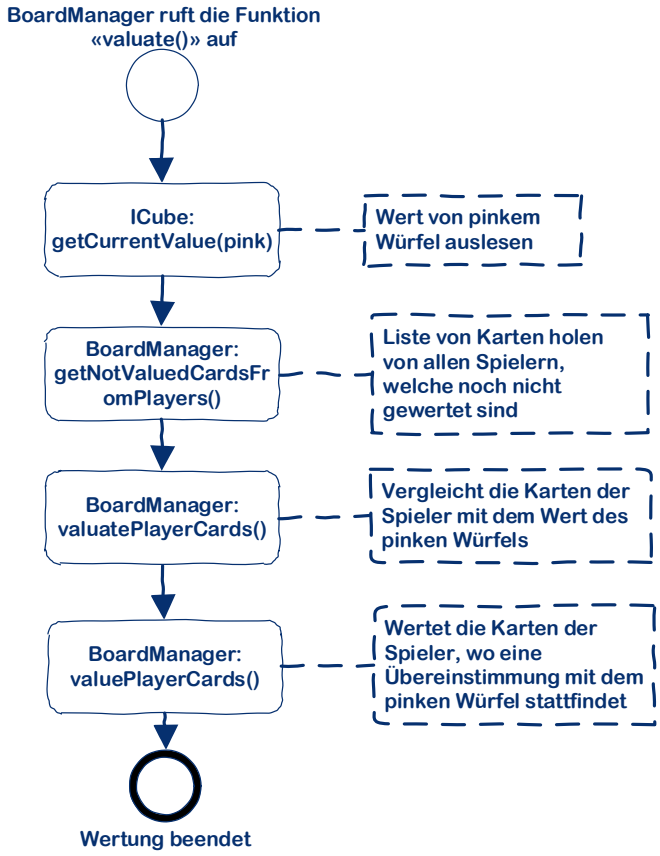

Abbildung 22: Ablauf Methode valuate von Klasse Valuation

5.2.11 Würfeln

Für das Würfel-Szenario erstellen wir die Klasse CubeManager:

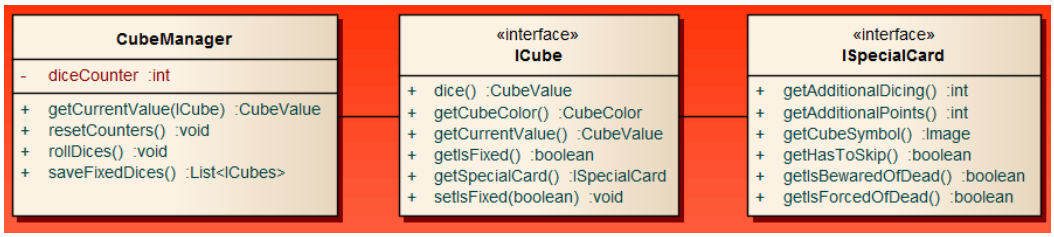

Abbildung 23: Klasse CubeManager

Diese Klasse merkt sich wie oft der aktive Spieler schon gewürfelt hat und merkt sich die fixierten Würfel. Die Logik hinter der Methode „rollDices“ sieht wie folgt aus:

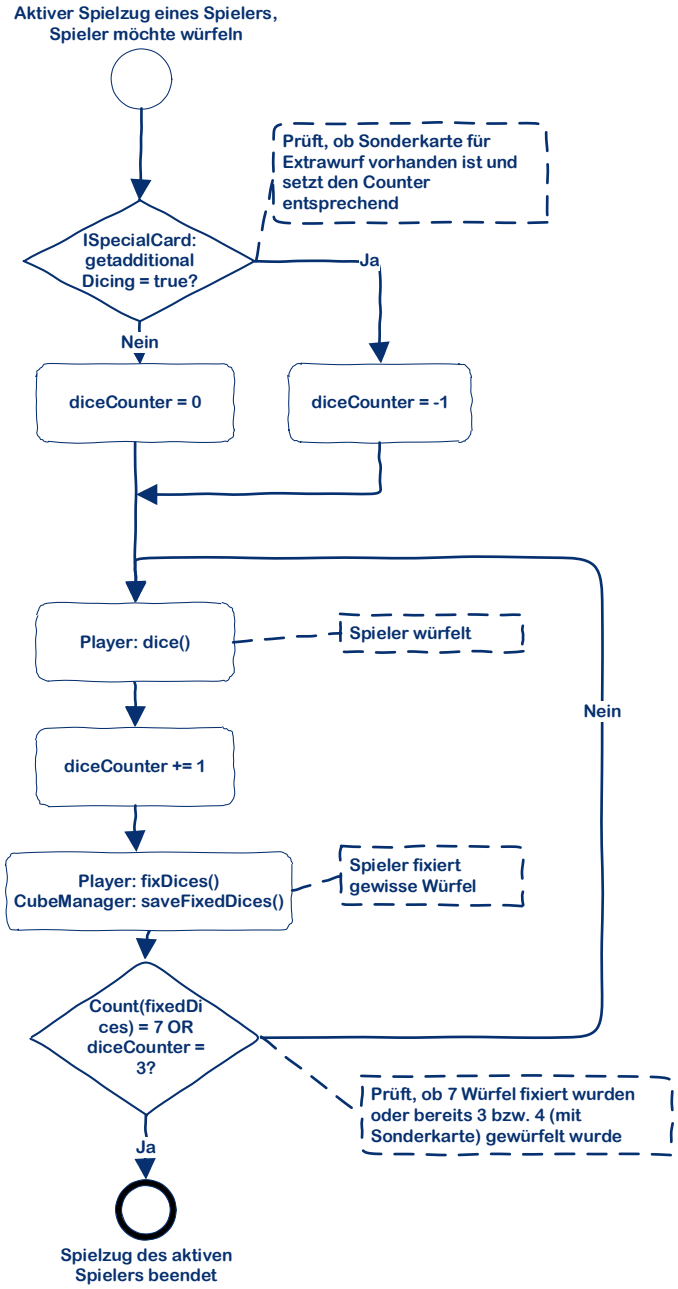

Abbildung 24: Ablauf der Methode rollDices der Klasse CubeManager

5.3 Projekt Client

5.3.1 Main

Die Mainklasse ist der Einstiegspunkt für die Client Anwendung. Sie erstellt den Splashscreen und instanziiert die benötigten Objekte.

5.3.2 Client

Der Client ist für die Verbindung zum Server verantwortlich. Er baut die Verbindung zum Server auf, stellt Anfragen und wandelt die Antworten wieder in Java-Objekte um. Der genaue Ablauf ist im Kapitel „Server - Client Zusammenspiel“ beschrieben.

Zusätzlich bietet der Client für die Controller-Klassen der GUIs eine Methode an, um sich als Listener zu registrieren. Die GUIs werden über diese Funktionalität stetig über Änderungen informiert und können sich aktualisieren.

5.3.3 Server Verbindung

Um die Verbindung zum Server aufzubauen und aufrechtzuerhalten haben wir zwei Hilfsklassen. Wie diese Klassen im Detail zusammenarbeiten können Sie im Kapitel „Server - Client Zusammenspiel“ nachlesen.

5.3.3.1 ServerConnector

Die ServerConnector Klasse stellt die Verbindung zum Server her. Sobald die Anfrage vom Server akzeptiert wurde erstellt sie den ServerConnection Thread.

5.3.3.2 ServerConnection

Die ServerConnection ist ein Thread, welcher im Hintergrund läuft. Er hält die Verbindung zum Server offen und dient als Kommunikationsknoten. Benachrichtigungen vom Server leitet er an den Client weiter und der Client kann via Server Connection Anfragen verschicken.

6 Server- / Client-GUI

6.1 Übersicht

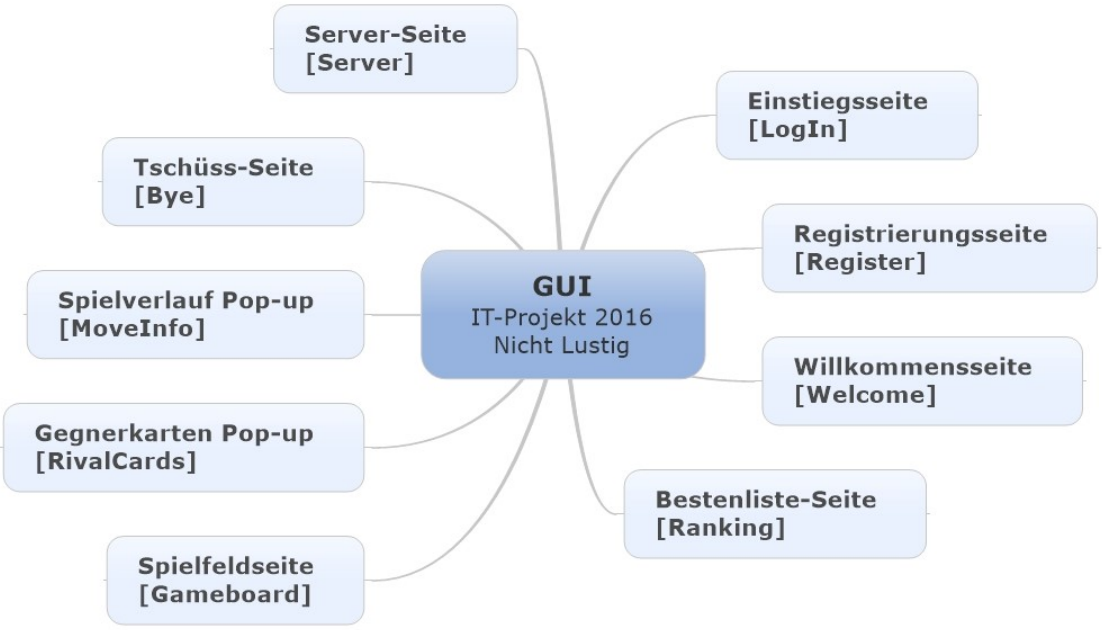

Abbildung 25: GUI-Übersicht

In dieser Übersicht sind sämtliche Seiten ersichtlich, welche wir für unser Spiel erstellen möchten. Ausserdem wurden in der Anforderungsspezifikation, welche wir mit Phase 1 abgegeben haben, die einzelnen Seiten bereits genauer vorgestellt. Im obenstehenden Mind Map sind zusätzlich bereits die Klassennamen ersichtlich (siehe eckige Klammern).

6.1.1 GUI-Konzept

Alle Grafischen Benutzeroberflächen unseres Spieles werden wir mit der Sprache JavaFX, welche wir im Programmieren 2 bei Lukas Frey gelernt haben, gemäss dem Model-View-Controller-Pattern (MVC) umsetzen.

Für das MVC-Pattern haben wir uns entschieden, da uns Bradley Richards dieses im Modul Software Engineering 2 näher gebracht hat. Insbesondere hat er uns sein JavaFX-Framework gezeigt, auf welchem wir unsere GUI-Programmierung aufsetzen möchten.

Das Pattern dient dazu, Programme mit Benutzeroberflächen und mit klaren Schnittstellen zu strukturieren.

Die Konzeption zur Umsetzung aller GUI-Seiten sieht daher wie folgt aus:

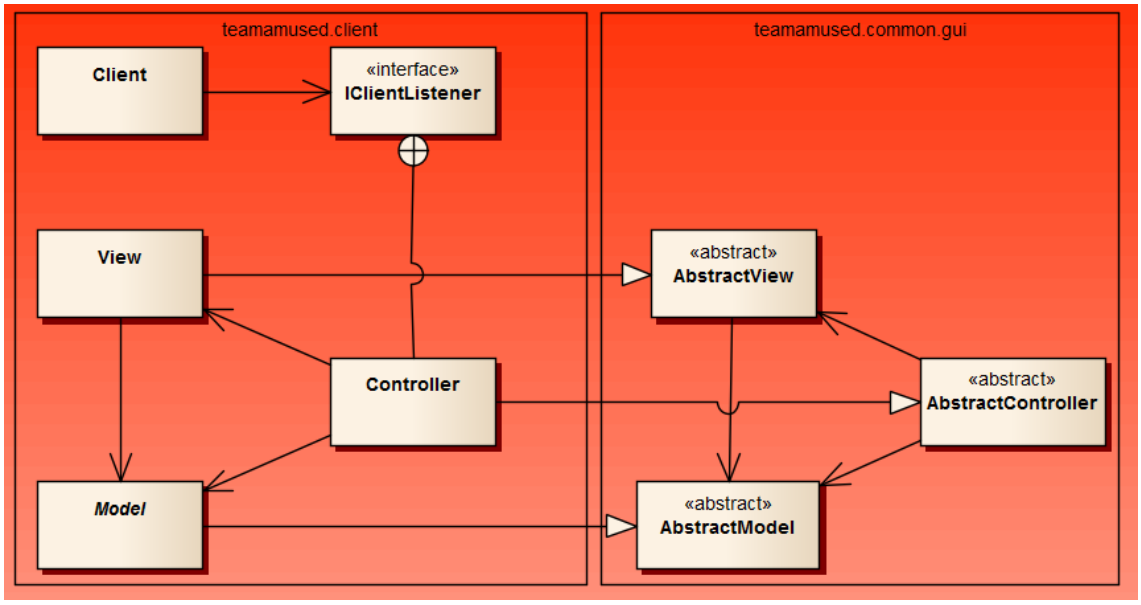

Abbildung 26: MVC Umsetzung

Pro Oberfläche werden wir uns an das obige Schema halten und je eine Model-, eine View- und eine Controllerklasse schreiben.

Alle Model-, View- und Controller-Klassen erben von den abstrakten Klassen, welche sich im Projekt „teamamused.common“ befindet. In diesen drei Klassentypen werden wir die Grundlogik des MVC-Patterns implementieren. Diese drei Klassen werden wir gemäss der Vorlage aus dem Software Engineering 2 Kurs von Brad übernehmen.

Die Controller-Klassen übernehmen die Steuerung und sorgen dafür, dass Benutzerinteraktionen wirksam werden. Damit die Controller erfahren, wann sich eine Aktualisierung im Client ergibt und sie die Views entsprechend steuern müssen, werden wir das Observer-Pattern anwenden. Die Controller werden als Observer auftreten und das Interface „IClientListener“ implementieren. Über das Interface registrieren sie sich in der Liste des Clients, welcher als Observable auftritt, zur Benachrichtigung bei Aktualisierungen dessen. Alle Methoden, welche bei Ereignissen aufgerufen werden, sind im Interface als Default-Methoden programmiert. So kann der Controller selber entscheiden, welche Aktualisierungen er braucht, indem er die entsprechenden Methoden überschreibt. Der Controller behandelt zusätzlich auch die Ereignisse der View. Wird beispielsweise ein Button betätigt, reagiert er entsprechend und erstellt gegebenenfalls eine Anfrage an den Client. Kommen vom Client neue Daten, übergibt er diese ans Model und teilt der View mit, dass sie sich aktualisieren muss.

Die View übernimmt reine Darstellungsaufgaben. Sie ist die Präsentationsschicht und greift auf die Daten des Models zu, um diese darzustellen. Sie wird keine Geschäftslogik enthalten.

Im Model finden sich alle für die View benötigten Daten. Auch das Modell ist völlig frei von irgendwelcher Geschäftslogik. Es kennt weder die View-, noch die Controllerklassen.

Ziel der gewählten Architektur ist es, dass wir die Darstellung von der Geschäftslogik trennen können. Wird beispielsweise eine Anpassung an der View erforderlich, sollte dies keinen Einfluss auf die Logik haben.

Die genaue Detailspezifikation der einzelnen Screens wird sich während der Implementierungsphase ergeben. Da wir der Agilität entsprechend vorgehen möchten, werden wir die Dokumentation laufend nachführen und konkretisieren.

7 Server - Client Zusammenspiel

Der Datenaustausch zwischen Client und Server erfolgt via Java Objekt Serialisierung über eine Websocket Verbindung. Die zentralen TransportObject Klassen sind im Projekt „Common“ definiert und daher im Client sowie im Server vorhanden. Eine genauere Beschreibung zu deren Eigenschaften finden Sie in der Klassenübersicht.

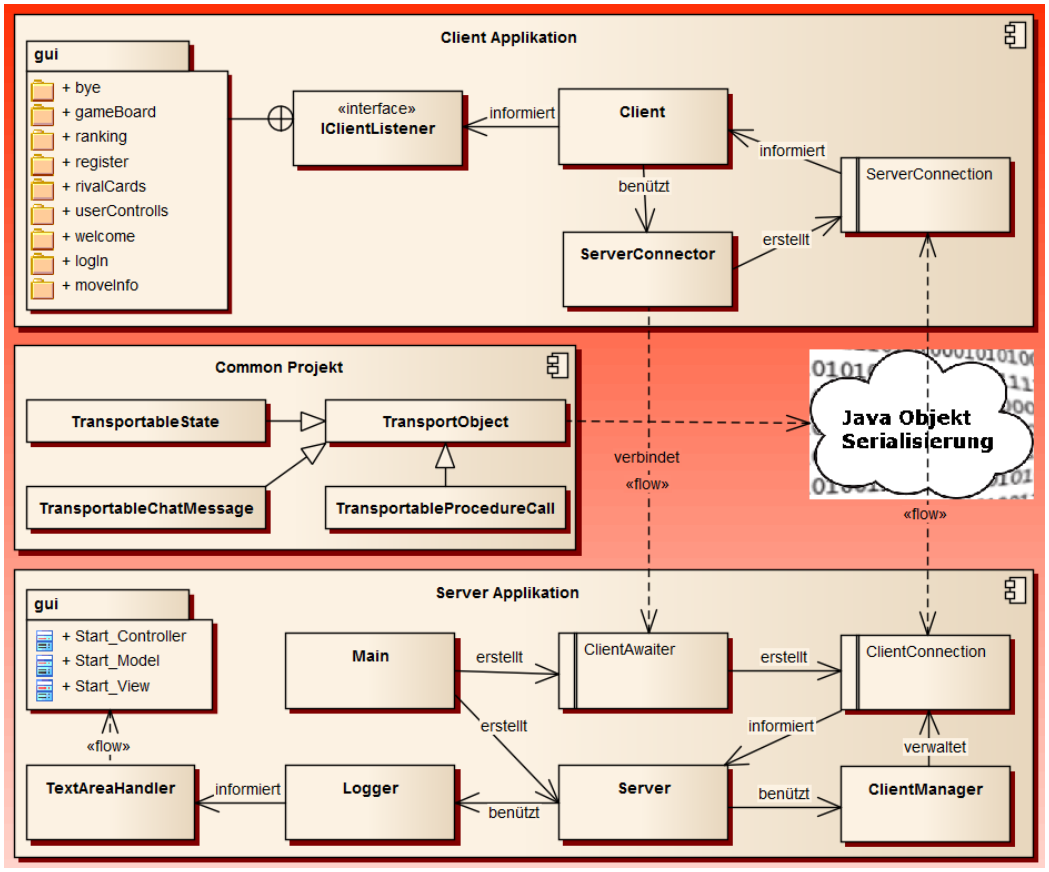

Abbildung 27: Client Server Kommunikation

Ablauf Clientseitig:

1. Der Client fordert den ServerConnector auf eine Verbindung zum Server aufzubauen.
2. Der ServerConnector baut via Websocket eine Verbindung zum Server auf.
3. Ist die Verbindung erfolgreich aufgebaut, erstellt er einen ServerConnection Thread, welcher ab diesem Moment die Verbindung zum Server aufrechterhält.
4. Der Client erhält vom ServerConnection Thread Informationen vom Server weitergeleitet und kann via der ServerConnection Anfragen an den Server schicken.

Ablauf Serverseitig:

1. Beim Start des Servers wird ein ClientAwaiter Thread erstellt.
2. Der ClientAwaiter erstellt ein ServerSocket auf welches sich die Clients Verbinden können.
3. Sobald sich ein Client verbindet, akzeptiert der ClientAwaiter die Verbindung und erstellt für jeden Client einen eigenen ClientConnection Thread.
4. Der ClientAwaiter registriert den neuen Client beim ClientManager und wartet weiter auf die nächste Anfrage.
5. Falls auf dem Server eine Zustandsänderung erfolgt, leitet der Server die Änderungen via „ClientManager“ an alle registrierten Clients weiter.
6. Erfolgt eine Anfrage eines Clients informiert der ClientConnection Thread den Server.

8 Datenstruktur

In der Anforderungsspezifikation haben wir als kann Anforderung (nice to have) den UseCase Rangliste nachführen. Um die Spiele und die Platzierungen zu verwalten, würden wir auf dem Server eine XML Datei erstellen.

Diese verwaltet folgende Entitäten:

- Spieler (Player)
- Spiel (Game)
- Platzierung (Ranking)

Dabei spielt ein Spieler in mehreren Spielen und ein Spiel hat mehrere Spieler.

Ein Spiel hat mehrere Platzierungen und jede Platzierung gehört zu einem Spiel sowie zu einem Spieler.

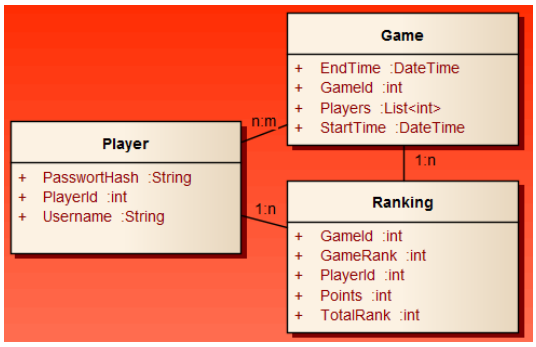

Abbildung 29: Datenstruktur

Wie die XML Datei genau aussehen soll überlassen wir dem java.beans.XMLEncoder, wir stellen uns das in etwa wie in der Abbildung rechts vor.

Beim Starten des Servers würden die Daten aus der Datei jeweils geladen werden und beim Beenden des Spieles wider gespeichert.

```
▼<Server>
  ▼<Players>
    ▼<Player>
       <PlayerId>1</PlayerId>
       <Username>Spieler 1</Username>
       <PasswordHash>xye86</PasswordHash>
     </Player>
    ▼<Player>
       <PlayerId>2</PlayerId>
       <Username>Spieler 2</Username>
       <PasswordHash>2as87</PasswordHash>
     </Player>
   </Players>
  ▼<Games>
    ▼<Game>
       <GameId>1</GameId>
       <StartTime>2016-05-21-10:25:56.123</StartTime>
       <EndTime>2016-05-21-10:55:04.567</EndTime>
      ▼<GamePlayers>
         <GamePlayer>1</GamePlayer>
         <GamePlayer>2</GamePlayer>
       </GamePlayers>
     </Game>
    ▼<Game>
       <GameId>2</GameId>
       <StartTime>2016-05-21-10:25:56.123</StartTime>
       <EndTime>2016-05-21-10:55:04.567</EndTime>
      ▼<GamePlayers>
         <GamePlayer>1</GamePlayer>
         <GamePlayer>2</GamePlayer>
       </GamePlayers>
     </Game>
   </Games>
  ▼<Rankings>
    ▼<Ranking>
       <GameId>1</GameId>
       <PlayerId>2</PlayerId>
       <Points>22</Points>
       <GameRank>1</GameRank>
       <TotalRank>2</TotalRank>
     </Ranking>
    ▼<Ranking>
       <GameId>1</GameId>
       <PlayerId>1</PlayerId>
       <Points>13</Points>
       <GameRank>2</GameRank>
       <TotalRank>3</TotalRank>
     </Ranking>
    ▼<Ranking>
       <GameId>2</GameId>
       <PlayerId>2</PlayerId>
       <Points>25</Points>
       <GameRank>1</GameRank>
       <TotalRank>1</TotalRank>
     </Ranking>
    ▼<Ranking>
       <GameId>2</GameId>
       <PlayerId>1</PlayerId>
       <Points>11</Points>
       <GameRank>2</GameRank>
       <TotalRank>4</TotalRank>
     </Ranking>
   </Rankings>
 </Server>
```

Abbildung 28: XML Datei der Daten