Universität Bremen
FB 3 – Informatik
Dr. Karsten Hölscher
TutorIn: Dr. Karsten Hölscher

Software–Projekt 2 2016

VAK 03-BA-901.02

Architekturbeschreibung

Fabian Schneekloth fa_sc1@uni-bremen.de 4015377
Benjamin Brennecke brennben@uni-bremen.de 4001280

Abgabe: 03. Juni 2016 — Version 1.1

Inhaltsverzeichnis

Version und Änderungsgeschichte

Version	Datum	Änderungen
1.0	05.04.2016	Dokumentvorlage als initiale Fassung kopiert
1.1	03.06.2016	Erste, dem Projekt entsprechend, ausgefüllte Fassung.

1 Einführung

1.1 Zweck

Dieses Dokument richtet sich an die Entwickler und Tester des Spieles "Gold and Greed", dabei handelt es sich um die Mitglieder der Gruppe "ProjectGG". Das Dokument soll die Architektur des Programms beschreiben und die verwendeten Design-Entscheidungen erläutern. Der Überblick über die Struktur des Programms dient als Grundlage für ein effizientes Programmieren und Testen der Anwendung.

1.2 Status

Die aktuelle Version ist v1.1, Stand dieser Version ist dass die bereit gestellten Inhalte der Dokumentvorlage entfernt wurden und mit den für das Spiel "Gold and Greed" relevanten Inhalten ersetzt wurden. Die Version stellt also die erste tatsächliche Architekturbeschreibung, für das Projekt dar.

Es ist aktuell nicht geplant weitere Änderungen an Version 1.1 vor zu nehmen, es ist also als finale Version angedacht. Sollte während des Projektes jedoch fest gestellt werden das gewisse Änderungen an der Architektur definitiv nötig sind um ein effizientes und erfolgreiches Fortführen des Projektes zu gewährleisten, wird der finale Status aufgehoben und das Dokument aktualisiert.

1.3 Definitionen, Akronyme und Abkürzungen

- **Spielsitzung:** Eine Spielsitzung beschreibt ein tatsächlich ablaufendes Spiel unabhängig von den anderen Funktionen des Programmes "Gold and Greed". Es besteht aus mindestens einer Karte auf der gespielt wird, mind. zwei Teams mit je mindestens einem Spieler und den Basisobjekten der beiden Spieler. Eine Spielsitzung kann verloren oder gewonnen werden. Ein Team hat gewonnen wenn für alle anderen Teams die Bedingungen für ein Verlieren erfüllt sind, diese sind dass alle Mitglieder des Teams jeweils alle ihre Basen verloren haben.
- **Einheit:** Eine Einheit stellt ein von einem Spieler kontrolliertes Objekt innerhalb einer Spielsitzung da. Einheiten stellen die einzigen bewegbaren Objekte in einer

Spielsitzung dar. Sie fungieren als eine Schnittstelle für die Interaktion zwischen Spielern.

- **Basis:** Eine Basis ist eine spezielle Einheit, welche nicht bewegt werden kann. Sie stellt ein Gebäude innerhalb des Spiels dar, welches verschiedene Funktionen zur Verfügung stellt. Ein Spieler kann dabei mehr als eine Basis besitzen, jedoch bedeutet ein Verlust aller Basen ein Verlieren des Spiels.
- **Buff:** Ein Buff ist ein autonomes Objekt innerhalb einer Spielsitzung, welches die Werte einer Einheit oder eines Spielers manipuliert. Die Manipulation kann dabei temporär oder permanent sein. Die Manipulation der Werte erfolgt in der Regel zum positiven, werden Werte ausschließlich negativ beeinflusst spricht man von einem De-Buff. Ein Buff wird nicht zwangsläufig grafisch dargestellt, wobei dies möglich ist.
- **Multiplayer:** Ein Multiplayer stellt einen Spielmodus dar bei dem mehr als ein Spieler innerhalb einer Spielsitzung menschlich sind, d.h. nicht von einem Computer gesteuert werden.
- **Architektur:** Eine Architektur beschreibt die Struktur der Klassen bzw. Dateien eines Programms. Außerdem beschreibt sie welche Elemente welche Rollen innerhalb des Programms übernehmen und wie diese Elemente zusammen arbeiten.
- **Derby:** Genauer, Apache Derby ist eine leichtgewichtige Javabasierte Datenbank. Sie wird verwendet um Daten eines Javaprogramms auch über die Laufzeit des Programmes hinaus zu speichern.
- **libGdx:** Hierbei handelt es sich um ein Javabasiertes Framework, welches auf die Entwicklung von Spielen ausgelegt ist. Das Framework übernimmt viele Aufgaben bezüglich der Darstellung der Spielelemente, sowie Eingaben und Ausgaben.
- **Framework:** Bei einem Framework handelt es sich um eine Sammlung von Klassen, welche in ein Programm eingebunden werden können um diesem neue Funktionen zur Verfügung zu stellen. Das Framework besitzt dabei selbst eine gewisse Struktur die zumindest teilweise auf das Programm übertragen wird. Man kann es also vereinfacht als ein Programm betrachten, welches erst durch dass einbetten in ein anderes Programm lauffähig wird.
- **RMI:** Bei RMI(Remote Method Invocation) handelt es sich um eine API, welche Teil der grundlegenden Java-Bibliotheken ist, also keine Inhalte dritter benötigt. Die API ermöglicht das Teilen von Java-Objekten über mehrere Anwendungen hinweg. Die Anwendungen greifen dabei über ein lokales Netzwerk oder das Internet auf einen Computer zu, welcher zuvor entsprechende Objekte zum Teilen zur Verfügung gestellt hat.
- **RemoteInterface:** Ein Remote-Interface ist ein spezielles Java-Interface, welches es ermöglicht die implementierenden Objekte über die RMI API zu Teilen. Die Interfaces müssen dabei lediglich "RemoteInterface" erweitern und unterscheiden sich ansonsten nicht von anderen Interfaces, wobei zusätzliche Aspekte definiert

werden können(was in "Gold and Greed" jedoch nicht relevant ist). Außerdem müssen alle Anwendungen welche die geteilten Objekte verwenden wollen, die gleichen Interfaces mit der gleichen Projektstruktur besitzen.

- **Registry:** Bei einer Registry handelt es sich um eine Datei, welche manuell oder durch Java-Code erstellt werden kann. An sie können Java-Objekte mit einem Namen als Identifikation angebunden werden, dies ermöglicht es anderen Anwendungen unter Verwendung von RMI auf die Objekte zu zu greifen.

1.4 Referenzen

[1] *Architekturbeschreibung LaTeX-Vorlage* aus StudIP.
Abgerufen am 05.04.2016.
[2] *Architekturbeschreibung der Gruppe BreakingBits*,
entstanden im Modul "Software-Projekt 2" WiSe 2012/13, Version 1.1
[3] *Architekturbeschreibung der Gruppe BitMonkeys*,
entstanden im Modul, "Software-Projekt 2" WiSe 2012/13, Version 1.0
[4] *report_CodeBreakers*, Bewertung der Architekturbeschreibung "CodeBreakers",
letzte Änderung: 21.01.2016
[5] *Mindestanforderungen re-SWP 2 SoSe 2016 aus StudIP.*
Abgerufen: 02.06.2016
[6] *Hinweise zur Abgabe der Architekturspezifikation* aus StudIP.
Abgerufen am 02.06.2016
[7] *Terminübersicht re-SWP 2 aus StudIP*,
Abgerufen: 02.06.2016
[8] *Foliensätze aus SWP 1, WiSe 2014/15*
Abgerufen:
Sätze 01-10: 14.07.2015
Sätze 11-12: 07.10.2015
[9] *UML 2 glasklar, AuthorIn: Chris Rupp, Stefan Queins, die SOPHISTen*,
Nürnberg, 2012 Carl Hanser Verlag München

1.5 Übersicht über das Dokument

- **Kapitel 1 - Einführung:** Dieses Kapitel beschreibt den Sinn dieses Dokumentes und liefert einen Überblick über die verwendeten Definitionen und Begriffe, Quellen, das Dokument selbst und mögliche Änderungen, sollte es weitere Versionen geben.
- **Kapitel 2 - Globale Analyse:** In Kapitel 2 werden relevante Einflussfaktoren und Probleme identifiziert. Außerdem werden mögliche Strategien heraus gearbeitet und beschrieben welche davon verwendet werden.
- **Kapitel 3 - Konzeptionelle Sicht:** Hier wird die Architektur auf einer sehr abstrakten Ebene beschrieben. Der Fokus liegt hierbei auf den Aufgabenbereichen

und ihren Verbindungen innerhalb des Programms und nicht auf der unterliegenden Dateistruktur.

- **Kapitel 4 - Modulsicht:** In diesem Kapitel wird die vorherige Sicht verfeinert und genauer betrachtet. Dafür werden die Komponenten aus Kapitel 3 in Module aufgeteilt wie sie später auch im Projekt auftauchen. Die einzelnen Module werden dann durch Klassendiagramme weiter beschrieben, außerdem werden auch die Schnittstellen zwischen den Modulen beschrieben.
- **Kapitel 5 - Datensicht:** Die Klassendiagramme aus Kapitel 4 werden hier in reduzierter Form weiter verwendet um die Beziehungen zwischen den Daten und ihre Bedeutung zu erläutern. Funktionen werden hier nicht beachtet.
- **Kapitel 6 - Ausführungssicht:** In der Ausführungssicht wird das Programm zur Laufzeit dargestellt, es wird dabei beschrieben, welche Daten sich wo befinden und wie mit einander zusammen arbeiten.
- **Kapitel 7 - Zusammenhänge zwischen Anwendungsfällen:** Dieses Kapitel verwendet Sequenzdiagramme um den Zusammenhang zwischen Architektur und Anwendungsfällen zu erläutern. Dabei werden exemplarische Anwendungsfälle verwendet, welche möglichst viele Elemente der Architektur verwenden um die Zusammenhänge möglichst gut dar zu stellen.
- **Kapitel 8 - Evolution:** Das letzte Kapitel liefert einen Überblick über mögliche Änderung an der Software bzw. Architektur und betrachtet wie sich diese Änderungen auswirken würden.

2 Globale Analyse

2.1 Einflussfaktoren

1.Deadline	
Beschreibung:	Das Produkt muss am 05.08.16 abgegeben werden.
Flexibilität:	Unflexibel da vorgegeben.
Änderbarkeit:	Kann vom Dozenten verschoben werden.
Auswirkungen:	Die Architektur muss so gewählt werden dass die Mindestanforderungen zum gegebenen Zeitpunkt vollständig erfüllt werden können.

2.Entwicklungsbudget	
Beschreibung:	Der Gruppe besteht lediglich aus 2 Personen.
Flexibilität:	Gering, nur zum negativen, da nach Projektbeginn nur Gruppenmitglieder austreten jedoch nicht hinzu kommen können.
Änderbarkeit:	Es ist möglich das eine Person vollkommen aus fällt.
Auswirkungen:	Alle Gruppenmitglieder müssen in der Lage sein das Projekt alleine fort zu führen.

3.Gradle	
Beschreibung:	Das Spiel wird durch Gradle automatisch gebaut.
Flexibilität:	Es kann alternativ Maven verwendet werden.
Änderbarkeit:	Die Auswahl des Build-Systems kann sich im Laufe des Projektes ändern, sofern dies als Sinnvoll erachtet wird.
Auswirkungen:	Die Architektur muss auf die Struktur von Gradle angepasst werden.

4.RMI	
Beschreibung:	Es wird die Java API RMI zur Kommunikation zwischen Server und Client verwendet.
Flexibilität:	Es kann alternativ auch über TCP/IP Socketcommunication oder andere Protokolle kommuniziert werden.
Änderbarkeit:	Keine Änderbarkeit, da die Architektur auf die gewählte Kommunikation angepasst werden muss.
Auswirkungen:	Es muss sichergestellt sein dass alle geteilten Objekte ein entsprechendes Remote-Interface zur Verfügung stellen und das Server und Client Projekt die gleiche Package-Struktur inkl. der Interfaces aufweisen.

5.Datenbank	
Beschreibung:	Es muss eine leichtgewichtige, lokale, relationale Datenbank verwendet werden.
Flexibilität:	Die Datenbank ist frei wählbar, solange sie die Bedingungen erfüllt.
Änderbarkeit:	Keine Änderbarkeit.
Auswirkungen:	Die Architektur muss eine Schnittstelle zu der Datenbank aufweisen.

6.Funktionen	
Beschreibung:	Es sind einige Funktionen vorgegeben, welche für den Endbenutzer ausführbar sein sollen.
Flexibilität:	Keine.
Änderbarkeit:	Keine.
Auswirkungen:	Die Architektur muss entsprechende Funktionen in der Logik bereit stellen und durch eine Userschnittstelle zugänglich machen.

7.Multiplayer	
Beschreibung:	Das Spiel muss mehrere menschliche Spieler gleichzeitig zulassen.
Flexibilität:	Keine.
Änderbarkeit:	Keine.
Auswirkungen:	Die Architektur muss mehrere Client-Zugriffe auf ein, laufendes Spiel zulassen.

8.Spielfeld	
Beschreibung:	Die Elemente innerhalb eines Spiels(Einheiten, Gebäude etc.) werden auf einem Spielfeld dargestellt.
Flexibilität:	Eine grafische Darstellung ist nicht zwingend Notwendig, das Spiel könnte auch vollkommen Textbasiert sein.
Änderbarkeit:	Es kann sich im Projektverlauf dazu entschieden werden, anstatt der grafischen Darstellung, Texte mit entsprechenden Informationen aus zu geben.
Auswirkungen:	Die Architektur muss eine oder mehrere Schnittstellen zur Darstellung grafischer Elemente zur Verfügung stellen.

9.Rundenbasiert	
Beschreibung:	Der Spielablauf folgt einem rundenbasierten Muster.
Flexibilität:	Das dem Spielablauf zugrunde liegende System ist frei wählbar.
Änderbarkeit:	Die Gruppe kann im Projektverlauf entscheiden auf ein Echtzeitbasiertes System zu wechseln.
Auswirkungen:	Die Architektur muss den Zugriff für verschiedene Spieler innerhalb einer Runde ermöglichen/einschränken, sowie die eingehenden Befehle in entsprechender Form verarbeiten können.

10.Exzellenz Anforderungen	
Beschreibung:	Einige der Mindestanforderungen, sind als Exzellenz-Anforderung eingestuft und sind somit nicht für das Bestehen notwendig.
Flexibilität:	Entsprechende Anforderungen müssen nicht zwingend erfüllt werden.
Änderbarkeit:	Der Dozent hat die Möglichkeit Anforderungen zu Exzellenz-Anforderungen hoch zu stufen oder Exzellenz-Anforderungen zu zwingend notwendigen runter zu Stufen.
Auswirkungen:	Die Architektur darf die Exzellenz-Anforderungen nicht als zwingend erforderliches Modul umsetzen.

11.Server-Client zusammen	
Beschreibung:	Es gibt nur ein Programm, das Server und Client gleichzeitig umfasst.
Flexibilität:	Server und Client können auch in zwei separate Programme aufgeteilte werden.
Änderbarkeit:	Es kann sich im Projektverlauf dazu entschieden werden Server und Client auf zu teilen.
Auswirkungen:	Die Architektur muss eine Schnittstelle zur Verfügung stellen um Server und Client separat starten zu können, außerdem muss erkennbar sein welche Klassen jeweils für Server oder Client relevant sind.

2.2 Probleme und Strategien

Arbeitsverteilung
Es stehen nur zwei Entwickler zur Verfügung, daher müssen beide Entwickler in der Lage sein die Arbeit des jeweils anderen zu übernehmen, für den Fall dass einer der beiden aus fällt. **Einflussfaktoren** Deadline Entwicklungsbudget Gradle RMI Datenbank Spielfeld Server-Client zusammen
S1:Pair-Programming Diese Strategie sieht vor das gesamte Projekt in Pair-Programming zu entwickeln.
S2:Aufteilung nur Modul-intern Nach dieser Strategie wird die Implementation nur innerhalb der verschiedenen Bereiche auf die Entwickler aufgeteilt, d.h. z.B. nicht zwischen GUI und Logik.

Wir haben uns hier für S2 entschieden, da diese Strategie es ermöglicht dass alle Entwickler Überblick über alle Bereiche des Projektes erhalten, ohne dabei die Geschwindigkeit der Arbeitsteilung zu verlieren. Eine genauere Definition der "Bereiche" ist in Kapitel 3.4 zu finden.

Mehrere Verbindungen zu einem Server
Es müssen mehrere Verbindungen zu einem Server möglich sein. **Einflussfaktoren** RMI Multiplayer Server-Client zusammen
S1:Verwendung von RMI Bei dieser Strategie wird RMI verwendet, wodurch einkommende Zugriffe automatisch sequentiell verarbeitet werden.
S2:Direkte Verbindung in eigenem Thread Hierbei wird zu jedem Client eine direkte Verbindung erstellt und in einem eigenen Thread verarbeitet.

In diesem Fall wurde S1 gewählt, da diese Strategie es ermöglicht den Server-Client Aspekt bei der Implementation weniger zu beachten und es noch einige andere Probleme löst.

Gleiche Struktur für Server und Client
Die Package/Klassenstruktur muss für Server und Client gleich sein, damit die Remote-Interfaces gefunden werden können. **Einflussfaktoren** Gradle RMI Server-Client zusammen
S1:Paralleles Aufsetzen Hierbei werden neue Interfaces parallel beim Server und beim Client hinzu gefügt, um so eine gleiche Struktur zu gewährleisten.
S2:Einzelnes Programm Diese Strategie sieht vor Server und Client in einem Programm darzustellen und somit in die gleiche Struktur einzuordnen.

Wir haben uns hier für S2 entschieden, da diese Strategie weniger Fehlerpotential zu lässt und allgemein weniger Verwaltungsaufwand darstellt.

Schummeln durch Datenmanipulation
Es darf für Spieler nicht möglich sein durch ein Manipulieren der Daten auf ihrem Computer einen Vorteil im Spiel zu erhalten. **Einflussfaktoren** RMI Multiplayer Spielfeld
S1:Überprüfen der Client-Eingaben Bei dieser Strategie werden eingehen Befehle von den Clients auf valide Angaben überprüft.
S2:Don't trust the Client Nach dieser Strategie wird dem Client niemals vertraut und somit gar nicht erst die Möglichkeit gegeben Logik-bezogene Befehle selbst zu verarbeiten, stattdessen kann er nur Aufforderungen an den Server senden.

Hier haben wir uns für S2 entschieden. Diese Strategie liefert aufgrund der geteilten Objekte zwar eine geringere Sicherheit, ist jedoch einfacher um zu setzen, zusätzlich ist ein Schutz gegen Schummeln nicht zwingend Notwendig.

Synchronisation der Clients
Der Status aller Spielelemente muss bei allen Clients synchron sein. **Einflussfaktoren** Multiplayer Spielfeld RMI
S1:Geteilte Objekte - RMI Hierbei verwenden alle Clients, sowie der Server die gleichen Objekte, welche über RMI geteilt werden und so automatisch synchronisiert werden.
S2:Senden von Updates Bei dieser Strategie sendet der Server nach jeder Änderung eine Update-Nachricht an alle Clients mit Informationen über die Änderungen.

In diesem Fall haben wir uns für S1 entschieden, da dies weniger Aufwand bezüglich der Implementation darstellt und wie bereits in "Mehrere Verbindungen zu einem Server" genannt, vorteilhaft für andere Aspekte des Projektes.

3 Konzeptionelle Sicht

Im folgenden werden die Komponenten des Produktes beschrieben, dabei wird deren Verhalten und allgemeine Rolle beschrieben, jedoch nicht ihre konkrete Umsetzung. Obwohl, wie in Abb.1 zu sehen, es sich beim Server und Client um zwei verschiedene Komponenten handelt, ist an zu merken dass sich die Funktionalität für beide Elemente in einem einzelnen System befindet, sie also durchaus gleichzeitig auf einem Computer ausgeführt werden können, ohne dass verschiedene Programme benötigt werden.

3.1 Datenbank

Hierbei handelt es sich um eine externe Komponente, welche zwar verwendet aber nicht selbst implementiert wird. Wobei extern sich hier auf die ablaufenden Prozesse bezieht, die Datenbank selbst ist direkt in das Programm integriert. Die Datenbank kümmert sich um das Speichern von Daten, auch über ein Beenden des Programmes hinaus. Es wird eine Derby Datenbank verwendet. Die Kommunikation mit der Datenbank wird in der Komponente "DBController" beschrieben.

3.2 Server

Bei dieser Komponente handelt es sich um eine Sammelkomponente, welche alle Elemente umfasst die auf dem Server ausgeführt werden. Die Aufgabe der Server-Komponente ist es die Kommunikation zwischen verschiedenen Clients zu ermöglichen und zu verwalten. Außerdem finden hier jegliche darstellungs-unabhängige Logikprozesse statt, sowie die Kommunikation mit der Datenbank.

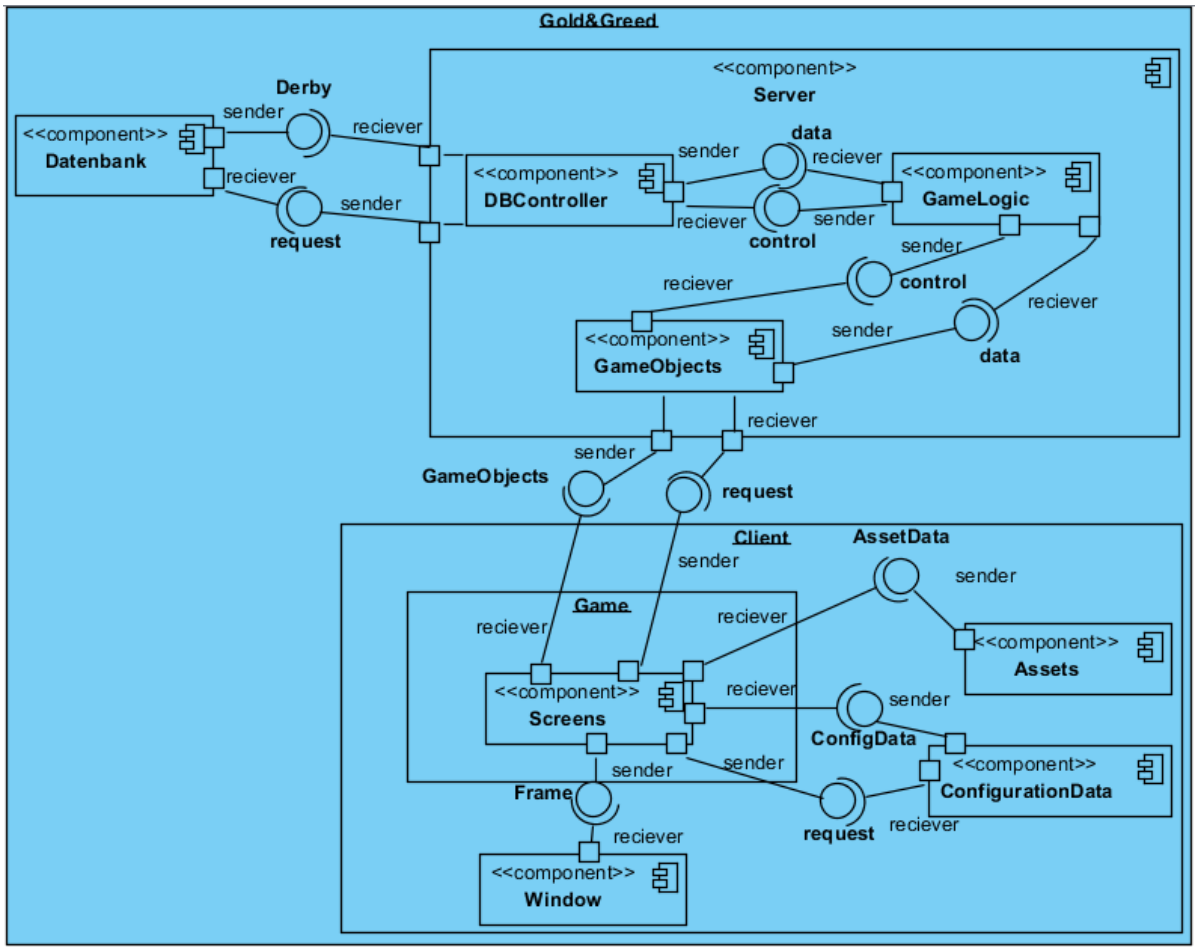

Abbildung 1: Die Komponenten des Programms

3.2.1 DBController

Diese Komponente stellt die Schnittstelle zwischen Server und Datenbank dar. Sie kommuniziert über Anfragen mit der Datenbank um Daten zu Speichern oder zu erhalten und an die übrige Logik zurück zu geben. Dabei ist zu beachten dass der DBController keinen Einfluss darauf nimmt welche Daten, wann gespeichert werden sondern lediglich auf Anfragen aus "GameLogic" reagiert.

3.2.2 GameLogic

Die GameLogic-Komponente umfasst jegliche darstellungs-unabhängige Logik, ausgenommen Datenobjekte. Dies beinhaltet alle Klassen welche nicht mit dem Client geteilt werden. Hier wird außerdem entschieden welche Daten, wann gespeichert oder für welche Clients zur Verfügung gestellt werden. Zum Speichern bzw. Laden sendet diese Komponente Aufforderungen und entsprechende Daten bzw. Anfragen an den DBController,

welcher dann die Kommunikation mit der Datenbank übernimmt. Die Verbindung zu "GameObjects" besteht hauptsächlich aus einem Verwenden der Komponente, wobei spezielle Elemente aus "GameObjects" auch Anfragen an die GameLogic senden können.

3.2.3 GameObjects

Dies ist das letzte Element innerhalb des Servers. In dieser Komponente befinden sich alle mit dem Client geteilten Objekte, das beinhaltet fast alle darstellbaren Objekte, sowie spezielle Verwaltungsobjekte für den Zugriff auf die "GameLogic". Diese Komponente stellt auch die Zugriffsschnittstelle für die Clients zur Verfügung, welche speziell durch Remote-Interfaces innerhalb von GameObjects realisiert wird. Obwohl sich hier die darstellbaren Objekte befinden, besteht hier keine Verbindung zu den Assets, stattdessen findet man hier nur die Information welche Assets zu einem Objekt gehören, um dann später in "Screens" verwendet zu werden.

3.3 Client

Beim Client handelt es sich um die Komponente welche bei jedem Spieler läuft, sie kümmert sich um die Darstellung eines Spiels und lokale Einstellungen.

3.3.1 Game

Die Game-Komponente dient lediglich zur Veranschaulichung, dass es sich hierbei um das tatsächliche Programm handelt, ist ansonsten jedoch nur eine Containerkomponente für "Screens".

3.3.2 Screens

Diese Komponente kommuniziert über die Remote-Interfaces mit "GameObjects" und so mit dem Server. Die Komponente fragt alle, für eine grafisch korrekte Darstellung der Spiellogik, relevanten Informationen ab und leitet Spieler-Eingaben weiter. Für die Darstellung greift die Komponente auf "Assets" zu, was jedoch in der entsprechenden Komponente erläutert wird. Außerdem werden hier lokale Einstellungen wie Ton-Lautstärke oder Auflösung verwaltet. Dazu weiteres in "ConfigurationData". Die tatsächliche Darstellung im Fenster wird vom LibGdx-Framework übernommen.

3.3.3 Assets

Hierbei handelt es sich nicht um implementierte Teile des System sondern grafische Dateien, welche auf dem Computer des Clients liegen. Diese Grafiken stellen die verschiedenen Objekte eines Spiels dar(z.B. Einheiten) und werden von "Screens" geladen

und unter Verwendung des LibGdx-Framework dargestellt. Da diese Darstellung nur für den Client relevant ist gibt es keine direkte Beziehung zu "GameObjects", da dort lediglich Zugehörigkeiten gespeichert werden, jedoch keine Verwendung statt findet.

3.3.4 ConfigurationData

Diese Komponente beschreibt ebenfalls nur Dateien und keinen implementierten Teil des System. Die Dateien enthalten Client-relevante Informationen, welche in erster Linie von den Präferenzen oder Bedürfnissen des jeweiligen Spielers abhängen. Darunter fallen vor allem Einstellungen bezüglich Darstellung und Ton, sowie ggf. Steuerung. Dabei ist jedoch an zu merken dass die Einstellungen pro Computer und nicht pro Spieler gespeichert werden. Die Informationen werden dann von der Screens-Komponente verwendet um ihre Aufgaben präziser zu erfüllen. Das Speichern und Laden der Informationen wird dabei vom LibGdx-Framework umgesetzt.

3.3.5 Window

Die Window Komponente beschreibt keine persistenten Elemente des Systems sondern lediglich das am Ende dargestellte Fenster, welches die Schnittstelle zum Spieler realisiert. Die einzelnen Bilder(Frames) werden dabei von den Screens definiert und durch das LibGdx-Framework an das System weiter gegeben, wo dann die endgültige Darstellung des Fensters statt findet.

3.4 Bezug zu den Strategien

Hier wird die Verbindung zwischen den Komponenten und den Strategien erläutert.

Die Strategie zum Verwenden vom "RMI" findet sich in den Komponenten "GameObjects" und "Screens" wieder, speziell ihrer Kommunikation, welche die in der RMI-Strategie beschriebenen Remote-Interfaces verwendet..

Die Mindestanforderungen werden hauptsächlich im Server umgesetzt und der Zugriff dann über den Client ermöglicht.

Die Strategie "Don't trust the Client" findet sich in den Inhalten des Server und Client Komponenten, beschrieben in ihren Aufgaben. Verdeutlicht durch die Screens-Komponente, welche die einzige implementierte Komponente in "Client" darstellt und entsprechend der Strategie nur Darstellung jedoch keine direkte Logik umsetzt.

Im Kommentar zu Anfang des Kapitels ist die Verbindung zu "Einzelnes Programm" zu finden.

Die in "Aufteilung nur Modul-intern" beschriebenen Bereiche entsprechenden den hier erläuterten (elementaren) Komponenten. Eine Aufteilung von zu implementierten Teilen findet also nur innerhalb dieser Komponenten statt.

4 Modulsicht

Die Modulsicht wird im Groben durch das folgende Paketdiagramm dargestellt, dass Ähnlichkeiten zur Konzeptionellen Sicht aufweist. Da wir im Verlauf dieses Kapitels den Inhalt dieser Pakete mit seinen Ports erläutern und diesen beschreibend auf die konzeptionelle Sicht abbilden, nutzen wir Klassendiagramme mit einer Erweiterung durch Ports. Spezifisch zum Aufbau erkennt man hierbei eine Trennung der Assets und Config-Datei vom Client-Paket, wodurch das Data-Paket entstanden ist. Es gibt also vier große Pakete: Der eben erwähnte Client und Data. Derby, dass die Datenbank enthält. Server, mit Spiellogik und der Kommunikation des Servers mit der Datenbank sowie dem Client. Jegliche im folgenden Kapitel genannten Interfaces erweitern die Klasse "RemoteInterface" aus der RMI API, um eine Verwendung über RMI zu ermöglichen.

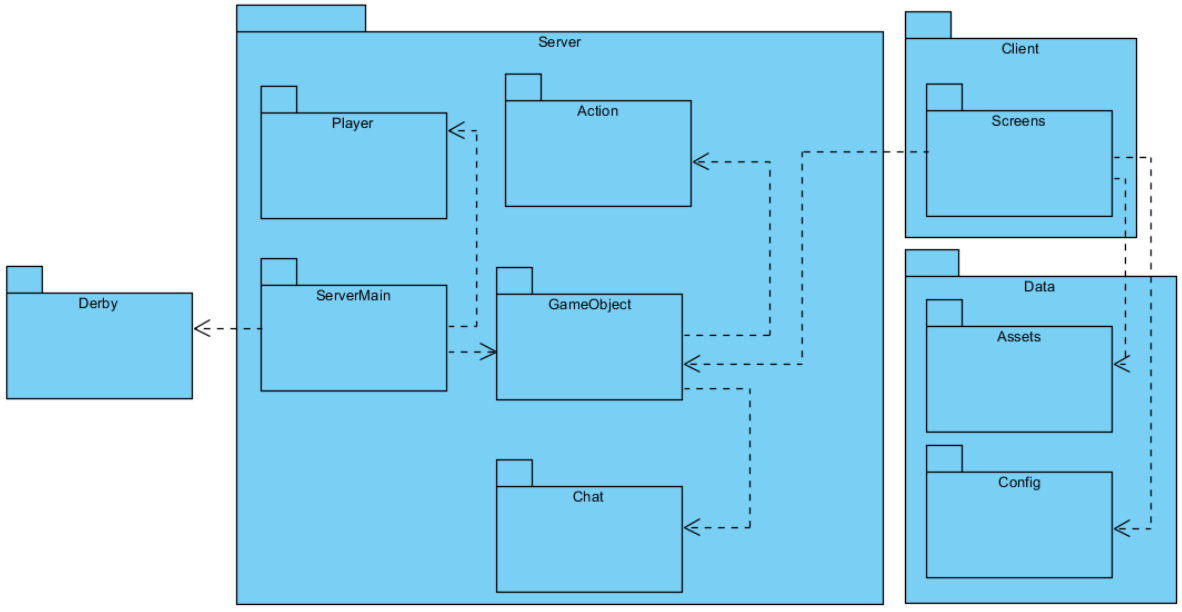

Abbildung 2: Ein Paketdiagramm das eine grobe Modulsicht und ihre Importbeziehungen zeigt.

4.1 Server-Modul

Das Server-Modul beinhaltet die Server-Klasse sowie den DBManager. Es stellt also eine Überschneidung der DBController-Komponente und GameLogic-Komponente dar. Seine Aufgabe ist zum Einen die Kommunikation mit der Datenbank und zum Anderen den initialen Zugriff auf den Server-Computer zu ermöglichen.

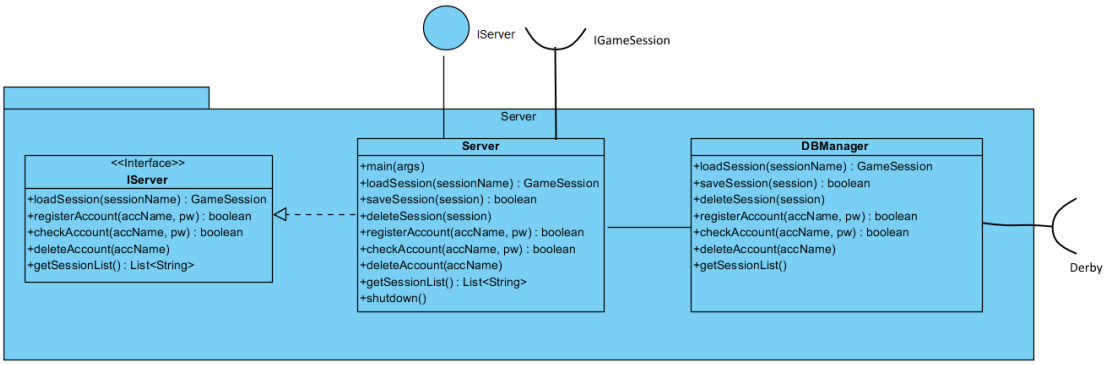

Abbildung 3: Das Server-Package als Klassendiagramm

Die Klasse Server besitzt eine main-Methode, da sie nicht nur den initialen Zugriff auf den Server ermöglicht sondern auch das zugrunde liegende Programm initialisiert. Beim initialisieren erstellt die Klasse eine neue Registry auf dem Computer und registriert sich selbst um so den Zugriff über RMI zu ermöglichen, ist bereits eine Registry vorhanden wird diese verwendet. Die Registry wird in einem statischen Attribut ermöglicht, wodurch der Server nicht automatisch beendet wird, da die Registry nicht vom Garbage-Collector eingesammelt wird. Ansonsten ist vor allem die getSessionList()-Methode interessant, diese liefert eine Auswahl von allen verfügbaren Spielen und dient so als Ausgangspunkt zum Laden eines Spiels.

Das Interface IServer besitzt alle für den Client relevanten Methoden und ist die für RMI eigentlich verwendete Komponente, also die initiale Schnittstelle nach außen.

Die Klasse DBManager besitzt nur eine erhaltende Schnittstelle, welche von Derby bereit gestellt wird. Ansonsten besteht nur eine Verbindung zu Server, da Server den DBManager für die Kommunikation mit der Datenbank verwendet.

4.2 Chat

Das Chat-Modul besitzt die Aufgabe jegliche Nachrichtensysteme um zu setzen, bisher umfasst dies nur die Klassen Chat und Message. Es ist eines der drei Module welche zusammen die GameObject-Komponente darstellen, die anderen beiden sind Player und GameObject.

Die Message-Klasse beschreibt eine einzelne Nachricht innerhalb eines Chats. Dies wurde als eigene Klasse realisiert, um zusätzliche Funktionen wie das ausschließen von Spielern

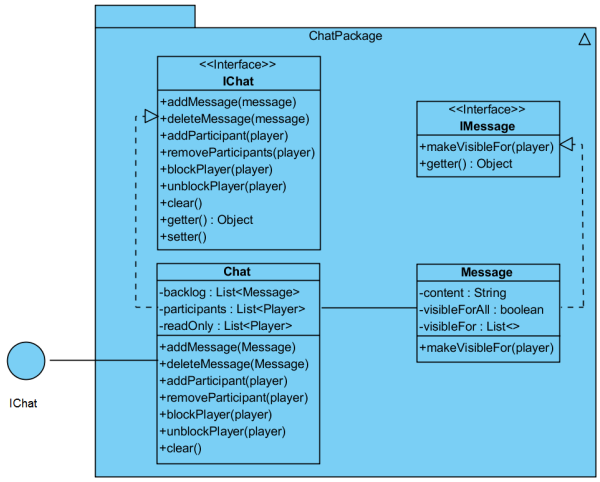

Abbildung 4: Das Chat-Modul als Klassendiagramm

zu vereinfachen, weitere Funktionen sind bisher nicht vorhanden.

Der Kern des Chat-Moduls ist jedoch die Chat Klasse, welche eine Sammlung von Nachrichten ist und Informationen darüber besitzt, wer am Chat mit welchen Rechten teil nimmt.

Beide Klassen besitzen Interfaces um ihre Verwendung auch vom Client aus zu ermöglichen. Die Interfaces besitzen dabei auch die getter-Methoden um sie dem Client zur Verfügung zu stellen. Da diese Methoden jedoch trivial sind und sich durch die Attribute in der implementierenden Klasse definieren wurden sie unter dem Punkt getter zusammen gefasst mit Object als Rückgabetyp.

Obwohl die Klassen durch ihre Interfaces direkt vom Client zugegriffen werden könnten ist die Schnittstelle hauptsächlich für GameSession aus GameObjects als direkten Zugriff gedacht und stellt für den Client lediglich die Funktionalität zur Verfügung. Weiteres zum Zugriff in GameObjects.

4.3 GameObject

Das GameObject-Modul ist der größte Teil aus der GameObjects-Komponente und gewissermaßen ihr Kern. Es beinhaltet jegliche Klassen, welche am Ende in einer Spielsession dargestellt und verwendet werden, sowie das Objekt für die Spielsession selbst(GameSession), ausgenommen davon sind die Chat-Objekte, da das Nachrichtensystem als eigenes Moduls betrachtet wird und jegliche Spieler-bezogenen Objekte, welche nicht direkt (grafisch) dargestellt werden, diese befinden sich im Player-Modul. Die Aufgabe des Moduls ist es dem Client weiteren Zugriff, nach dem initialen Zugriff zu ermöglichen und seine Anfragen mit den Spiel-Objekten, sowie der Spielelogik zu ver-

binden.

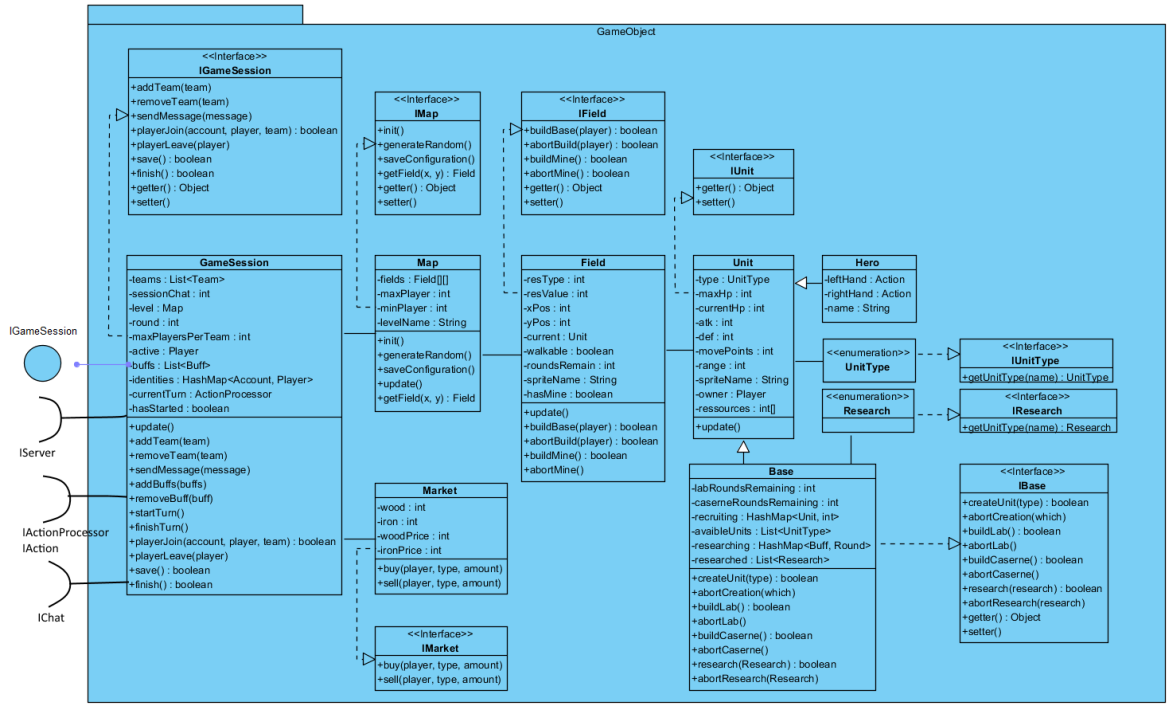

Abbildung 5: Das GameObjects-Modul als Klassendiagramm

Jegliche Klassen in diesem Modul besitzen Interfaces um ihre Funktionalität dem Client zur Verfügung zu stellen, dabei beinhalten sie nur die Methoden welche auch für den Client relevant sind. Updates werden z.B. von GameSession übernommen, weshalb entsprechende Methoden nicht für den Client zugänglich sind. Dafür besitzen sie getter-Methoden, welche auch hier allgemein abgekürzt wurden, mit Object als Rückgabewert. Wieder da sie trivial sind und sich durch die Attribute in der implementierenden Klasse definieren. Die Objekte dieses Moduls realisieren die meisten der Mindestanforderungen, ausgenommen sind dabei vor allem der Datenbank-Teil sowie die Server-Client Struktur.

Auch die beiden Enumerations, UnitType und Research besitzen Interfaces, welche lediglich eine get-Methode besitzen. Diese ermöglicht dem Client den gewünschten Enumwert zu einem String zu erhalten.

Obwohl die Interfaces, die Verwendung aller Objekte durch den Client ermöglichen und so Teil der Schnittstelle zwischen der Screens und GameObjects-Komponente sind, dienen sie nur zur Bereitstellung der Funktionalität, der Zugriff wird über GameSession geregelt.

Daher stellt GameSession den Kern diese Moduls dar. Da sie alle Objekte einer Spielsession besitzt, greift der Client auf die GameSession zu und erhält so auch Zugriff auf die anderen Objekte. Zusammen stellen die Interfaces also die Schnittstelle zwischen den Komponenten Screens und GameObjects dar. Neben der Verbindung zu Screens, verwendet GameSession auch Server aus dem Server-Modul und Action aus dem Action-

Modul, welche sich beide in der GameLogic-Komponente befinden. Die Verwendung zwischen Server und GameSession ist dabei beidseitig. Somit stellt GameSession also beide Schnittstellen der GameObject-Komponente dar, zum einen die zu Screens, zum anderen jedoch auch die zwischen GameObjects und GameLogic.

4.4 Action

Das Action-Modul beinhaltet vor allem Logik die auf die Interaktion zwischen Spielobjekten bezogen ist. Das umfasst die Action-Klassen, sowie den ActionProcessor. Der Zugriff erfolgt auch hier über die entsprechenden Interfaces. Die Klasse ist Teil der GameLogic-Komponente.

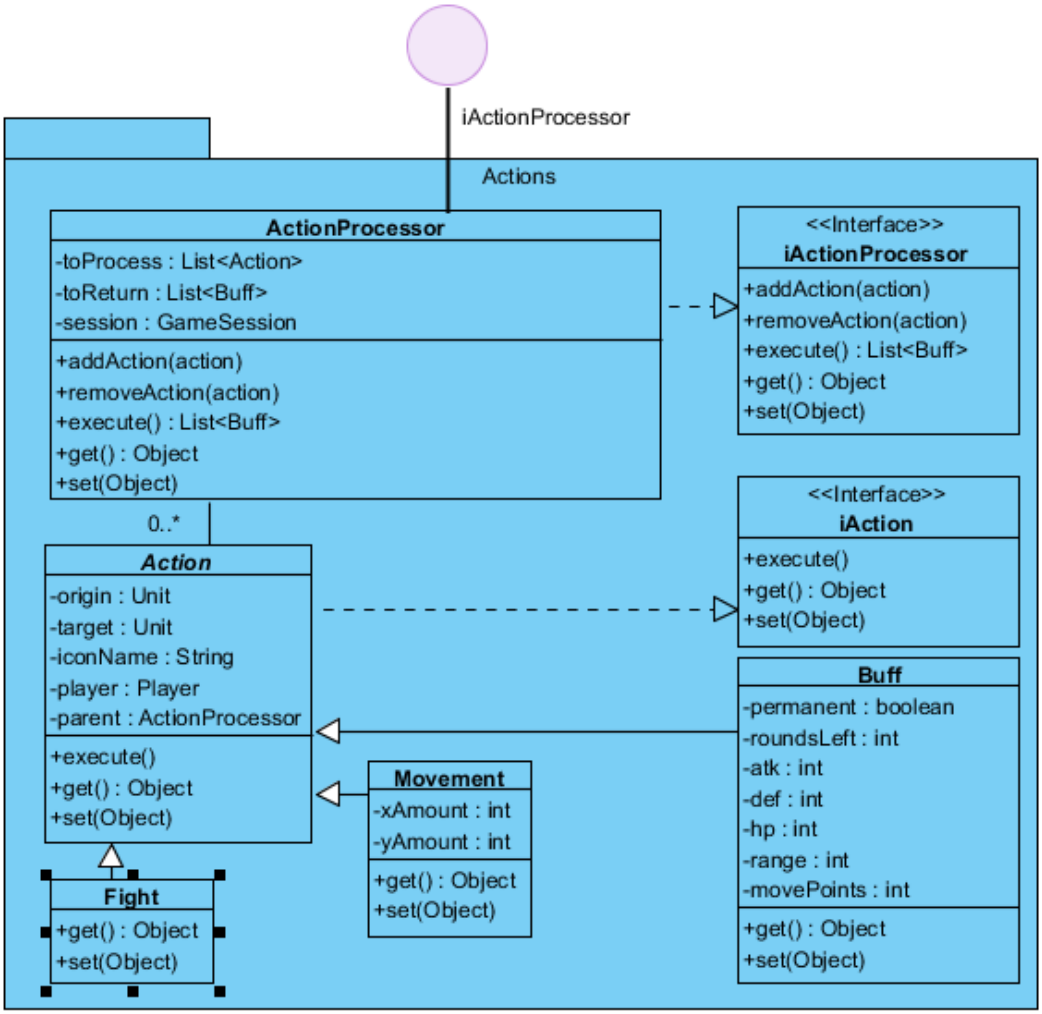

Abbildung 6: Das Action-Modul als Klassendiagramm

Der ActionProcessor ist die Schnittstelle des Moduls nach außen. Er besitzt Listen mit den zu verarbeitenden Objekten und Buff-Objekten, welche möglicherweise beim Verarbeiten der Actions generiert wurden. Der ActionProcessor wird dabei von GameSession für je einen Zug verwendet. Beendet ein Spieler seinen Zug wird der ActionProcessor angewiesen alle ihm zugeordneten Actions zu verarbeiten, dafür dient die execute-Methode. Sollten bei diesem Prozess Buffs entstehen werden diese am Ende als Liste zurück gegeben um in GameSession gespeichert werden zu können.

Die zweite besonders wichtige Klasse ist Action. Diese Klasse beschreibt eine ausführbare Aktion, welche in der Regel zwei Teilnehmer besitzt, wobei es auch nur einen Teilnehmer geben kann. Die Action kann über ihre execute-Methode ausgeführt werden, wobei der Inhalt dieser Methoden von der speziellen Implementierung ab hängt. Beispiele dafür sind die bereits definierten Actions Movement und Fight, welche eine Bewegung und einen Kampf realisieren, bisher sind keine weiteren Actions geplant.

Zuletzt ist noch die Buff-Klasse zu sehen. Dies ist eine spezielle Action, welche nach ihrem Ausführen nicht sofort verworfen wird und sogar permanent aktiv sein kann. Sie besitzt Attribute ähnlich zu denen von Einheiten, die Werte dieser Attribute bestimmen wie die Werte der Einheiten manipuliert werden. Dazu mehr im nächsten Kapitel.

4.5 Player

Das Player-Modul ist das letzte Modul der GameObjects-Komponente. Es umfasst alle Klassen, welche direkt Spieler bezogen sind und somit nur indirekt auf dem Spielfeld dargestellt werden(z.B. Farbe von Einheiten). Auch hier erfolgt der Zugriff von außen über die Interfaces.

Die Schnittstelle nach außen stellt hier die Teamklasse dar, da eine GameSession zwar die teilnehmenden Spieler kennt, das Spiel an sich aber über die Teams verwaltet wird. Sollte es nur zwei Spieler geben, gibt es dementsprechend zwei Teams mit je einem Spieler. Ein Team besitzt eine Liste mit allen Mitgliedern, einen Chat und eine Teamfarbe. Außerdem wird auch die in den Mindestanforderungen geforderte Teamkasse hier umgesetzt. Genaueres dazu findet sich in der Datensicht.

Neben dem Team beinhaltet das Modul auch die Playerklasse selbst. Diese stellt einen Spieler innerhalb einer Spielsitzung dar. Es ist also das Referenzobjekt wenn geprüft wird zu wem eine Einheit gehört. Dafür besitzt der Spieler außerdem das TechnologyTree-Objekt, welches darstellt welche Technologien der Spieler bereits erreicht hat. Die Technologien beziehen sich dabei auf Aufwertungen die den Spieler direkt betreffen, Forschungen bezüglich Einheiten werden in Basen, genauer dem Laboratorium verwaltet.

Zuletzt beinhaltet das Modul das Account-Objekt. Dieses Identifiziert einen Spieler auch über eine Spielsitzung hinaus und wird verwendet um z.B. beim Laden eines Spiel die Player korrekt den Teilnehmern zu zu ordnen. Bisher beinhaltet diese Klasse keine weiteren Informationen, es wäre jedoch denkbar die Klasse um zusätzliche Statistiken wie z.B. Sieg-Niederlage Verhältnis zu erweitern. Um diese Möglichkeit nicht aus zu schlie-

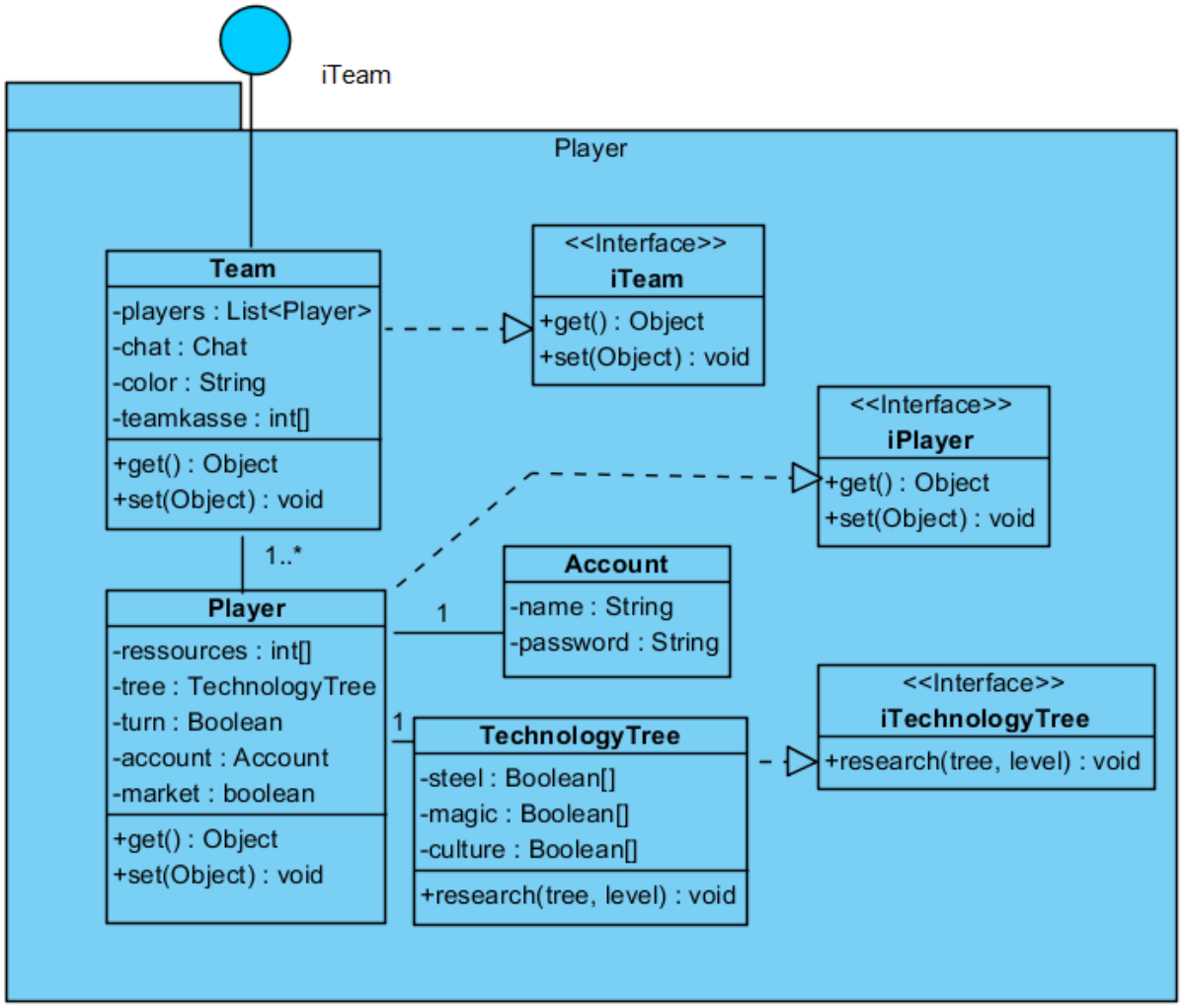

Abbildung 7: Das Action-Modul als Klassendiagramm

ßen wurde der Account als Objekt umgesetzt, ansonsten hätte man auch direkt den Accountnamen in Player speichern können.

4.6 Screens

Zuletzt haben wir noch das Screens-Modul, welches die Screens-Komponente beschreibt. Alle Klassen in diesem Modul sind Unterklassen von AbstractGameScreen und dienen zur Darstellung verschiedener Bildschirme. Dieses Modul besitzt keine Interfaces, da es nur auf andere Objekte zu greift, selbst aber nicht zugegriffen wird, da die Darstellung die höchste Instanz bezüglich der Aufrufhierarchie des Programms ist.

Der Kern des Moduls ist wie bereits genannt die Klasse AbstractGameScreen. Sie beschreibt welche Methoden und Attribute von allen Screens implementiert werden müssen, diese Ergeben sich durch das libGdx-Framework, wobei es sich dabei Teilweise nur um

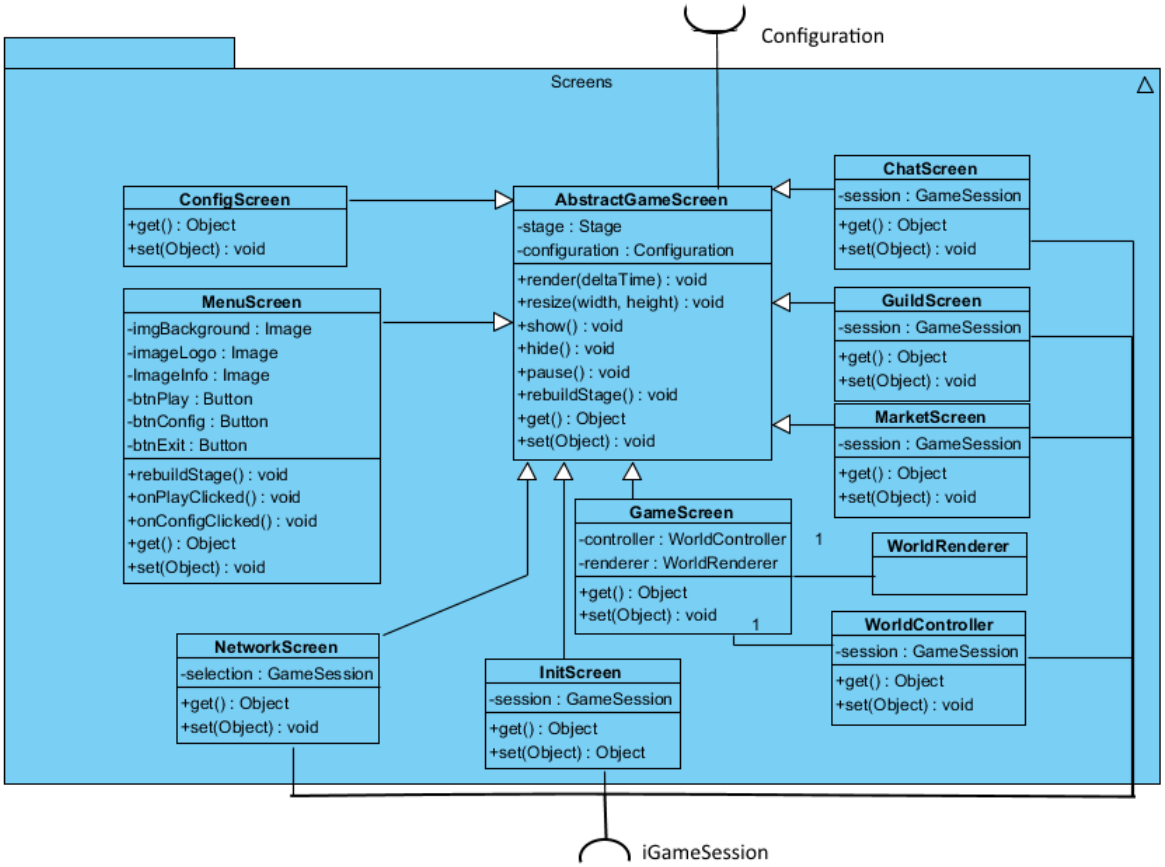

Abbildung 8: Das Screens-Modul als Klassendiagramm

Konventionen handelt. Die Methoden und Attribute befassen sich dabei hauptsächlich mit der Darstellung der Bildschirme.

Die anderen Klassen implementieren diese Methoden dann lediglich mit den für den jeweiligen Bildschirm relevanten Informationen, weshalb sie hier nicht weiter erläutert werden, stattdessen folgt eine kurze Beschreibung der Screens und eine Auflistung der benötigten Funktionen:

- **ConfigScreen:** In diesem Bildschirm können die Werte für das Configuration Objekt eingestellt werden.
  Benötigte Funktionen:
  - **Einstellen der Auflösung**
  - **Einstellen der Lautstärke**
  - **Musik deaktivieren/aktivieren**

  - **Zu MenuScreen navigieren**

- **MenuScreen:**Dieser Bildschirm stellt das Hauptmenü dar und ist der initiale Bildschirm nach dem Start.
  Benötigte Funktionen:
  - **Wechseln zu "NetworkScreen"**
  - **Wechseln zu "ConfigScreen"**
  - **Beenden des Spiels**
  - **Registrieren von neuem Account**
- **NetworkScreen:** Der NetworkScreen liefert eine Liste mit allen verfügbaren Spielen, sowie die Möglichkeit ein neues Spiel zu erstellen.
  Benötigte Funktionen:
  - **Einloggen in Account.**
  - **Wechseln zu "MenuScreen".**
  - **Auswählen eines existenten Spiels zum Beitreten bzw. Laden.** Führt zu "InitScreen", wobei das ausgewählte Spiel übergeben wird.
  - **Erstellen eines neuen Spiels.** Führt zu "InitScreen", ohne das ein Spiel übergeben wird.
- **InitScreen:** Dieser Screen liefert einen Überblick über die Einstellungen des Spiels und ermöglicht es sie zu bearbeiten sofern das Spiel neu erstellt wird.
  Benötigte Funktionen:
  - **Zurückkehren zu "NetworkScreen".**
  - **Auswählen eines Teams.**
  - **Minimale und maximale (pro Team) Spieleranzahl angeben.** Host exklusiv.
  - **Auswählen der Spielmap.** Host exklusiv.
  - **Spiel starten.** Host exklusiv.
- **GameScreen:** Hier wird das laufende Spiel dargestellt. Um eine einwandfreie Darstellung zu gewährleisten, sollen hier die Hilfsklassen "WorldController" und "WorldRenderer" verwendet werden, daher werden diese als Attribute gespeichert.
  Benötigte Funktionen:
  - **Alle in den Mindestanforderungen genannten Möglichkeiten eines Spielers innerhalb eines laufenden Spiels.** Details: Siehe Mindestanforderungen.
- **MarketScreen:** In diesem Bildschirm wird der Marktplatz eines Spiels dargestellt.
  Benötigte Funktionen:
  - **Kaufen/Verkaufen von Holz.**

  - **Kaufen/Verkaufen von Eisen.**
  - **Zurückkehren zum GameScreen.**
- **GuildScreen:** Hier werden Teamrelevante Informationen angezeigt und Funktionen zur Verfügung gestellt.
  Benötigte Funktionen:
  - **Teamnachrichten schreiben und lesen.**
  - **In die Teamkasse einzahlen und Kassenstand betrachten.**
  - **Zurückkehren zum GameScreen.**
- **ChatScreen:** Hier erhält der Spieler Zugriff auf seine Chatunterhaltungen.
  Benötigte Funktionen:
  - **Zurückkehren zum vorherigen Screen.**
  - **Nachrichten in einem Chat schreiben und lesen.**

Die eingehende Schnittstelle von IGameSession verweist zwar auf InitScreen, jedoch ist dies lediglich die initiale Verbindung, das GameSession-Objekt wird dann an die anderen Screens weiter gegeben und dort ebenfalls durch das Remote-Interface von GameSession weiter verwendet.

Die Configuration und Assets werden dabei direkt aus den jeweiligen Dateien geladen, wobei libGdx zur Hilfe verwendet wird.

5 Datensicht

Für eine bessere Übersicht wurden die Interfaces in diesem Diagramm weg gelassen, da sie ohnehin nur für die Verwendung von RMI benötigt werden.

5.1 Client

5.1.1 MainGame

Diese Klasse dient lediglich zum Initialisieren des Clients und besitzt daher nur die allgemein bekannte Methode "void main(String args[])". In dieser Methode wird dann die Verbindung zum Server aufgebaut und der erste Bildschirm(MenuScreen) initialisiert.

5.1.2 Configuration

Diese Objekt dient zum Speichern und Laden von lokalen Einstellungen wie z.B. Auflösung oder Lautstärke von Ton. Die Einstellungen werden durch das Framework automatisch auf dem Computer gespeichert und geladen.

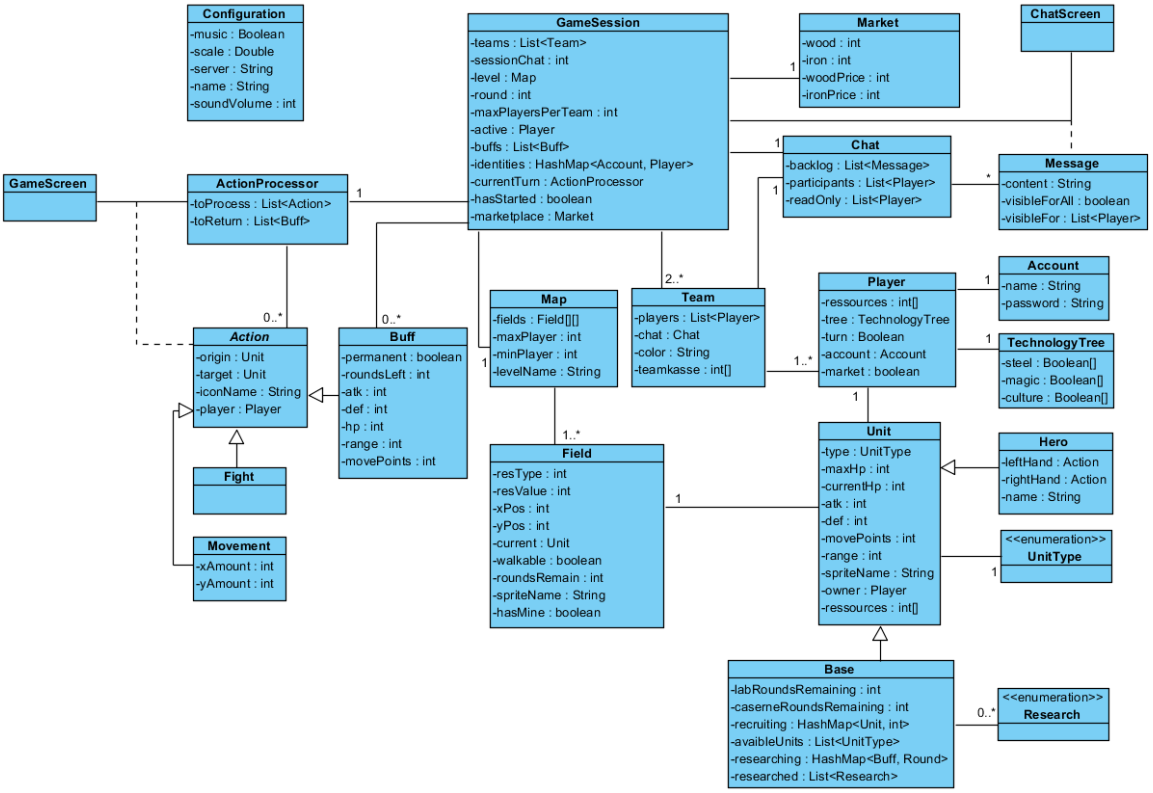

Abbildung 9: Die Datenklassen von Gold and Greed

Attribute:

- **music:** Bestimmt ob Musik gespielt werden soll oder nicht.
- **scale:** Bestimmt die Skalierung aller dargestellten Elemente.
- **server:** Beinhaltet eine Liste mit allen zur Verfügung stehenden Servern.
- **soundVolume:** Gibt die Lautstärke aller Töne an.

5.2 Server

5.2.1 GameSession

Diese Klasse realisiert das zentralste Objekt des Programms, eine Spielesitzung. Sie dient dazu alle in einem Spiel existenten Objekt zu verwalten und ihren Zugriff zu ermöglichen. Hier wird außerdem bestimmt wann welche Informationen gespeichert werden und die nicht geteilten Spielobjekte verwaltet.

Attribute:

- **teams:** Eine Liste mit allen Teams des Spiels.

- **sessionChat:** Der globale Chat des Spiels.
- **level:** Die Karte auf der gespielt wird.
- **round:** Die Nummer der aktuellen Runde.
- **maxPlayersPerTeam:** Gibt an wie viele Spieler in einem Team maximal sein dürfen.
- **active:** Der Spieler, welcher am Zug ist.
- **buffs:** Eine Liste mit allen im Spiel aktiven Buffs, Zuordnung findet zu Spielobjekten findet innerhalb von "Buff" statt.
- **identities:** Eine HashMap, welche jedem Account einen Spieler zu ordnet. Fürs Laden eines Spiels relevant.
- **currentTurn:** Ein ActionProcessor, welcher alle Aktionen des aktuellen Zugs verwaltet.
- **hasStarted:** Gibt an ob das Spiel noch Konfiguriert wird oder schon gestartet ist.

5.2.2 ActionProcessor

In dieser Klasse werden die in einem Zug gegebenen Befehle verarbeitet und umgesetzt.

Attribute:

- **toProcess:** Eine Liste mit allen Befehlen(Actions) die in dem aktuellen Zug gegeben werden.
- **toReturn:** Eine Liste mit Buffs die beim Verarbeiten der Befehle generiert wurden. Wird später an GameSession zurück gegeben.
- **session:** Eine Referenz auf die GameSession, welche dem ActionProcessor gehört, als Zugriffsmöglichkeit.

5.2.3 Action

Eine abstrakte Klasse für alle Arten von Aktionen. Sie kann von GameScreen erstellt und an den ActionProcessor übergeben werden, ein solcher Prozess findet z.B. bei Usereingaben statt.

Attribute:

- **origin:** Die Einheit von der die Aktion ausgeht bzw. auf der sie ausgeführt. Z.B. die zu bewegende Einheit.
- **target:** Die Zieleinheit, sofern es eine gibt. Z.B. die angegriffene Einheit.
- **iconName:** Gibt den Namen des Iconsprites zur Darstellung auf dem Client an.
- **player:** Der Spieler für den die Aktion gilt. Relevant sofern es direkte Auswirkungen auf einen Spieler und nicht auf seine Einheiten hat.

- **parent:** Der ActionProcessor, der diese Action verwaltet, als Zugriffspunkt. Bei Buffs null.

5.2.4 Fight

Eine Implementation einer Action zum Umsetzen eines Kampfes zwischen "Origin" und "Target".

Attribute: Keine speziellen.

5.2.5 Movement

Eine Implementation einer Action zum Bewegen von Einheiten("Origin").

Attribute:

- **xAmount:** Gibt an wie viele Felder in X-Richtung gegangen werden sollen. Orientierung wird durch positive und negative Werte realisiert.
- **yAmount:** Gibt an wie viele Felder in Y-Richtung gegangen werden sollen. Orientierung wird durch positive und negative Werte realisiert.

5.2.6 Buff

Ein Buff ist eine spezielle Form einer Action, mit der Eigenschaft dass sie nicht nur einmal ausgeführt wird sondern mehrfach. Ihre Aktion besteht aus dem Manipulieren von Einheiten Werten.

Attribute:

- **permanent:** Gibt an ob es sich bei dem Buff um eine dauerhafte Verbesserung handelt oder eine Temporäre.
- **roundsLeft:** Gibt an wie viele Runden der Buff noch aktiv bleibt. Falls "permanent" **true** ist, ist dieser Wert -1. Ist dieser Wert 0, löscht sich der Buff selbst beim nächsten Update(execute).
- **atk:** Dieser Wert wird auf den gleichnamigen Wert der "target" Einheit einmalig addiert. Löscht sich der Buff wird der Wert wieder subtrahiert. Der Wert darf negativ sein und so Minderungen darstellen.
- **def:** Analog zu atk.
- **hp:** Analog zu atk. Bezieht sich auf "maxHp"
- **range:** Analog zu atk.
- **movePoints:** Analog zu atk.

5.2.7 Market

Über das Marktobjekt können Spieler handeln, wobei man nicht direkt mit anderen Spielern handeln kann.

Attribute:

- **wood:** Die Menge an Holz die auf dem Markt verfügbar ist. Erhöht sich wenn Holz verkauft wird.
- **iron:** Analog zu "wood" für Eisen.
- **woodPrice:** Gibt an wie viel Gold man für das Verkaufen von Holz erhält oder für das Kaufen verliert. Preis ist pro 1 Holz angegeben und wird bei Aufruf von "buy" oder "sell" neu generiert.
- **ironPrice:** Analog zu "woodPrice" für Eisen.

5.2.8 Chat

Ein Chat ermöglicht das Kommunizieren zwischen Spieler über Textnachrichten.

Attribute:

- **backlog:** Eine Liste mit allen bisher gesendeten Nachrichten.
- **participants:** Alle Spieler die an dem Chat teilnehmen und somit Lesen und Schreiben können.
- **readOnly:** Eine Liste mit allen Spielern die zwar die Nachrichten lesen dürfen, aber keine Schreiben(z.B. weil sie andere beleidigen).

5.2.9 Message

Eine Message ist eine Chatnachricht, welche jedoch Information über ihre Sichtbarkeit besitzt. Sie kann von ChatScreen erstellt und an GameSession übergeben werden.

Attribute:

- **content:** Der Inhalt der Nachricht, welcher im Chat angezeigt wird, als Zeichenkette.
- **visibleForAll:** Gibt an ob jeder diese Nachricht lesen kann.
- **visibleFor:** Falls "visibleForAll" **false** ist, werden in dieser Liste alle Spieler gespeichert, welche die Nachricht lesen dürfen.

5.2.10 Team:

Ein Team ist eine verbündete Menge von Spielern in einer Sitzung. Damit ein Spiel gewonnen werden kann ist es ausreichend dass alle übrigen Spieler dem gleichen Team

angehören, es muss also keinen einzelnen Sieger geben.

Attribute:

- **players:** Eine Liste mit allen Mitgliedern des Teams.
- **chat:** Ein Chat der nur für das Team sichtbar ist.
- **color:** Die Teamfarbe, Haupterkennungsmerkmal für Teams.
- **teamkasse:** Die gemeinsame Kasse des Teams. Verliert ein Teammitglied eine Einheit erhält er einen Teil der Kosten aus dieser Kasse zurück, sofern genug vorhanden sind. Die Werte werden dabei direkt gespeichert und über ihren Index den Ressourcen zugeordnet(0=Holz,1=Eisen,2=Gold). Mana kann nicht zurück gewonnen werden.

5.2.11 Player

Ein Player ist die Repräsentation eines Accounts innerhalb einer Spielsitzung, also des agierenden Spielers.

Attribute:

- **ressources:** Die Ressourcen über die der Spieler verfügt. Die Menge wird direkt als Wert gespeichert und der Typ über den Index identifiziert(0=Holz,1=Eisen,2=Gold,3=Mana).
- **tree:** Der TechnologyTree des Spielers, welcher angibt welche Technologien ein Spieler bereits entwickelt hat.
- **turn:** Gibt an ob der Spieler am Zug ist.
- **account:** Der zum Spieler gehörende Account.
- **market:** Gibt an ob der Spieler bereits Zugriff auf den Marktplatz hat.

5.2.12 Account

Die eindeutige Identifikation eines Spielers, auch über Spielsitzungen hinaus. Kann eventuell mit optionalen Werten erweitert werden(z.B. Statistiken über Sieg-Niederlage Verhältnis).

Attribute:

- **name:** Der Name des Accounts.
- **password:** Das Passwort des Accounts, genauer der Vergleichswert. Wird bei genug übriger Zeit des Projektes zu einem Hash geändert umso die Sicherheit zu erhöhen.

5.2.13 TechnologyTree

Ein Technologie-Baum gibt an welche Technologien ein Spieler entwickelt hat.

Attribute:

- **steel:** Realisiert den Zweig für Holz und Eisen Erweiterungen. Jede Erweiterung wird durch einen Boolean wert dargestellt.
- **magic:** Analog zu "steel" für Mana-bezogene Erweiterungen.
- **culture:** Analog zu "steel" für Gold-bezogene Erweiterungen. Hier kann auch der Zugriff auf den Marktplatz freigeschaltet werden.

5.2.14 Map

Eine Map stellt ein Level bzw. eine Karte für eine Spielsitzung da. Hier werden auch alle Felder der Karte gespeichert.

Attribute:

- **fields:** Ein X*Y großes Array, welches die einzelnen Spielfelder beinhaltet.
- **maxPlayer:** Die maximale Spieler Anzahl für diese Karte.
- **minPayer:** Die minimale Spieler Anzahl für diese Karte.
- **levelName:** Der Name der Karte.

5.2.15 Field

Ein Feld ist ein einzelnes Element einer Karte. Es kann eine Einheit oder eine Ressource besitzen, in diesem Fall kann das Feld nicht von anderen Einheiten überquert werden.

Attribute:

- **resType:** Gibt an welche Ressource das Feld besitzt(-1=Keine,0=Holz,1=Eisen,2=Mana).
- **resValue:** Gibt an wie viel von der Ressource vorhanden ist, falls es eine gibt, ansonsten -1.
- **xPos:** Die X-Position des Feldes auf der Karte.
- **yPos:** Die Y-Position des Feldes auf der Karte.
- **current:** Die Einheit die sich aktuell auf dem Feld befindet oder **null**.
- **walkable:** Gibt an ob das Feld aktuell überquert werden kann.
- **roundsRemain:** Gibt an wie viele Runden noch fehlen bis der Bau einer Basis abgeschlossen ist. Ist -1 falls Bau noch nicht begonnen hat.
- **spriteName:** Der Name des zugehörigen Sprites.

- **hasMine:** Gibt an ob das Feld eine Mine hat, nur möglich wenn die Ressource Eisen ist.
- **roundsRemainMine:** Analog zu roundsRemain, für die Mine.

Methoden:

- **update:** Aktualisiert alle Werte die nicht direkt bearbeitet werden(z.B. roundsRemain).
- **buildBase:** Lässt den übergebenen Spieler den Bau einer Basis starten. Rückgabewert gibt an ob der Vorgang möglich ist oder nicht. Eine Baueinheit muss sich in der Nähe befinden damit der Bau fortgeführt wird.
- **abortBuild:** Lässt den übergebenen Spieler den Bau einer Basis abbrechen. Rückgabewert gibt an ob der Spieler den Vorgang ausführen darf bzw. ob der Vorgang erfolgreich war. Falls es keinen Bau gab ist Rückgabewert **true**. Wird der Bau erfolgreich abgebrochen, erhält der Spieler einen Teil der zum Bau benötigten Ressourcen zurück, abhängig vom Baufortschritt(umso mehr Ressourcen verbraucht sind desto weniger erhält der Spieler zurück).
- **buildMine:** Startet den Bau einer Mine. Rückgabewert gibt an ob der Vorgang erfolgreich war. Bau kann nicht abgebrochen werden, alle Baueinheiten in der Nähe werden beachtet unabhängig von ihrer Zugehörigkeit. Eine fertige Mine kann von allen Spielern verwendet werden.

5.2.16 Unit

Eine Einheit(Unit) stellt alle von Spielern kontrollierbaren Spielelemente da.

Attribute:

- **type:** Gibt an um welchen Einheitentyp es sich handelt.
- **maxHp:** Gibt an wie viele Lebenspunkte die Einheit maximal haben kann, also wie viel Schaden sie nehmen kann bevor sie stirbt.
- **currentHp:** Gibt den aktuellen Wert der Lebenspunkte an. Fällt dieser Wert auf 0 oder weniger stirbt die Einheit.
- **atk:** Gibt die Angriffskraft an, d.h. wie viel Schaden die Einheit mit einem Angriff maximal verursachen kann. Wird bei einem Kampf durch einen zufälligen Wert erweitert, welcher prozentual Abhängig ist und die Verteidigung des Gegners ignoriert.
- **def:** Gibt die Verteidigung der Einheit an, d.h. wie viel Schaden absorbiert wird, wenn die Einheit angegriffen wird.
- **movePoints:** Gibt an wie viele Felder die Einheit noch gehen kann. Wegfindung wird nicht beachtet, d.h. ein Gehen ist erlaubt wenn die Entfernung zum Zielfeld die movePoints nicht überschreitet und es einen beliebigen unblockierten Weg gibt, dessen Länge jedoch die movePoints überschreiten darf.

- **range:** Gibt an wie viele Felder eine andere Einheit entfernt sein darf und trotzdem noch ohne zusätzliche Bewegung angegriffen werden kann.
- **spriteName:** Der Name des zugehörigen Sprites.
- **owner:** Der Spieler zu dem die Einheit gehört.
- **ressources:** Die Anzahl der Ressourcen die eine Einheit bei sich trägt, verhält sich von der Verwaltung äquivalent zu "ressources" in Player.

5.2.17 UnitType

Eine enumeration welche alle Arten von Einheiten beinhaltet, für eine einfache und Performanz-günstige Identifikation. Hier werden auch die konkreten Attributs Werte für die Einheitentypen bestimmt.

5.2.18 Hero

Der Held ist eine spezielle Einheit, welche den Spieler als Spielfigur repräsentiert. Fällt seine "currentHp" auf 0 oder weniger, hat der Spieler nicht verloren, jedoch muss der Held sich für den Rest den Spiels vom Kampfgeschehen zurück ziehen. Er kann später über einen Einheiteneditor erstellt werden.

Attribute:

- **leftHand:** Eine spezielle Action(bisher undefiniert), welche die Fähigkeit der linken Hand des Helden wider spiegelt(bzw. der Ausrüstung in seiner linken Hand). Sie benötigt Mana zum ausführen.
- **rightHand:** Analog zu "leftHand" für die rechte Hand.
- **name:** Der Name des Helden.

5.2.19 Base

Eine Basis ist eine spezielle Einheit die sich nicht bewegen kann, dafür jedoch Forschungen und Erweiterungen, sowie das Ausbilden von Einheiten ermöglicht. Fällt ihre "currentHp" auf 0 wird sie von dem Spieler übernommen, dem die Einheit gehört, welche den finalen Angriff gegen die Basis ausführte. Verliert ein Spieler alle seine Basen hat er verloren und scheidet aus dem Spiel aus.

Attribute:

- **labRoundsRemaining:** Gibt an wie viele Runden noch fehlen bis der Bau des Laboratoriums abgeschlossen ist, -1 wenn der Bau noch nicht begonnen hat.
- **caserneRoundsRemaining:** Analog zu "labRoundsRemaining" für die Kaserne.
- **recruiting:** Eine HashMap welche die zu rekrutierenden Einheiten speichert. Ihr

Wert entspricht den verbleibenden Runden, bis sie erscheinen und wird über "update" aktualisiert.

- **avaibleUnits:** Eine Liste mit allen Einheitentypen die rekrutiert werden können.
- **researching:** Analog zu "recruiting" für zu erforschende Verbesserungen, wenn eine Verbesserung erforscht ist kommt sie in "researched".
- **researched:** Alle vollständig erforschten Verbesserungen.

5.2.20 Research

Eine enumeration, welche alle zur Verfügung stehenden Verbesserungen beinhaltet, für eine einfache und Performanz-günstige Identifikation.

6 Ausführungssicht

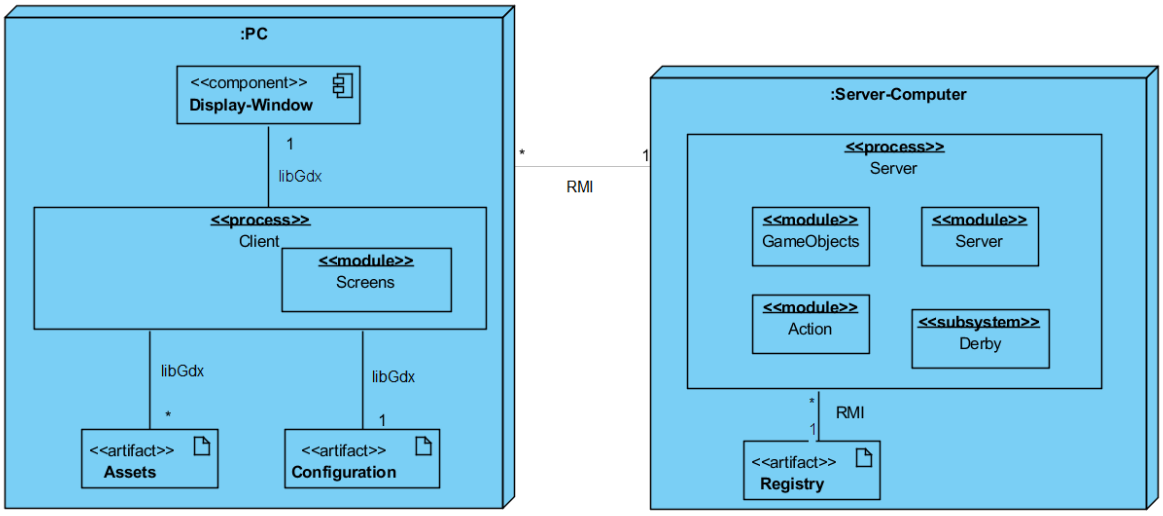

Abbildung 10: Ausführungssicht als Deployment-Diagramm

Wie auf dem Diagramm zu sehen läuft der Client-Prozess auf dem PC des Benutzers. Der Prozess selbst beinhaltet lediglich das Screens-Modul da, wie bereits im Paket-Diagramm der Modulsicht zu sehen war, die Module Assets und Configuration, kein direkter Teil der Implementation sind sondern Dateien auf dem PC. Der Client-Prozess gibt seine Informationen auf einem Fenster als Schnittstelle für den Benutzer aus. Die Verwendung der Assets, Configuration und des Ausgabefensters wird vom libGdx-Framework übernommen. Wie man sieht findet direkt auf dem Client nur Darstellungsrelevantes statt, wodurch wir die Möglichkeit zum Schummeln für den Client minimieren.

Über RMI kommuniziert der Client mit dem Server Computer. Auf dem Server-Computer laufen der Server-Prozess und die für RMI benötigte Registry. Dabei liegt immer nur eine Registry gleichzeitig auf dem PC, da RMI keine Verwendung verschiedener Registrys vor sieht und diese daher nicht unterschieden werden könnten. Die Registry kann jedoch von beliebig vielen Clients und Server-Prozessen verwendet werden was die Strategie mehrere Verbindungen zu zu lassen realisiert, außerdem werden Clients dadurch automatisch synchronisiert. Die Registry wird falls nötig vom Server Prozess angelegt. Im Server-Prozess finden sich dann die GameObjects-Module(welche hier zur Übersichtlichkeit zusammen gefasst wurden und somit den gleichen Inhalt wie die GameObjects-Komponente aus Kapitel 3 haben), das Action-Modul, sowie das Server-Modul. Das Server-Modul initialisiert das System und die Registry und ermöglicht dadurch den Zugriff für den Client. Danach greift der Client nur noch auf GameObjects zu und falls nötig minimal auf Action.

Außerdem findet sich in dem Server-Prozess noch die Derby-Datenbank. Diese liegt auf dem Server-Computer da nach den Anforderungen die Datenbank nicht installiert werden soll, sie muss also direkt vom Programm verwaltet werden und nicht einfach nur eine Verbindung aufgebaut.

Zuletzt ist an zu merken dass die Client und Server Prozesse nicht zwangsläufig auf zwei verschiedenen Computern laufen müssen, es ist durchaus möglich, durch die Strategie Client und Server in ein einzelnes Programm zu schreiben, dass beide Systeme auf dem gleichen Computer laufen. In diesem Fall bleiben die Inhalte der Prozesse gleich und auch an den Artefakten bzw. Komponenten ändert sich nichts, die Kommunikation erfolgt weiterhin über RMI, wobei lediglich "localhost" als Zielcomputer verwendet wird, um so auf den aktuellen Computer zu verweisen.

7 Zusammenhänge zwischen Anwendungsfällen und Architektur

In diesem Kapitel wird nun der Zusammenhang zwischen Anwendungsfällen und der Architektur mithilfe von Sequenzdiagrammen veranschaulicht. Bei den Diagrammen fällt eventuell auf dass die Methoden von auf Client-Ebene sehr generische Namen besitzen, dies liegt daran, dass diese wie bereits in der Modulsicht beschrieben noch nicht genau definiert sind, die Namen sind daher Repräsentanten für entsprechende Funktionen, wobei es mehrere Variationen dieser Methoden geben kann, abhängig von der speziellen Auswahl, welche in den Beispielen nicht genauer definiert ist. Diese Variationen unterscheiden sich jedoch nur von speziellen Werten und nicht im Ablauf.

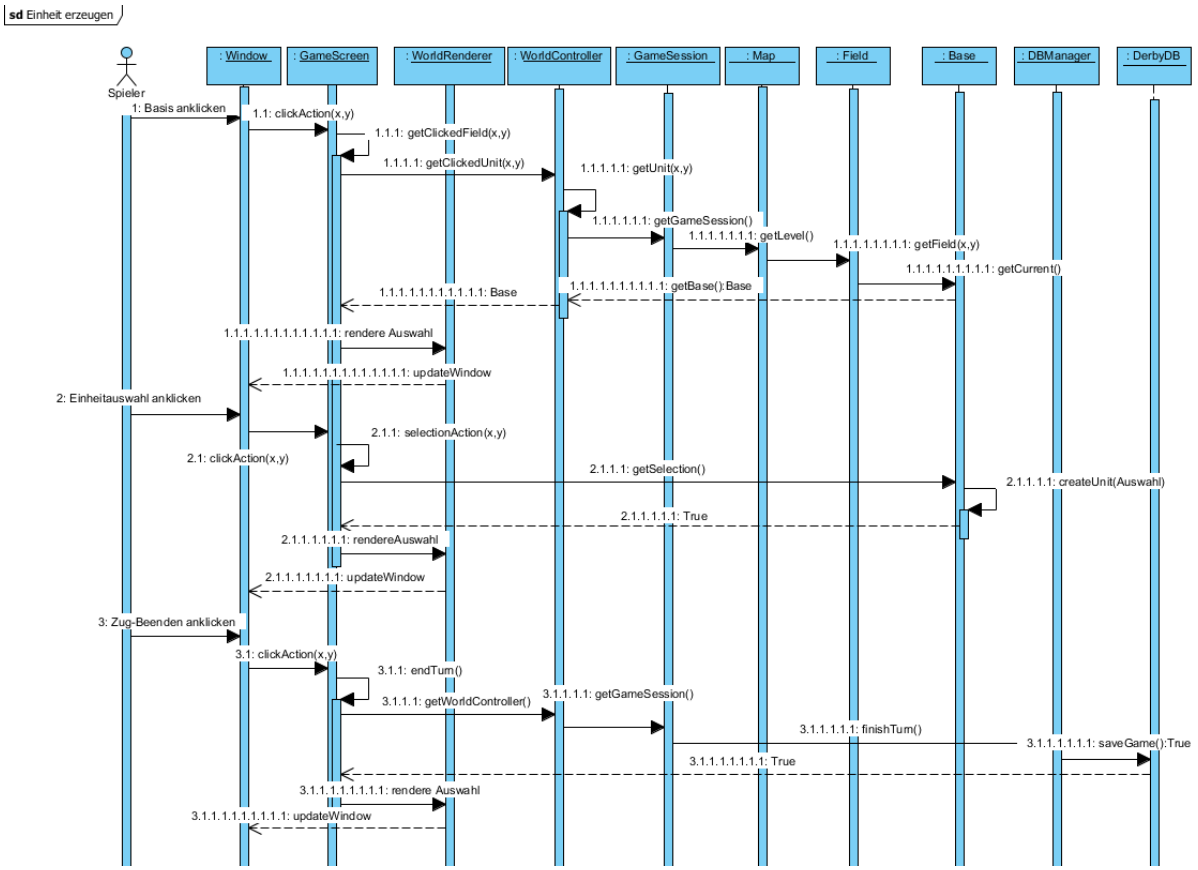

Abbildung 11: Das Erstellen einer Einheit

7.1 Erstellen einer Einheit

Bei diesem Anwendungsfall möchte der Spieler eine Einheit erstellen und danach seinen Zug beenden. Er beginnt damit auf eine Basis zu klicken welche auf seinem Fenster angezeigt wird um diese aus zu wählen. Dies führt zu einer Kette von Abfragen, welche jeweils ein Element abfragen um von diesem dann dass nächste Element zu erhalten, bis Schlussendlich die Basis zurück gegeben wird. Die Anfrage beginnt dabei in GameScreen(bzw. Window) und wird durch WorldRenderer und WorldController an die GameSession übersetzt. Diese leitet die Anfrage dann an ihre Elemente weiter, gemäß der Klassenhierarchie.

Ist die Frage dann beim Feld angekommen liefert dieses die Basis zurück, welche bis zum GameScreen durchgereicht wird. Nun haben sich die Informationen des Bildschirms geändert, was beim nächsten Rendern sichtbar wird(das das tatsächliche Rendern wird wie in den vorherigen Kapiteln bereits beschrieben von libGdx übernommen). Da die Basis nun ausgewählt und diese Auswahl im Fenster dargestellt ist, kann der Spieler nun die Einheitenauswahl anklicken. Dies bewirkt dass der GameScreen die an die Einheitenauswahl gebundene Funktion auszuführen, was in diesem Fall ein Aufrufen von

createUnit auf der ausgewählten Basis entspricht. Die Parameter für die Methoden sind dabei durch die spezielle Einheitenauswahl definiert. Innerhalb der createUnit-Methode wird dann überprüft ob die Methoden ausgeführt werden kann. Dieser Prozess ist für den Anwendungsfall nicht von Interesse, da wir davon ausgehen dass alle Bedingungen zum Erstellen der Einheit erfüllt sind, darum betrachten wir diesen Aspekt nicht genauer. Da alle erforderlichen Bedingungen erfüllt sind, gibt die Methode true zurück und die Render-Informationen aktualisieren sich erneut.

Zuletzt wählt der Spieler "Zug beenden" aus. Diese Anfrage wird durch den GameScreen weitergeleitet und vom WorldRenderer und WorldController an die GameSession übersetzt. Dort wird dann finishTurn aufgerufen, was das Ende der Runde kennzeichnet und den DBManager anweist die Session zu speichern. Da dieser Prozess in unserem Beispiel erfolgreich ist, wird nun noch true zurück gegeben und die Render-Informationen ein letztes mal aktualisiert.

7.2 Einheit wurde bewegt

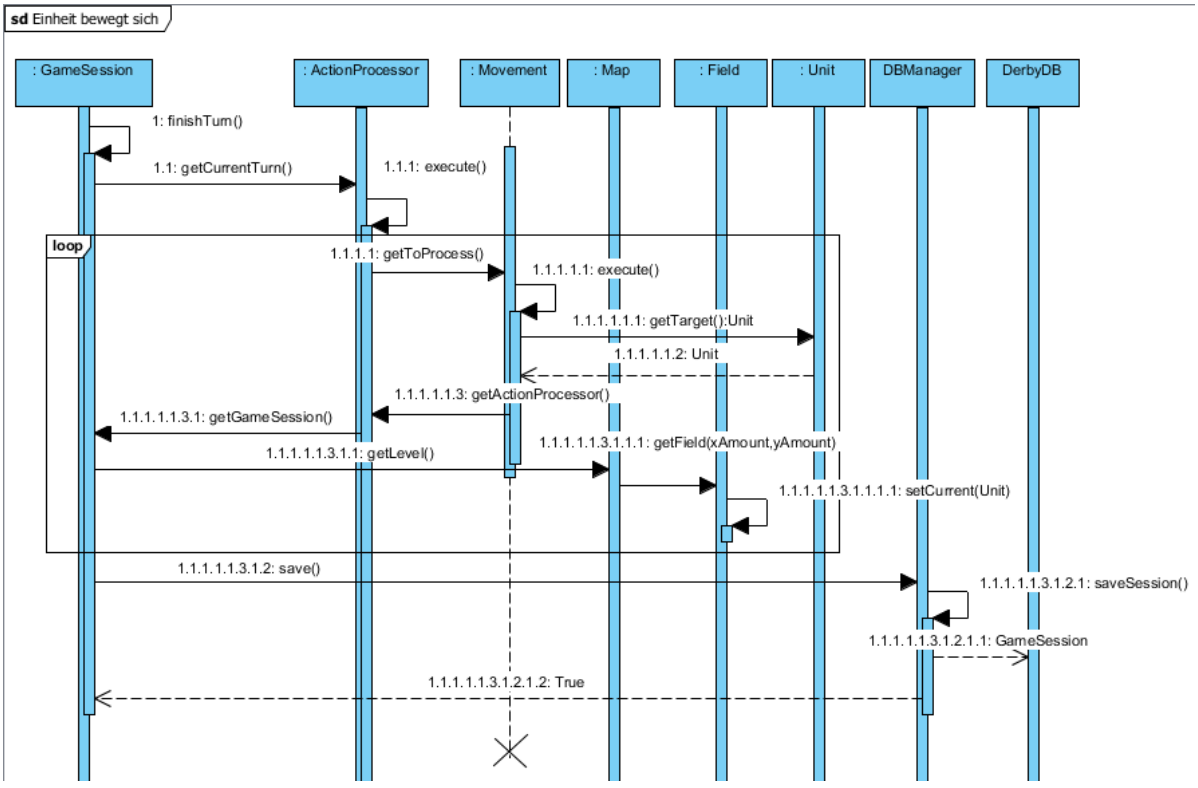

Abbildung 12: Der Prozess nachdem eine Einheit bewegt wurde

Dieser Anwendungsfall beginnt nachdem eine Einheit bewegt wurde und finishTurn aufgerufen wurde.

Daraufhin wird im ActionProcessor die execute-Methode Aufgerufen, wodurch eine

Schleife beginnt in der für jede Action innerhalb des ActionProcessors execute() Ausgeführt wird. In diesem Beispiel ist die einzige existierende Action die Movement-Action. Diese bewegt die Einheit, bei ihrer Ausführung auf ein leeres Feld. Wurde diese Methode ausgeführt, wird ein Movement-Objekt nicht weiter verwendet. Sind alle Actions des ActionProcessor-Objektes ausgeführt worden, wird die GameSession in der Datenbank gespeichert. Danach ist der nächste Spieler am Zug.

8 Evolution

Eine mögliche Änderung wäre der Wechsel von Gradle auf Maven. Dieser hätte für die Architektur selbst keine Auswirkungen, allerdings müssten alle Gradle bezogenen Dateien(z.B. build Dateien) entfernt werden und entsprechende Maven Dateien integriert werden, speziell müssten die Dependencys aus der allgemeinen build.gradle Datei in die pom.xml von Maven übertragen werden.

Würde man von RMI auf TCP/IP Kommunikation wechseln würde man einige weitere Klassen hinzu fügen welche die Kommunikation ermöglichen und müsste vermutlich jegliche Methodenaufrufe anpassen, also das gesamte System halb neu schreiben.

Eine weitere mögliche Änderung wäre ein Wechsel der Datenbank auf z.B. SQLite. In diesem Fall müsste man lediglich den Inhalt der Klasse DBManager an die jeweilige neue Schnittstelle anpassen, Änderungen an der Architektur wären jedoch nicht möglich, sofern die neue Datenbank keine speziellen Anforderungen an den Aufbau von Objekten hat.

Neben der grafischen Darstellung ist auch eine textliche Darstellung möglich. Ein Wechsel würde keine Änderung in der Architektur benötigen. Stattdessen müsste die Darstellung in den Screenklassen angepasst werden und der Großteil der Assets-Komponente würde weg fallen.

Um die Exzellenz-Anforderungen wie z.B. den Einheiteneditor zu erfüllen muss die Architektur um entsprechende Klassen erweitert werden welche den Zugriff für den Spieler ermöglichen und die Werte in den Einheiten die editierbar sein sollen entsprechend zu manipulieren.

Eine bisher nicht betrachte Änderung, da sie nicht geplant ist, wäre weitere Statistiken für den Spieler an zu legen. Dafür müsste man das Player bzw. Account-Objekt um entsprechende Inhalte erweitern und gegebenenfalls zusätzliche Screens zur Betrachtung dieser Statistiken hinzu fügen.

Zu guter Letzt ist eine Aufteilung von Server und Client auf zwei Programme denkbar. In diesem Fall müsste man die Server-Komponente auf ein Maven-Projekt auslagern um die Mindestanforderungen weiterhin zu erfüllen. Außerdem müsste man das übrige Client-Programm um die Interfaces der Server-Komponente erweitern, wobei zu beachten ist, dass diese die gleiche Struktur bezüglich der Pakete aufweisen müssen.

9 Dokumentation

Hier werden die Methoden der verschiedenen Klassen Dokumentiert, eine Beschreibung der Attribute ist bereits in der Datensicht zu finden. Für die Klassen, welche in der Datensicht als irrelevant betrachtet worden sind, finden sich hier auch Beschreibungen der Attribute. Sollte eine Klasse hier nicht aufgeführt sein, bedeutet dass das sie keine speziellen Methoden besitzt oder diese für die Architektur nicht relevant sind und somit noch nicht genau definiert. Außerdem werden teilweise Hilfsklassen kurz beschrieben, welche zwar, der Vollständigkeit halber im Diagramm auftauchen, aber für die restliche Architektur nicht relevant. Für eine bessere Orientierung zeigt Abb. 13 noch einmal das gesamte Programm als Klassendiagramm.

Die Dokumentation ist dabei nach Paketen geordnet.

9.1 Screens:

9.1.1 AbstractGameScreen

Diese Klasse stellt das Template dar welches alle Bildschirme erfüllen müssen. Die Funktionen sind dabei durch das LibGdx-Framework gegeben.

Attribute:

- **game:** Diese Attribut speichert dass aktuell laufende Programm und dient als Schnittstelle um den Bildschirm zu wechseln.
- **gamesession:** Diese Attribut speichert das aktuelle Spiel und wird verwendet um auf alle benötigten Informationen des Spiels zu zugreifen. Da das Spiel frühestens im NetworkScreen geladen wird, ist diese Attribut für alle vorherigen Screens(ConfigScreen und MenuScreen) **null**.
- **stage:** Diese Attribut stellt das aktuelle Level des Spiels dar.
- **configuration:** Hier wird das ConfigObjekt gespeichert, wodurch lokale Einstellungen eines Computers abgerufen werden können.

Methoden:

- **render:** Die Methoden bestimmt was im nächsten Bild dargestellt wird. Der Parameter "deltaTime" wird vom Framework gegeben und gibt die Zeit zum letzten Bild an. Dies wird benötigt um die Berechnungen Frame-unabhängig zu gestalten.In dieser Methode wird auch auf die Assets zu gegriffen.
- **resize:** Diese Funktion wird verwendet um die Größe des dargestellten Fensters zu ändern(ausgenommen Vollbild). Die Parameter sind selbsterklärend.
- **show:** Hiermit wird ein Bildschirm aktiv geschaltet.

- **hide:** Diese Methode deaktiviert einen Bildschirm wieder.
- **pause:** Mit dieser Methode kann das Rendern des Spiels pausiert werden, ist durch das Framework gegeben, für Desktop-Applikationen jedoch irrelevant.
- **rebuildStage:** Hiermit kann das Level auf den Ursprungszustand zurück gesetzt werden.

9.1.2 Config:

Methoden:

- **addServer/removeServer:** Fügt einen Server hinzu oder entfernt einen. Bekommt die IP des Servers als Parameter in Stringform übergeben.

9.1.3 WorldController

Dies ist eine Hilfsklasse, für die Kontrolle des Spiellevels, welche nach dem Framework verwendet wird, daher keine genauere Erklärung.

9.1.4 WorldRenderer

Dies ist eine Hilfsklasse, für die Darstellung des Spiellevels, welche nach dem Framework verwendet wird, daher keine genauere Erklärung.

9.2 ServerMain:

9.2.1 Server:

Methoden:

- **main:** Initialisiert das System und stellt eine Registry zur Verfügung.
- **loadSession:** Leitet die Anfrage an die gleichnamige DBManager Funktion weiter. Daher siehe DBManager.
- **saveSession:** Leitet die Anfrage an die gleichnamige DBManager Funktion weiter. Daher siehe DBManager.
- **registerAccount:** Leitet die Anfrage an die gleichnamige DBManager Funktion weiter. Daher siehe DBManager.
- **checkAccount:** Leitet die Anfrage an die gleichnamige DBManager Funktion weiter. Daher siehe DBManager.
- **shutdown:** Speichert alle ungespeicherten Spiele und beendet die Registry, sowie den Server.

- **getSessionList:** Liefert eine Liste mit den Namen aller auf dem Server verfügbaren Spielen, unter Verwendung des DBManagers zurück.
- **deleteAccount:** Leitet die Anfrage an die gleichnamige DBManager Funktion weiter. Daher siehe DBManager.
- **deleteSession:** Leitet die Anfrage an die gleichnamige DBManager Funktion weiter. Daher siehe DBManager.

9.2.2 DBManager:

Methoden:

- **loadSession:** Lädt das Spiel mit dem übergebenen Namen aus der Datenbank und gibt es zurück. Falls kein passendes Spiel existiert wird **null** zurück gegeben.
- **saveSession:** Speichert das übergebene Spiel in der Datenbank. Rückgabewert gibt an ob das Speichern funktioniert hat.
- **registerAccount:** Speichert einen neuen Account mit dem übergebenen Name und Passwort in der Datenbank. Rückgabewert gibt an ob der Vorgang erfolgreich war.
- **checkAccount:** Überprüft die übergebenen Werte auf einen korrekten Account in der Datenbank. Rückgabewert gibt an ob die Werte korrekt sind oder nicht.
- **deleteAccount:** Löscht den angegebenen Account aus der Datenbank, sofern er existiert, sonst passiert nichts.
- **deleteSession:** Löscht die GameSession aus der Datenbank, sofern sie existiert, sonst passiert nichts.

9.3 GameObject:

9.3.1 GameSession:

Methoden:

- **update:** Fordert alle Objekte des Spiels auf sich zu aktualisieren.
- **addTeam:** Fügt "teams" das übergebene Objekt hinzu.
- **removeTeam:** Entfernt das übergebene Team aus "teams", wenn es nicht vorhanden ist passiert nichts.
- **sendMessage:** Fügt dem Spielchat die übergebene Nachricht hinzu.
- **addBuffs:** Fügt eine Liste von Buffs zu "buffs" hinzu. Wird nur von ActionProcessor verwendet.
- **removeBuff:** Analog zu "removeTeam" für "buffs".

- **startTurn:** Initialisiert den nächsten Zug, d.h. wechselt den aktiven Player.
- **finishTurn:** Beendet den aktuellen Zug, d.h. fordert den ActionProcessor auf seine Inhalte zu verarbeiten, speichert alle Änderungen und ruft startTurn auf.
- **playerJoin:** Fügt einen neuen Spieler mit dem übergebenen Account zu dem übergebenen Team hinzu(sowie zu identities). Rückgabewert gibt an ob der Vorgang erfolgreich war.
- **playerLeave:** Entfernt den übergebenen Spieler aus dem Spiel und damit auch alle zugehörigen Objekte.
- **save:** Weist den Server an die Sitzung zu speichern. Rückgabewert gibt an ob der Vorgang erfolgreich war.
- **finish:** Beendet das Spiel. Es wird nicht beachtet ob das Spiel gespeichert ist. Gibt es einen Sieger wird dies angegeben und das Spiel danach gelöscht. Rückgabewert gibt an ob der Vorgang erfolgreich war.
- **createAction:** Dient zum Erstellen einer neuen Action, so dass das Action-Interface nur für die Funktionalität benötigt wird. Der erste Paramter gibt an um was für eine Action es sich handeln soll(z.B. "Fight"). Die weiteren Parameter entsprechen den Grundattributen einer Action. Alle spezifischeren Attribute werden danach über die setter gesetzt. Der Rückgabewert entspricht der erstellen Action.

9.3.2 Map:

Methoden:

- **init:** Diese Methode generiert die Felder der Karte, sowie max/minPlayer. Zu Beginn wird eine Karte Hard-gecoded. Wenn im Laufe des Projektes genügend Zeit übrig ist werden Karten in Dateien gespeichert und bei init ausgelesen, wobei die Datei über "levelName" identifiziert wird.
- **generateRandom:** Generiert eine zufällige Karte. Wird nur bei genug übriger Zeit des Projektes implementiert.
- **saveConfiguration:** Speichert die Karte in einer Datei, wird erst implementiert wenn genug Zeit im Projektverlauf übrig ist.
- **update:** Fordert alle Felder dazu auf sich zu aktualisieren.
- **getField:** Gibt das Feld mit den Koordinaten X und Y zurück.

9.3.3 Unit:

Methoden:

- **update:** Aktualisiert die Einheit, derzeit nur für Base nötig, später jedoch eventuell noch für andere.

9.3.4 Base:

Methoden:

- **createUnit:** Rekrutiert eine Einheit des übergebenen Typs. Rückgabewert gibt an ob der Vorgang erfolgreich war.
- **abortCreation:** Bricht den Rekrutierungsvorgang der übergebenen Einheit ab, sofern vorhanden. Analog zu "abortBuild" von Field.
- **buildLab:** Startet den Bau des Labors. Rückgabewert gibt an ob der Vorgang erfolgreich war.
- **abortLab:** Analog zu "abortCreation" für das Labor(wobei es immer nur ein Labor-Bauauftrag geben kann, daher kein Parameter).
- **buildCaserne:** Analog zu "buildLab" für die Kaserne.
- **abortCaserne:** Analog zu "abortLab" für die Kaserne.
- **research:** Erforscht die übergebene Forschung bzw. Verbesserung. Rückgabewert gibt an ob der Vorgang erfolgreich war.
- **abortResearch:** Bricht den Forschungsauftrag der übergebenen Forschung ab, wenn vorhanden, sonst passiert nichts.

9.3.5 Market:

Methoden:

- **buy:** Lässt den angegebenen Spieler die angegebene Menge Holz oder Eisen kaufen. Type bestimmt ob Holz oder Eisen(0=Holz, 1=Eisen). Kein Rückgabewert, die Werte werden direkt manipuliert. Zuletzt wird der jeweilige Preis neu berechnet in Abhängigkeit von einem Standardwert und der vorhandenen Menge.
- **sell:** Analog zu "buy", nur dass hier die Ressourcen verkauft und nicht gekauft werden.

9.4 Action:

9.4.1 ActionProcessor

: **Methoden:**

- **addAction:** Fügt die übergebene Action zu "toProcess" hinzu.
- **removeAction:** Entfernt die übergeben Action aus "toProcess", sofern vorhanden, ansonsten passiert nichts.
- **execute:** Führt alle Actions in toProcess aus und überprüft ob sich dadurch neue Actions oder Buffs ergeben. Neue Actions werden ebenfalls ausgeführt, Buffs wer-

den zu "toReturn" hinzu gefügt. Am Ende wird "toReturn" zurück gegeben, damit die Buffs in der GameSession gespeichert werden können.

9.4.2 Action:

Methoden:

- **execute:** Führt die Aktion aus. Konkreter Inhalt ist von den jeweiligen Aktionen abhängig.

9.5 Chat:

9.5.1 Chat(Klasse):

Methoden:

- **addMessage:** Fügt die übergebene Nachricht zu "backlog" hinzu.
- **removeMessage:** Entfernt die übergebene Nachricht aus "backlog", sofern vorhanden, ansonsten passiert nichts.
- **addParticipant:** Analog zu addMessage für "participants".
- **removeParticipant:** Analog zu removeMessage für "participants".
- **blockPlayer:** Verschiebt den angegebenen Spieler von "participants" zu "readOnly". Ist der Spieler nicht vorhanden oder bereits in "readOnly" passiert nichts.
- **unblockPlayer:** Analog zu blockPlayer in umgekehrter Richtung(also "readOnly" zu "participants").
- **clear:** Leert den backlog des Chat.

9.5.2 Message:

Methoden:

- **makeVisibleFor:** Fügt einen Spieler zu "visibleFor" hinzu und ermöglicht ihm so die Nachricht zu lesen. Der Vorgang kann nicht rückgängig gemacht werden.

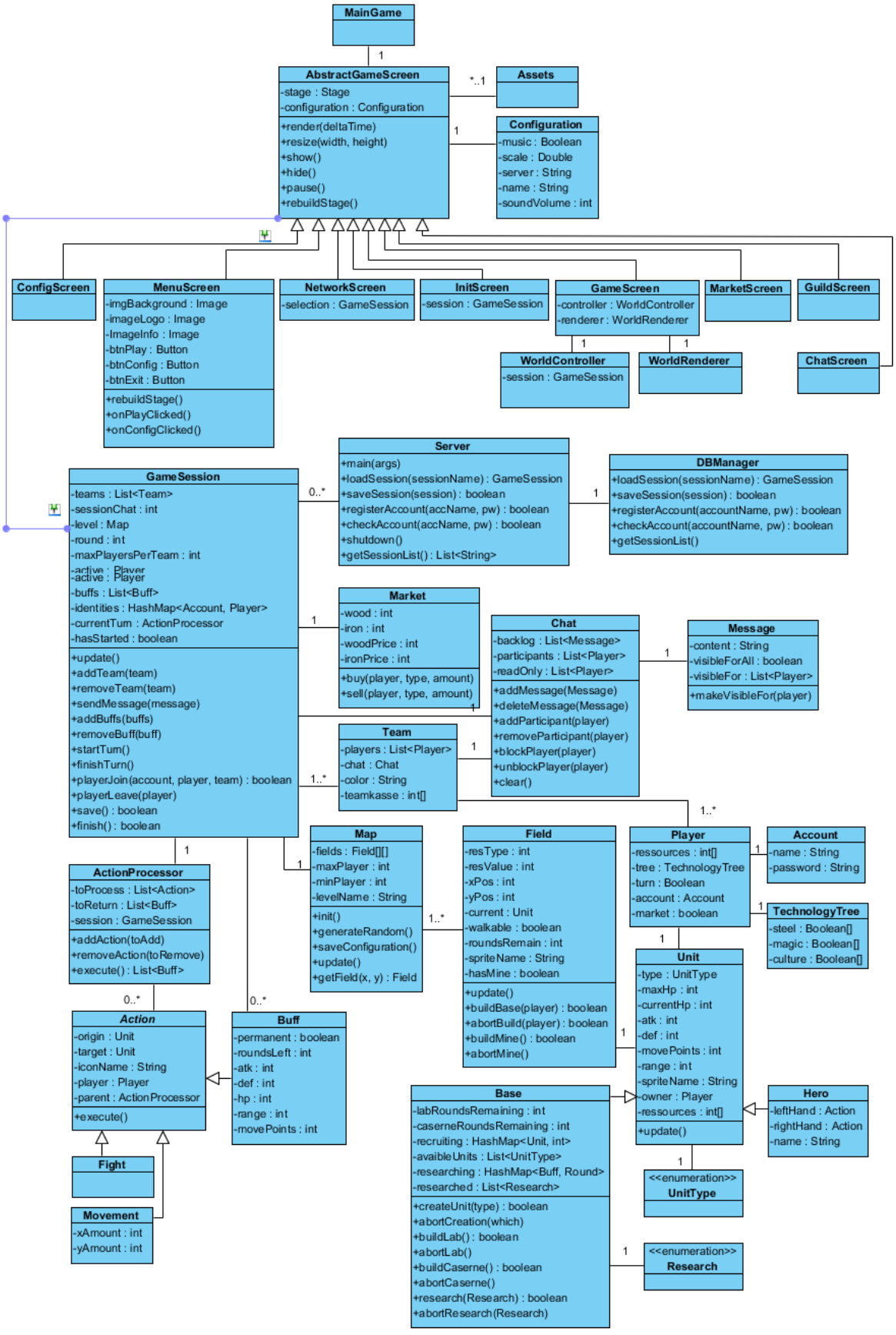

Abbildung 13: Gold and Greed als Klassendiagramm