UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Heinz Nixdorf Institut
Fachgruppe Softwaretechnik
Zukunftsmeile 1
33102 Paderborn

Analyse & Entwurf

im Rahmen des Softwaretechnikpraktikums 2017

Team 10

smarten

professional software development

Betreuerin: Christin Löer

Paderborn, den 7. Juni 2017

Autoren:

David Bock	Niklas Doppelstein
Max Krömker	Anke Küstner
Björn Luchterhandt	Sebastian Pranger
Jost Rossel	Wolfgang Schaperdot
René Scherf	Hanna Siek
Moritz Thiele	Robin Wulfes

Inhaltsverzeichnis

1 Analyse - Übersicht über Systemkomponenten **1**

1.1 Architektur . . . 1
1.1.1 Modularität . . . 1
1.1.2 Server-Client-Architektur . . . 1
1.1.3 Entity-Control-Boundary . . . 2
1.1.4 Model-View-Controller . . . 3
1.1.5 Strategy-Pattern . . . 4
1.1.6 Observer-Pattern . . . 5
1.1.7 Singleton-Pattern . . . 6
1.2 Hauptkomponenten . . . 6
1.2.1 GameEngine . . . 6
1.2.2 Spielkonfigurator . . . 6
1.2.3 PlayerClient . . . 6
1.2.4 SmartphonePlayerClient . . . 7
1.2.5 AIPlayerClient . . . 7
1.2.6 ObserverClient . . . 7
1.2.7 SmartphoneObserverClient . . . 7
1.2.8 DesktopObserverClient . . . 7
1.3 Externe Komponenten . . . 7
1.3.1 Eclipse . . . 7
1.3.2 Java Development Kit . . . 7
1.3.3 Android 4.4 (API Level 19) . . . 8
1.3.4 Eclipse Modeling Framework . . . 8
1.3.5 Plugin Development Environment . . . 8
1.3.6 JavaFx . . . 8
1.3.7 Android Development Tools / Android Studio . . . 8
1.3.8 emfjson Jackson . . . 9

2 Analyse - Clients **10**

2.1 Sequenzdiagramme . . . 10
2.1.1 Registrieren . . . 10
2.1.2 Wegfelder organisieren . . . 11
2.1.3 Spielzug durchführen . . . 13
2.2 Klassendiagramm . . . 14
2.2.1 GUI . . . 15
2.2.2 ViewerGUIControl . . . 15
2.2.3 PlayerGUIControl . . . 15
2.2.4 GameList . . . 15
2.2.5 ClientControl . . . 15
2.2.6 GameModel . . . 15

2.3 TileOrganizer 15

3 Analyse - GameEngine **16**
3.1 Sequenzdiagramme 16
3.1.1 Partie ausrichten 16
3.2 Klassendiagramm 17
3.2.1 EngineControl 18
3.2.2 GUI 18
3.2.3 OpenFileDialog 18
3.2.4 GUIControl 18
3.2.5 TournamentControl 18
3.2.6 GameControl 18
3.2.7 ClientInterface 18
3.2.8 Game 19
3.2.9 TournamentBracketModel 19
3.2.10 Pool 19
3.2.11 GameControlList 19
3.2.12 Token 19
3.2.13 Placement 19

4 Analyse - Spielkonfigurator **19**
4.1 Sequenzdiagramme 19
4.1.1 Spielkonfiguration erstellen 21
4.2 Klassendiagramm 23
4.2.1 GUI 23
4.2.2 GUIControl 23
4.2.3 ConfigurationControl 23
4.2.4 Configuration 24
4.2.5 OpenFileDialog 24
4.2.6 SaveFileDialog 24
4.2.7 TileList 24
4.2.8 Tile 24

5 Entwurf **25**

6 Entwurf - Clients **25**
6.1 Entwurfs-Klassendiagramm 25
6.1.1 GUI 26
6.1.2 ViewerGUIControl 26
6.1.3 PlayerGUIControl 26
6.1.4 TileOrganizer 26
6.1.5 ClientControl 26

6.1.6 GameList . . . 27
6.1.7 ClientToServerConnection . . . 27
6.1.8 GameModel . . . 27

7 Entwurf - KIClient **27**
7.1 Generelles Vorgehen . . . 27
7.2 Entwurfs-Klassendiagramm . . . 28
7.3 Elemente der KI . . . 28
7.3.1 AIController . . . 28
7.3.2 TimeTracker . . . 28
7.3.3 Strategy und StrategyPicker . . . 28
7.3.4 Move Generator und MoveTree . . . 29

8 Entwurf - GameEngine **30**
8.1 Entwurfs-Klassendiagramm . . . 30
8.1.1 EngineControl und GameControlList . . . 30
8.1.2 ClientInterface . . . 31
8.1.3 ServerToClientConnection . . . 31
8.1.4 GameModel . . . 31
8.1.5 Pool . . . 31

9 Entwurf - Spielkonfigurator **32**
9.1 Entwurfs-Klassendiagramm . . . 32
9.1.1 View Package . . . 32
9.1.2 Control Package . . . 32
9.1.3 Model Package . . . 33

10 Entwurf - Gemeinsame Komponenten **33**
10.1 GameModel . . . 33

A Anhang **35**
A.1 Registrieren (GameEngine) . . . 35
A.2 Partie spielen (Client) . . . 37
A.3 Partie spielen (GameEngine) . . . 38
A.4 Spielzug durchführen (GameEngine) . . . 41
A.5 Turnier ausrichten (GameEngine) . . . 42

Abbildungsverzeichnis **44**

1 Analyse - Übersicht über Systemkomponenten

Im Pflichtenheft wurden der Problembereich und die Systemanforderungen bereits beschrieben. In diesem Dokument soll näher auf das Design unserer Software eingegangen werden. Dazu werden die Architekturentscheidungen beschrieben und erläutert, sowie das Design der einzelnen Hauptkomponenten im Detail aufgeführt.

1.1 Architektur

1.1.1 Modularität

In Übereinstimmung mit dem Lastenheft soll die Software aus sechs verschiedenen Hauptkomponenten bestehen – dem Spielkonfigurator, der Spiel-Engine, dem PC-Beobachter, dem Smartphone-Beobachter, dem Smartphone Teilnehmer, sowie dem KI-Teilnehmer. Um eine möglichst zeiteffiziente Umsetzung des Projektes zu gewährleisten, sollen diese Komponenten weitestgehend modular umgesetzt werden, sodass die Komponenten unabhängig voneinander entwickelt und ausgeführt werden können. Für notwendige Interaktionen zwischen den Komponenten sorgen vorher vereinbarte Interfaces, die das Zusammenfügen der Komponenten erleichtern sollen.

1.1.2 Server-Client-Architektur

Um ein netzwerkbasiertes Mulitplayer-Design zu realisieren, empfiehlt sich eine Server-Client-Architektur. Bei dieser kommunizieren mehrere Clients (Spieler, Beobachter) mit einem Server (GameEngine). Der Server speichert den Zustand des Spieles, fordert die Spieler-Clients zu Zügen auf, nimmt deren Anfragen entgegen und beachtet die Einhaltung der Spielregeln. Die Clients bekommen vom Server Benachrichtigungen über den aktuellen Spielzustand und können so das Spiel visualisieren. Aufgeforderte Spieler-Clients müssen dem Server zudem in einer vorgegebenen Zeitspanne einen gültigen Spielzug senden, um ihren Spieler nicht verlieren zu lassen.

Das folgende Diagramm visualisiert die vorgesehenen Kommunikationsschnittstellen der einzelnen Komponenten. Zur Netzwerkkommunikation wurde vom Interface-Komitee ein EMF-Modell vorgegeben, dessen Einhaltung verpflichtend ist. Auf die Spezifikation dieses Modells soll daher in diesem Dokument nicht näher eingegangen werden. In der folgenden Abbildung sind die durch das EMF-Modell ausdrückbaren Nachrichten symbolisch im Interface ITsuroEMF zusammengefasst.

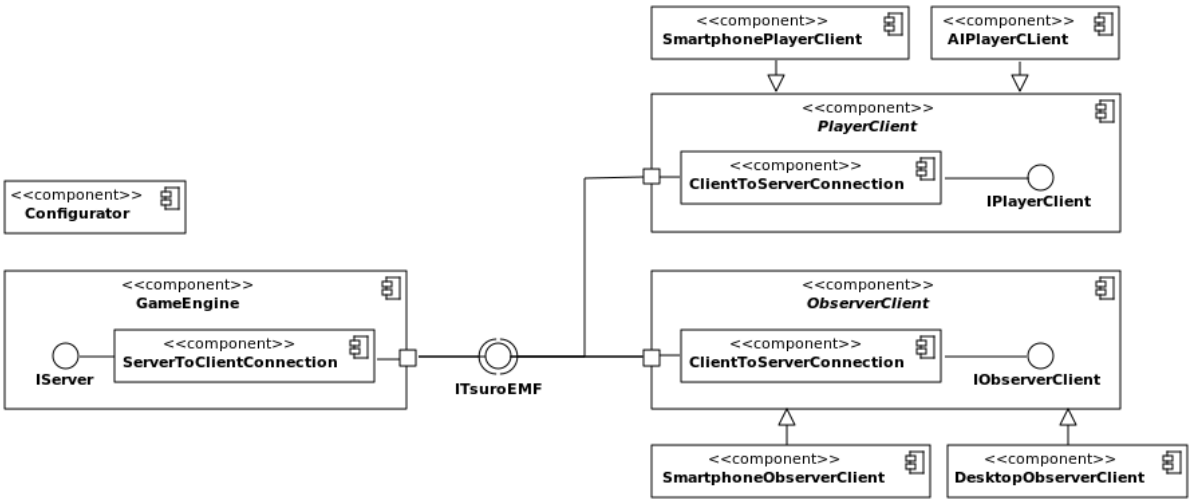

Abbildung 1: UML-Komponentendiaramm des gesamten Projektes

IServer Das Interface *IServer* findet in der GameEngine Anwendung und wird von der *ServerToClientConnection* implementiert. Es definiert alle Nachrichten, die in Form von Instanzen des EMF-Modells vom Server an die Clients gesendet werden können. Die Control-Klassen des Servers konsumieren dieses Interface, um mit den Clients zu kommunizieren.

IObserverClient Das Interface *IObserverClient* wird von den *ClientToServerConnection*-Klassen implementiert und bietet Funktionen, um Nachrichten von eine Beobachter-Client zum Server zu senden. Es wird von den Control-Klassen der Beobachter-Clients konsumiert.

IPlayerClient Das Interface *IPlayerClient* erweitert *IObserverClient* um die für Spieler-Clients relevanten Methoden und wird analog zu *IObserverClient* in den Spieler-Clients eingesetzt.

1.1.3 Entity-Control-Boundary

Für die Analyse des Tsuro-Systems bietet es sich an, anhand des Entity-Control-Boundary-Architekturmusters (ECB) vorzugehen. Diese 3-Schichten Architektur definiert folgende Klassentypen, sowie deren Funktionen:

Entity-Klassen sind für die Datenhaltung zuständig. Sie übernehmen das Speichern der Daten und stellen sie den Control Klassen zur Verfügung.

Control-Klassen sind für die Logik der Komponente zuständig.

Boundary-Klassen bieten eine Schnittstelle zwischen Nutzer (Actor) und der Logikschicht (also den Control-Klassen).

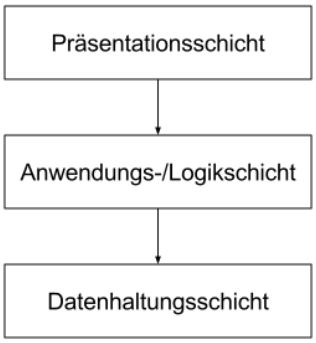

Abbildung 2: Schema der Drei-Schichten-Architektur

Dabei sieht das Architekturmuster vor, dass diese drei Typen von Klassen je nur mit den angrenzenden Schichten kommunizieren. Das bedeutet unter anderem, dass Boundary-Klassen nur über Control-Klassen auf Entity-Klassen zugreifen können, und der Nutzer nie direkt mit Entity- oder Control-Klassen interagiert. Dies hat den Vorteil, dass die drei Schichten so wenig wie möglich voneinander abhängen, was die Wartung und Änderung des Systems vereinfacht.

1.1.4 Model-View-Controller

Da die Software einige Komponenten besitzt, die ein Grafisches-User-Interface (GUI) benötigen, bedarf es einer Strukturierung der Interaktion zwischen Nutzer und System. Dafür sollen die Klassen der Software wie oben beschrieben in Entity, Control und Boundary aufgeteilt werden, wobei die Boundary-Klassen hier die Interaktionen zwischen Nutzer und System ermöglichen sollen.

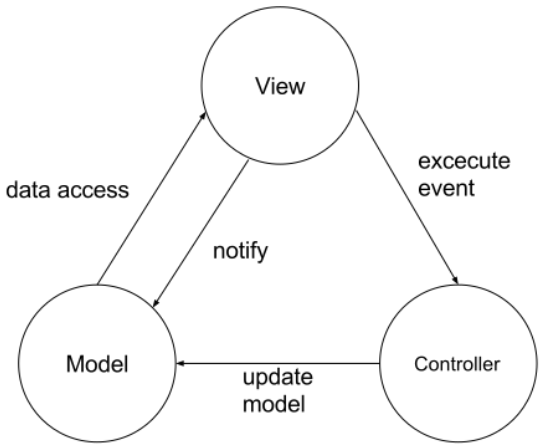

Abbildung 3: Schema der Model-View-Controller-Architektur

Das Modell-View-Controller-Architekturmuster sieht vor, dass dem Nutzer eine View zur Verfügung gestellt wird. Auf Basis der Informationen, die in dieser View darstellt werden (hier beispielsweise der Zustand des Spielbretts und die vom Teilnehmer gehaltenen Wegfelder), gibt der Nutzer Anweisungen an einen Controller (z.B. durch das Drücken einer Schaltfläche). Dieser Controller bestimmt daraufhin, welche Aktionen der Anweisung des Nutzers folgen. Dies geschieht im Model, dass den internen Zustand des Systems beschreibt. Geschehen durch den Controller veranlasst Änderungen in diesem Model, so wird die View angewiesen, sich dem neuen Zustand entsprechend zu aktualisieren.

Damit stellen View und Controller beide einen Teil der Boundary dar, da sie direkt mit dem User in Verbindung stehen. Das Model soll dementsprechend die Control und Entity-Klassen einschließen. Die Projektion der Begriffe des MVC-Musters auf die des ECB-Musters ist zwar nicht vollständig eindeutig (beispielsweise könnte der Controller des MVC auch den Control-Klassen des ECB-Musters zugewiesen werden), soll aber im Rahmen dieser Analyse wie beschrieben genutzt werden, um die Interaktion zwischen Boundary-Klassen und Usern besser erläutern zu können.

1.1.5 Strategy-Pattern

Das Strategy-Pattern ermöglicht es, Algorithmen einfach auszutauschen, ohne das dafür beträchtlicher Implementierungsaufwand geleistet werden muss. In diesem Projekt wird das Strategy-Pattern von der KI verwendet, sodass unterschiedliche und voneinander abgegrenzte

Algorithmen entwickelt werden können und später frei zwischen ihnen gewählt werden kann. Dies ist sowohl während der Entwicklung nützlich, wenn getestet wird welcher Algorithmus sicherer bzw. schneller abläuft, als auch während der Ausführung, wenn z.B. eine kurze Überlegungszeit für die KI bedeutet, dass kein langwieriger Brute-Force Algorithmus verwendet werden darf und dynamisch auf eine andere schnellere Strategie umgeschaltet werden kann.

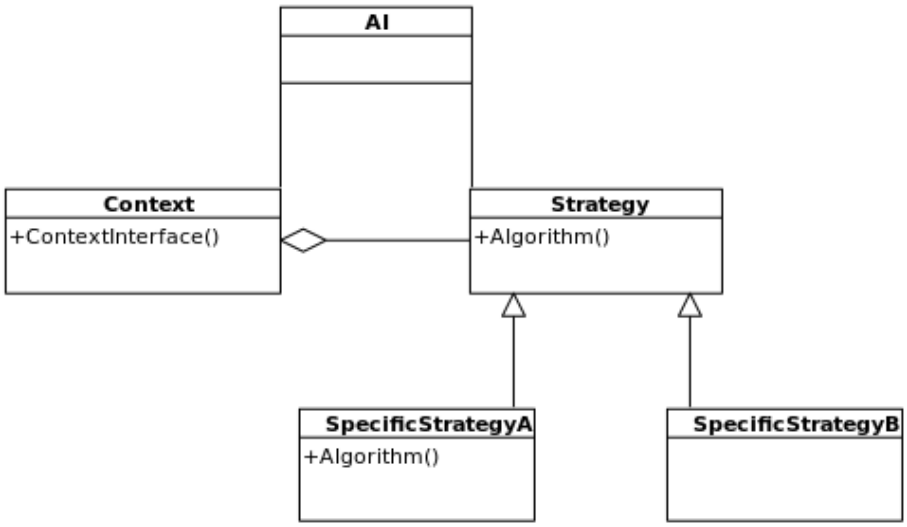

Abbildung 4: Schema des Strategy-Patterns

1.1.6 Observer-Pattern

Das Observer-Pattern dient zur Benachrichtigung verschiedene Komponenten, sobald sich ein für sie relevanter Zustand ändert. Im Tsuro-Projekt wird das Observer-Pattern genutzt, indem alle Klassen, die das Interface *IGameModelListener* implementieren, andere Komponenten über *notify()* benachrichtigen. Dies ist konkret z.B. sowohl für GUI-Controller nützlich, wenn eine Eingabe des Users eintrifft, als auch Clientseitig, wenn sich der Spieler, der am Zug ist, ändert.

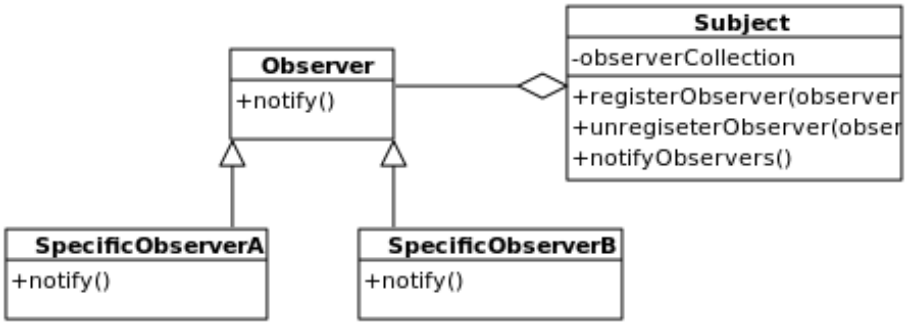

Abbildung 5: Schema des Observer-Patterns

1.1.7 Singleton-Pattern

Das Singleton-Pattern wird verwendet, wenn von einer Klasse nur genau ein Objekt existieren soll. Bei einer Singleton-Klasse wird dies sichergestellt und außerdem ein einfacher Zugriff auf die Instanz der Klasse ermöglicht. In Java wird das Singleton-Pattern wie folgt realisiert. Die Instanz der Klasse wird in einer privaten Klassenvariable gespeichert. Der Konstruktor der Klasse ist privat, sodass er nur durch die Klasse selbst aufgerufen werden kann. Die öffentliche Methode `getInstance()` ermöglicht Zugriff auf die Instanz.

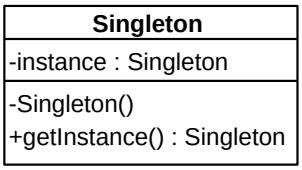

Abbildung 6: Schema des Singleton-Patterns

1.2 Hauptkomponenten

Zur Umsetzung des netzwerkbasierten Tsuro-Spiels werden gemäß der Server-Client-Architektur mehrere Hauptkomponenten benötigt. Die Kommunikation dieser Komponenten ist durch Interfaces spezifiziert, deren Grundlage das vom Interface-Komitee veröffentlichte Interface-Dokument bildet. Auf diese Weise lassen sich Komponenten leicht austauschen, warten und erweitern. Dieser Abschnitt bietet eine Übersicht über die zu entwickelnden Komponenten.

1.2.1 GameEngine

Die *GameEngine* fungiert als Server im Netzwerk und bietet den in den Abschnitten 1.2.3 und 1.2.6 genannten Clients die Grundlage für die Ausführung von Spielen. Die *GameEngine* richtet Spiele und Tuniere aus und ist für die Verwaltung der vorhandenen Spiele und Spieler zuständig.

1.2.2 Spielkonfigurator

Mit dem *Spielkonfiurator* lassen sich Konfigurationsdateien für Tsuro-Spiele erstellen, die von der *SpielEngine* eingelesen werden können. Der *Spielkonfiurator* wird als Eclipse-Plugin entwickelt. Die Konfigurations-Daten werden auf Grundlage des EMF-Modells aus dem Interface-Dokument als Java-Objekte angelegt, als JSON serialisiert und in eine XMI-Datei geschrieben.

1.2.3 PlayerClient

Unter der abstrakten Komponente *PlayerClient* sind sowohl der *SmartphonePlayerClient*, als auch der *AIPlayerClient* zusammengefasst. Beide können sich mit der *GameEngine* verbinden und auf dieser Spiele spielen.

1.2.4 SmartphonePlayerClient

Der *SmartphonePlayerClient* dient zum Spielen auf Android-Geräten.

1.2.5 AIPlayerClient

Der *AIPlayerClient* tritt an die Stelle eines Spielers und führt Spielzüge automatisch aus. Dazu verwendet er das Strategy-Pattern.

1.2.6 ObserverClient

Die Komponente *ObserverClient* ist abstrakt und beschreibt den *SmartphoneObserverClient*, sowie den *DesktopObserverClient*. Diese beiden Clients können eine Verbindung mit der *SpielEngine* herstellen und darauf laufende Spiele visualisieren.

1.2.7 SmartphoneObserverClient

Der *SmartphoneObserverClient* visualisiert Spiele auf Android-Geräten.

1.2.8 DesktopObserverClient

Der *DesktopObserverClient* wird als Eclipse-Plugin umgesetzt und visualisiert Spiele auf einem Windows-Computer.

1.3 Externe Komponenten

1.3.1 Eclipse

Eclipse ist eine Entwicklungsumgebung, welche ursprünglich für Java entwickelt wurde. Sie ist eine Open-Source Komponente und somit für jeden frei zugänglich. Nach Vorgabe des Lastenhefts wird die Version *Eclipse Mars SR2(4.5.2)* verwendet. Mit dieser Version lässt sich das für den Spiel-Konfigurator geforderte *Eclipse*-Plugin realisieren, da auf das „Plug-in Development Environment" zugegriffen werden kann.

1.3.2 Java Development Kit

Das *Java Development Kit (JDK)* wird zur Entwicklung von Programmen mit der Programmiersprache Java verwendet. Es ist über *Oracle* erhältlich. Das *JDK* beinhaltet neben der *Java Runtime Environment (JRE)* zahlreiche andere Komponenten wie bspw. den Java-Compiler, den Java-Debugger, das Java-Dokumentationswerkzeug(javadoc) und viele weiter Optionen. Wir werden das *Java SE Development Kit 8* verwenden.

1.3.3 Android 4.4 (API Level 19)

Android ist eine auf *Linux* basierende Softwareplattform und ein Betriebssystem für mobile Endgeräte. Es handelt sich um eine quelloffen entwickelte Software. In unserem Projekt werden wir die Version *Android 4.4 (API Level 19)* verwenden. Ziel ist es, für diese Version die Nutzung unseres Endprodukts auf dem Smartphone garantieren zu können.

1.3.4 Eclipse Modeling Framework

Das *Eclipse Modelling Framework* (*EMF*) ist ein Open-Source-Projekt, welches es ermöglicht, aus einem strukturierten Datenmodell (*EMF*-Modell) Java-Klassen zu generieren. Das *EMF*-Modell ist also ein Metamodell für die erzeugten Instanzen. Zur Netzwerkkommunikation ist uns die Verwendung von Java-Klassen basierend auf einem vorgegebenen *EMF*-Modell vorgeschrieben. Weiterhin soll der Spielkonfigurator auf diesem Modell basierende Objekte serialisieren und der SpielEngine als Datei zur Verfügung stellen.

1.3.5 Plugin Development Environment

Die *Plug-in Development Environment (PDE)* bietet Tools zum Erstellen, Entwickeln und Testen von Eclipse-Plug-ins. Dieses Plug-in beinhaltet einige weitere Fähigkeiten und wird außerdem in drei Komponenten aufgeteilt. Diese Komponenten wären zum einen die *UI*, die *API Tools* und *Build*. Die UI beinhaltet einen Satz von Modellen, Werkzeugen und Editoren zur Entwicklung von Eclipse-Plug-ins. Die API Tools vereinfachen die API-Dokumentation und Wartung. *Build* sind Tools, die dabei helfen Build-Prozesse zu automatisieren. Dementsprechend lässt sich das geforderte Eclipse-Plug-in für den Spiel-Konfigurator mit der PDE gut umsetzen.

1.3.6 JavaFx

JavaFx ist ein Framework für Graphische Benutzeroberflächen. Es beinhaltet eine Bibliothek die das Implementieren der GUI erleichtert. Das Framework ist modular und objektorientiert aufgebaut, dadurch lassen sich auch komplexe Anwendungen gut realisieren. Bei *JavaFx*-Komponenten gibt es keine Kompatibilitätsprobleme im Bezug auf unterschiedliche Plattformen. Die Struktur der Benutzeroberfläche ist unabhängig vom Programmcode. Da die Expertise bezüglich JavaFx am größten ist, haben wir uns für dieses Framework entschieden.

1.3.7 Android Development Tools / Android Studio

Android Studio ist eine frei Integrierte Entwicklungsumgebung, welche von *Google Inc.* zur Verfügung gestellt wird. Des Weiteren ist es auch eine Entwicklungsumgebung für Android, so dass die Smartphone-Komponenten einfacher realisieren lassen. Die *Android Development Tools (ADT)* sind Plug-ins für Eclipse. Somit ist die Implementierung der Smartphone-

Komponenten für das Betriebssystem Android möglich. Wir werden zur Realisierung der Smartphone-Komponente hauptsächlich Android Studio nutzen.

1.3.8 emfjson Jackson

JSON (JavaScript Object Notation) ist ein kompaktes Format zur Übertragung von Daten in Textform. Es wird bevorzugt in Webanwendungen verwendet, dient aber auch als Datenformat bei der Serialisierung. Durch einen einfachen hirarchischen Aufbau entsteht nur ein geringer Strukturierungs-Overhead bei der Datengröße, weshalb JSON für Netzwerkanwendungen z.B. XML zu bevorzugen ist. Auch in diesem Projekt soll JSON für die Netzwerkkommunikation eingesetzt werden.

Um die Instanzen des Interface-EMF-Modells in JSON zu serialisieren wird das quelloffene Projekt *emfjson Jackson* verwendet. Es ist Teil einer Gruppe von Projekten namens *emfjson* (http://emfjson.org/), deren Ziel es ist, JSON für das Eclipse Modeling Framework zu unterstützen. Damit ist der technische Grundstein für die Server-Client-Kommunikation, sowie für die Spielkonfigurations-Dateien gelegt.

2 Analyse - Clients

2.1 Sequenzdiagramme

Um die Konstruktion des Analyse-Klassendiagramms nachvollziehbar zu machen, sind im Folgenden Sequenz-Diagramme zu wichtigen Use-Cases abgebildet. Weitere Sequenzdiagramme finden sich im Anhang.

2.1.1 Registrieren

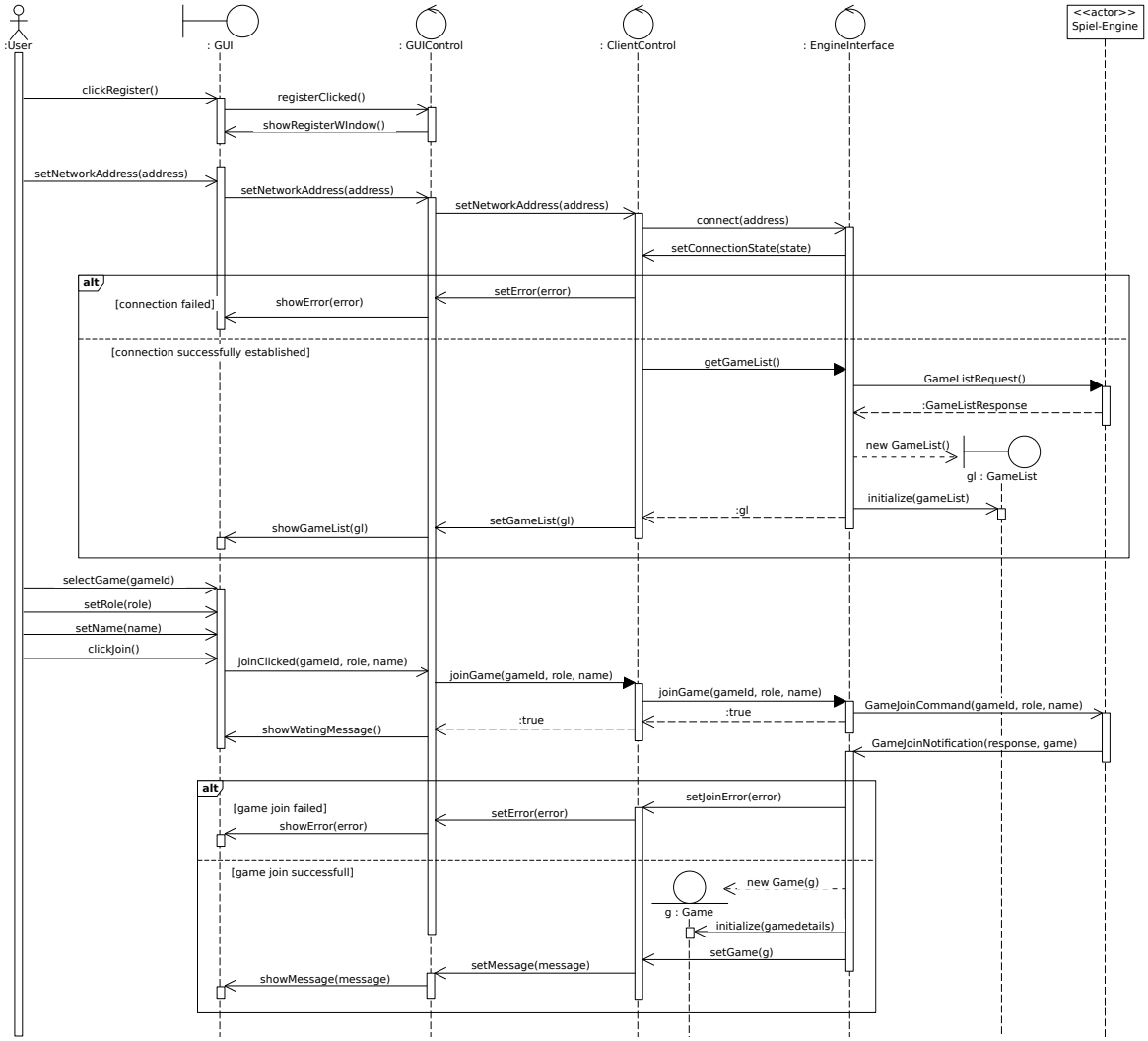

Abbildung 7: Sequenzdiagramm - Registrieren - Client

2.1.2 Wegfelder organisieren

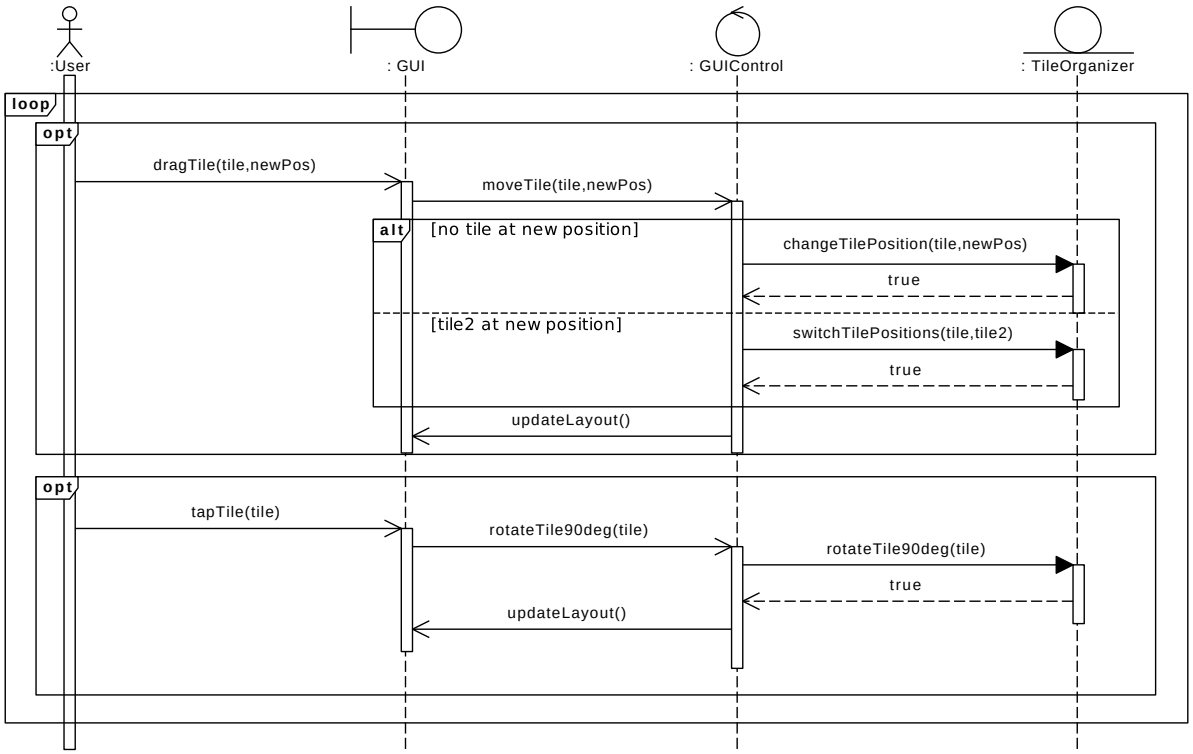

Abbildung 8: Sequenzdiagramm - Wegfelder organisieren

Das Sequenzdiagramm beschreibt die Abläufe während der ”Wegfelder organisieren” Phase, in der der Benutzer die Wegfelder in seinem Deck nach Belieben umordnen und drehen kann. Dabei gibt der User über die GUI den Befehl für das Bewegen/Rotieren des Wegfeldes. Die GUIControl gibt dies weiter an den TileOrganizer. Anschließend wird die GUI bezüglich der Änderungen aktualisiert.

2.1.3 Spielzug durchführen

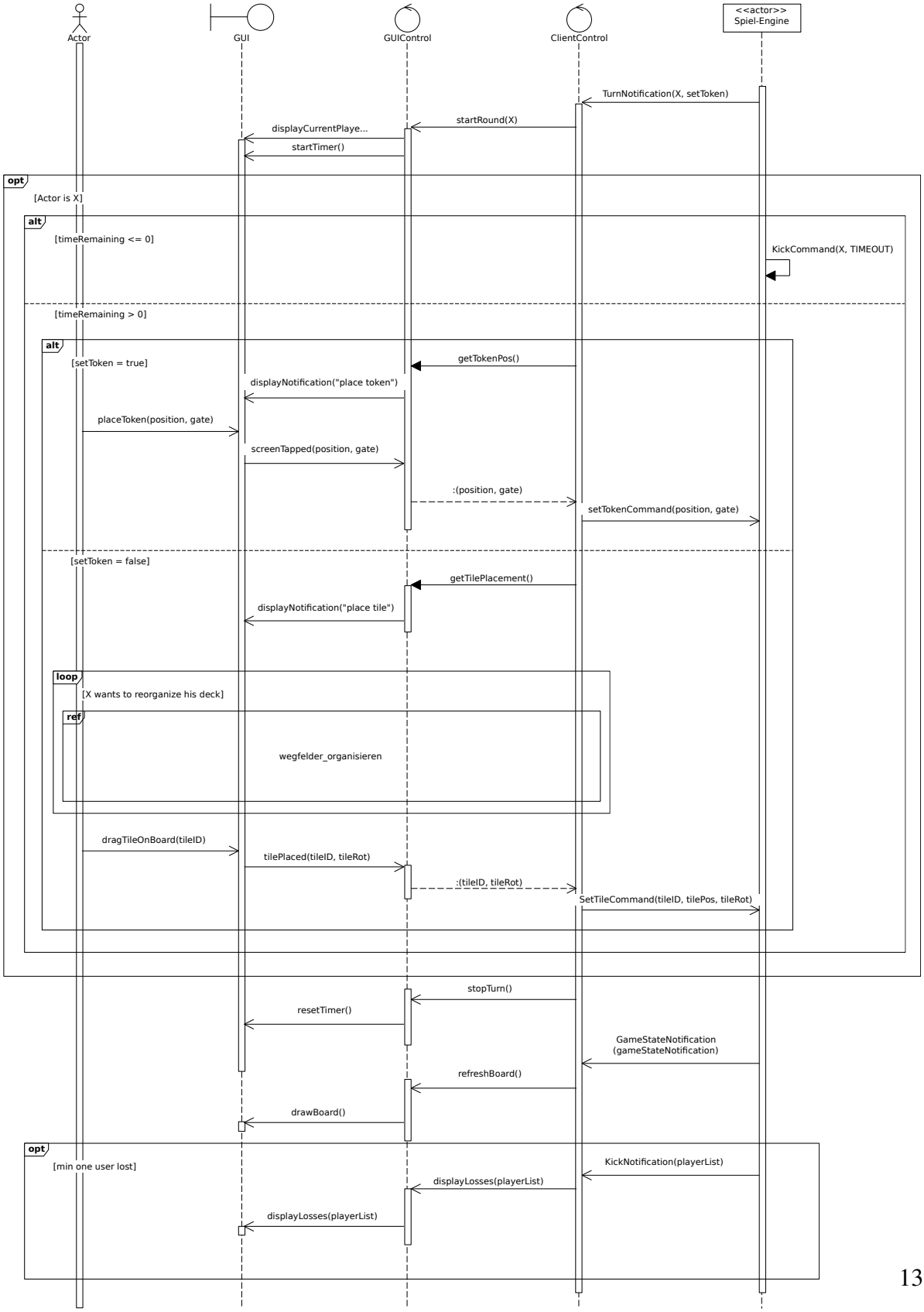

Abbildung 9: Sequenzdiagramm - Registrieren - Client

Das Analyse-Sequenzdiagramm ”Registrieren - Client” gibt Aussage darüber, wie der Client sich mit einem Server verbindet und bei einem speziellen Spiel oder dem allgemeinen Spiele-Pool registriert. Zuerst öffnet sich ein weiteres Fenster, welches eine IP-Adresse erwartet, die dem Server zugehörig ist, auf welchem die Spiel-Engine läuft. Nach dem Verbinden sendet der Server dann eine Liste mit den aktuell geöffneten Spielen, aus denen sich der Nutzer ein Spezielles auswählen kann. Außerdem muss noch vom Nutzer angegegeben werden, ob er Beobachter oder Spieler sein möchte und wie sein Name lautet. Es gibt die Möglichkeit kein Spiel auszuwählen. Daraufhin wird man dann dem allgemeinen Spiele-Pool hinzugefügt, so dass man zufällig einem Spiel hinzugefügt werden kann. Sobald man seine Auswahl bestätigt hat (und man evtl. zufällig einem Spiel hinzugefügt wurde), sendet der Server das entsprechende Game als EMF-Model, welches dann durch das EngineInterface zu einem Objekt der Klasse Game übersetzt wird.

2.2 Klassendiagramm

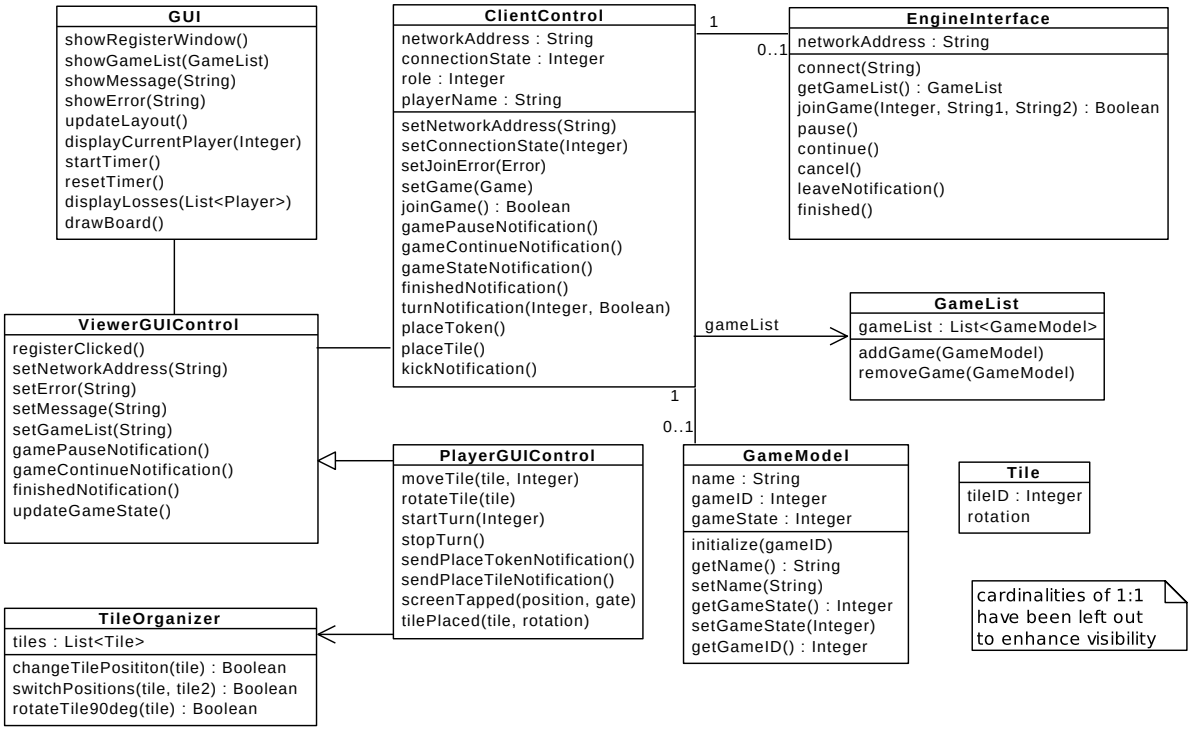

Abbildung 10: Analyse-Klassendiagramm der Clients

2.2.1 GUI

Die Klasse GUI ist für die Darstellung für den Benutzer zuständig und ermöglicht dem Benutzer die Interaktion mit der Software, indem Eingaben durch GUI registriert und weitergeleitet werden. Bei Änderungen durch die darunter befindlichen Kontrollklasse wird auch GUI aktualisiert, um besagte Änderungen anzuzeigen. Dazu kennt die Kontrollklasse ViewerGUIControl GUI und andersherum.

2.2.2 ViewerGUIControl

ViewerGUIControl ist die zugehörige Kontrollklasse zu GUI. Sie hält Reaktionen auf beobachtende Interaktionen eines Benutzers - sowohl Beobachter als auch Spieler - bereit. ViewerGUIControl kann auch Benachrichtigungen entgegen nehmen, wie z.B. Fehlermeldungen, damit diese dem Benutzer angezeigt werden können.

2.2.3 PlayerGUIControl

Die Klasse PlayerGUIControl erbt von ViewerGUIControl, enthält aber zusätzlich zu den Funktionen des Beobachters Funktionalitäten, die für einen Spieler nötig sind, um am Spiel teilzunehmen. Von der PlayerGUIControl interpretierte Aktionen werden an die ClientControl-Klasse weitergeleitet.

2.2.4 GameList

Die Klasse GameList speichert die auf dem Server verfügbaren Spiele, nachdem sich mit einem Server verbunden wurde.

2.2.5 ClientControl

Die Klasse ClientControl enthält die Funktionen für den Verbindungsaufbau zur Spiel-Engine (ClientToServerConnection), sowie die Identifikations- und Verbindungsdaten des Clients wie z.B. die Netzwerkadresse. Außerdem kennt sie die GameList und die GameModel Klasse.

2.2.6 GameModel

Die Klasse GameModel ist für die Initialisierung eines neuen Spiels verantwortlich.

2.3 TileOrganizer

Die Klasse TileOrganizer ist für die Organisation der einzelnen Wegfelder (“Tiles”) auf der Hand des Spielers zuständig und wird über die Klasse PlayerGUIControl angesteuert.

3 Analyse - GameEngine

3.1 Sequenzdiagramme

Um die Konstruktion des Analyse-Klassendiagramms nachvollziehbar zu machen, sind im Folgenden Sequenz-Diagramme zu wichtigen Use-Cases abgebildet. In den folgenden Diagrammen wird auf die interne Funktionaltität des Clients nicht weiter eingegangen, daher ist dieser als *actor* dargestellt. Weitere Sequenzdiagramme finden sich im Anhang.

3.1.1 Partie ausrichten

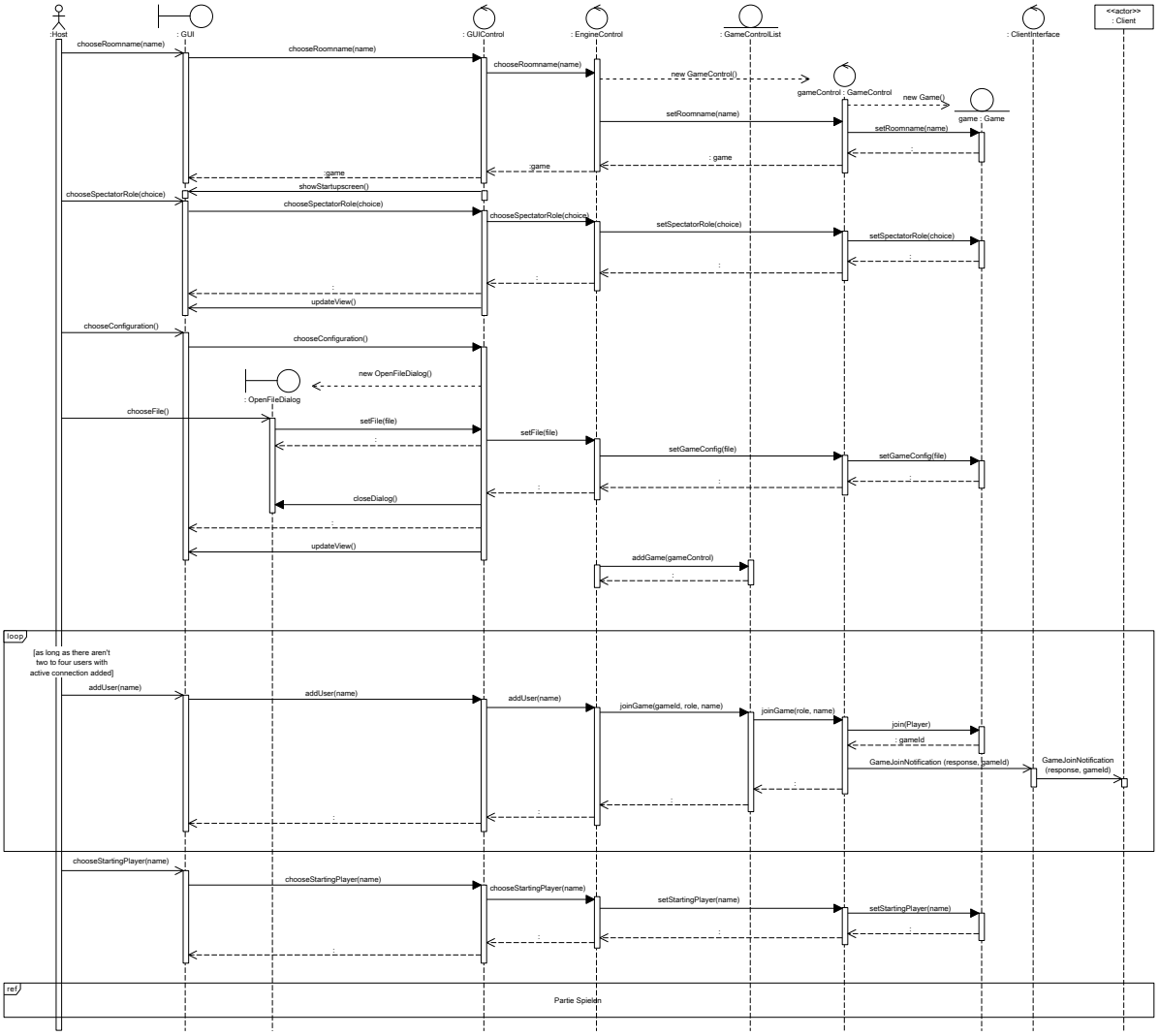

Abbildung 11: Sequenzdiagramm - Partie ausrichten - GameEngine

Das Sequenzdiagramm aus Abbildung 11 stellt aus der Sicht der Engine dar, wie sich das Ausrichten einer Partie gestaltet.
Der Host bekommt bei dem Ausrichten einer Partie die Möglichkeit diverse Parameter der Partie zu gestalten.
Zunächst gibt der Host einen Raumnamen an, woraufhin intern ein *Game*-Objekt und das zugehörige *GameControl*-Objekt erzeugt werden. Die *GameControl* verwaltet die Daten des *Games* und hält den Status aktuell. Folgend kann der Host entscheiden, ob er dem Spiel selber zusehen möchte, oder nicht. Die Entscheidung wird in dem *Game*-Objekt abgespeichert. Zum Nutzen einer vorher angelegten Spielkonfiguration wird ein Dialog geöffnet, welcher dem Host die Möglichkeit gibt eine Datei zu öffnen. Die übergebene Datei wird dekodiert und im *Game*-Objekt abgespeichert.
Nachfolgend kann der Host zwei bis vier Spieler zu dem Spiel hinzufügen und einen startenden Spieler festlegen. Nach dem Hinzufügen wird der entspechende Client benachrichtigt.

3.2 Klassendiagramm

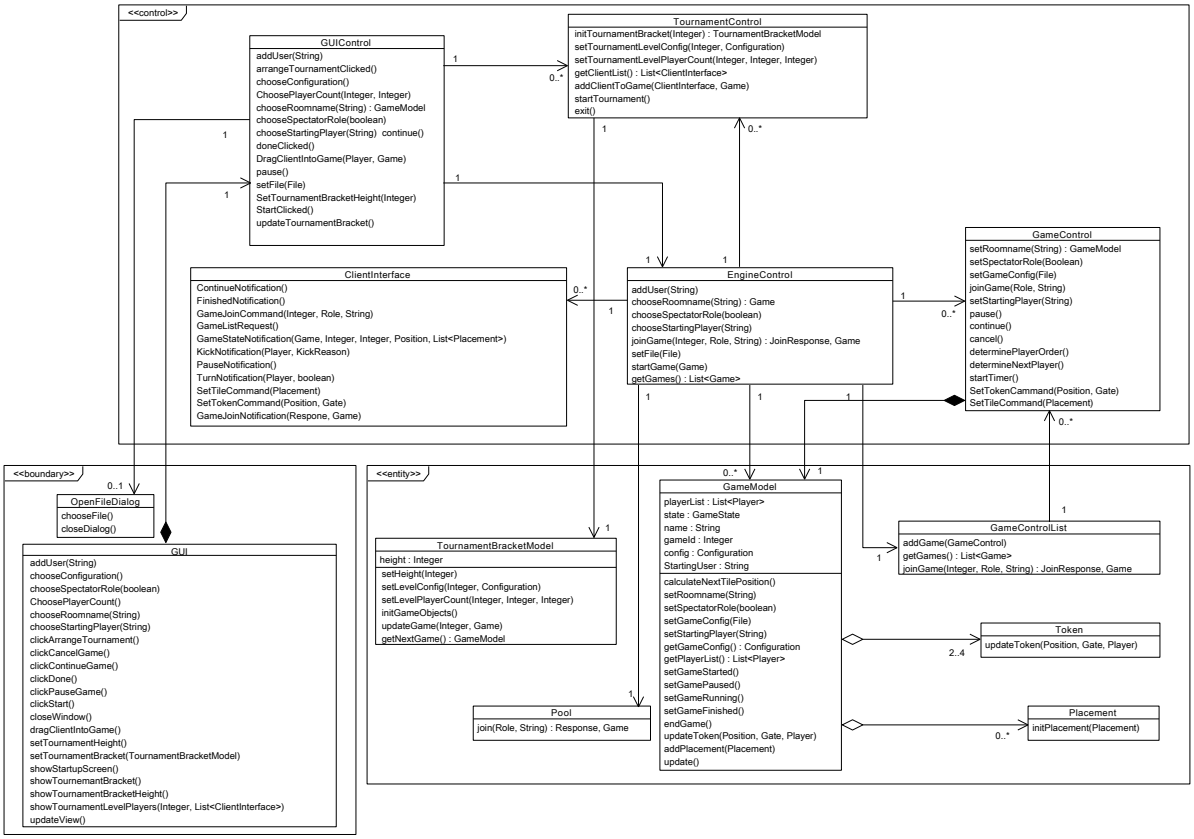

Abbildung 12: Analyse-Klassendiagramm der GameEngine

Im Folgenden werden die Klassen der Game-Engine, wie im Klassendiagramm in Abbildung 12 abgebildet, beschrieben.

3.2.1 EngineControl

Die *EngineControl*-Klasse kann es pro Server nur einmal geben. Sie ermöglicht es Clients sich zu registrieren und dem Host einzelne Partien oder ganze Turniere zu organisieren und dem Server hinzuzufügen.

3.2.2 GUI

Die Klasse *GUI* visualisiert Informationen für den Host und ermöglicht diesem die Interaktion mit der Software, indem Eingaben durch *GUI* registriert und weitergeleitet werden. Bei Änderungen der darunter befindlichen Kontrollklasse (*GUIControl*) wird die *GUI* aktualisiert, um diese Änderungen anzuzeigen. Um dies zu ermöglichen kennen sich die Boundary-Klasse *GUI* und die Control-Klasse *GUIControl*.

3.2.3 OpenFileDialog

Die Klasse *OpenFileDialog* ermöglicht es dem Ausrichter, Dateien auszuwählen und diese in die Spiel-Engine zu laden. Genutzt wird der *OpenFileDialog* hier zum Laden von Spielkonfigurationsdateien.

3.2.4 GUIControl

Die Klasse *GUIControl* dient zur Verarbeitung der von der *GUI* mitgeteilten Benutzereingaben und leitet diese an die angesprochenen Control-Klassen weiter. Hierzu kennt *GUIControl* sowohl *EngineControl*, als auch *TournamentControl* und *GameControl*.

3.2.5 TournamentControl

Die Klasse *TournamentControl* dient zum Organisieren eines Turnies. Sie erstellt den Turnierbaum, startet nacheinander die Spiele und speichert die Ergebisse im *TournamentBracketModel*.

3.2.6 GameControl

Die Klasse *GameControl* verwaltet alle Ereignisse die das entsprechende *Game*-Objekt betreffen.

3.2.7 ClientInterface

Diese Klasse repräsentiert die Verbindung zu den Clients. Für jeden registrierten User gibt es ein *ClientInterface*. Die Funktionen der Klasse dienen entweder dem Senden von Daten an den jeweiligen Client oder dem Empfangen von Daten vom jeweiligen Client.

3.2.8 Game

Die Klasse *Game* speichert alle Daten die zu einer Partie gehören. Darunter befinden sich der Zustand der Partie und des Spielfelds, repräsentiert durch die Klassen *Token* und *Placement*, sowie alle teilnehmenden Spieler.

3.2.9 TournamentBracketModel

Das *TournamentBracketModel* repräsentiert einen Turnierbaum und enthält Informationen wie die Höhe und alle zum Turnier gehörenden *Game*s.

3.2.10 Pool

Jeder Server hat genau einen *Pool*. In diesem werden Spieler verwaltet, die sich zwar beim Server registrierern, aber keinem *Game* beitreten.

3.2.11 GameControlList

Die *GameControlList* beinhaltet die *GameControl*-Objekte zu allen Partien, die auf dem Server laufen.

3.2.12 Token

Ein *Token* repräsentiert die Spielfigur eines Spielers und speichert die Position auf dem Spielfeld und seinen Besitzer.

3.2.13 Placement

Die Klasse *Placement* beschreibt die Platzierung eines Wegfeldes auf dem Spielfeld.

4 Analyse - Spielkonfigurator

4.1 Sequenzdiagramme

Um die Konstruktion des Analyse-Klassendiagramms nachvollziehbar zu machen, ist nachfolgend das Analyse-Sequenzdiagramm zum erstellen einer Konfiguration abgebildet.

4.1.1 Spielkonfiguration erstellen

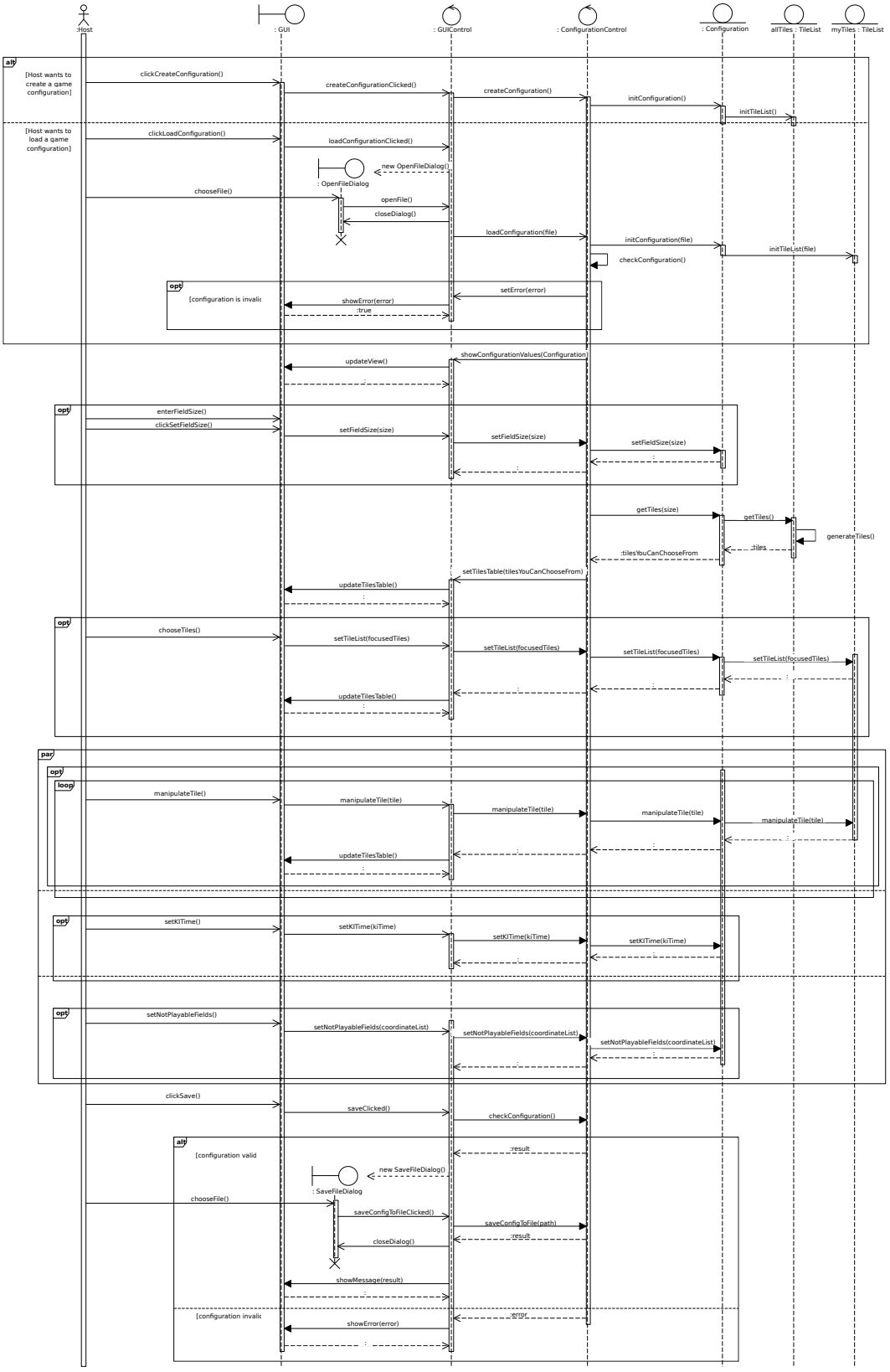

Abbildung 13: Sequenzdiagramm - Spielkonfiguration erstellen

Das Sequenzdiagramm aus Abbildung 13 zeigt, wie sich die Einstellungen der Spielkonfiguration gestaltet. Als Actor haben wir hier den *Host*. Dieser kann entweder eine neue Spielkonfiguration erstellen oder er wählt eine alte Spielkonfiguration aus. Unsere Boundary-Klasse *GUI* gibt die Auswahl des *Hosts* weiter an die *GUIControl*, so dass überprüft werden kann ob die Eingabe des *Hosts* zulässig ist.
Optional kann auch ein Error auftreten beim Laden einer älteren Konfiguration. Dies wird durch die *ConfigurationControl* überprüft.
Bei einem Durchlauf ohne Error wird eine *TileList* erzeugt. Diese kann entweder eine schon bekannte aus einer älteren Konfiguration sein, der ein *File* übergeben wird oder es wird eine neue *TileList* erzeugt.
Weiterhin kann man optional die Größe des Spielfeldes ändern als Host. Diese wird wieder überprüft durch die *GUIControl* und *ConfigurationControl*. Wenn die Größe des Spielfeldes zulässig ist, soll die Klasse *Configuration* die *Tiles* erzeugene.
Auf der *GUI* werden dann die aktuellen *Tiles* angezeigt aus denen der *Host* wählen kann. Nach dem der *Host* alle gewünschten *Tiles* ausgewählt hat, werden diese in der TileList *MyTiles* gespeichert.
Des Weiteren kann der *Host* noch die *Tiles* bearbeiten bezüglich der Pfade. Auch die bearbeiteten *Tiles* werden wieder in *MyTiles* gespeichert. Außerdem kann noch eine Spielzeit für die KI gesetzt werden und unbespielbare Felder auf dem Spielfeld. Beides passiert wieder durch den *Host* und wird dieses Mal jedoch in der *Configuration* gespeichert. Am Ende des Prozesses speichert der *Host* die Spielkonfiguration, welche wiederum auf der Klasse *ConfigurationControl* überprüft wird.
Wenn die Konfiguration zulässig ist, dann kann der *Host* sie als *File* abspeichern und bekommt zum Schluss eine Nachricht auf der *GUI*, die ihm anzeigt, dass das Speichern erfolgreich war. Wenn die Konfiguration unzulässig war, dann wird dem *Host* auf der *GUI* ein Error angezeigt.

4.2 Klassendiagramm

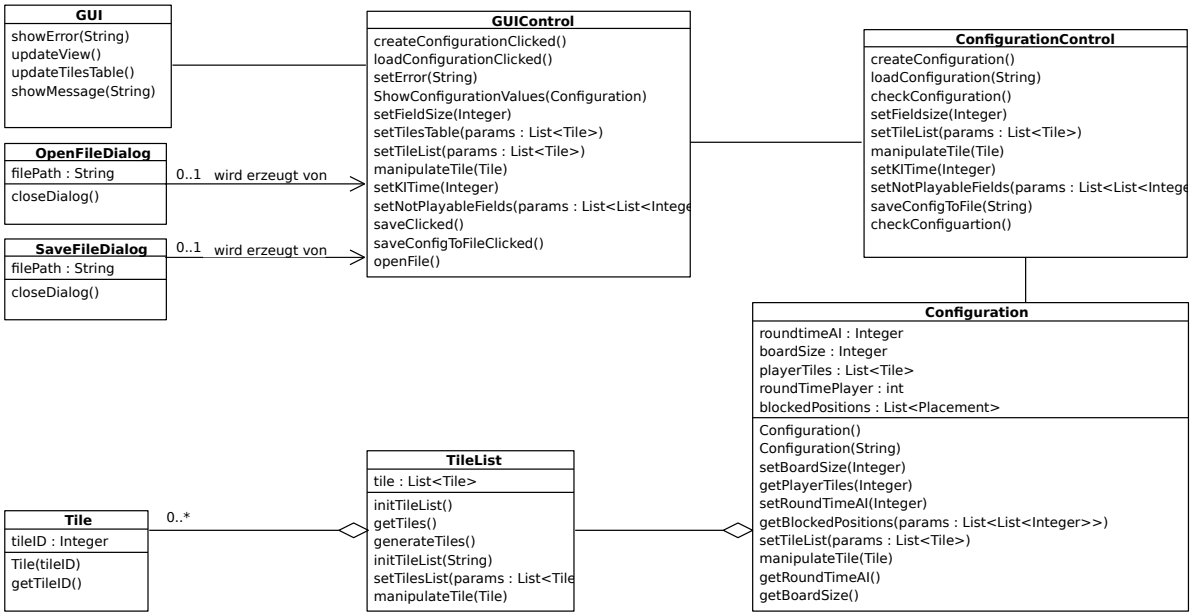

Abbildung 14: Analyse-Klassendiagramm des Spielkonfigurators

4.2.1 GUI

Die Klasse *GUI* visualisiert alle Informationen für den Host und bildet eine Schnittstelle zur Software. Sie ist außerdem eine Boundary-Klasse Über diese Schnittstelle können Eingaben des Hosts registriert werden und an die Control- und Entity-Klassen weitergegeben werden. Wenn Änderungen durch die darunter liegende Kontrollklasse, in unserem Fall *GUIControl* stattfinden, wird die *GUI* aktualisiert, um dem Host die Änderungen anzuzeigen. Durch diese Kommunikation kennen sich die Klassen *GUI* und *GUIControl*.

4.2.2 GUIControl

Die Klasse *GUIControl* verarbeitet die Daten der *GUI*. Über die*GUI* werden die Benutzereingaben an die *GUIControl* weitergeleitet. Diese gibt die Daten an die *ConfigurationControl* weiter. Dazu müssen sich die Control-Klassen *GUIControl* und *ConfigurationControl* kennen.

4.2.3 ConfigurationControl

Die Control-Klasse *ConfigurationControl* bekommt die geprüften Benutzereingaben von der *GUIControl* weitergegeben. Die *ConfigurationControl* prüft ob die Benutzereingabe auch gültig für die Klasse *Configuration* sind. Wenn gültige Daten übergeben wurden, werden diese

weitergeleitet an die *Configuration*. Somit kennen sich auch die Klassen *ConfigurationControl* und *Configuration*.

4.2.4 Configuration

Die Entity-Klasse *Configuration* speichert die Eingaben des Hosts, wie zum Beispiel die Zeiteinstellung für den KI-Teilnehmer und die Feldgröße. Sie besteht zusätzlich aus einer *TileList*, kennt also auch die Entity-Klasse *TileList*.

4.2.5 OpenFileDialog

Die Boundary-Klasse *OpenFileDialog* ermöglicht es dem Ausrichter, Dateien zu wählen und diese in die Spielkonfiguration zu laden.

4.2.6 SaveFileDialog

Die Boundary-Klasse *SaveFileDialog* ermöglicht es dem Ausrichter, eine Spielkonfiguration als Datei zu speichern und den Speicherort zu bestimmen.

4.2.7 TileList

Die Entity-Klasse *TileList* besteht aus der Entity-Klasse *Tile*. Eine TileList kann keine bis beliebig viele *Tiles* enthalten. Eine *TileList* ist Bestandteil einer Konfiguration. Also kennt die Klasse *TileList* auch die anderen beiden Entity-Klassen *Tile* und *Configuration*.

4.2.8 Tile

Die Entity-Klasse *Tile* ist Bestandteil der Klasse *TileList*. Ein Tile hat eine TileID, so dass man diese voneinander unterscheiden kann.

5 Entwurf - Clients

Nachdem in der Analyse die Aufgaben der einzelnen Klassen beschrieben wurden, wird im Entwurf erklärt, wie wir diese Funktionen umsetzen wollen. Im Folgenden wird also aufgelistet, wie die Software- und Hardwarebedingungen aussehen, die Datenstrukturen werden sowohl an die Plattform als auch an die Programmiersprache angepasst, die Architektur wird verfeinert und die Use Cases werden zu vollständigen Verhaltensbeschreibungen für einzelne Klassen erweitert.

5.1 Entwurfs-Klassendiagramm

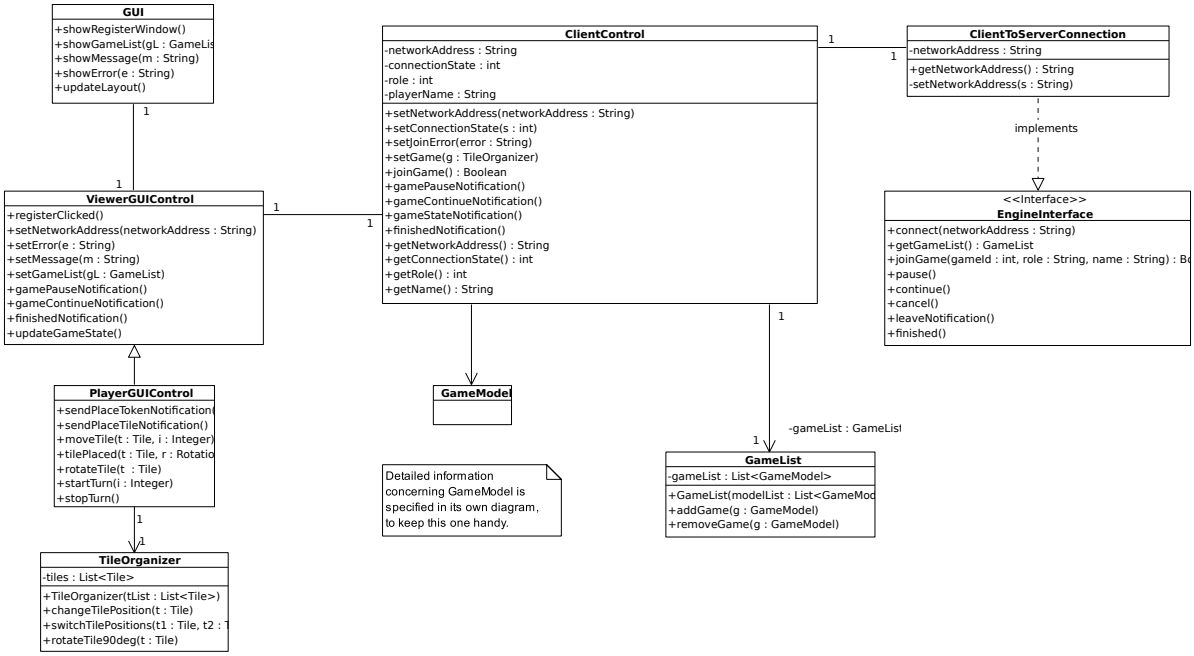

Abbildung 15: Entwurfs-Klassendiagramm Clients

Der Client enthält alle Klassen die erforderlich sind, um Tsuro auf einem Endgerät zu spielen oder andere Spieler bei einem Spiel zu beobachten. Da es sowohl Beobachter als auch Spieler gibt, beide aber zum Teil gleiche Funktionen (die graphische Darstellung des Spiels) benötigen, haben wir uns für eine Hierarchie entschieden, die über Vererbung realisiert wird. Desweiteren wird die Kommunikation zum Server über eine seperate Klasse (ClientToServerConnection) verwaltet, genauso wie die Kontrolle der Anzeige(ViewerGUIControl und PlayerGUIControl). Um die Kommunikation zwischen Benutzer (Anzeigeebene) und Server zu gewährleisten, gibt es die Klasse ClientControl, welche als Bindeglied zwischen beidem dient.

5.1.1 GUI

Die Klasse GUI des Clients ist für die Anzeige des Tsuro-Spiels zuständig. Über Die GUI Klasse kann der Benutzer mit der Software interagieren. Dabei dient die Klasse sowohl zur Anzeige für Beobachter als auch für Spieler gleichermaßen. Um das aktuelle Geschehen anzuzeigen, kommuniziert sie mit der Klasse ViewerGUIControl, über die Updates zum Empfangen des Spielstatus und Interaktionen des Benutzers weitergeleitet werden können.

5.1.2 ViewerGUIControl

Die Klasse ViewerGUIControl dient zum Interpretieren beobachtender Aktionen des Benutzers und hält für solche Interpretationen entsprechende Aktionen bereit. Sie kommuniziert sowohl mit der GUI, um vom Server gesendete Benachrichtigungen weiterzugeben und dafür zu sorgen, dass die GUI bei Änderungen in einem Spiel aktualisiert wird, als auch mit der ClientControl Klasse, um Aktionen, die vom Benutzer ausgelöst wurden, weiter zu senden.

5.1.3 PlayerGUIControl

Die Klasse PlayerGUIControl erbt von der Klasse ViewerGUIControl, womit es möglich ist, Spiele zu beobachten und beobachtende Aktionen auszuführen sowie mit der ClientControl Klasse und der GUI zu kommunizieren.Durch diese Vererbung ist eine saubere Trennung der Rollen verschiedener Clients und ein Wechsel der Rolle beim Verbinden mit einem Server gewährleistet. Zusätzlich enthält sie alle Funktionen die zum Spielen erforderlich sind, wie z.B. das aktive Bewegen von Wegfeldern. Für die Organisation der Wegfelder auf der Hand des Spielers kann die PlayerGUIControl auf die TileOrganizer Klasse zugreifen.

5.1.4 TileOrganizer

Die Klasse TileOrganizer ist ausschließlich für die Organisation der Wegfelder auf der Hand des Benutzers zuständig. Dazu werden die Wegfelder des Benutzers von ihr verwaltet und Aufrufe zum Organisieren können von der Klasse PlayerGUIControl empfangen werden.

5.1.5 ClientControl

Die Klasse ClientControl ist die Schnittstelle zwischen ViewerGUIControl und der Spiel-Engine. Über diese Klasse wird die Verbindung zur Spiel-Engine hergestellt sowie Informationen an diese weitergeleitet und von ihr empfangen. Weiterhin speichert sie nach der Registrierung wichtige Verbindungsinformationen.

5.1.6 GameList

Die Klasse GameList enthält speichert nach der Registrierung an einem Server die auf diesem verfügbaren Spiele, sodass die ClientControl Klasse sehen kann, welche Spiele verfügbar sind.

5.1.7 ClientToServerConnection

Die Klasse ClientToServerConnection implementiert das Engineinterface, wodurch über diese Klasse direkt mit einem Server kommuniziert werden kann. Sie kann sich mit einem Server verbinden sowie von diesem empfangene Informationen an die ClientControl Klasse weiterleiten. Außerdem ist ClientToServerConnection für das Beitreten zu Spielen verantwortlich.

5.1.8 GameModel

Siehe Beschribung in Abschnitt 10.1.

6 Entwurf - KIClient

Die Komponente KI-Teilnehmer soll wie ein menschlicher Teilnehmer an einer Partie teilnehmen können, aber dabei selbstständig Spielzüge durchführen. Als weitere Anforderung soll die künstliche Intelligenz so spielen, dass sie mögliche Spielzüge berechnet, evaluiert und anschließend einen möglichst optimalen Spielzug auswählt. Ein optimaler Spielzug soll dabei nicht nur die aus ihm resultierenden Siegchancen, sondern auch die für die Evaluation benötigte Zeit und die der KI zur Verfügung stehenden Informationen berücksichtigen. Die Vorgehensweise der KI zur Findung eines solchen optimalen Spielzugs soll im Folgenden erläutert werden.

6.1 Generelles Vorgehen

Da die Regeln des Spiels Tsuro den Teilnehmern verhältnismäßig überschaubare Möglichkeiten zur Interaktion bietet, nämlich die Platzierung eines gehaltenen Tiles an einer vorher bestimmten Position (der Position des Spieler-Token), sollte die Menge der möglichen Ausgänge eines gegebenen Spiels zumindest bei kleinen Spielfeldgrößen durchaus berechenbar sein. Dies legt natürlich die Verwendung eines BruteForce-Algorithmus zur Bestimmung der optimalen Züge nahe. Allerdings sollte auch die Möglichkeit eines eng gesteckten Zeitlimits und die eines großen Spielfeldes berücksichtigt werden und damit auch die Laufzeit einer solchen Berechnung. Daher ist der KI-Teilnehmer so konzipiert, dass er über verschiedene Strategien verfügt, aus denen er anhand der gesetzten Limitationen eine optimale Strategie bestimmen soll. Es sollte zudem in Betracht gezogen werden, den ”Grad des Vorrausdenkens” schrittweise aufzubauen, sodass beispielsweise auf Teilergebnisse zurückgegriffen werden kann.

6.2 Entwurfs-Klassendiagramm

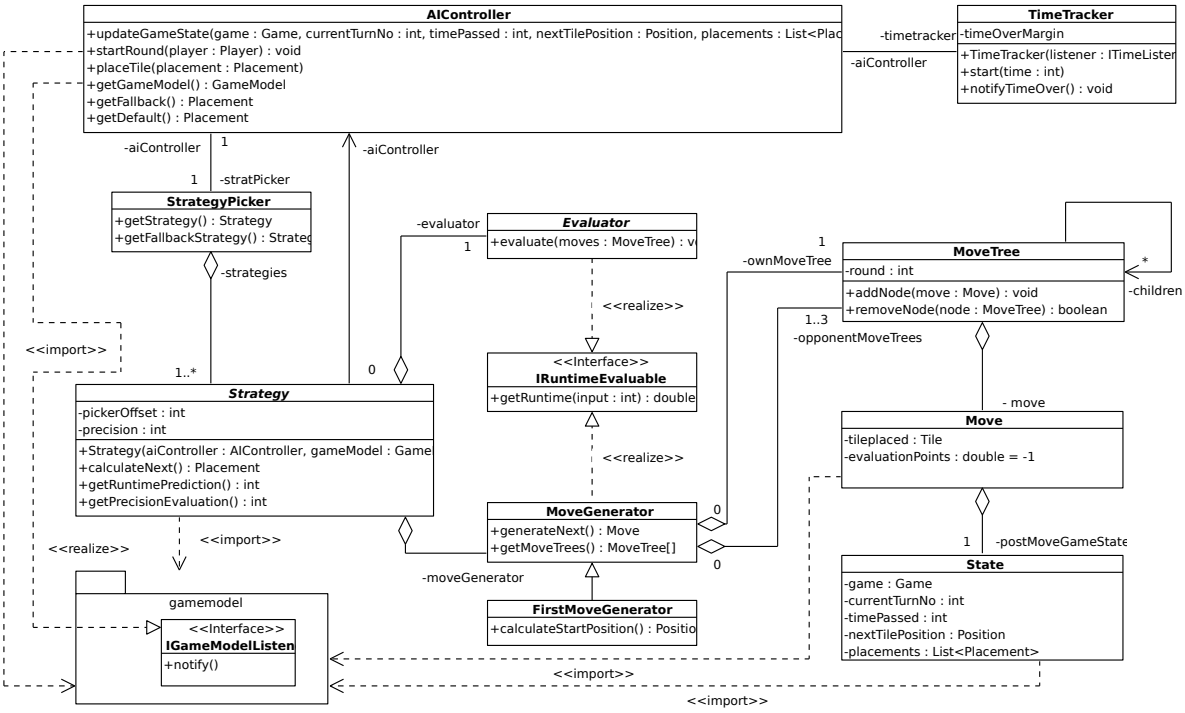

Abbildung 16: Entwurfs-Klassendiagramm des KI-Teilnehmers

6.3 Elemente der KI

6.3.1 AIController

Die zentrale Klasse der KI. Sie verbindet die KI mit der Clientsteuerung und hat damit auch Zugriff auf das GameModel.

6.3.2 TimeTracker

Eine Klasse, die parallel zur Berechnung der Strategy läuft. Dauert die Berechnung des nächsten Zuges zu lange, soll der AIController informiert und ein zuvor berechnet Fallback-Placement genutzt werden. Das Fallback ist dabei der nächstmögliche regelkonforme Zug, der, sofern möglich, nicht zu einer Niederlage führt.

6.3.3 Strategy und StrategyPicker

Der *StrategyPicker* ist eine Klasse, die entscheidet, welche der implementierten Strategien ausgewählt wird. Um eine geeignete Strategie zu finden, evaluiert der *StrategyPicker*

- Die Genauigkeit einer Strategie anhand eines manuell festgelegten *precision* Wertes. Die Genauigkeit einer Strategie errechnet sich aus der Größe des vom *MoveGenerator* generierten *MoveTree*. Dieser gibt mögliche Abfolgen von Zügen an.

- Einer Laufzeitschätzung, die sich aus den geschätzten Laufzeiten des *MoveGenerators* und des *Evaluators*. Beide Klassen sollen dazu ein Interface *IRunTimeEvaluable* implementieren

- Einer Laufzeitschätzung, die sich aus den geschätzten Laufzeiten des *MoveGenerators* und des *Evaluators*. Beide Klassen sollen dazu ein Interface *IRunTimeEvaluable* implementieren

Da der KI in manchen Szenarien nur eine begrenzte Zeit zur Verfügung steht, ist es Aufgabe des *StrategyPicker*, Laufzeit und Präzision gegeneinander abzuwägen. Wie der Name impliziert, sollen dabei Strategien an das *Strategy*-Pattern angelegt implementiert werden. Dies ermöglicht ein leichtes Austauschen und einfache Erweiterung bereits vorhandener Strategien. Die Strategien selbst sollen dabei jedoch stets einen *MoveGenerator*, der regelkonforme Züge generiert, und einen *Evaluator*, der diese generierten Züge auf Basis der sich aus ihnen ergebenden Siegchancen bewertet.

6.3.4 Move Generator und MoveTree

Der *MoveGenerator* speichert generierte Züge ab und erzeugt so nach und nach einen *MoveTree*. Jeder Ast in dieser Baum-Struktur steht dabei für einen möglichen Spielzug, also der Platzierung eines *Tiles*. Dabei soll nicht nur das gelegte *Tile* berücksichtigt werden, sondern auch dessen Rotation. Ein weiterer Anspruch an den *MoveGenerator* ist das Eliminieren von redundanten Zügen und solchen, die zu einer Niederlage führen. Ein Unentschieden wird hierbei *nicht* als Niederlage gewertet. Um festzustellen, ob ein Zug zu einer Niederlage oder einem Sieg führt, speichert der *MoveTree* über die Klasse *Move* auch den aus einem theoretischen Zug resultierenden Spielstatus, insbesondere die *placements* und *nextTilePosition*. Die Klasse *Move* erlaubt darüber hinaus das speichern eines Evaluation-Werts (*evaluationPoints*), der durch den entsprechenden *Evaluator* von -1 (”noch nicht bewertet”) berechnet wird. Der *FirstMoveGenerator* stellt einen *MoveGenerator* dar, der mögliche Züge von verschiedenen Startpositionen aus berechnet.

7 Entwurf - GameEngine

7.1 Entwurfs-Klassendiagramm

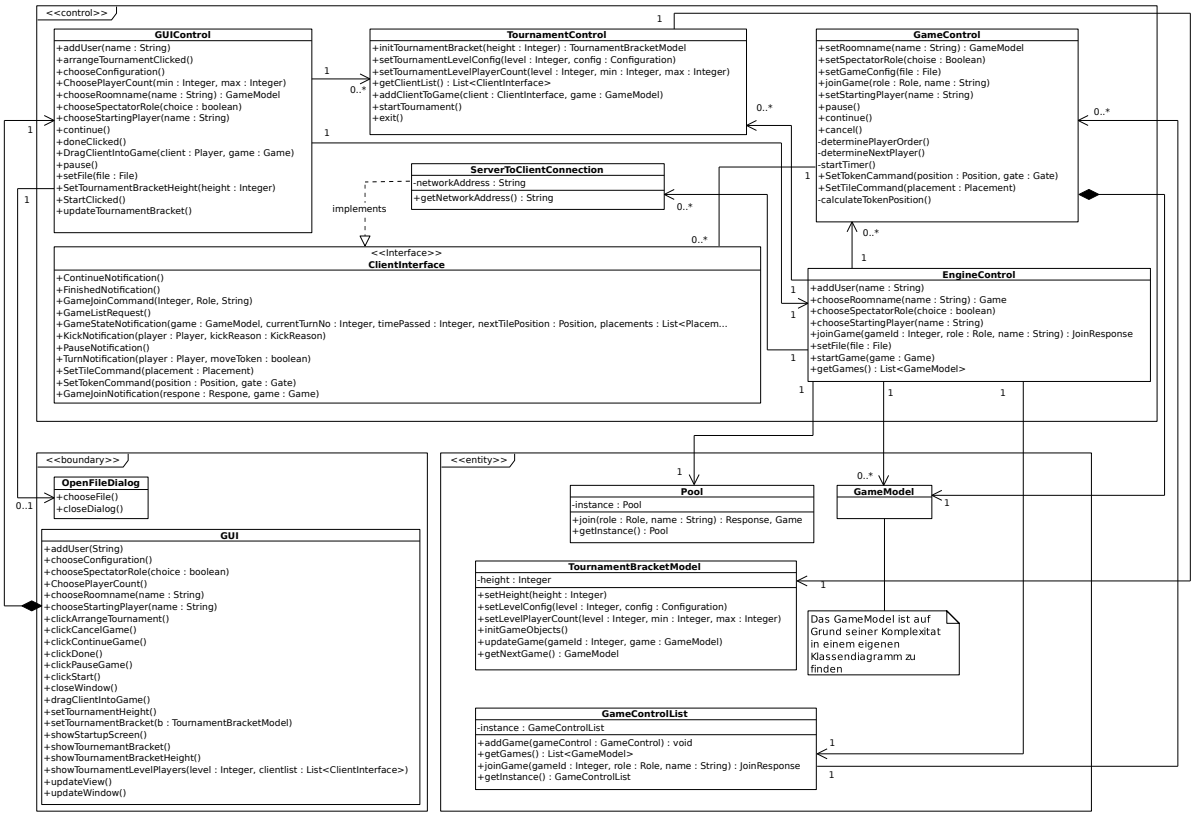

Abbildung 17: Entwurfs-Klassendiagramm GameEngine

Das Klassendiagramm aus der Analyse wurde im Enwurf weiter verfeinert und auf die Programmiersprache Java angepasst. Die Einteilung in *Control*, *Boundary* und *Entity* kann in diesem Zusammenhang als Einteilung in Packages verstanden werden. Da sich allerdings in der Implemtierung noch weitere Packages ergeben könnten wurden die Variablen und Methoden vorwiegend als *public* gesetzt, eine weitere Einschränkung dieser Sichtbarkeiten ist nicht ausgeschlossen.

7.1.1 EngineControl und GameControlList

Die *EngineControl* organisiert innerhalb der GameEngine alle laufenden Spiele und Turniere. In einer *EngineControl* ist genau eine *GameControlList* abgespeichter über welche alle

Spiele verwaltet werden, die *GameControlList* existiert genau einmal und wird daher mit dem Singelton Pattern gestaltet.

7.1.2 ClientInterface

Das *ClientInterface* ist ein Java Interface welches definiert über welche Methoden man mit den Clients kommunizieren kann. Als explizite implementierung ist die Klasse *ServerToClient-Connection* gedacht.

7.1.3 ServerToClientConnection

Die *ServerToCLientConnection* behandelt alle Methoden die vom Client gesendet werden.

7.1.4 GameModel

Siehe Beschribung in Abschnitt 10.1.

7.1.5 Pool

Der Pool verwaltet alle wartenden Spieler und kommt ebenfalls nur einmal vor, ist also auch als Singelton implementiert.

8 Entwurf - Spielkonfigurator

8.1 Entwurfs-Klassendiagramm

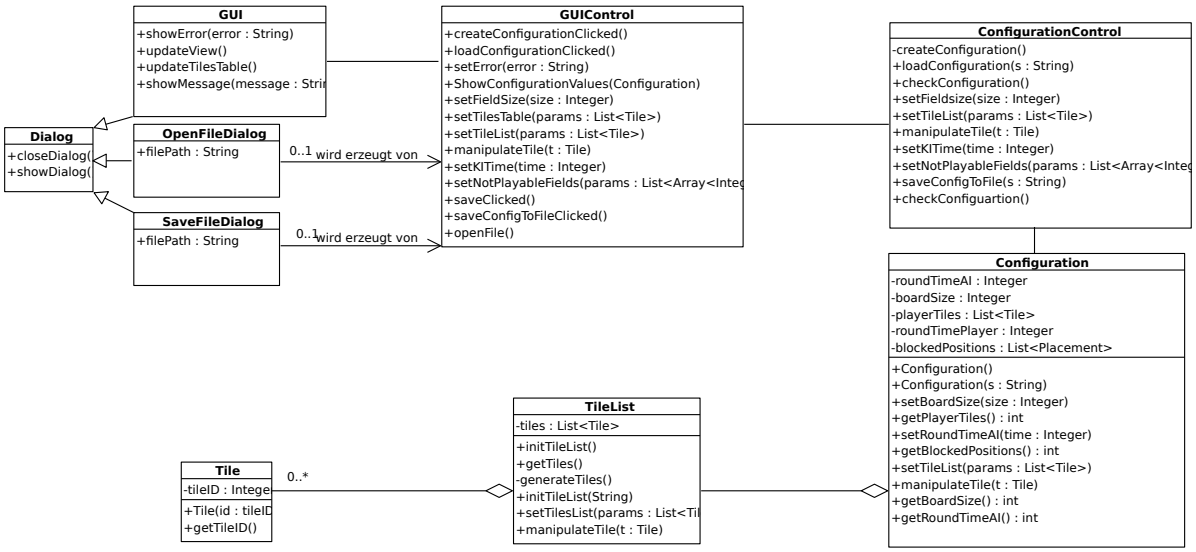

Abbildung 18: Entwurfs-Klassendiagramm Spielkonfigurator

Die Abbildung 18 zeigt das Entwurfsklassendiagramm des Spielkonfigurators. Zur besseren Übersichtlichkeit wurden Kardinalitäten von Eins weggelassen. Das Klassendiagramm unterteilen wir in drei Packages. Die Klassen GUI, OpenFileDialog und SaveFileDialog gehören dem Package View an. GUIControl und ConfigurationControl bilden das Control-Package. Tile, TileList und Configuration gehören dem Package Model an.

8.1.1 View Package

Das Package *View* unseres Spielkonfigurators ist für die Graphische Benutzeroberfläche des Konfigurators zuständig. Dieses Package behinhaltet die Klassen *GUI, OpenFileDialog* und *SaveFileDialog*. Mit Hilfe von OpenFile- und SaveFile-Dialogen können Konfigurationen geladen oder auch gespielt werden.
Die Main-GUI präsentiert dem Benutzer sämtliche zur Verfügung stehenden Einstellungsmöglichkeiten für das Spiel.

8.1.2 Control Package

Das Package Control unseres Spielkonfigurators überprüft Konfigurationen und leitet in der GUI konfigurierte Daten an die Datenhaltungsschicht weiter. Zu dem erzeugt die GUIControl die File-Dialoge. Es besteht aus den Klassen *GUIControl* und *ConfigurationControl*.

8.1.3 Model Package

Im Model-Package existiert die Klasse *Configuration*, diese führt sämtliche für eine Konfiguration benötigte Daten zusammen. Sie enthält zu dem eine *TileList* bestehend aus beliebig vielen *Tiles*. Die *TileList* ist zu dem in der Lage Vorschläge für *Tile*-Zusammenstellungen zu generieren.

9 Entwurf - Gemeinsame Komponenten

9.1 GameModel

Das GameModel ist eine gemeinsam genutzte Komponente, die die dem Spiel zugehörigen Daten speichert und verwaltet. Die Klassen sind denen des Interface-Dokuments nachempfunden, um den Datenaustausch zwischen Server und Client zu erleichtern. Das GameModel soll dem Observer-Pattern entsprechend mit interessierten Klassen kommunizieren, die sich mittels r*egisterListener()* und *unregisterListener()* am GameModel für Notifications über Änderungen an- und abmelden können. Klassen, die solche Notifications erhalten wollen, müssen dazu das Interface *IGameModelListener* implementieren.

Das GameModel bietet weiterhin die Möglichkeit, mittels eines *TileTranslators* die im Interface-Dokument definierten *TileIDs* in *ConfigTiles* zu überführen, welche Informationen über die konkrete Anordnung der Gates der Wegfelder enthalten. Dieses ist insbesondere für das Editieren von Wegfeldern durch den Spielkonfigurator, sowie bei der Berechnung von Verbindungen zwischen platzierten Tiles von Bedeutung.

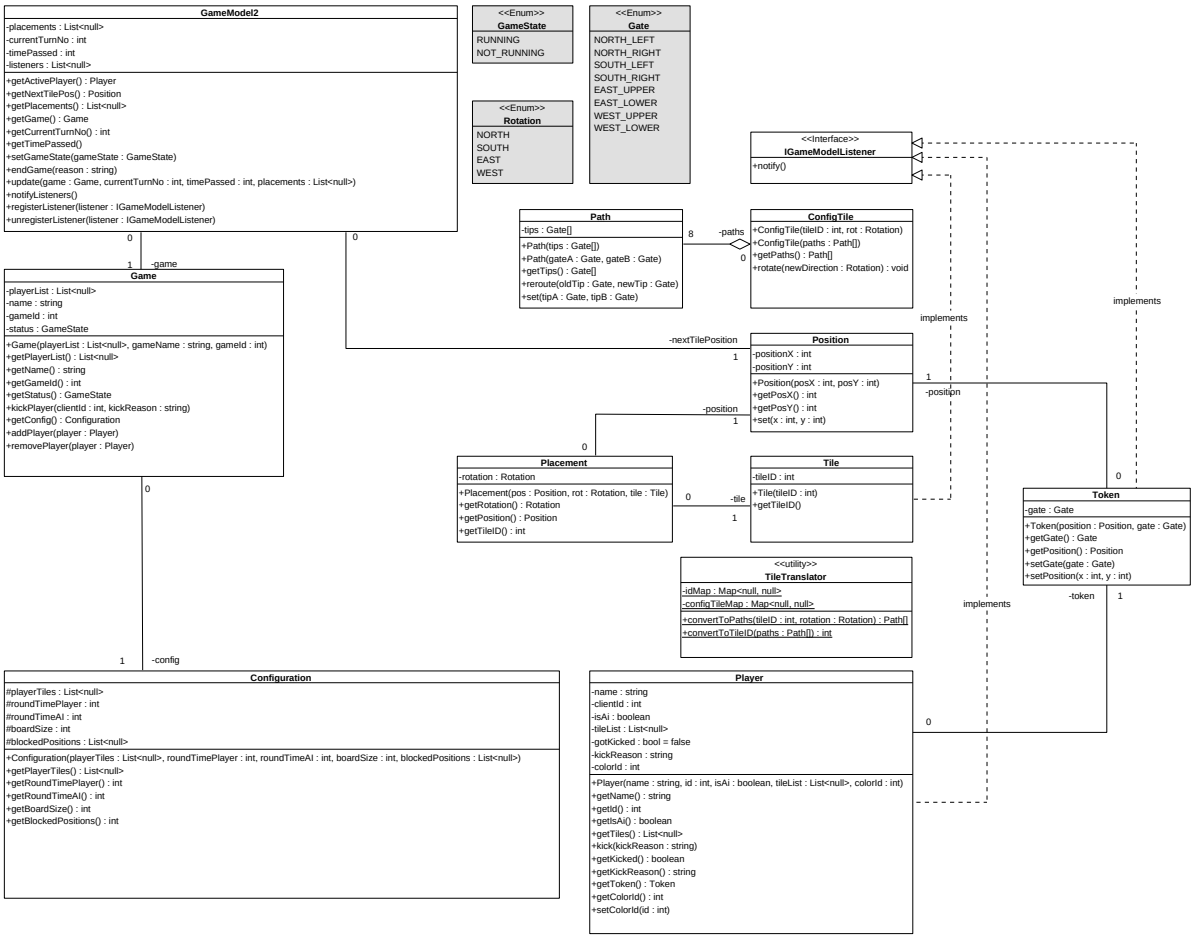

Abbildung 19: Entwurfs-Klassendiagramm des GameModels

A Anhang

A.1 Registrieren (GameEngine)

Das Analysesequenzdiagramm in Abbildung 20 zeigt den Use-Case ”Registrieren” aus Sicht des Servers.
Die GameEngine bietet einem verbundenen Client die Möglichkeit eine *GameList* anzufordern. Diese wird von der *EngineControl* aus dem *GameControlList* erstellt und an den Client geschickt.
Nach Auswahl verschiedener Parameter des Users auf der Client-Seite (näheres siehe Abbildung 7) verarbeitet die *EngineControl* den vom *ClientInterface* empfangenen *GameJoinCommand*. Je nach Parameter wird der Client dann dem allgemeinen Pool oder dem gewünschten Spiel in der *GameControlList* hinzugefügt. Abschließend schickt das *ClientInterface* eine Antwort, ob dem Pool oder dem gewünschten Game erfolgreich beigetreten werden kann.

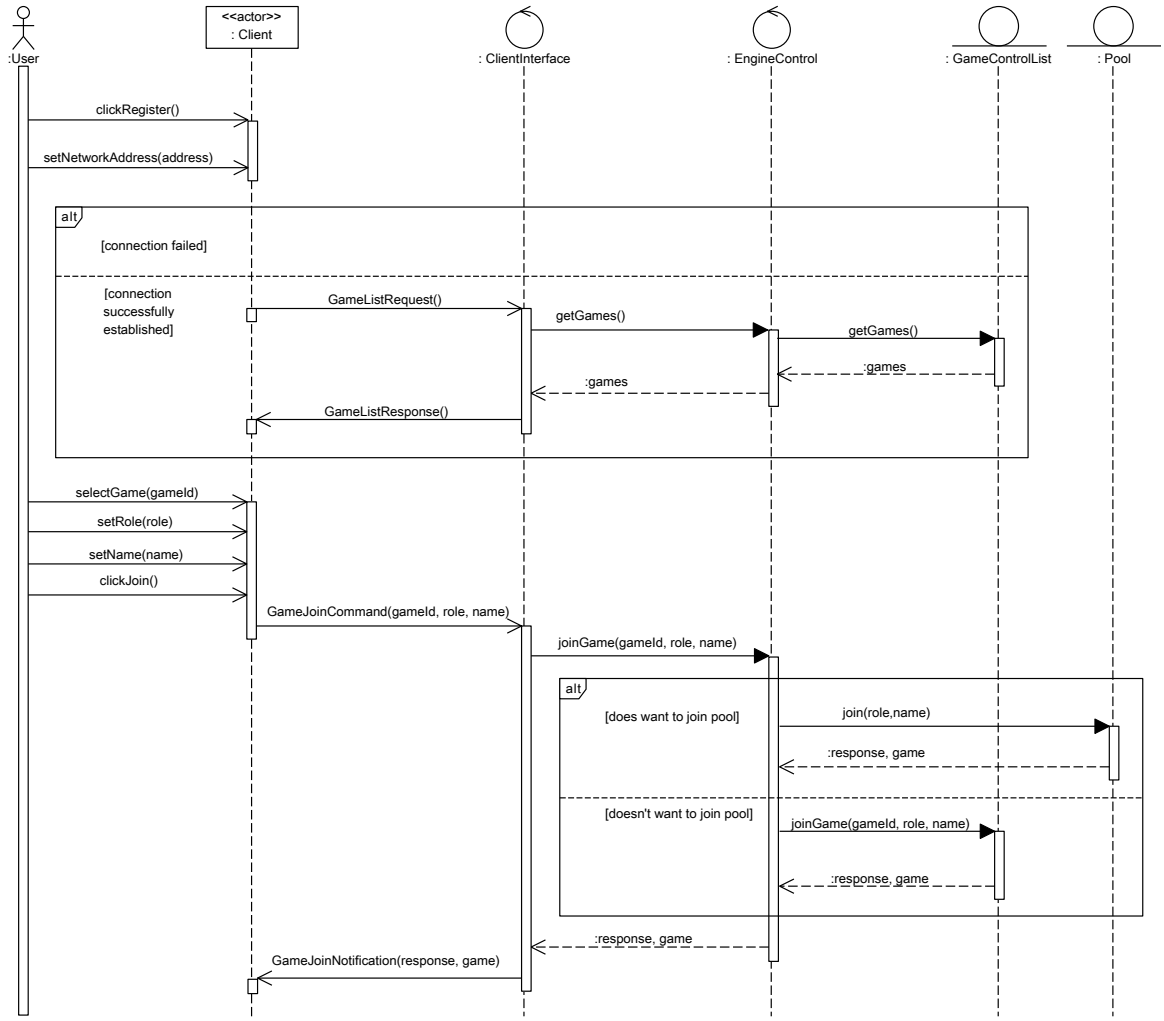

Abbildung 20: Sequenzdiagramm - Registrieren - GameEngine

Partie ausrichten (Client)

Das Sequenzdiagramm aus Abbildung 21 stellt dar, wie das Aufsetzen einer Tsuro-Partie aus Sicht des Hosts abläuft, der mit der Spiel-Engine interagiert. Zunächst wird ein Raumname gewählt und spezifiziert, ob der Host Beobachter der Partie sein möchte. Nachdem eine Spiel-Konfiguration ausgewählt wurde, können Teilnehmer hinzugefügt werden. Sollte es dabei zu keinen Problemen kommen, bekommen die Clients ihr Spiel erstellt. Sonst wird ein entsprechender Fehler beim Client angezeigt. Sind genügend Teilnehmer erfolgreich hinzugefügt worden, kann das Spiel gestartet werden.

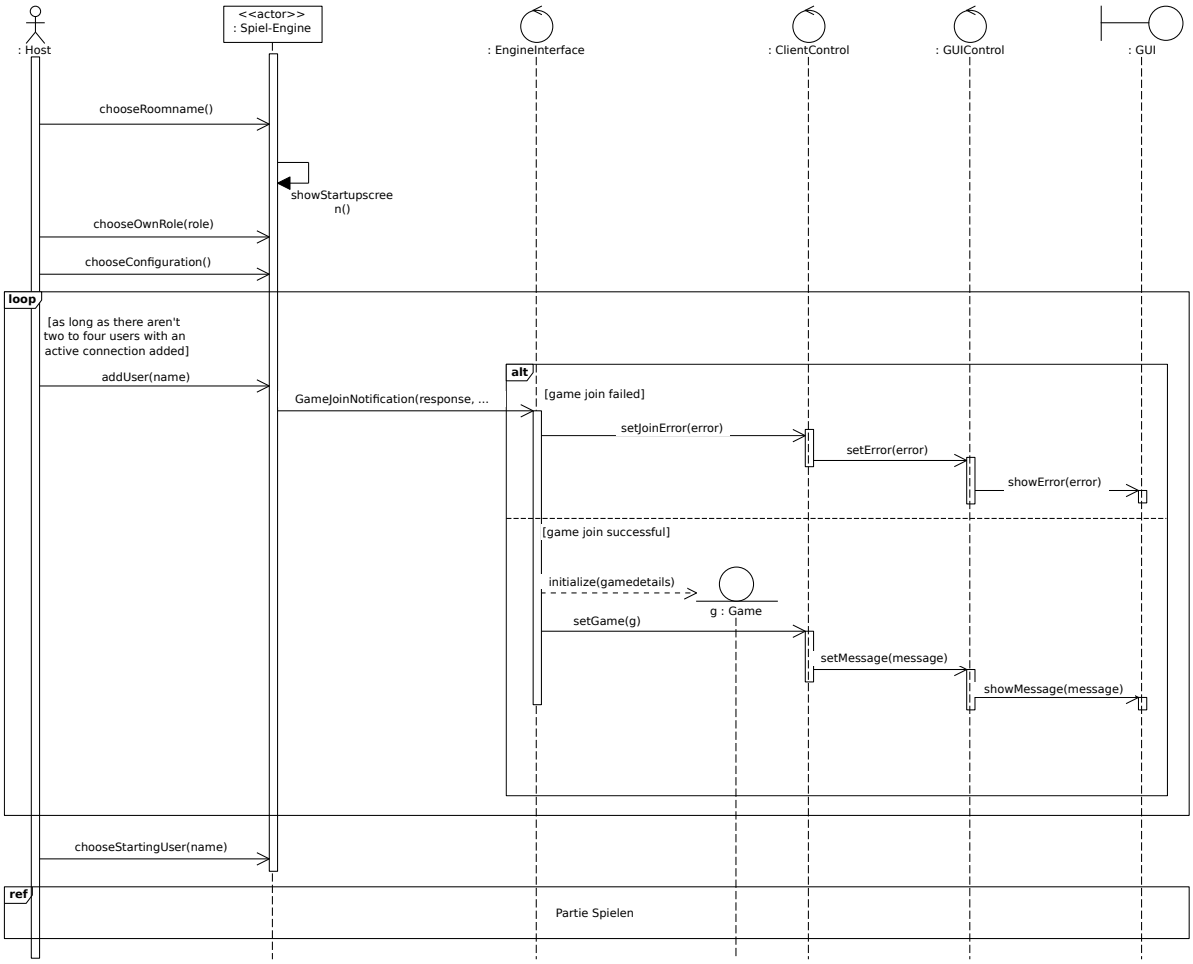

Abbildung 21: Sequenzdiagramm - Partie ausrichten - Client

A.2 Partie spielen (Client)

Das Sequenzdiagramm aus Abbildung 22 beschreibt aus der Sicht des Clients wie *Partie spielen* funktioniert. In diesem Diagramm wird nicht weiter auf die interne Funktionalität bezüglich der *Engine* eingegangen, deswegen taucht sie hier nur als *Actor* auf.
Während einer Partie werden Spielzüge durchgeführt. Zwischen den Spielzügen kann der *Host* Aktivitäten durchführen. Diese sind alle Optional.
Wenn der Host das Spiel pausiert, wird von der *Engine* über das *EngineInterface*, die *ClientControl* und die *GUIControl* an die *GUI* weitergegeben. Diese zeigt den Spielern eine Nachricht auf. Analog gilt dies für das Weiterspielen des Spiels. Die Pausierung des Spiels und das Weiterspielen sind nicht unabhängig voneinander machbar.
Während eines Spiels kann ein*Client* das Spiel verlassen, dann wird auf der *GUI* der Spielstatus aktualisiert. Wenn der Gewinner des Spiels feststeht, zeigt die *GUI* an, dass das Spiel beendet

wurde und den Gewinner. Des Weiteren kann der *Host* das Spiel abbrechen. In diesem Fall zeigt die *GUI* an, dass das Spiel abgebrochen wurde.
Alle Informationen die von der*Engine* weitergegeben werden an die *GUI*, werden über das*EngineInterface*, die *ClientControl* und die *GUIControl* an die *GUI* weitergegeben.

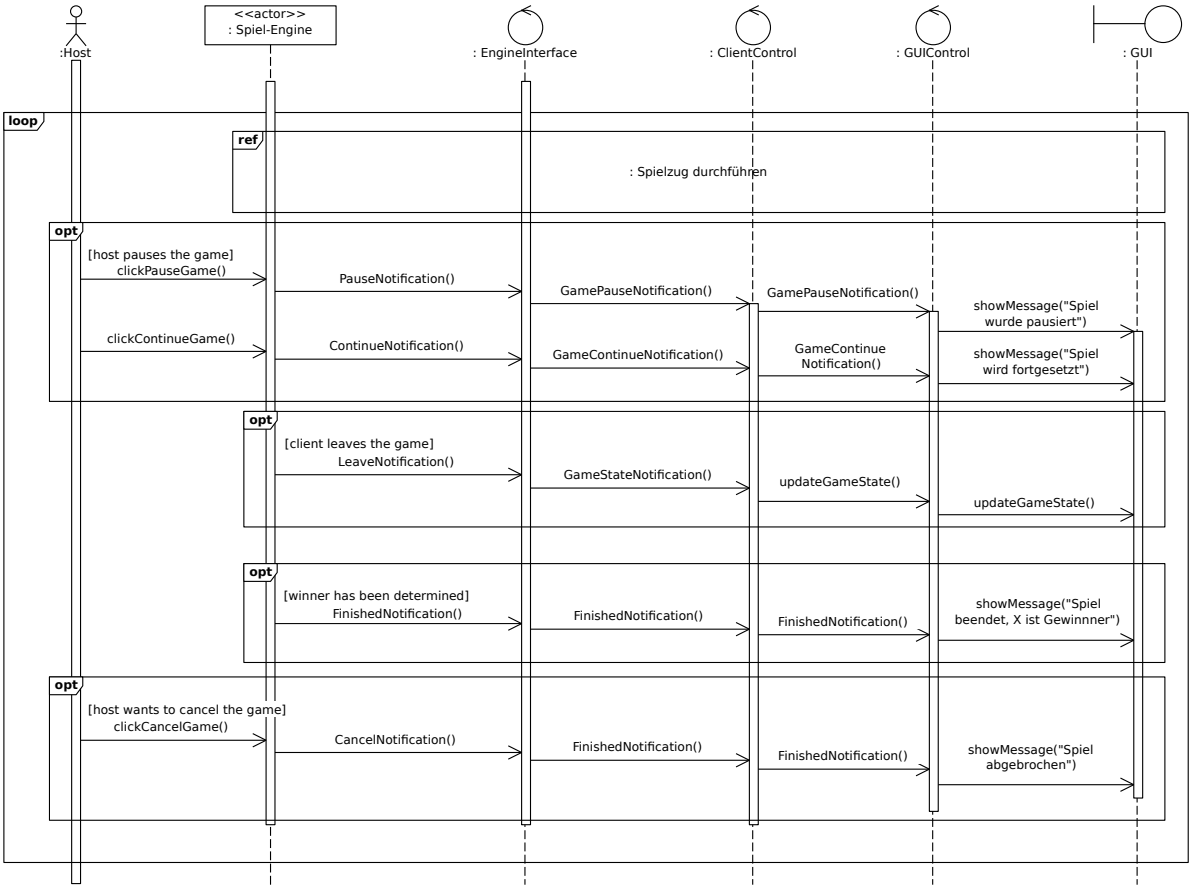

Abbildung 22: Sequenzdiagramm - Partie spielen - Client

A.3 Partie spielen (GameEngine)

Das Sequenzdiagramm aus Abbildung 23 stellt das Spielen einer Partie aus Sicht der Engine dar. In diesem Diagramm wird auf die interne Funktionaltität des Clients nicht weiter eingegangen, daher ist dieser als *actor* dargestellt.
Zu Beginn wird die Spielkonfiguration sowie die Liste der Spieler durch die *GameControl*-Klasse geladen. Darauf wird die Reihenfolge der Spieler bestimmt und der erste Spieler kann aufgefordert werden, seinen Spielzug durchzuführen.
Zu jedem Zeitpunkt hat der Host über die *GUI* die Möglichkeit, die Partie zu pausieren und anschließend wieder fortzusetzen. Außerdem kann die Partie jederzeit durch den Host

abgebrochen werden. Die *GUIControl* nimmt diese Ereignisse von der GUI entgegen und leitet sie an *GameControl* weiter. Die *GameControl* aktualisiert die Daten des *Game*-Objektes und benachrichtigt die Clients.
Falls ein Spieler die Partie verlässt, werden ebenfalls die Daten des *Games* aktualisiert und alle Clients über den neuen Zustand der Partie informiert.
Solange noch kein Gewinner feststeht, wird der nächste Spieler bestimmt, der zum Spielzug aufgefordert werden kann.
Wenn ein Client die Partie gewonnen, hat werden alle Clients über den Gewinn benachrichtigt und die Partie wird beendet.

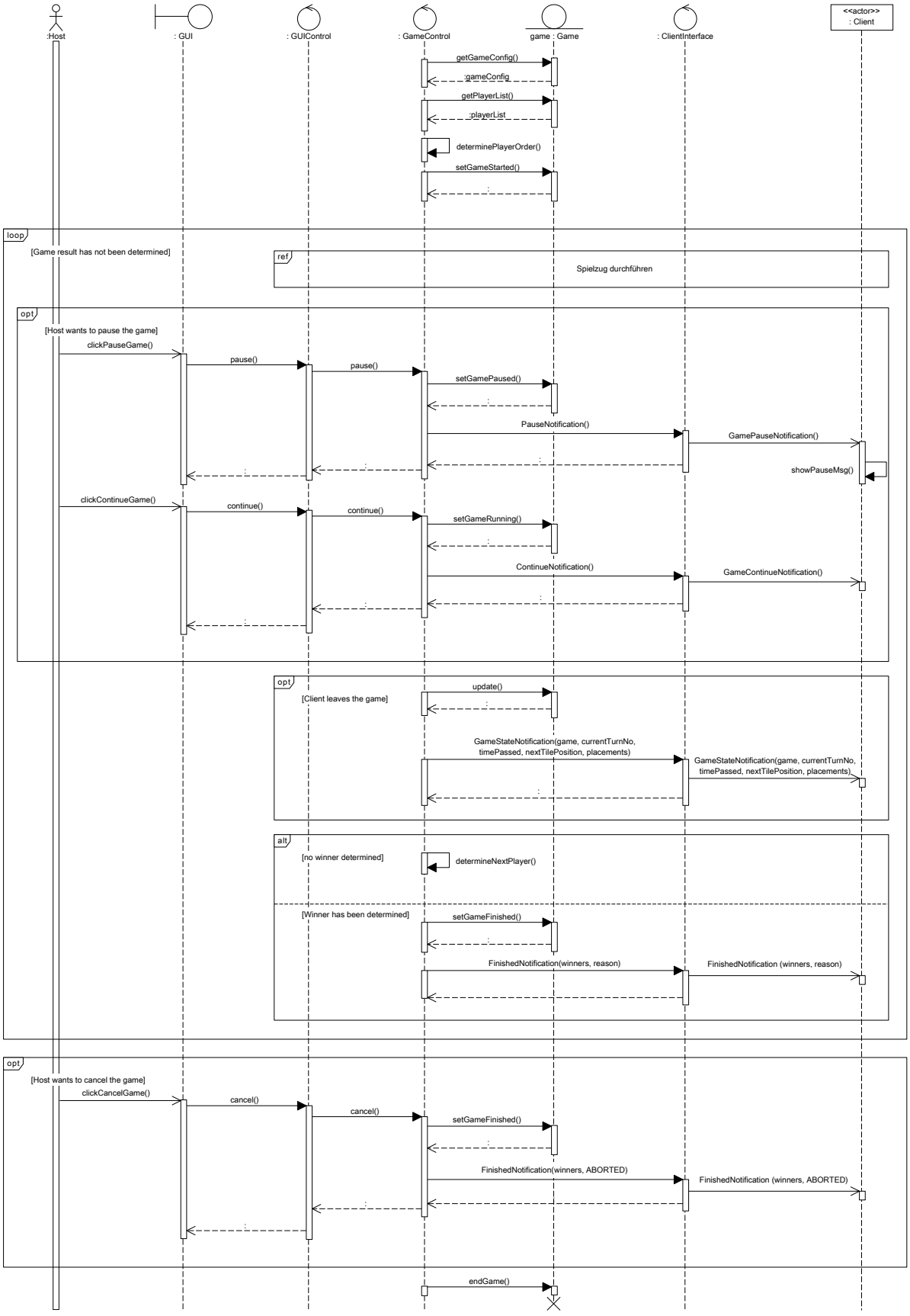

Abbildung 23: Sequenzdiagramm - Partie spielen - GameEngine

A.4 Spielzug durchführen (GameEngine)

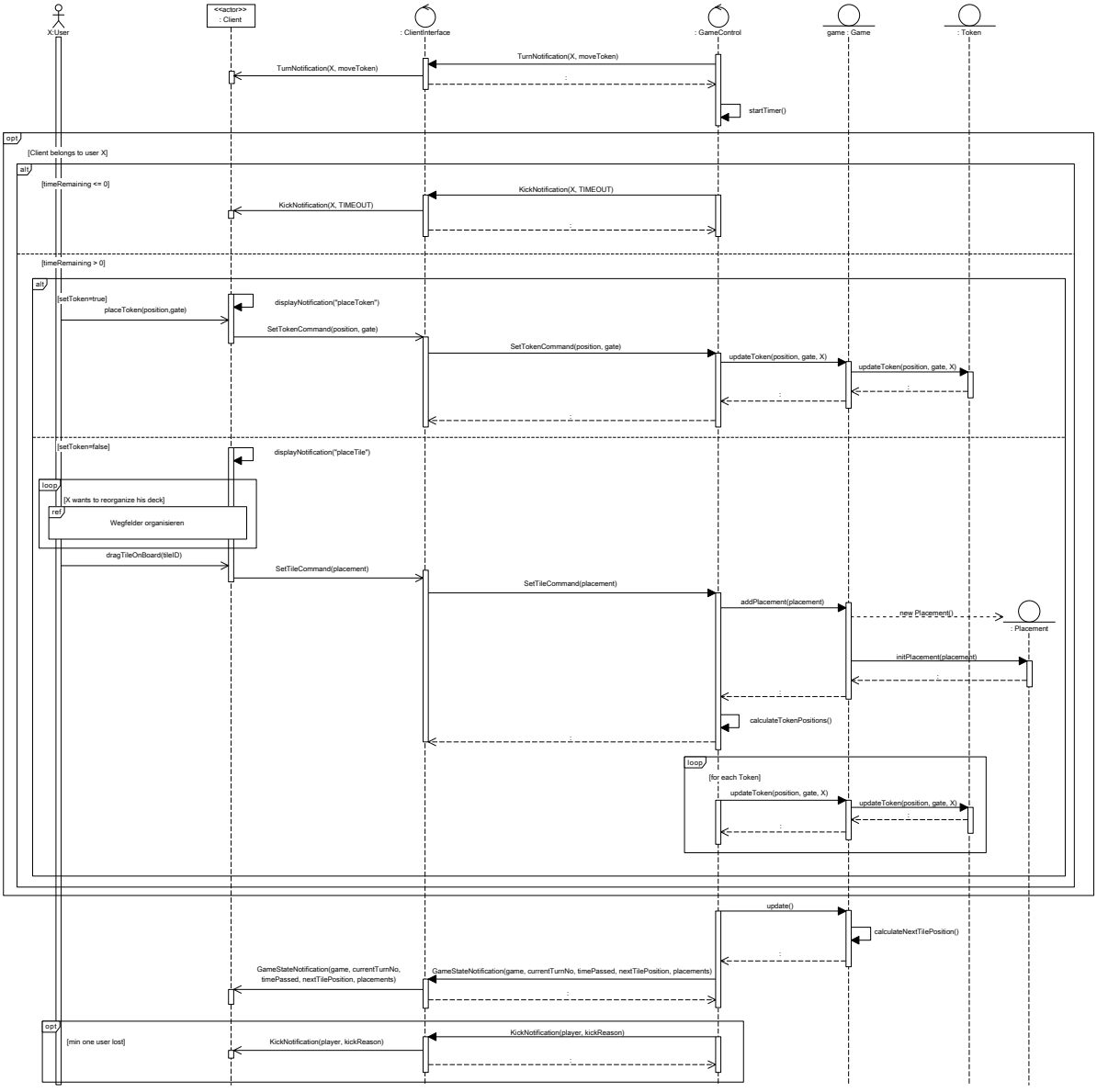

Abbildung 24: Sequenzdiagramm - Spielzug durchführen - GameEngine

Das Sequenzdiagramm aus Abbildung 24 stellt aus der Sicht der Engine dar, wie sich das Durchführen eines Spielzuges gestaltet. In diesem Diagramm wird auf die interne Funktionalität des Clients nicht weiter eingegangen, daher ist dieser als actor dargestellt.
Wie im Interface-Dokument festgelegt wird zu Beginn des Spiels eine *TurnNotification* an alle am Spiel registrierten Clients gesendet, diese Aufgabe übernimmt in unserer Modellierung die

GameControl-Klasse die das entsprechende Spiel verwaltet in welchem ein Spielzug durchgeführt werden soll.
Nur in dem Fall, dass der Client von *User X* angesprochen wurde darf dieser einen Spielzug durchführen. Nun kann es entweder sein, dass *User X* zu lange für seinen Zug braucht (dann wird eine *KickNotification* an seinen Client gesendet) oder nicht (*User X* setzt sein nächstes *Tile* oder, falls es der erste Zug ist, seine Spielfigur *Token*). In beiden Fällen werden die entsprechenden Werte in den jeweiligen Klassen gespeichert.
In jedem Fall, also auch, wenn *User X* nicht am Zug war, wird der entsprechende Client über den aktuellen Spielstand informiert. Eventuell wird eine *KickNotificaion* gesendet, falls ein Spieler nach diesem Spielzu gewonnen hat.

A.5 Turnier ausrichten (GameEngine)

Das Analyse-Sequenzdiagramm in Abbildung 25 stellt den Ablauf eines gesamten Turniers dar. Die einzelnen Schritte hierbei sind folgende:

- Organisiere Turnier
    - Der Ausrichter (X) erstellt ein neues Turnier
    - X legt Anzahl von Spielen fest
    - X spezifiziert alle Turnierebenen mit der jeweiligen Spieleranzahl
    - X teilt Clients Spielerrollen zu
- Turnierspiele werden ausgetragen und das Turnierdiagramm aktualisiert
- X beendet das Turnier

X interagiert hier mit der *GUI*- und *OpenFileDialog*-Boundary. Diese Aktionen werden an die *GUIControl* weitergereicht. Die *GUIControl* reicht die Informationen an die selbst erstellte *TournamentControl* weiter und ruft Nachrichten auf der *GUI*-Boundary auf. *TournamentControl* interagiert mit den Attributen des Turniers die in *TournamentBracketModel*-Entity gespeichert sind. Auch gibt *TournamentControl* der *EngineControl* Bescheid, dass ein neues Spiel gestartet wird.

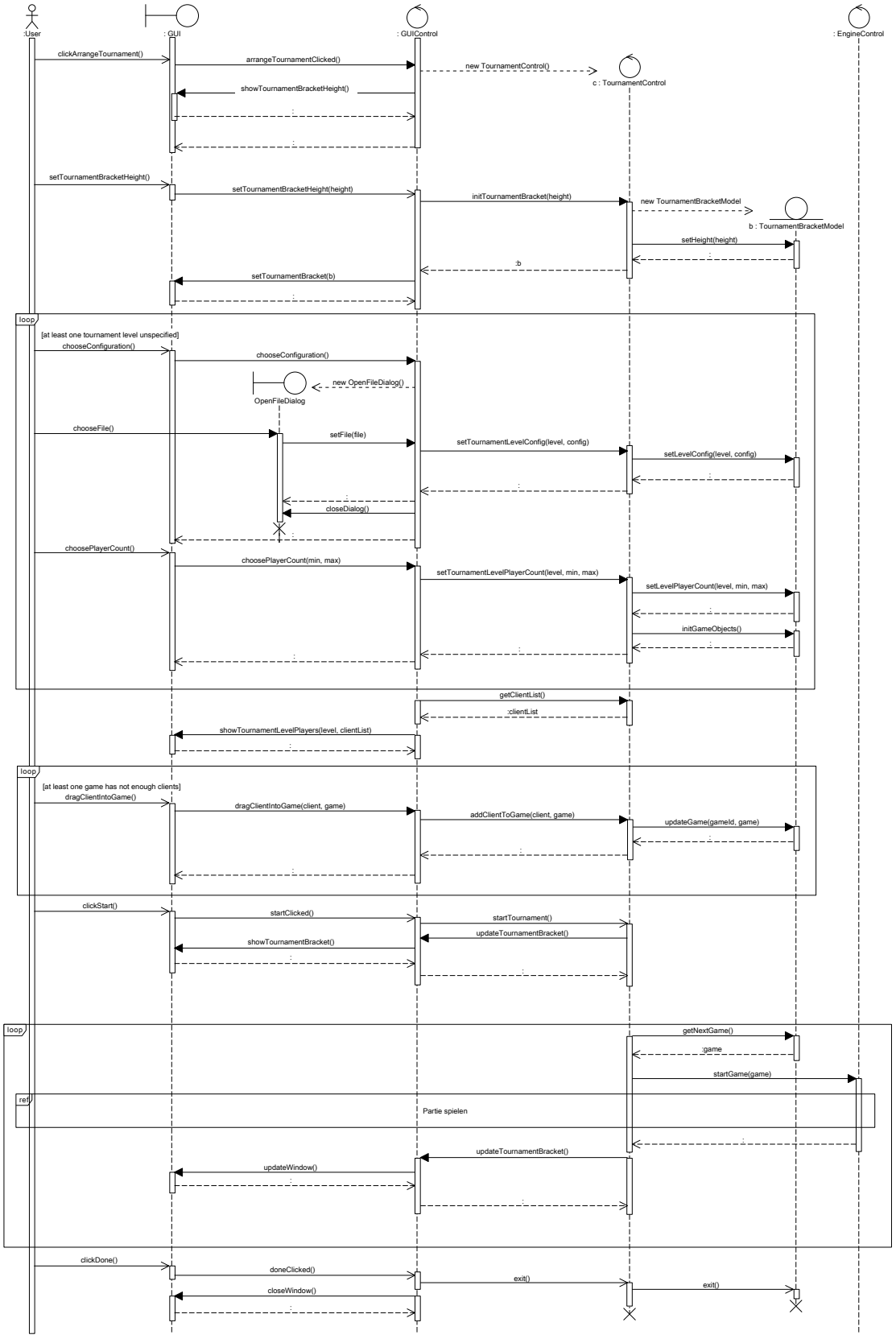

Abbildung 25: Sequenzdiagramm - Turnier ausrichten

Abbildungsverzeichnis

1 UML-Komponentendiaramm des gesamten Projektes 2
2 Drei-Schichten-Architektur - Schema 3
3 Model-View-Controller - Schema 4
4 Strategy-Pattern - Schema 5
5 Observer-Pattern Schema 5
6 OSingleton-Pattern Schema 6
7 Sequenzdiagramm - Registrieren - Client 10
8 Sequenzdiagramm - Wegfelder organisieren 11
9 Sequenzdiagramm - Spielzug durchführen - Client 13
10 Analyse-Klassendiagramm der Clients 14
11 Sequenzdiagramm - Partie ausrichten - GameEngine 16
12 Analyse-Klassendiagramm der GameEngine 17
13 Sequenzdiagramm - Spielkonfiguration erstellen 21
14 Analyse Klassendiagramm des Spielkonfigurators 23
15 Entwurfs-Klassendiagramm Clients 25
16 Entwurfs-Klassendiagramm KI 28
17 Entwurfs-Klassendiagramm GameEngine 30
18 Entwurfs-Klassendiagramm Spielkonfigurator 32
19 Entwurfs-Klassendiagramm des GameModels 34
20 Sequenzdiagramm - Registrieren - GameEngine 36
21 Sequenzdiagramm - Partie ausrichten - Client 37
22 Sequenzdiagramm - Partie spielen - Client 38
23 Sequenzdiagramm - Partie spielen - GameEngine 40
24 Sequenzdiagramm - Spielzug durchführen - GameEngine 41
25 Sequenzdiagramm - Turnier ausrichten 43