Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marius Klausa

Konzeption und Entwicklung eines Frameworks

für online basierte Aufbau- und Rollenspiele

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marius Klausa

Konzeption und Entwicklung eines Frameworks für online basierte Aufbau- und Rollenspiele

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Stefan Sarstedt
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 15. August 2011

Marius Klausa

Thema der Bachelorarbeit

Konzeption und Entwicklung eines Frameworks für online basierte Aufbau- und Rollenspiele

Stichworte

Browsergame, Framework, Online Games, Quasar, Hibernate

Kurzzusammenfassung

Im Rahmen dieser Arbeit werden zunächst diverse Browsergames untersucht, um eine fundierte Anforderungsanalyse für das zu entwickelnde Framework aufzustellen. Im Folgenden wird die Realisierung des Frameworks im Detail erörtert. Abschließend wird gezeigt, wie eine konkrete Spieleidee mithilfe des entwickelten Frameworks umgesetzt wird.

Marius Klausa

Title of the paper

Design and development of a Framework for online-based build-up- and role-playing games

Keywords

Browsergame, Framework, Online Games, Quasar, Hibernate

Abstract

Within this paper different Browsergames will be examined initially in order to set a profound requirement analysis. In the following, the realization of the Framwork will be discussed detailedly. In conclusion, the execution of a specific game-idea by means of the Browsergame Framework will be shown.

Inhaltsverzeichnis

1. Einleitung ... 4

1.1. Motivation ... 4

1.2. Zielsetzung ... 4

1.3. Abgrenzungen ... 4

1.4. Aufbau der Arbeit ... 5

2. Grundlagen ... 6

2.1. Browsergames ... 6

2.1.1. Definition Browsergame ... 6

2.1.2. Browsergame Thematik ... 6

2.1.3. Untersuchung diverser Browsergames ... 7

2.2. Browsergame Frameworks ... 9

2.2.1. Browsergame Framework .NET ... 9

2.2.2. Modular Gaming ... 9

2.2.3. Little Goblin ... 10

2.3. Browsergame Baukasten ... 10

2.4. Quasar ... 10

2.5. Hibernate ... 11

2.6. Windows Communication Foundation ... 12

3. Konzeption ... 13

3.1. Idee ... 13

3.2 Funktionale Anforderungen ... 13

3.3. Nichtfunktionale Anforderungen ... 18

4. Entwurf ... 19

4.1. Architektur des Anwendungskerns ... 19

4.2. Komponentenentwurf im Detail ... 21

4.2.1. Attribut-Komponente ... 21

4.2.2. Forschungs-Komponente ... 22

4.2.3. Ressourcen-Komponente ... 22

4.2.4. Einheiten-Komponente ... 22

4.2.5. Kampf-Komponente ... 26

4.2.6. Charakter-Komponente ... 26

4.2.7. Auftrags-Komponente ... 27

4.2.8. Spielfeld-Komponente ... 27

4.2.9. Technologiebaum-Komponente 27
4.2.10. User-Komponente 28
4.2.11. Quest-Komponente 28
4.2.12. Nachrichten-Komponente 29
4.2.13. Item-Komponente 29
4.2.14. Handels-Komponente 29
4.2.15. Gilden-Komponente 30
4.2.16. Accountverwaltungs-Komponente 31
4.2.17 Rechteverwaltungs-Komponente 31
4.2.18. Sitzungsverwaltungs-Komponente 32
4.3. Architektur des Gesamtsystems 33
4.4. Kompositionsmanager im Detail 34
4.5. Architektur der technischen Infrastruktur 34
4.6 Technik-Architektur 35
4.7. Persistenz-Manager 36
4.8. Umgang mit Transaktionen 37
4.9. Mail- und Common-Komponente 37
5. Implementierung 38
5.1. Einheiten-Komponente 38
5.2. Kampf-Komponente 40
5.3. Technologiebaum-Komponente 41
5.4. Spielfeld-Komponente 42
5.5. Auftrags-Komponente 43
5.6. Charakter-Komponente 43
5.7. Ausnahmebehandlung 45
5.8. Problematik mit parallelen Transaktionen 45
6. Test 46
6.1. Test der Auftrags-Komponente 46
6.2. Codeabdeckung 48
7. Realisierung eines Fallbeispiels mithilfe des Frameworks 49
7.1. Technologischer Rahmen 49
7.2. Spieleidee 49
7.3. Erzeugung der Daten und Konfiguration des Frameworks 50
7.4. Hosten des WCF-Dienstes 52
7.5. Entwicklung einer prototypischen Webseite. 52
8. Zusammenfassung und Ausblick 54

8.1. Zusammenfassung 54
8.3. Resümee 54
8.3. Ausblick 56
Literaturverzeichnis **57**
Tabellenverzeichnis **59**
Abbildungsverzeichnis **60**
Glossar **61**
Anhang CD

1. Einleitung

In diesem Kapitel wird zuerst die Motivation, die hinter dieser Arbeit steht, erörtert und anschließend die Zielsetzung formuliert. Daraufhin werden einige Abgrenzungen im Rahmen dieser Arbeit diskutiert. Abschließend wird die Gliederung der Arbeit vorgestellt.

1.1. Motivation

Online-Games erfreuen sich großer Beliebtheit. Das Unternehmen Bigpoint beispielsweise entwickelt stetig neue Browsergames[1]. Für Browsergames kann man sich in der Regel kostenlos registrieren. Die meisten Spieleinhalte werden kostenlos zur Verfügung gestellt. Wenn man bei einem Spiel aber erfolgreich und besser als andere Spieler sein möchte, kann man bei sehr vielen Browsergames spezielle Spiele-Inhalte kaufen. Dieser Aspekt macht Browsergames auch wirtschaftlich interessant.
Bei Recherchen im Internet liest man häufig in Foren, dass viele Leute ein eigenes Browsergame betreiben möchten. Doch fehlt ihnen meistens das technische Know-how und Durchhaltevermögen um solch ein Spiel zu entwickeln. Aus diesem Grund wurden Überlegungen angestellt ein Framework zu entwickeln, welches das Erstellen von Browsergames erleichtern soll.

1.2. Zielsetzung

Das Ziel ist es ein Framework zu entwickeln, welches das Erstellen von umfangreichen online-basierten Spielen vereinfacht. Das Framework soll Funktionalität für die meistbenötigten Features bereitstellen. Die Entscheidung, welche Features in das Framework aufgenommen werden, soll nicht willkürlich, sondern durch eine detaillierte Untersuchung diverser Browsergames erfolgen. Um eine Individualisierung der erstellten Spiele zu ermöglichen, soll das Framework an vielen Stellen konfigurierbar sein. Das Framework soll als alleinstehender Server konzipiert werden. Um zu verhindern, dass Benutzer betrügen und Spielstände manipulieren können, sollen alle Berechnungen auf dem Server stattfinden. Clients dienen lediglich der Interaktion und der Darstellung.

1.3. Abgrenzungen

Das zu entwickelnde Framework soll keine Funktionalität für grafische Animationen o.ä. anbieten. Die Optik der Spiele und die Interaktionsmöglichkeiten für die Benutzer sind ausschließlich den Clients überlassen. Die Arbeit beschäftigt sich hauptsächlich mit der Entwicklung des Frameworks. Die Entwicklung von Clients wird nur prototypisch anhand

---

[1] http://de.bigpoint.com

eines Fallbeispiels erläutert. Die Features des Frameworks sollen nur aus den Genres Aufbau- und Rollenspiel stammen.

1.4. Aufbau der Arbeit

Kapitel 2 beschäftigt sich mit den Grundlagen. Es behandelt die Themen Browsergames, Browsergame Frameworks und stellt einige in der Arbeit verwendete Technologien vor.

Kapitel 3 enthält eine ausführliche Liste aller an das Framework gestellten Anforderungen.

Kapitel 4 beschäftigt sich mit dem Entwurf des Frameworks. Es behandelt den detaillierten Entwurf aller Komponenten, sowie die Architektur des gesamten Systems.

Kapitel 5 enthält einige interessante Aspekte der Implementierung.

Kapitel 6 erläutert, wie das Framework getestet wurde und zeigt einen Ausschnitt aus einem Komponenten-Test. Darüber hinaus beinhaltet es eine tabellarische Übersicht der Codeabdeckung.

Kapitel 7 zeigt detailliert, wie ein konkretes Spiel mithilfe des Frameworks entwickelt wird.

Kapitel 8 enthält die Zusammenfassung, eine ausführliche Diskussion über die Stärken und Grenzen des Frameworks, sowie einen Ausblick über zukünftige Erweiterungen.

2. Grundlagen

Dieses Kapitel stellt die Grundlagen für das Verständnis der nachfolgenden Arbeit. Zunächst wird das Thema Browsergame erläutert, gefolgt von einer Betrachtung über einige existierende Browsergame Frameworks. Abschließend werden einige in der Arbeit verwendete Technologien vorgestellt.

2.1. Browsergames

In diesem Abschnitt wird zunächst eine allgemeine Definition für Browsergames aufgestellt und anschließend einige Einschränkungen gemacht, wie ein Browsergame im Kontext dieser Arbeit definiert wird. Nachfolgend werden einige aktuelle Browsergames und ihre Gemeinsamkeiten angeschaut, damit ersichtlich wird, welche Features in aktuellen Spielen vorkommen. Dies ist notwendig, um eine fundierte Anforderungsanalyse aufstellen zu können.

2.1.1. Definition Browsergame

Unter einem Browsergame wird ein Computerspiel verstanden, dass entweder in einem Web-Browser abläuft, oder über einen bedient wird.[2] Durch Browser Plug-Ins, wie beispielsweise der Flash Player von Adobe Systems, und die Einführung von HTML5 sind Browsergames theoretisch keine Grenzen gesetzt. Von Text-basierten, über einfache 2D, bis hin zu komplexen 3D-Spielen, ist alles realisierbar.
Im Kontext dieser Arbeit spricht man bei Browsergames von einfachen textbasierten Spielen, die serverseitig berechnet werden und lediglich über einen Web-Browser bedient werden. Es gibt keine grafisch animierten, interaktiven Elemente.

2.1.2. Browsergame Thematik

Nach näherer Betrachtung lassen sich die meisten Browsergames in die Genres Aufbau- und Rollenspiel einteilen.[3] Dabei können sie verschiedene Ausprägungen haben, wie z.B. Antike, Fantasy, Mittelalter oder Weltraum. Bei einem Aufbauspiel hat man meistens eine Basis, die z.B. ein Planet oder ein Dorf sein kann. Auf dieser Basis kann man verschiedene Gebäude ausbauen, wie z.B. eine Kaserne, ein Forschungslabor oder einen Hangar. Darüber hinaus können auf einer Basis Einheiten ausgebildet werden. Mit den Einheiten kann man die Basen anderer Spieler Angreifen oder die eigene Basis verteidigen. Ein Rollenspiel hat als Mittelpunkt immer einen Charakter (Spielfigur), welcher je nach Ausprägung z.B. ein Mittelalterlicher Krieger, Elf oder ein Raumschiff sein kann. Charaktere haben fast immer ein Level (Stufe), welches durch Erfahrungspunkte aufsteigen kann, und

[2] http://de.wikipedia.org/wiki/Browsergame
[3] http://www.browsergamesliste.com

Attribute, wie z.B. Ausdauer, Stärke oder Geschicklichkeit. Attribute können je nach Spiel auf unterschiedliche Art und Weise ausgebaut werden. Zusätzlich hat ein Charakter fast immer ein Inventar und Ausrüstungsplätze für Items (Gegenstände), wie Schwert, Brustpanzer oder Stiefel. Items verbessern den Charakter, indem sie die Attribute erhöhen oder mehr Schaden machen. Es gibt Quests (Aufgaben), welche vom Charakter absolviert werden können. Dabei kann man Items oder Erfahrungspunkte erhalten.

2.1.3. Untersuchung diverser Browsergames

Untersucht wurden die folgenden Spiele:

- Stargate-Galaxys
- Die Stämme
- Gondal,
- OGame
- Knight Fight
- Shakes & Fidget
- Pennergame
- Kartellwar
- Travian
- Zaren Kriege.

Die Links zu den genannten Spielen sind im Literaturverzeichnis zu finden

Wie in Abschnitt 2.1.2. erwähnt, sind die meisten Spiele aus dem Genre der Aufbau- und Rollenspiele. Bei allen Spielen des Aufbau-Genres gibt es eine Basis, auf der man verschiedene Gebäude und Einheiten ausbauen kann. Einheiten und Gebäude haben Attribute. Der Ausbau der Gebäude und Einheiten kostet Ressourcen. Ressourcen sind z.B. Holz, Eisen oder Wasserstoff. Durch Forschungen können die Attribute der Gebäude oder Einheiten verbessert werden. Mit den Einheiten kann man die Basen anderer Spieler angreifen und dabei Ressourcen erbeuten. Befreundete Spieler können sich gegenseitig Ressourcen schicken. Für diese Bewegungen muss man die Koordinaten der Zielbasis eintragen. Dafür gibt es eine zwei- oder dreidimensionale Karte, in welcher die Koordinaten der Basen verzeichnet sind. Spieler können untereinander Handel treiben, das bedeutet Ressourcen gegen andere Ressourcen tauschen zu können. Jedes Aufbauspiel hat zusätzlich einen Technologiebaum, welcher die Voraussetzungen zum Bau von Gebäuden oder Einheiten definiert. Bei den Browsergames des Rollenspiel-Genres gibt es einen Charakter, der Attribute hat, welche man durch Forschungen oder Ressourcen upgraden (verbessern) kann. Spieler können mit ihrem Charakter gegen die Charaktere anderer Spieler kämpfen. Der Charakter kann Items ausrüsten und hat ein Inventar, in dem man Items ablegen kann. Zusätzlich kann der Charakter bei den meisten Spielen durch erhalt von Erfahrungspunkten Stufen aufsteigen. Bei Händlern kann man Items im Tausch gegen Ressourcen erhalten. Bei allen untersuchten Spielen gibt es ein Gilden- und Nachrichtensystem. Spieler können Gilden (Gruppen) gründen und andere Spieler in diese einladen. Gilden haben ein eigenes Rechtesystem. Man kann Ränge mit verschiedenen Rechten definieren und sie den Gildenmitgliedern zuweisen. Rechte sind z.B. Spieler-einladen, Spieler-rauswerfen oder Gilde-auflösen. Bei einigen Spielen gibt es zusätzlich eine Gildenkasse, in welche die Gildenmitglieder Ressourcen einzahlen können.
Im Folgenden werden die Features der untersuchten Browsergames tabellarisch dargestellt.

	Stargate Galaxy	Die Stämme	Gondal	Ogame	Knight Fight	Shakes & Fidget	Pennergame	Kartellwar	Travian	Zaren Kriege
Es gibt Ressourcen zum Bau von	Ja, 3	Ja, 3	Nein	Ja, 5	Nein	Nein	Nein	Nein	Ja, 4	Ja, 4
Man kann Gebäude auf Zeit bauen	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Ja	Ja
Spezielle Gebäude produzieren	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Ja	Ja
Man kann Kampfeinheiten auf Zeit	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Ja	Ja
Einheiten haben Attribute	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Ja	Ja
Es gibt Basen auf der man Gebäude bauen kann	Ja, maximal 10 Planeten	Ja, unbegrenzt Dörfer	Nein	Ja, maximal 8 Planeten	Nein	Nein	Nein	Nein	Ja, mehrere Dörfer	Ja, ein Dorf
Man kann Einheiten zwischen Basen	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Ja	Ja
Man kann andere Spieler angreifen	Ja, die Planeten und dabei Ressourcen erbeuten	Ja, die Dörfer und dabei Ressourcen erbeuten oder die Dörfer erobern	Ja, den Charakter anderer Spieler (Gold erbeuten)	Ja, die Planeten und dabei Ressourcen erbeuten	Ja, den Charakter anderer Spieler (Gold erbeuten und Erfahrungspunkte erhalten)	Ja, den Charakter anderer Spieler (Gold erbeuten und Ehrenpunkte erhalten)	Ja, den Charakter anderer Spieler (Geld erbeuten)	Ja, den Charakter anderer Spieler	Ja, Dörfer anderer Spieler	Ja, Dörfer anderer Spieler
Es gibt Forschungen, die Einheiten verbessern (auf Zeit)	Ja	Ja	Nein	Ja	Nein	Nein	Ja, verbessern die Attribute des Charakters	Ja, verbessern die Attribute des Charakters	Ja	Ja
Es gibt einen Technologiebaum	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Ja	Ja
Einheitenbewegungen dauern pro	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Ja	Ja
Es gibt ein Gildensystem	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
Die Gilde hat ein Rechtesystem	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
Es gibt ein Gildenkonto für	Ja	Nein	Ja	Nein	Ja	Ja	Ja	Ja	Nein	Ka
Man kann Handel mit anderen Spielern treiben	Ja, Ressourcen	Ja, Ressourcen	Nein	Nein	Nein	Nein	Nein	Ja, Gegenstände	Ja, Ressourcen und Gegenstände	Ja, Ressourcen
Es gibt eine Karte mit Koordinaten	Ja, 3	Ja, 2 Koordinaten	Nein	Ja, 3 Koordinaten	Nein	Nein	Nein	Nein	Ja, 2 Koordinaten	Ja, 2 Koordinaten
Spieler können sich Nachrichten	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
Es gibt ein Punktesystem	Ja	Ja	Ja	Ja	Nein	Ja	Ja	Nein	Ja	Ja
Es gibt Ranglisten	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Nein	Ja	Ja
Es gibt einen Charakter	Nein	Nein	Ja	Nein	Ja	Ja	Ja	Ja	Ja	Ja
Der Charakter hat	Nein	Nein	Ja	Nein	Ja	Ja	Ja	Ja	Ja	Nein
Es gibt ein Inventar	Nein	Nein	Ja	Nein	Ja	Ja	Ja	Ja	Ja	Nein
Der Charakter hat eine Stufe die er aufsteigen kann	Nein	Nein	Ja, durch Erfahrungspunkte	Nein	Ja, durch Erfahrungspunkte	Ja, durch Erfahrungspunkte	Nein	Nein	Ja	Ja, durch Erfahrungspunkte
Der Charakter hat Lebensenergie	Nein	Nein	Ja	Nein	Ja	Ja	Nein	Ja	Ja	Nein
Der Charakter hat Attribute die man upgraden kann	Nein	Nein	Ja, durch Gold	Nein	Ja, durch Gold	Ja, durch Gold	Ja, durch Forschungen	Ja, durch Forschungen	Ja	Ja
Es gibt neben Ressourcen eine	Nein	Nein	Ja, Gold und Kristalle	Nein	Ja, Gold und Edelsteine	Ja, Gold und Pilze	Ja, Euro	Ja, Geld	Ja, Geld	Ja, Goldmünzen
Es gibt ein auf Zeit basierendes Questsystem	Nein	Nein	Ja, als Belohnung gibt es Gold, Kristalle und Erfahrungspunkte	Nein	Ja, als Belohnung gibt es Gold, Edelsteine und Erfahrungspunkte	Ja, als Belohnung gibt es Gold, Pilze, Gegenstände und Erfahrungspunkte	Ja, als Belohnung gibt es Items die man verkaufen kann	Ja, als Belohnung gibt es Geld	Ja, als Belohnung gibt es Ressourcen	Nein
Es gibt verschiedenen Gegenstände	Nein	Nein	Ja, z.B. Kopf und Brust	Nein	Ja, z.B. Kopf und Brust	Ja, z.B. Kopf und Brust	Ja	Ja	Ja	Nein
Es gibt Händler	Nein	Nein	Ja, für Gegenstände	Ja, für Ressourcen	Ja, für Gegenstände	Ja, für Gegenstände	Ja, für Gegenstände	Ja	Nein	Nein
Es gibt Mittel um die Questzeit zu	Nein	Nein	Ja	Nein	Nein	Ja	Nein	Nein	Ja	Nein
Es gibt Computergegner gegen die	Nein	Nein	Ja, bei Quests	Nein	Nein	Ja, bei Quests	Nein	Nein	Nein	Nein

Tabelle1: Feature-Matrix

2.2. Browsergame Frameworks

Browsergame Frameworks sollen Entwicklern die Erstellung von Browsergames erleichtern. Dafür stellen sie einige oft gebrauchte Komponenten zur Verfügung. Im Folgenden werden drei verschiedene Ableger, die alle auf unterschiedlichen Technologien basieren, betrachtet.

2.2.1. Browsergame Framework .NET

Das Browsergame Framework .NET ist ein Open-Source-Framework, welches in C# geschrieben ist und sich in Entwicklung befindet. Zurzeit bietet es lediglich einige technische Komponenten und nur eine Anwendungskomponente an. Die technischen Komponenten dienen der Registrierung und Aktivierung von Benutzern. Das Aktivieren erfolgt mittels Aktivierungscode, der per E-Mail an den Benutzer geschickt wird. Eine Message-Komponente zum Verschicken von Nachrichten ist ebenfalls implementiert. Zusätzlich gibt es einige Hilfsklassen zum Auslesen von Werten in der Konfigurationsdatei, zum Transformieren von XML-Dateien und für das Logging. Das Framework bietet die Anbindung an eine Datenbank, allerdings wird zurzeit nur der Microsoft SQL-Server 2008 unterstützt. Die Anwendungskomponente bietet eine Unit-Klasse an (für Spiele-Einheiten). Diese hat lediglich einige Eigenschaften und sonst keinerlei Funktionalität. Zusätzlich wird eine Kampf-Klasse angeboten, welche einen kleinen Rahmen für einen Kampf vorgibt, aber keine richtige Implementierung enthält. Die Dokumentation des Frameworks ist sehr rudimentär. Auf der Webseite gibt es für jede Komponente einen beschreibenden Satz. Da es keine Anleitung gibt, ist es unklar, wie man mithilfe des Frameworks ein Browsergame veröffentlicht.[4]

2.2.2. Modular Gaming

Modular Gaming ist ein Open-Source-Framework, das auf Kohana[5] basiert. Es ist komplett in PHP geschrieben. Modular Gaming bietet neben den technischen Komponenten für Authentifizierung/Autorisierung und Nachrichten auch einige funktionierende Anwendungskomponenten. Diese umfassen ein Kampfsystem, Charakter, Inventar, Shops und Zonen. Der Charakter kann zwischen Zonen reisen und gegen Monster kämpfen. Die Eigenschaften eines Charakters sind im Quellcode definiert. So hat ein Charakter z.B. die Eigenschaften Lebensenergie, Stärke, Verteidigung, Beweglichkeit, Level und Erfahrung, um nur einige zu nennen. Es gibt keinerlei Dokumentation, allerdings steht eine kurze Anleitung zur Framework Installation zur Verfügung.[6]

[4] http://bgframework.codeplex.com
[5] http://kohanaframework.org
[6] http://modulargaming.com

2.2.3. Little Goblin

Little Goblin ist einerseits ein Open-Source-Framework, das mit Grails[7] entwickelt wird und andererseits ein Browsergame, welches auf dem Framework basiert. Es wird in der Programmiersprache Groovy entwickelt. Neben technischen Komponenten für das Login und Nachrichten, bietet Little Goblin auch einige Anwendungskomponenten an. So gibt es beispielsweise eine Charakter-Komponente. Ein Charakter hat diverse Ausrüstungsplätze, z.B. für Waffen und Rüstungen. Des Weiteren bietet Little Goblin ein Questsystem und ein einfaches rundenbasiertes Kampfsystem an. Auch bei diesem Framework wird die Erstellung eines Browsergames durch den Mangel an Dokumentation erschwert. Es gibt lediglich ein Klassendiagramm. Zu erwähnen ist jedoch, dass der zur Verfügung stehende Quellcode eine Demoversion des Browsergames Little Goblin enthält. Darüber hinaus verfügt die Webseite über eine Anleitung zur Installation des Frameworks und der dazugehörigen Demoversion.[8]

2.3. Browsergame Baukasten

Das Unternehmen HPM Kommunikation GmbH bietet auf der Webseite mybrowsergame.com einen Browsergame Baukasten an, mit dem man ohne zu Programmieren ein eigenes Spiel erstellen kann. Es gibt eine frei lizensierte Version die kostenlos ist. Bei dieser wird im fertigen Spiel lediglich Werbung eingeblendet. Bei der professionellen Lizenz darf man kommerzielle Spiele erstellen. Es fallen Kosten von 50 € im Monat pro angefangene tausend Spieler an[1], die Werbung entfällt und man hat vollständige Kontrolle über die Vermarktung des Spiels. Um ein Spiel zu erstellen muss man umfangreiche Einstellungen vornehmen. Es gibt Einstellungsmöglichkeiten für z.B. Ressourcen, Spielobjekte, Spielfeld, Spielereigenschaften und Spielerinteraktionen. Spielerobjekte können z.B. ein Dorf oder ein Planet sein. Bei dem Spielfeld kann man einstellen, ob es ein-, zwei-, oder dreidimensional sein soll und welche Ressource bei Bewegungen auf dem Spielfeld verbraucht werden soll. Spielereigenschaften können Attribute eines Charakters sein. Bei den Spielerinteraktionen kann man u.a. den Kampfablauf konfigurieren. Ob Ressourcen von einem Dorf entwendet werden können und ob der Spieler mit mehr Einheiten einen Vorteil beim Kampf erhält. Der Browsergame Baukasten ist zwar relativ mächtig, man hat aber keinen Zugriff auf den entstehenden Quellcode und ist somit auf die verfügbaren Einstellungsmöglichkeiten beschränkt.[9]

2.4. Quasar

Quasar ist die Standard-Architektur des Unternehmens sd&m. Es definiert für einige oft benötigte Komponenten Standardschnittstellen, wie z.B. für den Datenbankzugriff. Quasar unterscheidet ganz strikt zwischen fachlichen und technischen Komponenten. Fachliche Komponenten sollten ausschließlich mit der Fachlichkeit einer Anwendung zu tun haben, z.B. der Kundenverwaltung. Technische Komponenten dagegen befassen sich mit der Technologie, welche Anwendungsunabhängig ist, z.B. die Datenbankanbindung. Alle

[7] http://grails.org
[8] http://dewarim.de
[9] http://www.mybrowsergame.com

fachlichen Komponenten, auch Anwendungskomponenten, befinden sich im Anwendungskern. Jede Anwendungskomponente besteht aus beliebig vielen Anwendungsfällen, Entitätstypen, Datentypen und ihren Verwaltern. „Entitätstypen sind die Pfeiler, an denen eine Anwendung hochgezogen wird: das Buch, der Buchtitel, die Ausleihe […]“[10]. Entitätstypen haben zusätzlich einen Schlüssel, der ihre Gleichheit definiert. Die Verwalter sind für das Lesen, Schreiben, Ändern und Löschen von Entitätstypen verantwortlich. Datentypen hingegen haben keinen Schlüssel, sie sind gleich, wenn ihre Felder übereinstimmen. Beispiele für Datentypen sind die primitiven Typen wie String oder Integer und komplexere Typen wie eine Adresse oder eine Bankverbindung. Anwendungsfälle können andere Anwendungsfälle aufrufen. Zur Veranschaulichung beschreibt die Abbildung1 die Standardarchitektur des Anwendungskerns.

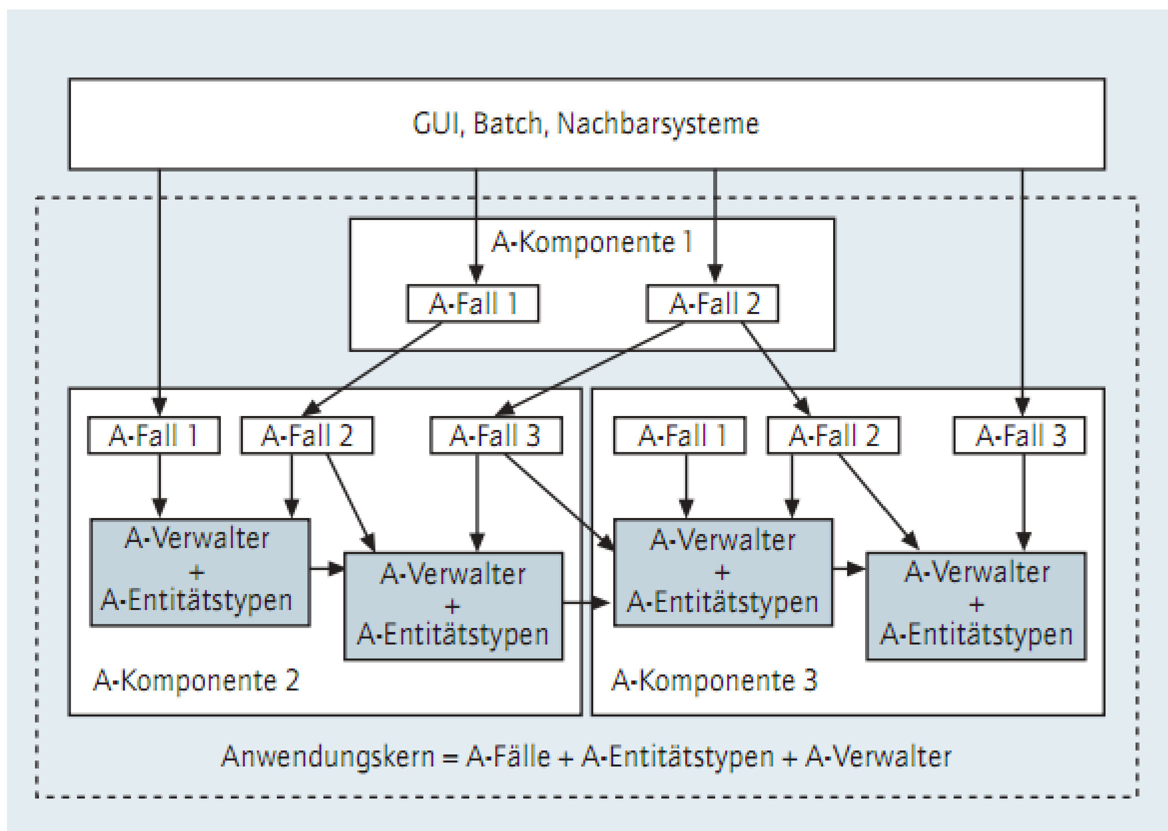

Abbildung1: Quasar: Die sd&m Standardarchitektur Teil 1. Online PDF S.16

2.5. Hibernate

Hibernate ist ein Open-Source Persistenz-Framework für Java-Anwendungen[11]. Es speichert Objekte in einer relationalen Datenbank. Hierfür müssen Mapping-Definitionen erstellt werden, um eine Abbildung der Eigenschaften einer Klasse, auf die Spalten einer Datenbanktabelle zu schaffen. Dies erfolgt in XML. Mit Hibernate kann man verschiedene Beziehungen zwischen Klassen abbilden. Zum Beispiel 1:1, 1:n und n:m Beziehungen, aber

[10] Siedersleben, J.: Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar. dpunkt.Verlag 2005. S. 176
[11] http://www.hibernate.org

auch Vererbung ist abbildbar. Hibernate bietet außerdem Caching, Lazy Loading und Transaktionen. Abfragen können in Hibernate unter anderem in einer eigenen Abfragesprache, der Hibernate Query Language (kurz HQL) erfolgen. Für das .NET-Framework gibt es eine Portierung namens NHibernate, für welche es eine API namens FluentNHibernate gibt, die es ermöglicht Mapping-Klassen in C# Code zu erstellen. FluentNHibernate wurde vom Autor dieser Arbeit mit folgenden Datenbanken getestet:

- SQLite
- Microsoft SQL-Server 2005, 2008

2.6. Windows Communication Foundation

Windows Communication Foundation, kurz WCF, ist eine Klassenbibliothek innerhalb des .NET-Frameworks. Sie stellt Klassen für die Kommunikation von verteilten Anwendungen zur Verfügung. Es gibt mehrere Nachrichtenmuster. Ein weit verbreitetes Muster ist das Request-response-Muster. Dabei fragt ein Endpunkt Daten bei einem anderen Endpunkt an, welcher auf die Anfrage antwortet. Man kann Datenverträge definieren, also Methoden in Schnittstellen kennzeichnen, sodass sie vom WCF-Dienst angeboten werden. Solch ein Dienst kann Metadaten zur Verfügung stellen, die z.B. über HTTP oder HTTPS abgefragt werden und zur automatischen Erstellung von Clients genutzt werden können. Die gängigste Methode zum Verschicken von Nachrichten ist, sie als SOAP-Nachricht zu codieren und per HTTP zu versenden. Es ist aber auch möglich Nachrichten ohne SOAP, also als reine XML-Daten oder mit JSON zu codieren.[12]

[12] http://msdn.microsoft.com/de-de/library/ms731082.aspx

3. Konzeption

In diesem Abschnitt wird zunächst die Idee des Frameworks vorgestellt. Anschließend werden die Anforderungen an das System betrachtet, welche auf Basis der in Abschnitt 2.1.3. untersuchten Browsergames, aufgestellt wurden.

3.1. Idee

Die Idee ist es, ein Framework zu erstellen, das neben den wichtigsten technischen Komponenten auch viele Anwendungskomponenten anbietet. Um ein Spiel zu erstellen, braucht man nicht mehr die komplette Spielelogik selbst zu implementieren. Das Framework soll genügend Komponenten anbieten um ein umfangreiches Spiel aufzusetzen. Der Entwickler soll dabei trotzdem die Möglichkeit haben an vielen Stellen Konfigurationen vorzunehmen, oder Teile durch eigene Implementierungen zu ersetzen. Das fertige Spiel soll nicht an einen Web-Server gebunden sein, sondern als alleinstehende Applikation laufen. Auf diese Art kann ein Spiel nicht nur über einen Web-Browser, sondern auch über einen Windows-Client bedient werden.

3.2 Funktionale Anforderungen

Benutzer

1. Benutzer sollen sich mit einem Nicknamen, Passwort und ihrer E-Mail Adresse am System registrieren können.
2. Das System speichert für jeden Benutzer einen Status. Es gibt mindestens die Status inaktiv, aktiv und admin. Nach dem Registrieren ist der Status auf inaktiv gesetzt.
3. Benutzer sollen nach dem Registrieren einen Aktivierungscode per E-Mail an ihre, bei der Registrierung angegebene, E-Mail Adresse geschickt bekommen.
4. Benutzer sollen ihren Aktivierungscode im System eingeben können. Nach korrekter Eingabe, wird ihr Status auf aktiv gesetzt.
5. Solange der Status eines Benutzers inaktiv ist, kann er am System keine Aktionen durchführen, bis auf eine neue Aktivierungsmail anzufordern.
6. Benutzer haben eine eindeutige Nummer. Beispiel: *User1*.

Sitzungen

7. Benutzer sollen sich mit ihrem Nicknamen und Passwort am System einloggen können.
8. Nach einem erfolgreichen Login, generiert das System ein Ticket mit einer eindeutigen ID und gibt das Ticket an den Benutzer zurück. Dieses Ticket wird für jede weitere Interaktion mit dem System benötigt.

9. Ein Ticket hat eine konfigurierbare Gültigkeitsdauer. Beispiel: 20 Minuten. Nach Ablauf der Zeit wird das Ticket aus dem System entfernt und ist somit nicht länger gültig.
10. Ein Daemon überprüft in regelmäßigen Abständen (konfigurierbare Zeitspanne) alle Sitzungen und entfernt abgelaufene.

Nachrichten

11. Benutzer sollen sich gegenseitig Nachrichten schicken können.
12. Nachrichten haben einen Absender, Empfänger, Betreff, Inhalt und einen Zeitstempel.
13. Benutzer sollen ihre Nachrichten löschen können.
14. Nachrichten haben eine eindeutige Nummer. Beispiel: *Msg1*.

Attribute

15. Es gibt Attribute die einen Namen und eine Beschreibung haben.
16. Jedes Attribut hat einen ganzzahligen Wert.

Ressourcen

17. Es gibt Ressourcen wie z.B. Holz oder Kohle.
18. Ressourcen sind ein Zahlungsmittel innerhalb des Spiels und werden beispielsweise für den Ausbau von Gebäuden benötigt.

Kampfeinheiten

19. Es gibt Kampfeinheiten. Diese haben einen Namen, eine Beschreibung, Kosten, Ressourcenkapazität, Bewegungskosten, Reisedauer, eine Menge von Attributen, eine Bauzeit und eine Menge von Bewegungstypen (siehe Punkt 76).
20. Kampfeinheiten haben eine eindeutige Nummer. Beispiel: *Unit1*.

Gebäude

21. Es gibt Gebäude. Diese haben einen Gebäudetyp, eine Stufe, eine Menge von Attributen, eine Bauzeit und Kosten.
22. Gebäude haben eine eindeutige Nummer. Beispiel: *Gebäude1*.
23. Gebäude können ausgebaut werden. Nach einem Ausbau ist die Stufe des Gebäudes um 1 höher. Der Gebäudetyp bleibt derselbe.
24. Gebäude werden einer Basis (siehe Punkt 27) zugeordnet und können daher auf jeder Basis erneut ausgebaut werden.
25. Gebäudetypen bestehen aus einem Namen und einer Beschreibung.
26. Minen sind Gebäude mit einer Ressourcenproduktion und einem Ressourcenverbrauch.

Basis

27. Jeder Benutzer bekommt nach dem Registrieren eine Startbasis.
28. Eine Basis bekommt beim Anlegen eine Menge von Gebäuden. Darin enthalten sind Gebäude der Stufe 0 eines jeden Gebäudetyps.
29. Zusätzlich bekommt eine Basis beim Anlegen eine Menge von Kampfeinheiten. In der Menge ist für jede vorhandene Kampfeinheit ein ganzzahliger Wert enthalten, der beschreibt wie viele Kampfeinheiten auf der Basis sind. Zu Beginn ist dieser Wert 0.

30. Eine Basis hat einen Namen, eine Beschreibung, eine Menge von Gebäuden, eine Menge von Kampfeinheiten, Ressourcen, einen Zeitstempel der letzten Aktion und einen Bezeichner der angibt, ob es die Startbasis ist.
31. Kampfeinheiten können auf der Basis ausgebildet werden. Nach Fertigstellung wird die Anzahl des entsprechenden Wertes der Kampfeinheit erhöht.
32. Beim Ausbau von Gebäuden oder Ausbilden von Kampfeinheiten werden die Kosten von den Ressourcen der beteiligten Basis abgezogen.
33. Jeder Benutzer kann mehrere Basen haben.
34. Jede Basis hat eine eindeutige Nummer. Beispiel: *Basis1*.

Charakter

35. Jeder Benutzer bekommt nach dem Registrieren einen Charakter.
36. Ein Charakter hat einen Namen, eine Stufe, Erfahrungspunkte, Lebensenergie, ein Inventar, eine Menge von Ausrüstungsplätzen, eine Menge von Attributen und Ressourcen.
37. Die Formel zur Berechnung der Lebensenergie soll konfigurierbar sein. Beispiel: *Charakterstufe * Wert eines bestimmten Attributs*.
38. Der Charakter soll durch Erhalt von Erfahrungspunkten Stufen aufsteigen können. Bei jeder Stufe können unterschiedlich viele Erfahrungspunkte für den Aufstieg nötig sein.
39. Der Wert der Attribute soll durch Abgabe von Ressourcen erhöht werden können. Die Formel zur Berechnung der Kosten soll konfigurierbar sein. Eventuelle Forschungen oder Items, welche die Attribute des Charakters erhöhen, sollen von der Formel nicht berücksichtigt werden.
40. Jeder Charakter hat einige Ausrüstungsplätze für Items (siehe Punkt 45).
41. Jeder Ausrüstungsplatz hat einen Ausrüstungstyp mit Namen und Beschreibung. Beispiel: *Brust, Kopf und Hände*.
42. Der Charakter hat ein Inventar mit einer konfigurierbaren Maximalgröße. Im Inventar können Items abgelegt werden.
43. Der Charakter kann Quests absolvieren (siehe Punkt 49).
44. Jeder Charakter hat eine eindeutige Nummer. Beispiel: *Charakter1*.

Items

45. Es gibt verschiedene Items. Items haben einen Namen, eine Beschreibung, eine Mindeststufe, einen Einkaufspreis, einen Verkaufspreis, eine Menge von Attributen und einen Ausrüstungstyp.
46. Items können nur ausgerüstet werden, wenn der Charakter die erforderliche Mindeststufe des Items erfüllt.
47. Beim Ausrüsten eines Items werden die Werte der Attribute des Items auf die Werte der Attribute des Charakters dazu addiert. Beim Ablegen wieder abgezogen.
48. Jedes Item hat eine eindeutige Nummer. Beispiel: *Item1*.

Quests

49. Es gibt verschiedene Quests. Quests haben einen Namen, eine Beschreibung, eine Stufe, eine Dauer und können als Belohnung ein Item, Ressourcen oder Erfahrungspunkte haben.
50. Ein Charakter kann nur eine Quest zurzeit absolvieren.
51. Quests werden absolviert, indem der Benutzer eine Quest startet und die Questdauer abwartet.

52. Gestartete Quests können abgebrochen werden.
53. Wenn eine Quest erfolgreich absolviert wird, werden die Erfahrungspunkte auf die Erfahrungspunkte des Charakters addiert, die Ressourcen auf die Ressourcen des Charakters addiert und das Item in das Inventar des Charakters gelegt.
54. Jede Quest hat eine eindeutige Nummer. Beispiel: *Quest1*.

Forschungen

55. Es gibt Forschungen. Diese haben einen Forschungstyp, Kosten, eine Stufe, eine Ausbildungszeit und eine Menge von Attributen.
56. Der Forschungstyp besteht aus einem Namen und einer Beschreibung.
57. Forschungen können erforscht werden, wobei die Kosten von den Ressourcen der ausgewählten Basis abgezogen werden.
58. Jede Forschung hat eine Menge von Gebäudetypen und Kampfeinheiten, auf die sich die Forschung auswirkt. Zusätzlich kann sich eine Forschung auch auf den Charakter auswirken.
59. Die Attribute der Forschungen beziehen sich auf die Attribute der Gebäude, Kampfeinheiten und die des Charakters, welche von den Forschungen beeinflusst werden sollen.
60. Der Wert der Attribute wird als Prozent betrachtet und gibt an um wie viel Prozent die Werte der Attribute der Gebäude, Kampfeinheiten und die des Charakters verändert werden sollen.
61. Nachdem eine Forschung erfolgreich ausgebildet wurde, werden die Werte der Attribute der jeweiligen Gebäude, Kampfeinheiten und die des Charakters angepasst.
62. Forschungen werden einem Benutzer direkt zugeordnet, denn sie wirken sich auf alle Basen eines Benutzers aus.
63. Jede Forschungsstufe des gleichen Forschungstyps kann pro Benutzer nur einmal erforscht werden.
64. Jede Forschung hat eine eindeutige Nummer. Beispiel: *Forschung1*.

Technologiebaum

65. Es gibt Technologiebäume für Kampfeinheiten, Gebäude und Forschungen.
66. Kampfeinheiten, Gebäude und Forschungen können von Gebäuden und Forschungen abhängig sein. Ein Technologiebaum definiert diese Abhängigkeiten. Beispiel: *Unit1* kann auf *Basis1* nur dann ausgebildet werden, wenn der Benutzer *Forschung1* erforscht und auf *Basis1 Gebäude1* ausgebaut hat.

Handel

67. Ein Benutzer soll Ressourcenangebote anlegen können. Dabei bietet er Ressourcen von einer seiner Basen an und definiert einen Ressourcenpreis.
68. Ein Benutzer soll Ressourcenangebote von anderen Benutzern annehmen können. Dabei muss der Benutzer eine seiner Basen auswählen. Wenn auf den Basen des Verkäufers und Käufers genügend Ressourcen vorhanden sind, werden die angebotenen Ressourcen auf der Basis des Verkäufers abgezogen und auf der Basis des Käufers gutgeschrieben. Der Ressourcenpreis wird auf der Basis des Verkäufers gutgeschrieben und auf der Basis des Käufers abgezogen.
69. Ein Benutzer soll Itemangebote anlegen können. Dabei wählt er ein Item aus dem Inventar seines Charakters aus und definiert einen Ressourcenpreis. Das angebotene Item wird sofort aus dem Inventar entfernt.

70. Ein Benutzer soll Itemangebote von anderen Benutzern annehmen können. Wenn der Charakter des Käufers einen freien Platz im Inventar und genügend Ressourcen hat, wird das angebotene Item in das Inventar des Charakters des Käufers gelegt, der Ressourcenpreis vom Charakter des Käufers abgezogen und beim Charakter des Verkäufers gutgeschrieben.
71. Benutzer sollen ihre eigenen Angebote abbrechen können.
72. Jedes Angebot hat eine eindeutige Nummer. Beispiel: *Angebot1*.

Spielfeld

73. Ein Punkt hat drei Koordinaten (X, Y ,Z).
74. Einem Spielfeldpunkt sind ein Punkt, eine Basis und ein Benutzer zugeordnet.
75. Beim Registrieren eines Benutzers, wird für die Startbasis ein zufälliger noch nicht belegter Punkt erzeugt.

Bewegungen

76. Es gibt die Bewegungstypen Angriff, Spionage, Bewegung, Transport und Kolonisierung.
77. Eine Bewegung ist immer von genau einem Bewegungstyp. Sie hat eine Basis als Startort und einen Punkt als Zielort.
78. Ein Benutzer soll Bewegungen starten können. Dazu wählt er eine seiner Basen aus, gibt eine Menge von Kampfeinheiten an, trägt die Zielkoordinaten ein und wählt einen Bewegungstyp aus.
79. Die Bewegungskosten werden von den Ressourcen der ausgewählten Basis abgezogen.
80. Die Reisedauer bis zum Ziel ist die Reisedauer der Kampfeinheit mit der höchsten Reisedauer * Entfernung zwischen Start- und Zielort.
81. Bei dem Bewegungstyp Angriff, soll der Benutzer angeben können ob sein Charakter an dem Angriff beteiligt sein soll. Dies geht nur, wenn der Charakter zu dem Zeitpunkt keine Quest absolviert und an keinem anderen Angriff beteiligt ist.
82. Wenn die Kampfeinheiten bei einem Angriff am Zielort angekommen sind, wird ein Kampf ausgetragen (siehe Punkt 90) und es werden Kampfberichte in Form einer Nachricht an beide beteiligten Benutzer geschickt. Die nach dem Kampf noch übrig gebliebenen Kampfeinheiten werden zurück zur Startbasis geschickt.
83. Bei einem Angriff stiehlt man Ressourcen von der Zielbasis, wenn Ressourcen vorhanden sind, aber höchstens so viele Ressourcen, dass die Gesamtkapazität aller Kampfeinheiten nicht überschritten wird.
84. Bei einer Spionage bekommt der Benutzer der diese gestartet hat einen Spionagebericht, in dem er über die Kampfeinheiten und Ressourcen auf der Zielbasis informiert wird. Der ausspionierte Benutzer wird über die Spionage mittels Nachricht informiert.
85. Bei einem Transport werden Ressourcen von der Starbasis abgezogen und bei der Zielbasis gutgeschrieben.
86. Eine Bewegung ist wie ein Transport, nur bleiben zusätzlich die verschickten Kampfeinheiten bei der Zielbasis.
87. Bei den Bewegungstypen Angriff, Spionage, Transport und Bewegung, muss sich am Zielort eine Basis eines Benutzers befinden. Bei Angriff und Spionage dürfen die Basen bei Start- und Zielort nicht demselben Benutzer gehören.

88. Bei einer Kolonisierung darf sich am Zielort keine Basis befinden. Bei Ankunft am Zielort wird für den Benutzer eine neue Basis erzeugt. Mitgeschickte Kampfeinheiten und Ressourcen bleiben auf der neuen Basis.
89. Grundsätzlich gilt für Bewegungen, dass bei der Startbasis genügend Ressourcen und Kampfeinheiten vorhanden sein müssen. Zusätzlich muss mindestens eine Kampfeinheit den entsprechenden Bewegungstyp besitzen.

Kampfsystem

90. Ein Benutzer soll die Möglichkeit haben die Basen von anderen Benutzern mit Kampfeinheiten anzugreifen.
91. Der Charakter des Angreifers kann am Kampf beteiligt sein, der Charakter des Opfers nimmt am Kampf teil, sofern der Charakter nicht selber an einem Angriff teilnimmt und keine aktiven Quests hat.
92. Bei einem Kampf werden alle Attribute des Angreifers und Verteidigers berücksichtigt. Auf der Seite des Angreifers gibt es die Attribute der am Kampf beteiligten Kampfeinheiten und die Attribute des optionalen Charakters. Auf der Seite des Verteidigers gibt es die Attribute aller Kampfeinheiten die sich auf der Basis befinden, die Attribute der Gebäude, sowie die Attribute des optionalen Charakters.
93. Die genaue Formel zur Berechnung des Kampfes soll konfigurierbar sein.
94. Nach einem Kampf können beide Parteien Kampfeinheiten verloren haben (die Anzahl der Kampfeinheiten wird reduziert).
95. Der Charakter kann jedoch nicht sterben, er bleibt immer vorhanden.

Gilde

96. Benutzer sollen die Möglichkeit haben Gilden zu gründen.
97. Benutzer sollen Gilden beitreten können.
98. Gilden haben ein eigenes Rechtesystem, welches von den Mitgliedern angepasst werden kann.
99. Gilden haben ein eigenes Ressourcenkonto, welches von den Mitgliedern verwaltet wird.

3.3. Nichtfunktionale Anforderungen

- Zeitverhalten - Die Antwort vom System soll ohne spürbare Verzögerung stattfinden.
- Zuverlässigkeit - Das System soll möglichst ohne offline-Zeit verfügbar sein.
- Skalierbarkeit - Das System soll mit einer großen Anzahl von Benutzern umgehen können.
- Sicherheit - Angreifern soll es nicht möglich sein auf Daten anderer Benutzer zuzugreifen.
- Korrektheit - Das System soll fehlerfreie Ergebnisse liefern.
- Erweiterbarkeit - Das System soll leicht zu erweitern sein.

4. Entwurf

In diesem Abschnitt wird die Softwarearchitektur des Frameworks diskutiert. Zunächst wird auf die Architektur des Anwendungskerns eingegangen, wobei jede erzeugte Komponente detailliert beschrieben wird. Im Anschluss wird betrachtet wie sich der Anwendungskern in das Gesamtsystem einfügt.

4.1. Architektur des Anwendungskerns

Der Anwendungskern ist das Herz des Browsergame Frameworks und enthält alle fachlichen Komponenten. Auf Basis der in Abschnitt 3 vorgestellten Anforderungsanalyse, wurden folgende Komponenten angelegt, die im Folgenden erläutert werden:

- Rechteverwaltungs-Komponente
- Sitzungsverwaltungs-Komponente
- Accountverwaltungs-Komponente
- Attribut-Komponente
- Auftrags-Komponente
- Charakter-Komponente
- Einheiten-Komponente
- Forschungs-Komponente
- Gilden-Komponente
- Handels-Komponente
- Item-Komponente
- Kampf-Komponente
- Nachrichten-Komponente
- Quest-Komponente
- Ressourcen-Komponente
- Spielfeld-Komponente
- Technologiebaum-Komponente
- User-Komponente

Abbildung2 zeigt alle Komponenten des Anwendungskerns mit ihren importierten und exportierten Schnittstellen.

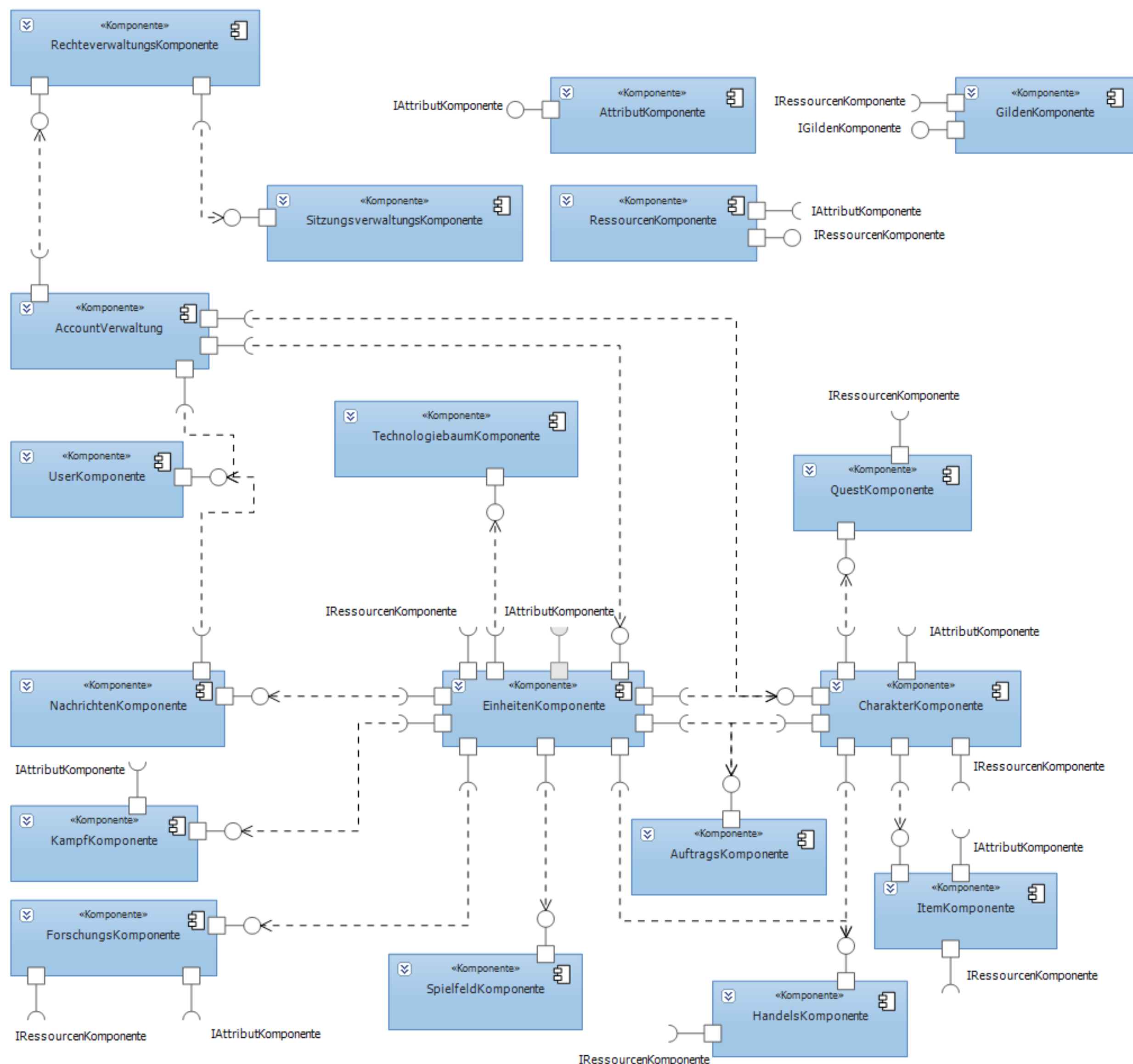

Abbildung2: Komponentenschnitt

Alle Komponenten sind lose gekoppelt. Entitätstypen werden über fachliche Schlüssel (Datentypen) referenziert. Es werden also nur dienstorientierte Schnittstellen zwischen Komponenten verwendet.[13] Um Abhängigkeiten zu reduzieren, wurden alle fachlichen Schlüssel in einer gesonderten Komponente untergebracht.[14] Dazu wurde neben den Komponenten aus dem Komponentenschnitt noch eine Komponente namens *Fachliche-Schlüssel* angelegt. Laut Quasar sollen alle Anwendungsfälle in speziellen Anwendungsfall-Klassen untergebracht sein. Da das Browsergame Framework bisher nur als Prototyp implementiert ist, wurde aus zeitlichen Gründen darauf verzichtet alle Anwendungsfälle auszulagern. Deshalb sind alle Anwendungsfälle in der Komponenten-Klasse untergebracht. Die Komponenten-Klasse implementiert die Komponenten-Schnittstelle und delegiert Aufrufe an die Verwalter und Anwendungsfälle. In diesem Fall beinhaltet sie die Anwendungsfälle selbst. Da WCF verwendet wird um das Browsergame Framework zu hosten, bedienen wir uns der Serialisierung von Objekten um Daten zwischen Clients und System auszutauschen.

[13] Vgl. Siedersleben, J.: Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar. dpunkt.Verlag 2005. S. 184

[14] Vgl. Siedersleben, J.: Quasar: Die sd&m Standardarchitektur Teil 1 Online PDF S. 17

Nach Quasar soll man die Fachlichkeit frei von technischen Details halten. Serialisierung ist ein technisches Konstrukt, daher wäre es sinnvoll die Annotationen für die Serialisierung in Transportklassen auszulagern. Allerdings wurde im Prototyp, aus demselben Grund weshalb auf Anwendungsfall-Klassen verzichtet wurde, auf Transportklassen verzichtet.

4.2. Komponentenentwurf im Detail

Es folgt eine detaillierte Diskussion über den Entwurf jeder Komponente, mit ihren Vor- und Nachteilen.

4.2.1. Attribut-Komponente

Viele Entitäten wie z.B. Gebäude, Kampfeinheiten oder der Charakter haben Attribute. Attribute sind z.B. Angriffskraft, Ausdauer oder Geschicklichkeit. Damit Attribute nicht statisch in der Implementierung definiert werden, sondern konfigurierbar sind, wurde dafür ein Datentyp erstellt. So ist es möglich beliebig viele Attribute zu definieren. Der Datentyp Attribut wird über einen Namen identifiziert und kann zusätzlich einen beschreibenden Text haben. Außerdem gibt es einen weiteren Datentyp namens *AttributTupel*. Dieser bringt ein Attribut mit seinem Wert in Verbindung, z.B. *AttributTupel(Angriffskraft, 20)*. Da die Attribut-Komponente keine fachliche Logik besitzt, gibt es hier keine Anwendungsfälle, sondern lediglich einen Verwalter. Der Verwalter ist in diesem Fall für das Anlegen und Auslesen von Attributen zuständig. Das Auslesen erfolgt über den Namen des Attributes und das konkrete Objekt wird als Parameter zurückgegeben. Dies verletzt nicht die Forderung nach loser Kopplung, da das Attribut ein Datentyp ist. Die Innensicht der Komponente wird in Abbildung3 als Kompositionsstrukturdiagramm gezeigt.

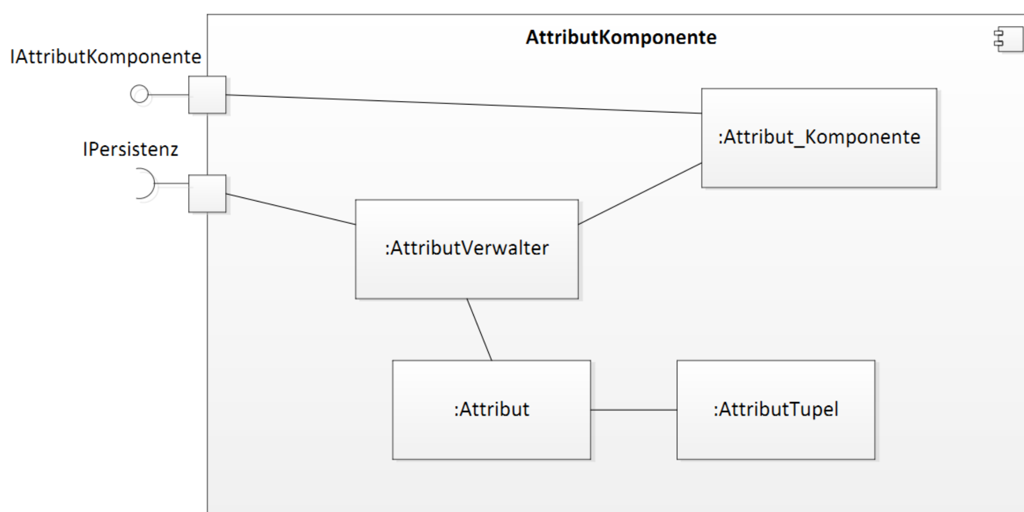

Abbildung3: Kompositionsstrukturdiagramm der Attribut-Komponente

Jede Komponente des Anwendungskerns ist nach diesem Muster aufgebaut.

4.2.2. Forschungs-Komponente

Die Forschungs-Komponente bietet Forschungen an, welche die Attribute von Gebäuden, Kampfeinheiten und die des Charakters erhöhen oder vermindern. Es ist frei definierbar auf welche konkreten Entitäten sich die Forschungen auswirken. Forschungen haben einen Forschungstyp und eine Stufe. Der Forschungstyp hat einen Namen und einen beschreibenden Text. Beim Ausbilden einer Forschung eines bestimmten Typs wird die Stufe um eins erhöht. Außerdem kostet die Erforschung Ressourcen (siehe Abschnitt 4.2.3). Zusätzlich haben Forschungen eine Menge von Attributen. Der Wert der Attribute gibt in diesem Fall allerdings an um wie viel Prozent das jeweilige Attribut einer Entität erhöht oder vermindert wird. Beispiel: *Forschung1* wirkt sich auf den Charakter aus und hat das Attribut *Stärke* mit einem Wert von 100. Der Charakter hat das Attribut *Stärke* mit einem Wert von 20. Nach der Erforschung hat das Attribut *Stärke* des Charakters einen Wert von 40, da die Forschung dieses um 100% erhöht.
Ausbildungszeiten von Kampfeinheiten oder Bauzeiten von Gebäuden werden auch als Attribut abgebildet. Dies ermöglicht es mit den Zeitwerten genauso zu verfahren wie mit herkömmlichen Attributen. Forschungen werden über einen fachlichen Schlüssel namens *ForschungID* referenziert. Außerdem verwaltet die Forschungs-Komponente die erforschten Forschungen für jeden Benutzer. Dafür gibt es den Datentyp *UserForschung*. Dieser speichert die User-ID zusammen mit einer Liste aller Forschungen die der Benutzer bereits erforscht hat. Dies ist notwendig damit Benutzer ihre Forschungsstufen ausbauen können, weil ermittelt werden muss welche Stufe die nächste zu erforschende ist.

4.2.3. Ressourcen-Komponente

Der Ausbau von Gebäuden, Forschungen oder Kampfeinheiten kostet Ressourcen. Ressourcen sind z.B. Holz, Metall oder Kohle. Diese werden von der Ressourcen-Komponente verwaltet. Ressourcen werden als Attribute abgebildet, damit sie im Kontext der Forschungen genauso wie herkömmliche Attribute behandelt werden können. Somit ist es möglich beliebig viele Ressourcen zu definieren. Allerdings gibt es einen zusätzlichen Datentyp namens *Ressource*, der ein Attribut als Ressource markiert, damit Ressourcen dennoch von Attributen unterschieden werden können. Ein weiterer Datentyp namens *Ressourcen* kapselt alle Ressourcen und definiert Rechenoperatoren zum Addieren, Subtrahieren und Multiplizieren. Beim Addieren werden alle Ressourcen desselben Typs miteinander addiert. Beispiel: *Ressourcen(R1.Holz+R2.Holz, R1.Kohle+R2.Kohle)*. Die Subtraktion folgt dem gleichen Muster. Bei der Multiplikation werden entweder alle enthaltenen Ressourcen mit einem Faktor multipliziert, oder man gibt explizit eine Ressource an. In diesem Fall wird nur die angegebene Ressource mit dem Faktor multipliziert. Die Gleichheit wird über die Werte aller Ressourcen definiert.

4.2.4. Einheiten-Komponente

Die Einheiten-Komponente ist die größte und komplexeste Komponente des Browsergame Frameworks. Sie umfasst die Gebäude, die Basis und die Kampfeinheiten. Ein Gebäude hat einen Gebäudetyp, eine Bauzeit, beliebig viele Attribute, eine Stufe und Kosten. Ein speziellerer Typ des Gebäudes ist die Mine. Die Mine erbt von Gebäude und hat zusätzlich

eine Produktion pro Stunde und einen Verbrauch pro Stunde. Diese sind als Ressourcen angegeben. Da Forschungen die Gebäudetypen, auf die sich beziehen, speichern, die Einheiten-Komponente aber die Forschungs-Komponente aufruft, wurde der Datentyp *GebäudeTyp* als Nulltyp in eine spezielle Nulltypen-Komponente ausgelagert, weil Microsoft Visual Studio zirkuläre Abhängigkeiten zwischen Komponenten verbietet. Eine Alternative wäre es, sich auf den primitiven Datentyp String zu beschränken, da ein Gebäudetyp lediglich an seinem Namen identifiziert wird.
Die Basis ist die Einheit, auf der Gebäude oder Kampfeinheiten gebaut werden. Sie besitzt die Ressourcen, von welchen die Kosten für den Bau von Einheiten abgezogen werden. Die Menge der Ressourcen, die eine Basis nach dem Erstellen hat, wird in der Konfigurationsdatei angegeben.
Die erste Entwurfsidee für die Gebäude bestand darin, eine Faktentabelle zu verwenden, in welcher Gebäudetypen und Stufen gespeichert sind. Die Faktentabelle sollte unveränderlich sein und praktisch als Schablone für die Gebäude der Benutzer dienen. Jeder Benutzer sollte seine eigene Gebäudekopie für die aktuelle Ausbaustufe haben, auf welcher Forschungen angewandt werden. Die durch Forschungen veränderten Attribute der Gebäude für jeden Benutzer individuell aus der Faktentabelle zu errechnen, wäre zu rechenintensiv gewesen, da dies für viele Aktionen des Benutzers von Nöten wäre. Der Nachteil dieser Entwurfsidee ist, dass bei einem Gebäudeausbau auf eine höhere Stufe, alle bisherigen Forschungen erneut auf die neue Gebäudekopie angewandt werden müssten. Aus diesem Grund werden für einen Benutzer direkt die Gebäudekopien für alle Ausbaustufen angelegt. Weil sich Forschungen auf alle Basen eines Benutzers auswirken, benötigt man diese Gebäudekopien nur einmal. Wenn ein Benutzer eine neue Basis bekommt, werden also dieselben Gebäudekopien verwendet. Wenn nun eine Forschung fertig wird, wird sie direkt auf alle Stufen der Gebäudetypen, auf welche sie sich auswirkt, angewandt. Forschungen müssen also nur noch einmal bei ihrer Fertigstellung berechnet werden. Dies ist ein Vorteil, weil die Ausbaustufen der Forschungen begrenzt sind, Benutzer aber theoretisch unbegrenzt viele Basen errichten können, auf denen wieder alle Gebäudestufen zur Verfügung stehen. Genauso verhält es sich mit den Kampfeinheiten. Ein Benutzer bekommt eigene Kopien der originalen Kampfeinheiten, auf welche die Forschungen angewandt werden. Jede Basis benutzt somit dieselben Kopien der Kampfeinheiten. Dies funktioniert nur, weil sich Forschungen auf alle Basen auswirken. Würde sich eine Forschung nur auf die Basis auf der sie erforscht wurde auswirken, hätte dies weitreichende Folgen. Wenn ein Benutzer z.B. auf 2 Basen, durch Forschungen unterschiedlich starke Kampfeinheiten desselben Typs hätte, würden die Kampfeinheiten beim Verschicken zur anderen Basis ihre Stärke verändern. Benutzer könnten diesen Umstand ausnutzen, indem sie andere Benutzer nur noch ausgehend von der Basis mit den stärksten Kampfeinheiten angreifen würden. Eine andere Möglichkeit wäre, eine komplexere Verwaltung der Kampfeinheiten, sodass die Kampfeinheiten ihre Stärke beim Verschicken beibehalten. Diese Ansätze wurden im Browsergame Framework allerdings nicht angewandt.
Kampfeinheiten haben eine Menge von Bewegungstypen. Diese sind Angriff, Transport, Bewegung, Spionage und Kolonisierung. Die Bewegung von Kampfeinheiten kostet Ressourcen. Dafür haben Kampfeinheiten einen Ressourcenverbrauch pro Koordinate (siehe Abschnitt 4.2.8). Die Kosten für die Bewegung werden über diesen Verbrauch und die Distanz von Start und Ziel berechnet. Bei einem Angriff wählt ein Benutzer eine Menge von Kampfeinheiten von einer seiner Basen aus und schickt sie zu einer Basis eines anderen Benutzers. Bei der Ankunft an der Zielbasis wird ein Kampf berechnet (siehe Abschnitt 4.2.5) und die überlebenden Einheiten werden zurück geschickt. Der Angreifer kann dabei

Ressourcen von seinem Gegner stehlen. Dazu haben Kampfeinheiten eine Ressourcenkapazität. Es können nur maximal so viele Ressourcen gestohlen werden, wie die Summe der Kapazitäten aller überlebenden Kampfeinheiten zulässt. Bei einem Angriff wird außerdem ein Kampfbericht erzeugt und an beide beteiligten Benutzer als Nachricht geschickt. Bei einem Transport kann man Ressourcen von einer Basis zu einer beliebigen anderen Basis transportieren. Die Einheiten kehren wieder zur Ursprungsbasis zurück. Bei einer Bewegung bleiben die verschickten Einheiten bei der Zielbasis. Die Einheiten gehören von da an dem Benutzer der Zielbasis. Die Spionage erlaubt es einem Benutzer herauszufinden wie viele Ressourcen und Kampfeinheiten sich auf einer anderen Basis befinden. Durch eine Kolonisierung ist es Benutzern möglich neue Basen zu erstellen. Dazu schickt man die Einheiten zu einer beliebigen Koordinate auf der sich keine Basis befindet. Wenn die Einheiten angekommen sind und auf dem Punkt in der Zwischenzeit eine Basis errichtet wurde, werden alle Einheiten zurück geschickt. Man kann Ressourcen und Kampfeinheiten mitschicken, welche auf der neuen Basis bleiben. Die Kampfeinheiten die den Bewegungstyp Kolonisierung enthalten, verschwinden allerdings. Wenn man die Einheit für die Kolonisierung sehr teuer macht, kann man verhindern, dass sich Benutzer sehr schnell beliebig viele Basen errichten.
Bei jeder Aktion eines Benutzers wird ein Update ausgeführt. Das Update berechnet die Bewegungen und die seit dem letzten Update produzierten Ressourcen. Des Weiteren wird bei einem Update überprüft ob Gebäude, Forschungen oder Kampfeinheiten fertiggestellt wurden (siehe Abschnitt 4.2.7). Bei einem Angriff oder einer Spionage auf einen anderen Benutzer, muss für diesen vorher ein Update berechnet werden. Der Angriff oder die Spionage müssen auf dem aktuellen Zustand berechnet werden. Benutzer könnten sonst ihre Kampfeinheiten vor dem offline gehen weg schicken, damit diese nicht zerstört werden können. Zur Veranschaulichung folgt Abbildung4, die das fachliche Datenmodell der Einheiten-Komponente darstellt.

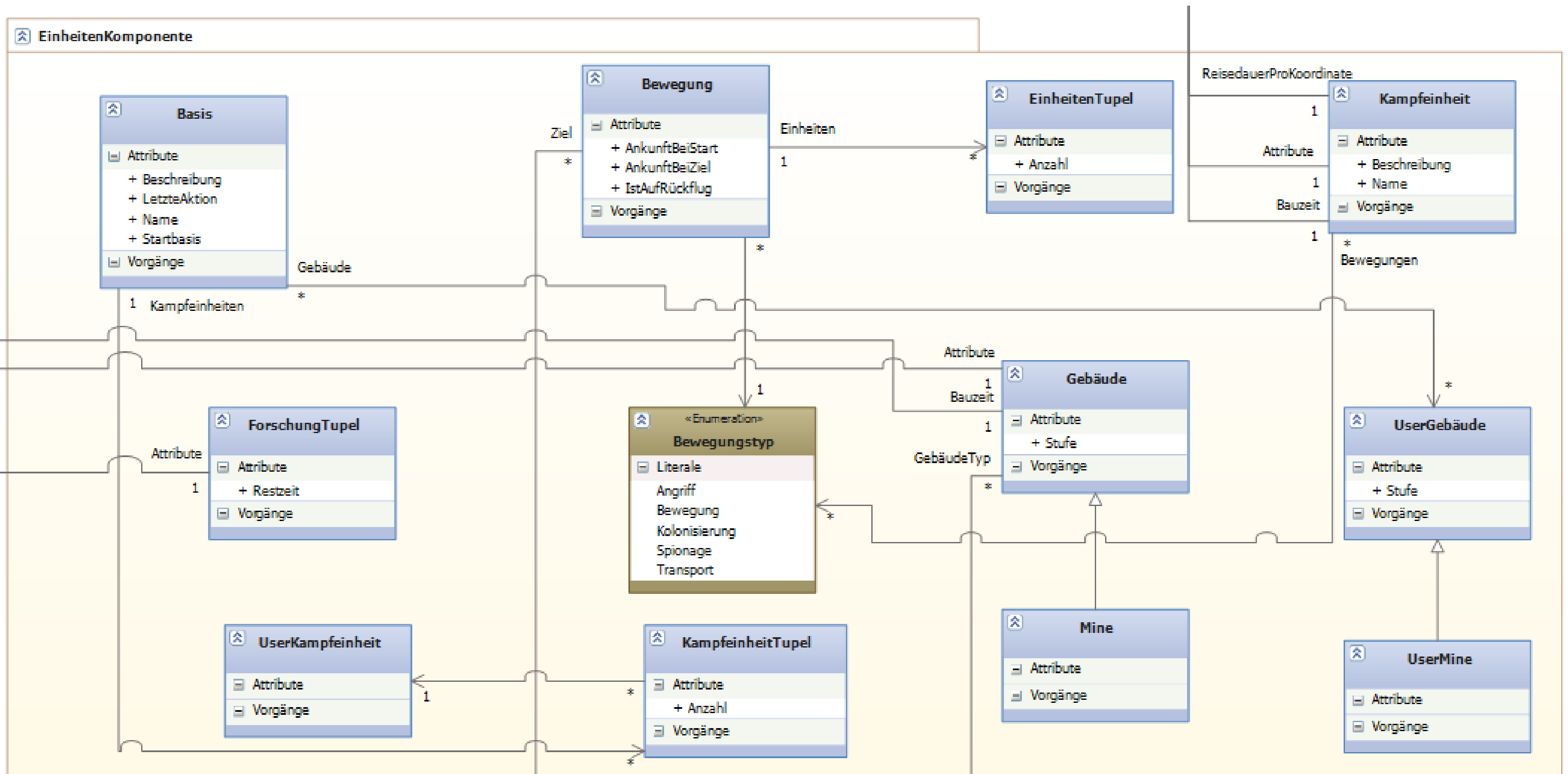

Abbildung4: Fachliches Datenmodell der Einheiten-Komponente

4.2.5. Kampf-Komponente

Die Kampf-Komponente ist für das Berechnen von Kämpfen zuständig. Es gibt eine abstrakte Klasse von der man erben kann und von der man eine Methode überschreiben muss. So kann jeder seine eigene Kampfberechnung implementieren. Die Methode definiert Parameter für alle Kampfrelevanten Informationen. Der Rückgabetyp gibt an, wer gewonnen, wer verloren hat und wie viele Kampfeinheiten jeweils übrig geblieben sind. Damit das Browsergame Framework aber voll funktionsfähig ist, gibt es eine fertige Implementierung die man einfach benutzen kann (siehe Abschnitt 5). Welche konkrete Implementierung verwendet werden soll, wird im Kompositionsmanager (siehe Abschnitt 4.4) festgelegt.

4.2.6. Charakter-Komponente

Die Charakter-Komponente ist die Komponente welche die Rollenspiel-Elemente beinhaltet. Der Charakter hat Attribute, konfigurierbare Ausrüstungsplätze und ein Inventar. Da der Charakter nicht alle existierenden Attribute hat, benötigt man eine Art Zuweisung. Dies erfolgt über einen speziellen Datentyp, welcher eine Liste von Attributen, die den Attributen des Charakters entsprechen, speichert. Ausrüstungsplätze könnten z.B. Brust, Kopf, oder Hände sein. Es gibt Items die man ausrüsten, oder im Inventar ablegen kann und welche die Attribute des Charakters verbessern. Items können gekauft und verkauft werden. Die Preise dafür sind als Ressourcen im Item selber definiert. Damit Benutzer nicht einfach die besten Items kaufen können, gibt es eine Klasse *ItemAuswahl*, in der eine in der Konfigurationsdatei definierte Anzahl von Items, die der Benutzer kaufen kann, gespeichert wird. Diese Auswahl wird bei jedem Stufenaufstieg des Charakters neu generiert und wählt zufällig Items aus der Datenbank, die der Stufe des Charakters entsprechen, aus. Ein Benutzer kann auch durch Bezahlung von Ressourcen eine neue Auswahl erhalten. Die Attribute des Charakters können ebenso durch Bezahlung von Ressourcen verbessert werden. Für diese Zwecke hat ein Charakter ein eigenes Ressourcenkonto. Man kann Formeln zur Berechnung der Lebensenergie des Charakters und zur Bestimmung des Ressourcenpreises für Attribut-Upgrades definieren. Die Lebensenergie wird über die Stufe des Charakters und ein beliebiges Attribut berechnet. In die Kosten für die Attribut-Upgrades fließt der Wert des jeweiligen Attributs mit ein. Die Stufe des Charakters kann durch Erhalt von Erfahrungspunkten aufsteigen, welche man durch das erfolgreiche Abschließen von Quests erhält. Der Charakter kann nur eine Quest zurzeit absolvieren. Wie die Konfiguration der Ausrüstungsplätze, der Lebensenergie und der Attribut-Upgrades konkret funktioniert, wird in Abschnitt 5, welcher sich mit der Implementierung beschäftigt, erläutert. Zwischen dem Charakter und der Startbasis, welche ein Benutzer nach der Registrierung erhält, können Ressourcen beliebig verschoben werden. Dies funktioniert nur mit der Startbasis, weil Benutzer den Charakter sonst zum Transport von Ressourcen zwischen Basen verwenden könnten und Kampfeinheiten somit für den Transport überflüssig wären. Da aber Kämpfe zwischen Charakteren zurzeit nicht implementiert sind, sodass man keine Ressourcen von einem Charakter erbeuten kann, ist es möglich Ressourcen von Basen auf dem Charakter sicher zu verwahren, ohne dass sie gestohlen werden können. Der Charakter kann allerdings an Angriffen auf andere Basen beteiligt sein, jedoch nur bei einem zurzeit und wenn keine Quest aktiv ist.

4.2.7. Auftrags-Komponente

Die Auftrags-Komponente verwaltet die Aufträge für Quests, Forschungen, Gebäude und Kampfeinheiten. Für Quests, Forschungen und Gebäude wird der Fertigstellungszeitpunkt gespeichert. Bei einem Update wird überprüft, ob dieser Zeitpunkt erreicht worden ist. Mit den Kampfeinheiten verhält es sich etwas komplizierter. Da man beliebig viele Kampfeinheiten gleichzeitig zur Ausbildung beauftragen kann, muss hier zusätzlich die Anzahl gespeichert werden. Dennoch wird nur eine Einheit zurzeit ausgebildet, weshalb man auch die Ausbildungszeit einer Einheit speichern muss. Der Fertigstellungszeitpunkt bezieht sich hier auf die aktuell in Ausbildung befindliche Einheit. Wenn dieser Zeitpunkt bei einem Update erreicht wurde, wird berechnet wie viele zusätzliche Einheiten in der Zwischenzeit fertiggestellt wurden und zu welchem Zeitpunkt die nächste Einheit fertig wird. Auf jeder Basis kann der Einfachheit halber nur ein Kampfeinheitentyp und ein Gebäudetyp gleichzeitig zum Bau beauftragt werden. Bei den Forschungen kann unabhängig von den Basen, für einen Benutzer nur eine Forschung zurzeit erforscht werden.

4.2.8. Spielfeld-Komponente

Die Spielfeld-Komponente stellt eine Karte mit drei Koordinaten dar. Jeder Basis ist ein eindeutiger Punkt auf dieser Karte zugeordnet. Der Abstand zwischen zwei Punkten entspricht der euklidischen Distanz[15], welche über den Satz des Pythagoras berechnet wird. Ein Punkt wird durch den Datentyp *Punkt3D* abgebildet, welcher lediglich die drei Werte für die X, Y und Z Koordinaten speichert. Die Zuordnung von Punkten und Basen erfolgt über den Datentyp *SpielfeldPunkt*. Dieser speichert neben der Basis-ID und einem *Punkt3D* auch die User-ID. Dies ist redundant weil man den Benutzer über die Basis herausfinden kann. Da Benutzer letztendlich in der Kartenübersicht zu jedem Punkt den Namen des Benutzers sehen sollen der dort eine Basis hat, wäre es zu aufwändig die User-ID aus dem Basis Objekt zu bekommen. Aus diesem Grund wurde sich bewusst dafür entschieden an dieser Stelle Datenredundanz einzuführen. Benutzer bekommen nach dem Registrieren eine Basis die bereits einem Punkt zugeordnet ist. Dafür generiert die Spielfeld-Komponente einen Punkt, welcher noch nicht vergeben ist. Dieser Punkt wird nicht zufällig aus dem gesamten Wertebereich ausgewählt, da Benutzer sonst zu weit voneinander entfernt sein könnten, was für die Bewegungszeit untragbar wäre. Deshalb werden die Punkte so gewählt, dass Benutzer relativ nah beieinander sind. Wie die Berechnung im Detail aussieht, wird in Abschnitt 5 erläutert.

4.2.9. Technologiebaum-Komponente

Die Technologiebaum-Komponente verwaltet die Abhängigkeiten zwischen Forschungen, Gebäuden und Kampfeinheiten. Konkrete Gebäude und Forschungen können als Voraussetzung zum Ausbau definiert werden. Der Ausbau eines bestimmten Gebäudetyps kann somit Abhängig von einer konkreten Forschungsstufe oder Gebäudestufe sein.

[15] Vgl. Butrynowski, C.: Entwicklung eines Systems zur Clusteranalyse von Benutzerprofilen Online PDF S.11

Beispiel: Der Gebäudetyp Kohlemine kann erst gebaut werden, wenn man den Gebäudetyp Hauptgebäude auf Stufe 5 und den Forschungstyp Minentechnik auf Stufe 1 hat.
Abbildung5 veranschaulicht dies noch einmal. Es kann nur der Typ einer Einheit von konkreten Einheiten abhängen, aber nicht umgekehrt. Beispielsweise ist es nicht möglich, dass Stufe 3 und Stufe 5 der Kohlemine unterschiedliche Voraussetzungen haben. Es gibt eine Oberklasse *Technologiebaum*, welche eine Menge von Gebäude-IDs und Forschungs-IDs speichert, weil sich die IDs auf konkrete Ausbaustufen dieser Einheiten beziehen. Somit gibt es für Gebäude, Forschungen und Kampfeinheiten Klassen, die von *Technologiebaum* erben. In diesen werden je nach Einheit, der Gebäudetyp, Forschungstyp oder die Kampfeinheit-ID gespeichert. Die Kampfeinheit-ID deshalb, weil Kampfeinheiten keine Ausbaustufen haben und die ID somit den Typ der Kampfeinheit angibt.

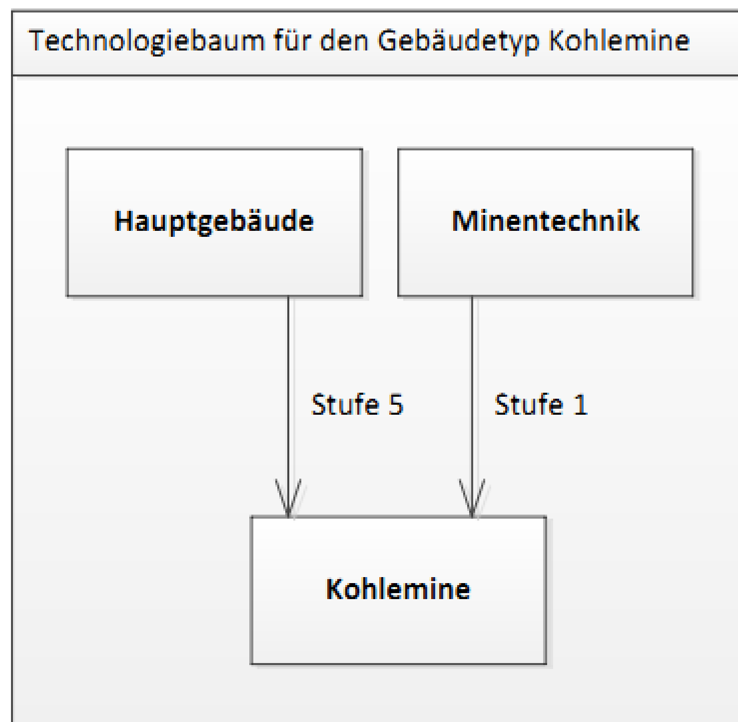

Abbildung5: Technologiebaum Beispiel

4.2.10. User-Komponente

Die User-Komponente dient zurzeit lediglich der Identifizierung eines Benutzers, sodass dieser nur mit seinen eigenen Daten arbeiten kann. Damit dies gewährleistet ist, werden Entitäten wie z.B. eine Basis, ein Gebäude, oder der Charakter, einer User-ID zugeordnet. Die Klasse *User,* speichert somit lediglich eine User-ID und einen Namen, den ein Benutzer beim Registrieren eingegeben hat und einzigartig ist.

4.2.11. Quest-Komponente

Die Quest-Komponente besteht neben dem Verwalter und der Komponenten-Klasse, aus den Klassen *Quest* und *QuestAuswahl*. Eine Quest hat unter anderem einen Namen, einen

beschreibenden Text, eine Dauer und eine Mindeststufe die ein Charakter benötigt um die Quest durchzuführen. Des Weiteren speichert sie alle Belohnungen die ein Charakter bei erfolgreicher Durchführung erhält, welche Erfahrungspunkte, Ressourcen, oder ein Item sein können. Obwohl Kämpfe zwischen Charakteren zurzeit nicht implementiert sind, kann eine Quest eine Charakter-ID eines Gegners speichern. Der Gegner ist ein Charakter der keinem Benutzer, sondern dem System gehört. Geplant war es, dass der Charakter der die Quest durchführt, die Belohnungen nur dann erhält, wenn er den Quest-Gegner besiegt. Zurzeit ist es allerdings so, dass man lediglich die Dauer der Quest abwarten muss um die Belohnungen zu erhalten. Damit ein Benutzer nicht jede in der Datenbank vorhandene Quest starten kann, gibt es die *QuestAuswahl*, welche eine in der Konfigurationsdatei definierte Maximalgröße hat und einem bestimmten Charakter zugeordnet ist. Die Zuordnung erfolgt über die jeweilige Charakter-ID. Die *QuestAuswahl* hält eine Liste von Quest-IDs, die der Charakter ausführen kann. Beim Starten einer Quest wird überprüft ob die Quest-ID in der *QuestAuswahl* vorkommt. Ein Benutzer kann eine neue *QuestAuswahl* anfordern. Dabei werden zufällig Quests aus der Datenbank, die der Stufe des Charakters entsprechen, ausgewählt.

4.2.12. Nachrichten-Komponente

Die Nachrichten-Komponenten ist für das Verwalten von Ingame-Nachrichten zuständig, welche Benutzer sich gegenseitig schicken, oder vom System (z.B. Kampfberichte) erhalten können. Die erste Idee für Nachrichten war es, eine Liste mit Nachrichten in der User-Klasse selber zu definieren. Da ein User-Objekt allerdings auch in vielen anderen Anwendungsfällen geladen wird, würden Nachrichten immer unnötig mitgeladen werden. Aus diesem Grund wurde die Beziehungsrichtung umgedreht, sodass Nachrichten die User-ID speichern und nur noch geladen werden, wenn sie wirklich angefordert werden. Da Benutzer aber wohl kaum die User-ID anderer Benutzer kennen, können Nachrichten auch über den Namen der Benutzer verschickt werden. Dazu erfragt die Nachrichten-Komponente bei der User-Komponente die User-ID zu dem jeweiligen Namen.

4.2.13. Item-Komponente

In der Item-Komponente werden zum einen Items gespeichert und zum anderem die Ausrüstungstypen, die auch vom Charakter verwendet werden, definiert. Ein Item ist genau einem Ausrüstungstypen zugeordnet und hat eine Mindeststufe die ein Charakter haben muss um das Item zu tragen. Da Items gekauft und verkauft werden können, hat jedes Item einen Einkaufs- und einen Verkaufspreis.

4.2.14. Handels-Komponente

Die Handels-Komponente ermöglicht den Handel zwischen Benutzern. Es können Ressourcen und Items gehandelt werden. Jedes von Benutzern erstellte Angebot, egal ob für Ressourcen oder Items, hat einen Ressourcenpreis und eine eindeutige Angebots-ID, und ist dem Benutzer über die User-ID zugeordnet. Dafür gibt es die Klasse *Angebot*. Für die

zwei Angebotstypen gibt es die Klassen *RessourcenAngebot* und *ItemAngebot*, welche von der Klasse *Angebot* erben. Im *RessourcenAngebot* werden die angebotenen Ressourcen gespeichert und die Basis-ID der Basis von der man die Ressourcen anbieten möchte. Im *ItemAngebot* hingegen wird nur die Item-ID des anzubietenden Items gespeichert. Der Charakter des Benutzers der das Angebot erstellt, muss das Item im Inventar haben. Wenn nun ein anderer Benutzer das Item kauft, werden die Ressourcen dem Charakter des Verkäufers gutgeschrieben und das Item wird in das Inventar des Käufers gelegt. Beim Erstellen eines Ressourcen-Angebots werden die Ressourcen nicht sofort von der Basis abgezogen, sondern erst wenn das Angebot von einem Käufer akzeptiert wird. Ein Ressourcen-Angebot kann also nur angenommen werden, wenn zum Zeitpunkt der Annahme genügend Ressourcen auf der Basis des Verkäufers vorhanden sind. Das hat den Vorteil, dass Benutzer das Handelssystem nicht missbrauchen können um Ressourcen vor Angriffen zu schützen. Da man Items von Charakteren aber nicht stehlen kann, wird ein angebotenes Item beim Erstellen eines Item-Angebots sofort aus dem Inventar entfernt.

4.2.15. Gilden-Komponente

Benutzer können sich zu Gruppen in Form einer Gilde zusammentun. Eine Gilde hat ein Ressourcenkonto, auf welches die Mitglieder Ressourcen einzahlen können. Wofür die Ressourcen verwendet werden, ist den Mitgliedern selbst überlassen. Beispielsweise könnten sie Anfänger unterstützen, damit diese schneller vorankommen. Für die Gilden gibt es ein eigenes Rechtesystem, wobei jedes Recht eine klar definierte Funktion hat. Die Rechte sind als Enumeration abgebildet:

- *Gildenleiter*: Mitglieder mit diesem Recht dürfen alles, selbst die Gilde löschen.
- *MitgliederEinladen*
- *Rundmail*: Mit diesem Recht kann man eine Nachricht an alle Gildenmitglieder verschicken.
- *KontoEinzahlen*
- *KontoAuszahlen*
- *KontostandSehen*
- *MitgliederSehen*: Mit diesem Recht kann man eine Liste aller Mitglieder angucken.
- *RängeBearbeiten*
- *BewerbungenVerwalten*: Mit diesem Recht kann man Bewerbungen von anderen Benutzern ablehnen oder annehmen.
- *EinladungenVerwalten*: Mit diesem Recht kann man Einladungen an andere Benutzer löschen, auch diejenigen die von anderen Gildenmitgliedern erstellt wurden.
- *MitgliederRauswerfen*

Der Gründer einer Gilde erhält zu Beginn das Recht Gildenleiter. Innerhalb der Gilde können beliebig viele Ränge definiert werden. Jeder Gildenrang hat eine Menge von Rechten. Damit man neuen Mitgliedern nicht manuell einen Rang zuweisen muss, hat jede Gilde einen Standard-Rang, welcher vom Gildenleiter verändert werden kann. Um Benutzer einer Gilde zuzuordnen, gibt es die Klasse *GildenMitglied*, in der die User-ID zusammen mit der Gilden-ID und dem Gildenrang gespeichert werden. Um einer Gilde beizutreten, kann man entweder von einem Gildenmitglied eingeladen werden, oder man kann sich bei einer Gilde bewerben. Ein Benutzer kann nur einer Gilde gleichzeitig angehören.

4.2.16. Accountverwaltungs-Komponente

Die Accountverwaltung dient der Registrierung und Aktivierung von Accounts. Um das System auch ohne Mail-Server testen zu können, kann man in der Konfigurationsdatei angeben, ob eine Aktivierung erforderlich ist. Beim Registrieren gibt der Benutzer einen Account-Namen, einen Ingame-Namen, einen Charakter-Namen, ein Passwort und eine E-Mail-Adresse an. Wenn eine Aktivierung erforderlich ist, wird der Status des Benutzers auf inaktiv gesetzt und eine E-Mail mit einem zufällig generierten Aktivierungscode an die angegebene Adresse gesendet. Andernfalls wird der Status auf aktiv gesetzt (siehe 4.2.17). In jedem Fall werden ein neuer Benutzer, ein neuer Charakter und eine neue Basis angelegt. Der Account-Name für das Login, ist für andere Benutzer nicht sichtbar und unterscheidet sich vom Ingame-Namen, welcher innerhalb des Spiels verwendet wird. Dies erhöht die Sicherheit, da Angreifer neben dem Passwort zunächst den Account-Namen herausfinden müssten.

4.2.17 Rechteverwaltungs-Komponente

Eine Rechteverwaltung ist zwingend notwendig, da sich Benutzer dem System gegenüber authentifizieren müssen. Benutzer sollen schließlich nicht in der Lage sein, Anwendungsfälle anonym oder für andere Benutzer zu tätigen. Außerdem dürfen nicht alle Benutzer alle Anwendungsfälle ausführen. Benutzer benötigen also Rechte, die sie für bestimmte Aktionen autorisieren. Es gibt die Rechte *gebannt*, *inaktiv*, *aktiv* und *admin*. Diese sind kumulativ aufgebaut, das heißt, dass ein Recht alle darunterliegenden Rechte beinhaltet. Das hat den Vorteil, dass man keine komplexen Zugriffsteuerungslisten oder Matrizen verwalten muss. Jedes Recht bekommt eine natürliche Zahl zugewiesen.

Gebannt = 1
Inaktiv = 2
Aktiv = 3
Admin = 4

Für jede ausführbare Aktion muss man lediglich eine dieser Zahlen speichern. Beispiel für die Überprüfung der Rechte: Die Aktion *Nachricht-Verschicken,* benötigt das Recht aktiv. Das bedeutet, dass nur Benutzer mit dem Recht *aktiv* oder *admin*, Nachrichten verschicken dürfen. Dazu überprüft man ob das Recht des Benutzers größer oder gleich (>=) dem Recht der Aktion ist.
Alle Aktionen die im Kontext der Rechteverwaltung unterscheidbar sind, werden als Enumeration abgebildet. Z.B. die Aktionen *Nachricht-Verschicken*, *Nachricht-Löschen* und *Get-Alle-Nachrichten* werden zu *Nachrichten-Verwalten* zusammengefasst, da sie alle mit demselben Recht ausführbar sind. Da NHibernate Aufzählungstypen als Zeichenkette in der Datenbank speichert, spielt die Reihenfolge der Aufzählungselemente keine Rolle. Somit muss man beim Hinzufügen neuer Anwendungsfälle keine Änderungen vornehmen.
Der resultierende Nachteil dieser Rechteverwaltung ist, dass man Anwendungsfälle und Rechte nicht beliebig miteinander kombinieren kann.
Beim Einloggen in das System werden der Account-Name und das Passwort benötigt. Wenn die Daten übereinstimmen, generiert das System ein Ticket, welches die User-ID und eine

Sitzungs-ID speichert (siehe Abschnitt 4.2.18.). Das Passwort wird somit nur einmalig für das Einloggen benötigt, was wiederum die Sicherheit erhöht, da das Passwort nicht für jede Aktion übertragen wird. Abbildung6 veranschaulicht den Login-Prozess in Form eines Sequenzdiagramms.

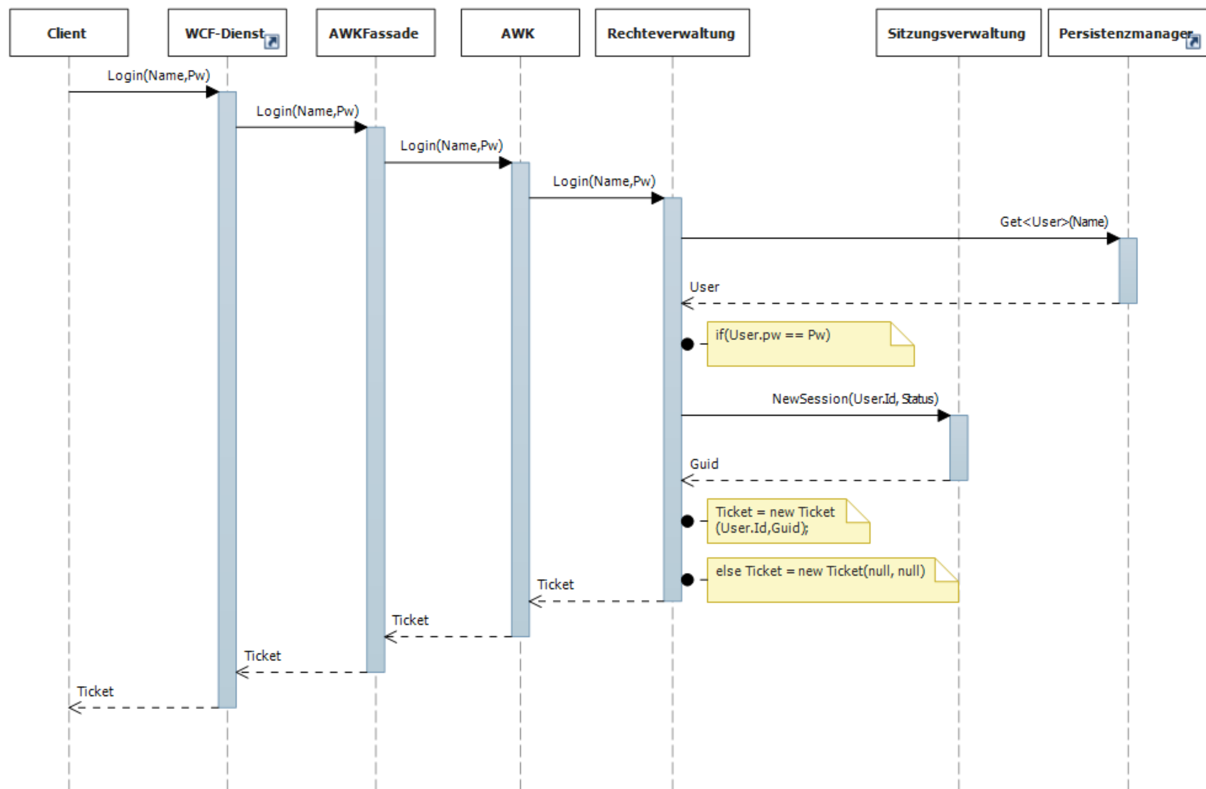

Abbildung6: Login-Prozess

4.2.18. Sitzungsverwaltungs-Komponente

Die Sitzungsverwaltung wird benötigt, damit sich Benutzer nach dem Einloggen mithilfe eines Tickets authentifizieren können. Nach dem Einloggen speichert das System, für eine bestimmte Zeitspanne, eine Sitzungs-ID zusammen mit der User-ID, dem Status des Benutzers, sowie einem Zeitstempel der dem Zeitpunkt des Einloggens entspricht. Die Zeitdauer der Speicherung entspricht der maximalen Sitzungsdauer und ist einstellbar. Der Status des Benutzers wird deshalb mitgespeichert, damit der Status nicht bei jeder Aktion, die der Benutzer tätigt, aus der Datenbank abgefragt werden muss. Das System generiert eine einzigartige, komplexe Sitzungs-ID, damit diese von Angreifern nicht erraten werden kann. Das Ticket, welches der Benutzer nach dem Einloggen erhält, speichert die Sitzungs-ID und die User-ID. Die Sitzungs-ID würde prinzipiell ausreichen, aber dann müsste ein Angreifer, so unwahrscheinlich es auch ist, nur eine gültige Sitzungs-ID erraten und könnte Aktionen für andere Benutzer tätigen. Auf die Weise, wie es beim Browsergame Framework gehandhabt wird, muss ein Angreifer zusätzlich noch die User-ID, die mit der Sitzungs-ID verknüpft ist, erraten. HTTP-Cookies werden zwar auch für Sitzungs-IDs verwendet, aber diese sollten nur für die Sitzung zwischen Web-Browser und Web-Server verwendet werden. Da man zum einen beliebig viele Web-Server einsetzen könnte, die mit dem Browsergame Framework kommunizieren, wäre es in der Theorie denkbar, dass eine Sitzungs-ID zweimal

generiert wird. Zum anderen kann man auch Windows-Forms oder WPF-Clients einsetzen, welche nicht mit einem Web-Server kommunizieren und somit ihre eigenen Sitzungs-IDs generieren müssten. Da die Sitzungs-ID in diesem Fall sicherheitskritisch ist, sollte das Browsergame Framework selber die Kontrolle über die Sitzungs-IDs haben. Aus diesem Grund generiert das System einzigartige und komplexe Sitzungs-IDs. Das Aufräumen abgelaufener Sitzungen geschieht über einen Daemon, welcher beim Start des Systems mitgestartet wird. Der Daemon überprüft nach Ablauf einer einstellbaren Zeitspanne, z.B. alle 5 Minuten, die Zeitstempel aller gespeicherten Sitzungen und entfernt alle ungültigen.

4.3. Architektur des Gesamtsystems

Bei der Entwicklung des Browsergame Frameworks wurde die Quasar Standardarchitektur als Vorbild genommen. Im Zentrum steht der Anwendungskern, welcher alle Anwendungskomponenten enthält. Der Zugriff auf die Funktionalität des Anwendungskerns erfolgt ausschließlich über den Kompositionsmanager, welcher das Fassaden-Muster[16] implementiert und eine Schnittstelle zur Kommunikation anbietet. Zum persistenten Speichern von Informationen wurde ein O/R-Mapper[17] gewählt. Der Zugriff auf diesen wird allerdings durch einen Persistenz-Manager (siehe Abschnitt 4.7) gekapselt, der die Technologie dahinter verbirgt und somit Abhängigkeiten reduziert.
Da das Framework als alleinstehender Server laufen soll, gibt es eine Service-Komponente die die Schnittstelle des Kompositionsmanagers als Dienst bereitstellt. Für die Clients wurde das Proxy-Muster angewandt[18]. Der Client verwendet ein Proxy-Objekt mit der gleichen Schnittstelle wie die des Kompositionsmanagers. Die Kommunikation mit dem Dienst wird somit durch den Proxy gekapselt. Zur Realisierung des Systems wurde das .NET-Framework ausgewählt. Mithilfe von WCF können Dienste sehr leicht implementiert werden und Proxys lassen sich automatisch generieren (siehe Abschnitt 7.5).
Als OR-Mapper wurde NHibernate ausgewählt, da der Autor dieser Arbeit bereits einige Erfahrung mit NHibernate hat und Hibernate in zahlreichen Projekten verwendet wird[19]. Zusätzlich wird die API FluentNHibernate verwendet, damit die Mapping-Definitionen in C# Code geschrieben werden können. Dies bietet Typsicherheit, was bei XML nicht gegeben ist. Durch die Verwendung von LINQ[20] für die Datenbankabfragen sind die fachlichen Komponenten dennoch NHibernate unabhängig, weil kein direktes SQL oder HQL verwendet wird. Dies hat den positiven Nebeneffekt, dass das Browsergame Framework gegen Angriffe durch SQL-Injection abgesichert ist.
Clients können in Form von Windows Forms-, WPF-, oder ASP.Net-Anwendungen realisiert werden. Abbildung7 veranschaulicht die Architektur des Gesamtsystems. Auf die Komponenten Mail- und Common wird in Abschnitt 4.9 detailliert eingegangen.

[16] Vgl. Gamma, E. [u.a.]: Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley 2011. S.212 ff
[17] Vgl. Eilebrecht, K., Starke, G.: Patterns Kompakt: Entwurfsmuster für effektive Software-Entwicklung. Spektrum Akademischer Verlag 2010. S.116
[18] Vgl. ebd., S.75 ff
[19] http://community.jboss.org/wiki/WhoUsesHibernate
[20] http://msdn.microsoft.com/de-de/library/bb397676.aspx

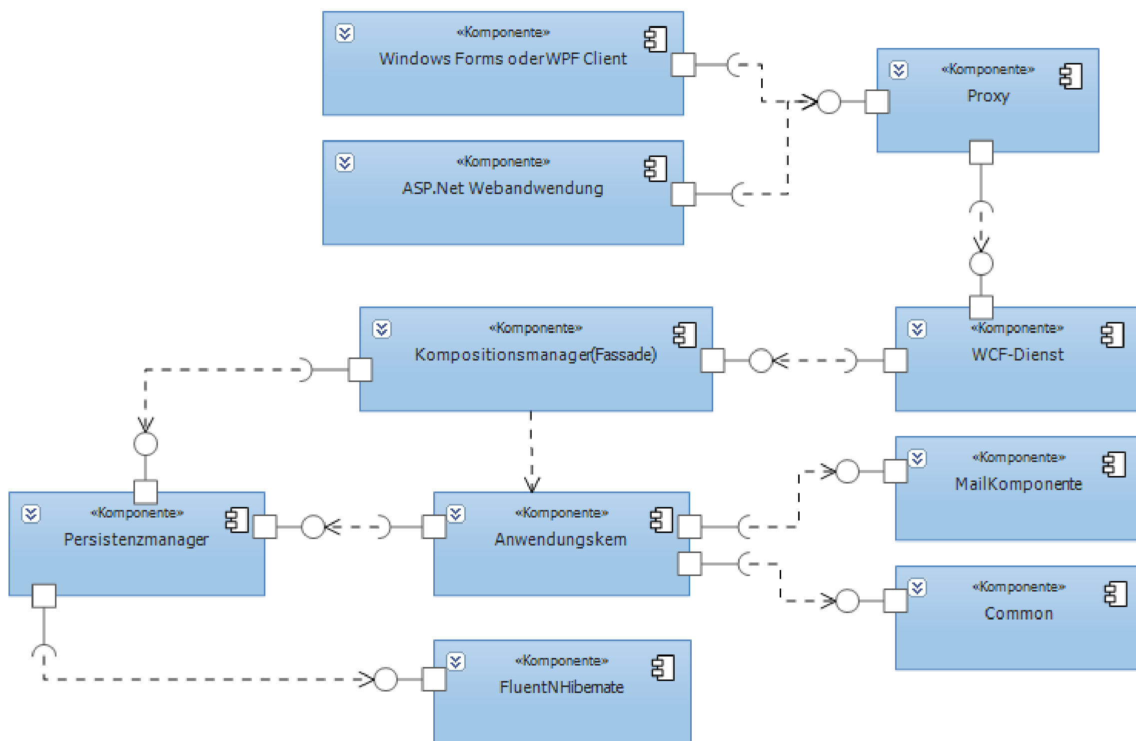

Abbildung7: Gesamtsystem-Architektur

4.4. Kompositionsmanager im Detail

Der Kompositionsmanager erzeugt je eine Instanz jeder Komponente des Anwendungskerns und versorgt die importierten Schnittstellen dieser, mit konkreten Implementierungen. Da die Fassade ein Spezialfall der Komposition ist, wird sie im Kompositionsmanager untergebracht.[21] Die Fassade wird hier als Sicherheitsfassade verwendet, das heißt sie übernimmt zusätzlich die Ausnahmebehandlung.[22] Außerdem übernimmt die Fassade die Aufgabe der Zugangskontrolle.[23] Benutzer müssen sich authentifizieren und die Fassade überprüft bei jedem Aufruf ob der Benutzer dazu berechtigt ist. Die Fassade übernimmt auch die Steuerung der NHibernate Sitzungen. Für jeden Aufruf an die Fassade wird eine Sitzung geöffnet, der Aufruf transaktional ausgeführt und am Ende die Sitzung wieder geschlossen.

4.5. Architektur der technischen Infrastruktur

Das Browsergame Framework kann prinzipiell auf jedem Rechner, auf dem das .NET Framework Version 4 installiert ist, betrieben werden. Im Rahmen dieser Arbeit stand ein Server mit dem Betriebssystem Windows Server 2008 R2 zu Verfügung. Zusätzlich waren der Microsoft SQL-Server 2008 und der Internet Information Services 7 (kurz IIS7) installiert.

[21] Vgl. Siedersleben, J.: Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar. dpunkt.Verlag 2005. S. 63
[22] Vgl. ebd., S.105
[23] Vgl. Eilebrecht, K., Starke, G.: Patterns Kompakt: Entwurfsmuster für effektive Software-Entwicklung. Spektrum Akademischer Verlag 2010. S.73

Alternativ können Web-Server und Datenbank auch auf physikalisch unterschiedlichen Rechnern laufen. Der IIS hostet eine ASP.NET Webanwendung, auf welche man über einen normalen Web-Browser zugreifen kann. Die ASP.NET Webanwendung kommuniziert über HTTP mit dem WCF-Dienst des Browsergame Frameworks. Alternativ kann man auch einen Windows-Forms oder WPF-Client für die direkte Kommunikation mit dem Framework nutzen. Abbildung8 veranschaulicht den zuvor beschriebenen Sachverhalt.

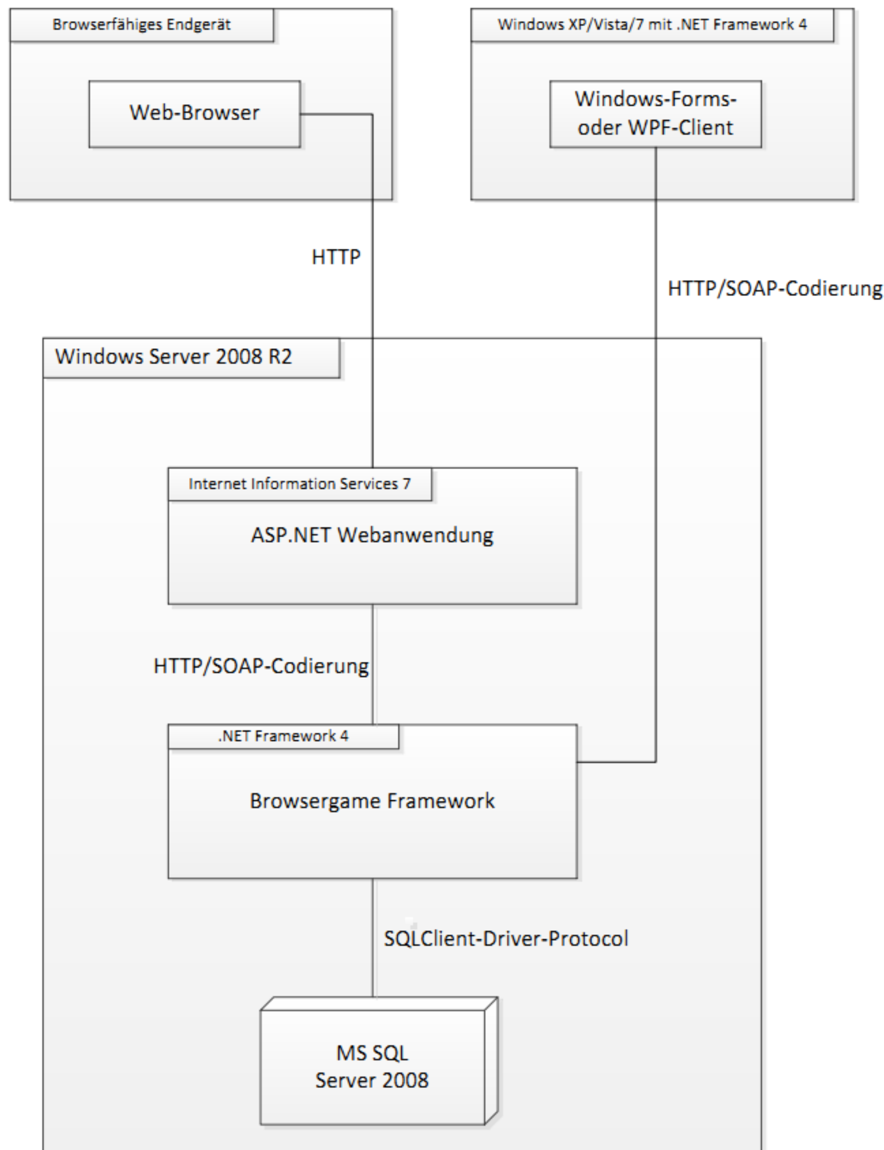

Abbildung8: Architektur der technischen Infrastruktur

4.6 Technik-Architektur

Die Architektur der Technik beschreibt das Zusammenspiel des Anwendungskerns mit den technischen Komponenten. Abbildung9 verdeutlicht, dass die ASP.NET Webanwendung und der Proxy auch auf physikalisch unterschiedlichen Rechnern laufen können als der WCF-Dienst. Daraus geht hervor, dass problemlos mehrere Web-Server eingesetzt werden können.

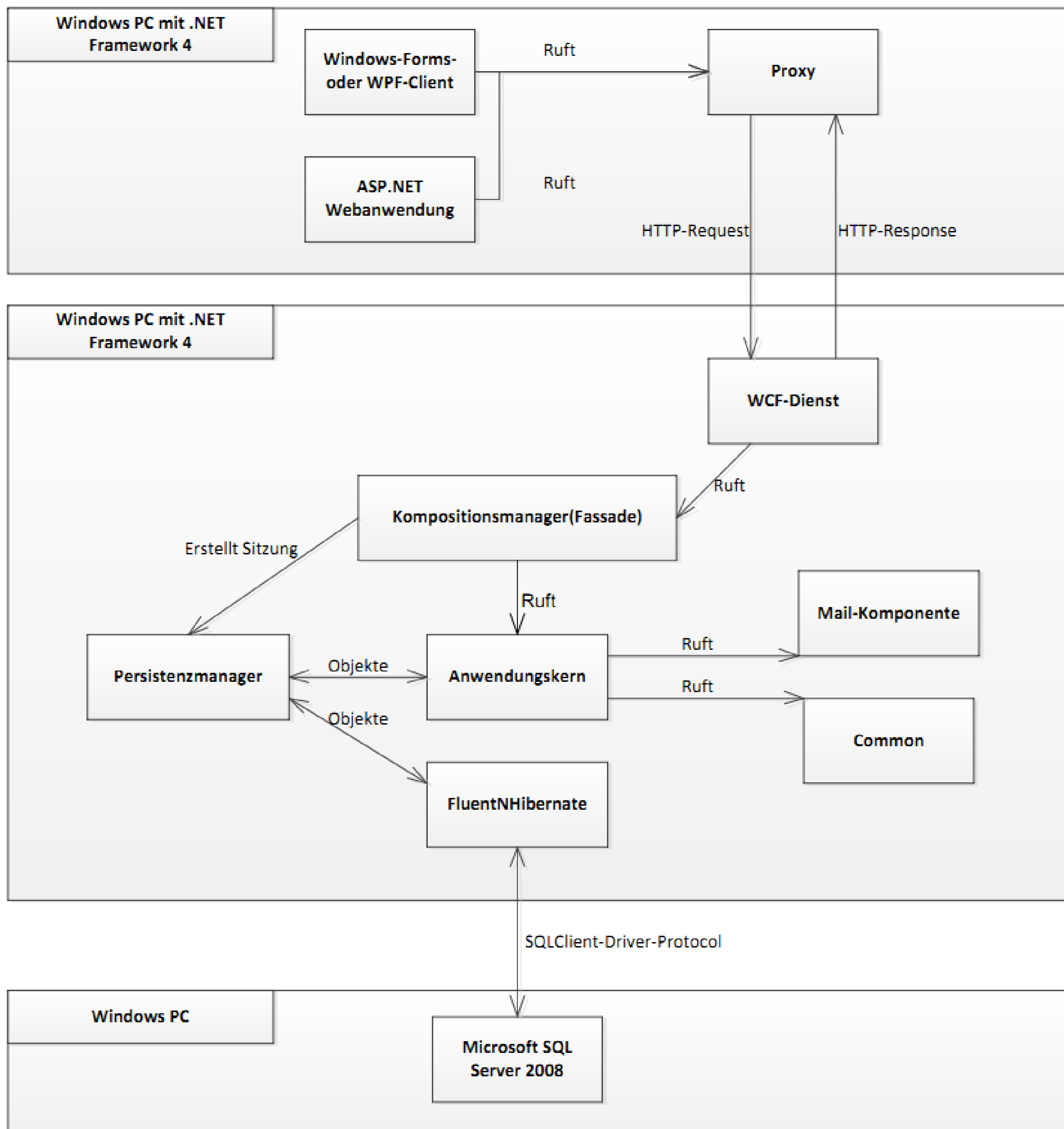

Abbildung9: Technik-Architektur

4.7. Persistenz-Manager

Der Persistenz-Manager wurde nach dem Persistenz-Manager von Prof. Dr. Stefan Sarstedt entworfen. Dabei wurden nur die Teile übernommen, die für das Funktionieren unbedingt notwendig sind. Per Konstruktor kann man die Datenbank auswählen und angeben, ob die vorhandene Datenbank verwendet oder ob sie überschrieben werden soll. Zurzeit kann man zwischen SQLite und MS SQL-Server 2005 und 2008 wählen. Es gibt eine Schnittstelle die von der Fassade verwendet wird, um Sitzungen zu öffnen. Innerhalb dieser Schnittstelle gibt es eine Methode, die einen Code-Block als Delegat annimmt und transaktional ausführt. Für die Verwalter-Klassen des Anwendungskerns gibt es eine Schnittstelle, um Objekte zu speichern, zu löschen und abzufragen.

4.8. Umgang mit Transaktionen

Für jeden Aufruf an die Fassade wird eine neue NHibernate Session erzeugt. Innerhalb dieser Session wird eine Transaktion erstellt. Der Aufruf wird an den Anwendungskern delegiert und läuft vollständig im Kontext der Transaktion. Wenn innerhalb des Anwendungskerns eine Ausnahme auftritt, wird die Transaktion zurückgerollt, damit in der Datenbank keine Inkonsistenzen auftreten können. Dies wird in Abschnitt 5.7 detailliert dargelegt. Sofern Transaktionen nicht auf denselben Daten operieren, sollen sie parallel ablaufen. Wenn aber beispielsweise ein Angriff und ein Update derselben Basis in demselben Moment stattfinden, könnten durch Parallelität inkonsistente Werte auftreten, weshalb Transaktionen die auf dem gleichen Datenbestand arbeiten, nicht parallel ausgeführt werden dürfen. Weshalb dies zurzeit nicht wie gewünscht funktioniert wird in Abschnitt 5.8 erläutert.

4.9. Mail- und Common-Komponente

Die Mail-Komponente ist eine rein technische Komponente und dient dem Versand von E-Mails. Zum Versenden von E-Mails loggt sich die Komponente bei einem SMTP-Server ein und versendet über diesen die Nachricht. Die Adressinformationen des Servers, sowie die Accountdaten, werden dabei aus der Konfigurationsdatei der Anwendung ausgelesen. Es gibt zwei Methoden zum Versenden von E-Mails. Eine Methode ist zum Versenden von einfachen Texten, die andere zum Versenden von HTML-Nachrichten.

In der Common-Komponente befindet sich zurzeit lediglich eine Helfer-Klasse, die das Auslesen von Werten aus der Konfigurationsdatei erleichtert. Es gibt Methoden um Strings, Integer und Boolesche-Werte auszulesen. Um den gewünschten Wert zu erhalten gibt man den String, der dem Key in der Konfigurationsdatei entspricht, an.

5. Implementierung

In diesem Abschnitt werden einige Aspekte der Implementierung von Komponenten, die bereits im Entwurf besprochen wurden, aufgezeigt. Realisiert wurde das Browsergame Framework in der Programmiersprache C# mit der Entwicklungsumgebung Microsoft Visual Studio 2010. Für die Gilden-Komponente wurden bisher nur das fachliche Datenmodell im Code abgebildet, das FluentNHibernate Mapping erzeugt und alle benötigten Anwendungsfälle in der Fassade definiert. Die fachliche Logik der Gilden-Komponente wurde noch nicht implementiert. Das Browsergame Framework wurde bisher nur prototypisch realisiert, ist dennoch lauffähig und bietet Funktionalität für fast alle aufgestellten Anforderungen. Da FluentNHibernate verwendet wird, müssen alle Eigenschaften Public virtual sein, damit NHibernate die Eigenschaften durch Proxys ersetzen kann. Die Getter sind somit alle öffentlich aufrufbar. Der Einfachheit halber wurden die Setter auch alle auf Public gesetzt, damit bei den Tests Werte einfach geändert werden können. Die Mapping-Definitionen stehen alle in den jeweiligen Dateien der Klassen selbst, damit sie sofort auffindbar sind. Da es sich bei den Mapping-Definitionen allerdings um ein Mittel handelt, das ausschließlich von dem technischen Framework FluentNHibernate benötigt wird, wäre es ratsam diese in gesonderten Klassen auszulagern und somit von der Fachlichkeit zu trennen.

5.1. Einheiten-Komponente

Ressourcen berechnen

Bei der Berechnung, der zwischen zwei Updates erzeugten Ressourcen, wird zwischen folgenden Situationen unterschieden:

- Seit dem letzten Update wurden keine Forschungen, welche die Ressourcenproduktion verändern, fertiggestellt und die Ausbaustufe der Mine nicht verändert.

In diesem Fall werden für die Zeitspanne einfach die Ressourcen mit der aktuellen Minenproduktion berechnet.

- Seit dem letzten Update wurden beliebig viele Forschungen, welche die Ressourcenproduktion verändern, fertiggestellt.

In diesem Fall müssen die Fertigstellungszeitpunkte jeder Forschung bekannt sein und die Ressourcen müssen für jeden resultierenden Zeitraum mit der jeweils veränderten Produktion berechnet werden.

- Seit dem letzten Update wurde die Ausbaustufe der Mine verändert.

In diesem Fall werden die Ressourcen bis zur Fertigstellung der Ausbaustufe mit der alten Produktion und nach der Fertigstellung mit der neuen Produktion berechnet.
Es gibt noch einen weiteren Fall, welcher eine Kombination aus den beiden Letzten darstellt.

Letztendlich wird für jeden resultierenden Zeitraum folgende Methode der Mine aufgerufen:

```
public virtual Ressourcen BerechneRessouren(TimeSpan zeitspanne)
{
    Ressourcen produktion = ProduktionProStunde * (decimal)zeitspanne.TotalHours;
    Ressourcen verbrauch = VerbrauchProStunde * (decimal)zeitspanne.TotalHours;
    return produktion - verbrauch;
}
```

Diese Berechnung führt dazu, dass sehr kleine Produktionen bei ganzzahligen Werten nicht berechnet werden können, wenn die Zeitspanne zwischen zwei Updates zu kurz ist. Beispielsweise ist es einem Benutzer durchaus möglich, jede Sekunde ein Update anzufordern. Damit die Ressourcenproduktion dann aber stimmt, müsste die Produktion mindestens 3600/Std. betragen, also eins pro Sekunde. Eine mögliche Lösung wäre es eine Tick-Rate[24] einzuführen, sodass Ressourcen z.B. alle fünf Sekunden produziert werden. Wenn ein Benutzer dann nach einer kürzeren Zeitspanne ein Update anfordert, bekommt er noch keine Ressourcen, sondern erst wenn 5 Sekunden vergangen sind. Diese Lösung verlagert das Problem allerdings nur. Wenn die Tick-Rate beispielsweise eine Sekunde betragen würde, hätte man dasselbe Problem bei einer geringen Produktion von weniger als 3600/Std. Die Tick-Rate müsste also relativ groß gewählt werden, was natürlich dazu führt, dass Benutzer länger warten müssen um Änderungen an ihren Ressourcen festzustellen. Eine andere denkbare Lösung wäre es Zahlen mit Nachkommastellen zu verwenden. Dadurch wäre es realisierbar, geringere Ressourcenproduktionen von weniger als 1/Sek. zu verwenden. Der Einfachheit halber wurden diese Möglichkeiten vorerst nicht in Betracht gezogen.

Kampf-Komponente rufen

Beim Bewegungstyp Angriff, wird ein Kampf zwischen zwei Benutzern ausgetragen, welcher von der Kampf-Komponente berechnet wird. Bei der Berechnung der Bewegung wird zuerst eine Methode aufgerufen, welche alle Paramater für die Kampf-Komponente vorbereitet, anschließend die Kampf-Komponente aufruft und das Kampfergebnis zurückgibt.

```
KampfErgebnis ergebnis = KampfAustragen(basis, bewegung, zielBasis);
```

Die Methode *KampfAustragen* bekommt über die Bewegung die Anzahl der beteiligten Kampfeinheiten des Angreifers und die Charakter-ID, sofern der Charakter mitgeschickt wurde. Mithilfe der Zielbasis werden alle Kampfeinheiten und die Gebäude des Verteidigers ermittelt und sofern der Charakter des Verteidigers verfügbar ist, auch dessen ID. In der Bewegung werden nur die Kampfeinheit-IDs der originalen Einheiten aus der Faktentabelle gespeichert. Deshalb ist die Basis des Angreifers von Nöten, um die eventuell durch Forschungen verbesserten Attribute der Kampfeinheiten zu ermitteln. Der Datentyp *KampfErgebis* speichert die User-ID für den Gewinner und Verlierer, sowie die nach dem Kampf übrig gebliebenen Kampfeinheiten.

```
public UserID Sieger { get; set; }
```

[24] Eine Tick-Rate ist eine definierte Zeitspanne, nach deren Ablauf ein wiederkehrendes Ereignis eintritt.

```
public UserID Verlierer { get; set; }
public IDictionary<KampfeinheitID, int> Siegereinheiten { get; set; }
public IDictionary<KampfeinheitID, int> Verlierereinheiten { get; set; }
```

Nachdem alle Parameter vorbereitet wurden, wird die Kampf-Komponente aufgerufen und das Kampfergebnis zurückgegeben.

```
return kampfKomponente.KampfAustragen(angreiferBasis.UserID, verteidigerBasis.UserID,
angreiferEinheiten, verteidigerEinheiten, gebäudeAttribute, angreiferCharakter,
verteidigerCharakter);
```

Die Parameter enthalten alle für den Kampf relevanten Informationen. Dies sind im Prinzip die Anzahl aller Kampfeinheiten, sowie Listen von allen Attributen der Kampfeinheiten, der Gebäude und der Charaktere.

5.2. Kampf-Komponente

Die Kampf-Komponente unterscheidet zwischen Angriffs- und Verteidigungsattributen. Welche dies sind, wird in der Konfigurationsdatei der Anwendung definiert.

Beispiel:

```
<add key="angriffsAttribute" value="Angriffskraft,Stärke"/>
<add key="verteidigungsAttribute" value="Verteidigung,Geschicklichkeit"/>
```

Ausgelesen werden können diese mithilfe der *AppConfHelper* Klasse der Common-Komponente. Die Kampf-Komponente enthält eine abstrakte-Klasse mit einer abstrakten Methode, die in der Schnittstelle *IKampfKomponente* definiert ist, die beim Erben überschrieben werden muss. Der Konstruktor stellt sicher, dass die Attribut-Komponente aufrufbar ist.

```
public abstract class Kampf_Komponente : IKampfKomponente
{
        protected IAttributKomponente attributKomponente = null;

        protected Kampf_Komponente(IAttributKomponente attributKomponente)
        {
                this.attributKomponente = attributKomponente;
        }

        public abstract KampfErgebnis KampfAustragen(UserID angreifer, UserID
        verteidiger, IList<KampfTupel> angreiferEinheiten, IList<KampfTupel>
        verteidigerEinheiten, IDictionary<Attribut, long> gebäudeAttribute,
        IList<AttributTupel> angreiferCharakter, IList<AttributTupel>
        verteidigerCharakter);
}
```

Folgendes ist aus der beispielhaften Implementierung der Kampf-Komponente.

Die Angriffsattribute werden aus der Konfigurationsdatei gelesen und anschließend wird mithilfe der Attribut-Komponente für jedes in der Zeichenkette enthaltene Attribut das Attribut-Objekt in einer Liste gespeichert.

```
IList<Attribut> angriffsAttribute = new List<Attribut>();
string tmpStr = AppConfHelper.GetValueAsString("angriffsAttribute");
if (tmpStr == null)
	tmpStr = "";
tmpStr = tmpStr.Trim();
foreach (string s in tmpStr.Split(','))
{
	Attribut tmpAtr = attributKomponente.GetAttribut(s);
	if (tmpAtr != null)
		angriffsAttribute.Add(tmpAtr);
}
```

Dasselbe wird für die Verteidigungsattribute gemacht. Die resultierenden Attribut-Listen werden dafür benutzt, um alle über die Parameter definierten Attribute zu sortieren. Letztendlich hat man am Schluss zwei Werte. Einen für die Summe aller Angriffsattribute des Angreifers und einen für die Verteidigungsattribute des Verteidigers. Mithilfe dieser wird entschieden, welcher Benutzer gewonnen und welcher verloren hat.

5.3. Technologiebaum-Komponente

Forschungen und Gebäude die ein Benutzer bereits ausgebaut hat, müssen gespeichert werden, damit man überprüfen kann, ob die Voraussetzungen zum Bau anderer Einheiten erfüllt sind. Da Forschungen Basis-übergreifend wirken, gibt es die Klasse *UserTechnologie*, welche alle Forschung-IDs, der vom Benutzer erforschten Forschungen, speichert.

```
public class UserTechnologie
{
	public virtual UserID User { get; set; }
	public virtual IList<BasisTechnologie> BasisTechnologien { get; set; }
	public virtual IList<ForschungID> Forschungen { get; set; }
}
```

Für jede Basis des Benutzers werden die Gebäude-IDs gespeichert, die auf der Basis ausgebaut wurden.

```
public class BasisTechnologie
{
	public virtual BasisID Basis { get; set; }
	public virtual IList<GebäudeID> Gebäude { get; set; }
}
```

Auf diese Weise ist die Überprüfung der Voraussetzungen sehr simpel. Die Methode zur Überprüfung bekommt als Parameter die User-ID des Benutzers, die Basis-ID der Basis auf der man das Gebäude, die Forschung oder die Kampfeinheit ausbilden möchte und je nachdem, den Gebäudetyp, Forschungstyp oder die Kampfeinheit-ID der auszubildenden Einheit.

Nachfolgend ein Beispiel zur Überprüfung der Voraussetzung zum Ausbau eines Gebäudes.

Der Technologiebaum des Gebäudes und die *UserTechnologie* des Benutzers werden mittels Verwalter aus der Datenbank geholt.

```
TechnologiebaumGebäude techTree = verwalter.GetTechnologiebaum(gebäude);
UserTechnologie userTech = verwalter.GetUserTechnologie(user);
```

Aus der *UserTechnologie* wird die entsprechende *BasisTechnologie* rausgesucht.

```
BasisTechnologie bt =(from tmp in userTech.BasisTechnologien
                      where tmp.Basis.Equals(basis)
                      select tmp).SingleOrDefault<BasisTechnologie>();
```

Anschließend wird geprüft, ob der Benutzer jede benötigte Forschung und jedes Gebäude des Technologiebaumes hat.

```
foreach (GebäudeID g in techTree.Gebäude)
       if (!bt.Gebäude.Contains(g))
              return false;
foreach (ForschungID f in techTree.Forschungen)
       if (!userTech.Forschungen.Contains(f))
              return false;
return true;
```

5.4. Spielfeld-Komponente

Die drei Koordinaten des Spielfeldes entsprechen jeweils einem 32-Bit-Integer. Damit die Startbasen der Benutzer aber nicht zu weit voneinander entfernt liegen, bekommt die Startbasis eines Benutzers beim Registrieren einen Punkt vom System zugewiesen. Damit soll sichergestellt werden, dass alle Benutzer in angemessener Zeit erreichbar sind[25]. Die Berechnung eines neuen Punktes ist für den Prototyp ziemlich simpel gehalten. Dabei wird die Anzahl der bereits vergebenen Punkte verwendet, damit der Wertebereich bei neuen Benutzer-Registrierungen oder beim Erstellen neuer Basen mitwächst. Man nimmt die Anzahl aller vorhandenen Punkte und nimmt diesen Wert als Wertebereich für den neuen Punkt. Dieser Wertebereich lässt wesentlich mehr Punkte zu, als tatsächlich vorhanden sind. Da aber der neue Punkt zufällig aus dem Wertebereich generiert wird, muss der Wert größer als der Tatsächliche sein, damit es unwahrscheinlich ist einen bereits vorhandenen Punkt zu erzeugen.

```
public Punkt3D NeuenPunktAnlegen(BasisID basis, UserID user)
{
       long anzahl = verwalter.GetAlleSpielfeldPunkte().Count;
       anzahl++;
       Random rnd = new Random();
       int x = rnd.Next((int)anzahl);
       int y = rnd.Next((int)anzahl);
       int z = rnd.Next((int)anzahl);
       Punkt3D p = new Punkt3D(x, y, z);
```

[25] Da die Reisezeit von Kampfeinheiten von der Entfernung der Punkte abhängt, würde eine Bewegung von dem Punkt (0,0,0) nach (Int.MaxValue,Int.MaxValue,Int.MaxValue), bei einer Reisedauer pro Koordinate von einer Sekunde, etwa 117,9 Jahre betragen.

```
        while (verwalter.GetPunkt(p) != null)
        {
                x = rnd.Next((int)anzahl);
                y = rnd.Next((int)anzahl);
                z = rnd.Next((int)anzahl);
                p = new Punkt3D(x, y, z);
        }
SpielfeldPunkt sp = new SpielfeldPunkt(basis, user, p);
verwalter.SpielfeldPunktAnlegen(sp);
return p;
}
```

5.5. Auftrags-Komponente

Die Auftrags-Komponente hat Warteschlangen für Kampfeinheiten, Gebäude, Forschungen und Quests. Bei einem Update wird geprüft ob eine Warteschlange seit dem letzten Update fertig wurde. Die Methoden zur Überprüfung geben *true* zurück, wenn dies der Fall ist und *false*, wenn der Fertigstellungszeitpunkt noch nicht erreicht wurde oder kein Bauauftrag vorliegt. Damit man dies nicht für jeden Gebäudetyp, Forschungstyp, etc. tun muss, geben die Methoden über Out-Parameter zusätzlich die IDs der fertigen Einheiten zurück.

Beispiel:

```
public bool IstGebäudeFertig(BasisID basis, out GebäudeID gebäude, out DateTime
fertigstellung)
```

Da nur jeweils ein Bauauftrag pro Basis vorliegen kann, reicht es nur eine ID und keine Liste von IDs zurückzugeben. Der Fertigstellungszeitpunkt wird benötigt, um die Ressourcen korrekt zu berechnen, falls ein Produktionsgebäude fertig wurde. Obwohl man die Verwendung von Out-Paramatern meiden sollte[26], wurden sie hier benutzt, weil es überflüssig schien für zwei zusätzliche Rückgabewerte eine extra Klasse zu erzeugen, die alle Rückgabeparameter enthält.

5.6. Charakter-Komponente

Die Charakter-Komponente ist neben der Einheiten-Komponente auch eine sehr umfangreiche Komponente. Deshalb werden lediglich einige interessante Aspekte der Implementierung aufgezeigt. Unter anderem die Verwendung von Lambda-Ausdrücken, was ein Sprachelement von C# ist.

Ein Charakter hat viele Konfigurationsmöglichkeiten. Diese werden in der Konfigurationsdatei der Anwendung vorgenommen. Eine Quest zu absolvieren, dauert eine in der Quest definierte Zeitspanne. Da Forschungen die Questzeit verkürzen können, muss man ein Attribut angeben, welches für die Beeinflussung der Questzeit verantwortlich ist.

---

[26] Vgl. Cwalina, K., Abrams, B.: Richtlinien für das Framework-Design: Konventionen, Ausdrücke und Muster für wiederverwendbare .NET-Bibliotheken. Addison-Wesley Verlag 2007. S.172

Beispiel:

In der *app.config*:

```
<add key="questzeitAttribut" value="QuestZeitAttribut"/>
```

Beim Erzeugen eines neuen Charakters für einen Benutzer:

```
Attribut questzeitAttribut = attributKomponente.GetAttribut(
AppConfHelper.GetValueAsString("questzeitAttribut"));
if (questzeitAttribut == null)
       throw new Exception("QuestZeitAttribut wurde nicht gefunden");
```

Somit ist es möglich das definierte Attribut auf die herkömmliche Weise in Forschungen zu verwenden. Die maximale Inventargröße wird ebenfalls aus der Konfigurationsdatei ausgelesen.

Die Lebensenergie eines Charakters wird über seine Stufe und ein in der Konfigurationsdatei definiertes Attribut berechnet. Dafür stellt die Charakter-Komponente ein Delegat bereit.

In der *app.config*:

```
<add key="lebensAttribut" value="Ausdauer"/>
```

In der Charakter-Komponente:

```
public delegate int lifeCalc(AttributTupel atr, int stufe);
```

Die Formel zur Berechnung der Lebensenergie wird über einen Lambda-Ausdruck wie folgt definiert:

```
(x, y) => (int)((int)x.Wert * y))
```

Dies ist ein Beispiel für eine Formel, welche angibt, dass die Lebensenergie gleich der Stufe des Charakters multipliziert mit dem Wert des für die Lebensenergie relevanten Attributes entspricht. Die Ressourcenkosten für das manuelle upgraden von Attributen werden ebenfalls über einen Delegaten definiert.

```
public delegate Ressourcen ressCalc(int wert, Ressourcen ress);
```

Die Kosten sind demnach abhängig von dem Wert des jeweiligen Attributes.

Die Attribute des Charakters werden redundant gespeichert. Es gibt mehrere Listen von Attributen. Eine Liste speichert die originalen Werte der Attribute, ohne Beeinflussung durch Items oder Forschungen. Eine weitere Liste speichert die durch Items veränderten Attribute und eine Liste speichert die Attribute für alle wirkenden Forschungen, auf Basis der durch die Items beeinflussten Attribute. Beim Anlegen oder Ablegen von Items, beim manuellen Upgrade von Attributen, sowie bei der Fertigstellung von Forschungen, werden die Attribute erneut berechnet.

5.7. Ausnahmebehandlung

Für die Ausnahmebehandlung ist die Sicherheitsfassade innerhalb des Kompositionsmanagers verantwortlich. Der Anwendungskern behandelt somit keine Ausnahmen. Das ist notwendig, weil Anwendungsfälle transaktional ausgeführt werden müssen, damit keine Inkonsistenzen entstehen. Dies wird dadurch sichergestellt, dass wenn der Anwendungskern eine Ausnahme auslöst, der Persistenz-Manager diese abfängt und die Transaktion zurückrollt. Dafür darf der Anwendungskern die Ausnahme allerdings nicht selber abfangen, da der Persistenz-Manager sonst nicht darüber informiert wird und das Zurückrollen der Transaktion nicht veranlassen kann. Nachdem die Transaktion zurückgerollt wurde, löst der Persistenz-Manager die Ausnahme erneut aus, damit sie von der Sicherheitsfassade abgefangen werden kann. Aufräumarbeiten sind im Prinzip nicht notwendig, da durch das Zurückrollen der Transaktion nichts geschehen ist. Im fachlichen Code kann man in kritischen Abschnitten gezielt Ausnahmen produzieren, um abgeschlossene Vorgänge nicht manuell rückgängig machen zu müssen.

Ausschnitt aus dem Anwendungsfall *CharakterRessourcenAbheben*, welcher Ressourcen beim Charakter abhebt und bei einer Basis einzahlt.

```
if (!charakterKomponente.RessourcenAbheben(user, ressis))
       return false;
if (!BasisRessourcenEinzahlen(tmpBasis, ressis))
       throw new Exception("Konnte Ressourcen nicht bei Basis einzahlen");
return true;
```

Wenn das Abheben der Ressourcen beim Charakter fehlschlägt, ist noch nichts Kritisches passiert und man kann problemlos mit *false* aus dem Anwendungsfall rausgehen. Wenn das Abheben allerdings geglückt ist, das Einzahlen auf der Basis allerdings fehlschlägt, müsste man die Ressourcen beim Charakter manuell wieder einzahlen. Durch den verwendeten Entwurf ist es möglich, lediglich eine Ausnahme zu werfen. Das Zurückrollen der Transaktion stellt sicher, dass beim Charakter keine Ressourcen fehlen.

5.8. Problematik mit parallelen Transaktionen

Wenn man parallele Transaktionen durch Einstellen des Isolationslevels der Datenbank ermöglichen will, treten zwei Probleme auf. Zum einen benötigt die Datenbank etwa fünf Sekunden um einen Deadlock bei den Transaktionen festzustellen[27] und zum anderen stellt sich die Frage, wie man mit der nicht ausgeführten Transaktion umgeht. Man könnte sie noch einmal ausführen oder den Benutzer darüber informieren, dass seine Aktion gescheitert ist. Beide Möglichkeiten bieten keine optimale Lösung des Problems. Da der Schwerpunkt dieser Arbeit auf der Fachlichkeit des Browsergame Frameworks liegt, wurde dieses Problem vorerst sehr simpel gelöst. Der Persistenz-Manager hält ein Singleton-Objekt, worüber er für den Code-Abschnitt der die Transaktion ausführt einen Lock legt. Dadurch wird dieser Abschnitt für andere Prozesse gesperrt, damit Transaktionen nicht parallel ablaufen können und somit nicht auf demselben Datenbestand operieren. Der offensichtliche Nachteil ist natürlich, dass der Prototyp somit absolut keine Parallelität zulässt.

[27] Fünf Sekunden resultieren aus eigenen Testversuchen.

6. Test

In diesem Abschnitt wird gezeigt, wie die Anwendungskomponenten des Browsergame Frameworks getestet wurden. Es wurden ausschließlich Komponententests erstellt. Das bedeutet, dass jede Komponente isoliert, ohne Verwendung der abhängigen Komponenten getestet wurde. Da aber Komponenten zum korrekten Arbeiten andere Komponenten benötigen, wurden dafür Dummy-Komponenten[28] erzeugt, welche dieselbe Schnittstelle wie die echten Komponenten implementieren, allerdings keine Logik besitzen, sondern nur feste Werte zurückliefern. Eine Ausnahme ist der Persistenz-Manager. Für diesen wurde kein Dummy erstellt, damit die Tests mit der echten Datenbank arbeiten können. Am Anfang jeder Testmethode werden der Persistenz-Manager, die Dummy-Komponenten, sowie die zu testende Komponente instanziiert. Danach werden die für den Test benötigten Testdaten erzeugt und in der Datenbank gespeichert. Die darauf folgenden Tests bauen aufeinander auf. Änderungen, die durch eine zu testende Methode entstehen, sind also für die folgende Methode sichtbar. Dadurch spart man zwar den Overhead zum instanziieren aller Komponenten und Erzeugen der Testdaten, aber es ist aufwändiger den Test bei Änderungen zu pflegen.

Für folgende Komponenten wurden Tests erzeugt:

- Auftrags-Komponente
- Charakter-Komponente
- Einheiten-Komponente
- Handels-Komponente
- Kampf-Komponente
- Nachrichten-Komponente
- Quest-Komponente
- Ressourcen-Komponente
- Spielfeld-Komponente
- Technologiebaum-Komponente

6.1. Test der Auftrags-Komponente

Es wird nur ein kleiner Ausschnitt aus einem Test gezeigt, da die kompletten Tests zu umfangreich sind. In diesem Beispiel wird gezeigt, wie das Beauftragen eines Gebäudeausbaus getestet wird.

```
Persistenz tmp = new Persistenz(DBTyp.MSSQL2005, DBStartMode.DBÜberschreiben,
"auftragsTest");
tmp.AddMappingInformation<EinheitenWarteschlange>();
IPersistenz persistenz = tmp.Build();
IConversationFactory factory = persistenz as IConversationFactory;
IAuftragsKomponente auftragsKomponente = new Auftrags_Komponente(persistenz);
```

[28] Vgl. Link, J., Adler, F.: Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung. dpunkt-Verlag 2005. S.97 f

```
BasisID actualBasis = new BasisID(1, "basis");
GebäudeID expectedGebäude = new GebäudeID(1, "gebäude");
GebäudeID actualGebäude = null;
DateTime actualFertigstellung = DateTime.MinValue;
bool IsGbFertig = false;
bool IsGbBeauftragt = false;
TimeSpan dauer = new TimeSpan(1, 0, 0);
GebäudeWarteschlange gWart = null;
DateTime expectedFertigstellung = DateTime.Now + dauer;
using (var conversation = factory.NewConversation())
{
	conversation.ExecuteTransactional(new TransactionalCode(() =>
	{
		IsGbBeauftragt =
		auftragsKomponente.AddGebäudeToWarteschlange(actualBasis,
		expectedGebäude, dauer);
		gWart = auftragsKomponente.GetGebäudeWarteschlange(actualBasis);
		IsGbFertig = auftragsKomponente.IstGebäudeFertig(actualBasis, out
		actualGebäude, out actualFertigstellung);
	}));
}
Assert.IsFalse(IsGbFertig);
Assert.IsTrue(IsGbBeauftragt);
Assert.AreEqual(expectedGebäude, actualGebäude);
Assert.AreEqual(expectedFertigstellung.Ticks, gWart.Fertigstellung.Ticks, new
TimeSpan(0, 1, 0).Ticks);
Assert.AreEqual(gWart.Basis, actualBasis);
Assert.AreEqual(gWart.Gebäude, expectedGebäude);
```

Am Anfang wird der Persistenz-Manager konfiguriert, welcher das Datenbankschema exportiert. Da die Auftrags-Komponente sonst keine Komponenten zum Arbeiten benötigt, gibt es hier keine Dummy-Komponenten. Die zu testenden Methoden befinden sich innerhalb des Using-Blocks. Für das Gebäude *expectedGebäude* wird auf der Basis *actualBasis* ein Bauauftrag erzeugt. Der Aufruf an die Auftrags-Komponente sollte glücken, weshalb der Boolean *IsGbBeauftragt true* sein soll. Als nächstes wird das Warteschlangen-Objekt verlangt und überprüft ob seine Interna mit den Vorgaben übereinstimmen. Zuletzt wird gefragt ob ein Gebäude fertig wurde. Da die Bauzeit aber eine Stunde beträgt, muss der Boolean *IsGbFertig false* sein. Der Fertigstellungszeitpunkt in dem Warteschlangen-Objekt sollte der aktuellen Systemzeit + der Bauzeit betragen. Dies ist nicht eindeutig testbar, da die Abarbeitung des Codes etwas Zeit in Anspruch nimmt. Aus diesem Grund wurde bei der Überprüfung ein Delta Wert verwendet[29], welcher mit einer Minute sehr großzügig gewählt wurde.

[29] Der Delta Wert gibt die Toleranz an, um wie viel sich Werte voneinander unterscheiden dürfen, um dennoch als gleich zu gelten.

6.2. Codeabdeckung

Die Codeabdeckung beschreibt, wie viel Prozent des gesamten Codes einer Komponente beim Test durchlaufen werden. Dabei wird ein hoher Wert angestrebt, da ein Test nicht Aussagekräftig wäre, wenn nur ein kleiner Teil einer Komponente getestet wird. Die Codeabdeckung wurde mit dem Microsoft Visual Studio 2010 ermittelt.

Komponente	**Nicht abgedeckt**	**Abgedeckt**
ForschungsKomponente.dll	51,73%	48,27%
ItemKomponente.dll	38,15%	61,85%
AttributKomponente.dll	28,13%	71,88%
QuestKomponente.dll	18,59%	81,41%
EinheitenKomponente.dll	16,70%	83,30%
HandelsKomponente.dll	15,35%	84,65%
NachrichtenKomponente.dll	13,10%	86,90%
CharakterKomponente.dll	13,04%	86,96%
KampfKomponente.dll	11,42%	88,58%
SpielfeldKomponente.dll	10,85%	89,15%
TechnologiebaumKomponente.dll	9,70%	90,30%
RessourcenKomponente.dll	5,62%	94,38%
AuftragsKomponente.dll	4,87%	95,13%

Tabelle2: Codeabdeckung

Obwohl für die Forschungs-, Item- und Attribut-Komponenten keine eigenen Tests vorhanden sind, sind sie dennoch aufgeführt, da in anderen Tests, Objekte aus Klassen dieser Komponenten als Testdaten erzeugt wurden. Diese drei Komponenten besitzen kaum fachliche Logik, sondern fast ausschließlich Verwalter, weshalb erst einmal keine Tests für diese Komponenten erstellt wurden. Tests die 100% Codeabdeckung erzielen sind extrem aufwändig, da man Testfälle für jeden vorhandenen Code-Pfad erstellen muss. Die Einheiten-Komponente war am schwierigsten zu testen, da dort vieles zeitabhängig ist. Die Ressourcenproduktion ist beispielsweise kaum korrekt zu testen, weil man wissen müsste zu welchem konkreten Zeitpunkt die Berechnung durchgeführt wurde.

7. Realisierung eines Fallbeispiels mithilfe des Frameworks

In diesem Abschnitt wird gezeigt, wie ein konkretes Spiel mithilfe des entwickelten Frameworks realisiert wird. Als erstes wird eine Spieleidee ausgearbeitet. Danach werden alle Schritte, die zum Veröffentlichen eines Spiels benötigt werden, erläutert.

7.1. Technologischer Rahmen

Für die Realisierung des Spiels stand ein Server mit einem 1,2GHz Single-Core Prozessor, 2GB Arbeitsspeicher und einer 100MBit Internetanbindung zur Verfügung. Installiert ist das Betriebssystem Windows Server 2008 R2. Als Datenbank wurde der Microsoft SQL-Server 2008 verwendet und zum Hosten der Webseite wurde der IIS7 installiert.

7.2. Spieleidee

Als Fallbeispiel wurde ein nicht allzu umfangreiches Mittelalter-Spiel erstellt. Die Ressourcen sind Holz und Kohle. Die Basis ist ein Dorf in dem man Gebäude bauen kann. Es gibt insgesamt vier verschiedene Gebäudetypen mit je zwei Ausbaustufen. Die Gebäudetypen sind das Hauptgebäude, die Kohlemine, das Holzlager und die Kaserne. Jedes Gebäude hat einen Verteidigungswert, der mit dem Ausbau der Stufe größer wird. Das Hauptgebäude dient lediglich als Voraussetzung für die Kohlemine und die Kaserne. Die Kohlemine benötigt das Hauptgebäude auf Stufe eins und produziert Kohle, das Holzlager kann man sofort ausbauen und produziert Holz. Die Kaserne kann man ab Hauptgebäude Stufe zwei bauen und dient der Ausbildung von Kampfeinheiten. Mit der ersten Ausbaustufe kann man Infanteristen ausbilden und mit der zweiten Kavalleristen. Beide Kampfeinheiten haben die Attribute Angriffskraft und Verteidigung. Infanterieeinheiten können für jeden Bewegungstypen bis auf Kolonisierung verwendet werden. Kavallerieeinheiten dagegen können zusätzlich neue Dörfer gründen. Der Charakter kann bis Stufe vier aufsteigen und hat die Attribute Ausdauer, Stärke und Verteidigung. Der Preis zum Upgraden der Attribute berechnet sich über den Wert des Attributs mal einen bestimmten Ressourcenpreis.

Preise:

- Ausdauer: 20 Kohle, 10 Holz
- Stärke: 50 Kohle, 0 Holz
- Verteidigung: 1500 Kohle, 1000 Holz

Der Preis für die Verteidigung ist deshalb so hoch angesetzt, weil es eine Forschung gibt, welche den Verteidigungswert um 100% erhöht. Der Charakter kann Items für die Ausrüstungstypen Brust und Kopf anlegen. Items können entweder gekauft oder durch Quests erhalten werden. Es gibt insgesamt drei Forschungstypen, die man jeweils einmal

ausbauen kann. Diese sind Holztechnik, Panzertechnik und Bauwesen. Die Holztechnik verbessert die Produktion des Holzlagers um 50%. Die Panzertechnik wirkt sich auf den Charakter aus, auf alle Kampfeinheiten und auf alle Gebäude bis auf die Kohlemine und verbessert deren Verteidigungswerte um 75%. Das Bauwesen verkürzt die Bauzeiten aller Gebäude um die Hälfte.

7.3. Erzeugung der Daten und Konfiguration des Frameworks

Die Spieleidee wird als Sammlung von Daten in der Datenbank abgebildet. Man erzeugt innerhalb der Main-Methode der Anwendung alle benötigten Objekte und speichert sie in der Datenbank ab.

Das Erzeugen aller Attribute sieht beispielsweise wie folgt aus:

```
Attribut ZeitAttribut = null;
Attribut ReiseAttribut = null;
Attribut Ausdauer = null;
Attribut Stärke = null;
Attribut Angriffskraft = null;
Attribut Verteidigung = null;
Attribut QuestZeitAttribut = null;
using (var conversation = conversationFactory.NewConversation())
{
	conversation.ExecuteTransactional(new TransactionalCode(() =>
	{
		Ausdauer = new Attribut("Ausdauer", "Erhöht die Lebensenergie");
		persistenz.Save(Ausdauer);
		Stärke = new Attribut("Stärke", "Erhöht den Schaden");
		persistenz.Save(Stärke);
		Angriffskraft = new Attribut("Angriffskraft", "Erhöt die Stärke der
		Einheit");
		persistenz.Save(Angriffskraft);
		Verteidigung = new Attribut("Verteidigung", "Erhöht die Verteidung der
		Einheit");
		persistenz.Save(Verteidigung);
		ZeitAttribut = new Attribut("Zeit", "Beschreibt die Zeitdauer");
		persistenz.Save(ZeitAttribut);
		ReiseAttribut = new Attribut("Reseizeit", "Beschreibt die Resezeit pro
		Koordinate");
		persistenz.Save(ReiseAttribut);
		QuestZeitAttribut = new
		Attribut(AppConfHelper.GetValueAsString("questzeitAttribut"), "Beschreibt
		die Questzeit");
		persistenz.Save(QuestZeitAttribut);
		CharakterAttributZuweisung zuweisung = new CharakterAttributZuweisung(
			Ausdauer, Stärke, Verteidigung
			);
		persistenz.Save(zuweisung);
	}));
}
```

Die Attribute können dann beim Anlegen der anderen Objekte verwendet werden, wie beispielsweise beim erzeugen des Hauptgebäudes.

```
GebäudeTyp Hauptgebäude = null;
using (var conversation = conversationFactory.NewConversation())
{
```

```
        conversation.ExecuteTransactional(new TransactionalCode(() =>
        {
                Ressourcen kosten = new Ressourcen();
                kosten.Rohstoffe.Add(new AttributTupel(Kohle, 50));
                kosten.Rohstoffe.Add(new AttributTupel(Holz, 50));
                Gebäude hauptgebäude = new Gebäude(Hauptgebäude, 1, new
                AttributTupel(ZeitAttribut, new TimeSpan(0, 2, 0).Ticks), kosten);
                hauptgebäude.Attribute.Add(new AttributTupel(Verteidigung, 30));
                persistenz.Save(hauptgebäude);
                hauptgebäude.GebäudeID = new GebäudeID(hauptgebäude.ID, "Gebäude");
                persistenz.Save(hauptgebäude);
        }));
}
```

Für die Rechteverwaltung muss man Berechtigungsobjekte speichern, welche die Abhängigkeiten von Anwendungsfällen und Rechten definieren.

```
using (var conversation = conversationFactory.NewConversation())
{
        conversation.ExecuteTransactional(new TransactionalCode(() =>
        {
                Berechtigung b1 = new Berechtigung(Status.Inaktiv,
                Anwendungsfall.AccountAktivieren);
                persistenz.Save(b1);
                b1 = new Berechtigung(Status.Aktiv, Anwendungsfall.GebäudeAusbauen);
                persistenz.Save(b1);
        }));
}
```

Die Konfigurationsdatei enthält folgende für das Framework relevante Einträge:

```
  <appSettings>
    <add key="aktivierungsmail" value="true"/>
    <add key="angriffsAttribute" value="Angriffskraft,Stärke"/>
    <add key="verteidigungsAttribute" value="Verteidigung"/>
    <add key="questzeitAttribut" value="QuestZeitAttribut"/>
    <add key="inventarGröße" value="10"/>
    <add key="questAuswahlGröße" value="2"/>
    <add key="itemAuswahlGröße" value="5"/>
    <add key="lebensAttribut" value="Ausdauer"/>
    <add key="charakterAttributStartwert" value="10"/>
    <add key="basisRessourcenStartwert" value="100"/>
    <add key="defaultabsender" value="frameworkbai@googlemail.com" />
    <add key="username" value="frameworkbai@googlemail.com" />
    <add key="passwort" value="xxx" />
    <add key="smtpserver" value="smtp.googlemail.com" />
    <add key="smtpport" value="587" />
    <add key="ssl" value="true" />
  </appSettings>
```

Für das Fallbeispiel wurde ein Google Mail Account angelegt, damit der Aktivierungscode per E-Mail an die Benutzer gesendet werden kann. Die Lambda-Ausdrücke zum Berechnen der Lebensenergie und des Preises für Attribut Upgrades werden beim Erzeugen des Fassaden Objekts übergeben.

```
Program.fassade = new ÖffentlicheFassade(persistenz as IPersistenz,
        persistenz as IConversationFactory,
         (x, y) => y * x, //Wert des Attributes * Ressourcenpreis
         (x, y) => (int)((int)x.Wert * y)); //Wert des Lebensenergie-Attributes *
        Charakterstufe
```

Da die Fassade die Sessions verwaltet, benötigt sie den Persistenz-Manager.

7.4. Hosten des WCF-Dienstes

Um einen WCF-Dienst auszuführen muss man zunächst einmal die Adresse des Dienstes definieren.

```
Uri baseAddress = new Uri("http://localhost:8000/BGFramework");
```

Danach erstellt man ein *ServiceHost*-Objekt, welches die Adresse und den Typ der Klasse, die als Dienst angeboten werden soll, benötigt.

```
ServiceHost host = new ServiceHost(typeof(ServiceFassade), baseAddress);
```

Das Problem ist, dass der Dienst für jede Anfrage ein neues Objekt dieses Typs erzeugt, weshalb man die eigentliche Fassade hier nicht verwenden kann. Die Fassade hält nämlich die Objekte aller Komponenten-Klassen. Die *ServiceFassade* implementiert die Fassaden-Schnittstelle und delegiert aufrufe lediglich an das eigentliche Fassaden-Objekt, welches als statisches Member in der Main-Klasse gespeichert wird. Der letzte Schritt vor dem Starten des Dienstes ist das Einstellen der Erlaubnis um Metadaten abfragen zu können.

```
ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
smb.HttpGetEnabled = true;
host.Description.Behaviors.Add(smb);
```

Dies ist notwendig, damit der Proxy automatisch generiert werden kann. Wie das funktioniert wird im nächsten Abschnitt erklärt. Zuletzt startet man den Dienst mit folgender Zeile:

```
host.Open();
```

7.5. Entwicklung einer prototypischen Webseite.

Um den Proxy zu erstellen, kann man die *svcutil.exe* verwenden, welche den kompletten Code automatisch generiert[30]. Man gibt folgenden Befehl in einen Kommandozeileninterpreter:

```
svcutil.exe /language:cs /out:generatedProxy.cs /config:app.config
http://localhost:8000/BGFramework
```

Der resultierende Proxy ist in C# Code geschrieben und steht in der Datei *generatedProxy.cs*. Die Konfigurationsdatei heißt *app.config*. Den Proxy kann man in einer ASP.NET oder Windows-Forms-Anwendung genauso benutzen, als würde man die Fassade direkt aufrufen. Dazu erzeugt man ein Objekt der Proxy-Klasse und ruft Methoden auf.

```
IÖffentlicheFassade fassade = new ÖffentlicheFassadeClient();
Ticket ticket = fassade.Login("testuser","testpw");
```

Der Proxy verbirgt somit die komplette Kommunikation mit dem Framework. Die Webseite

[30] http://msdn.microsoft.com/de-de/library/aa347733.aspx

wurde ganz schlicht gehalten. Es gibt keine Bilder oder Animationen. Nach dem Login wird man zur Übersicht weitergeleitet. Dort kann man seine Gebäude sehen und ausbauen.

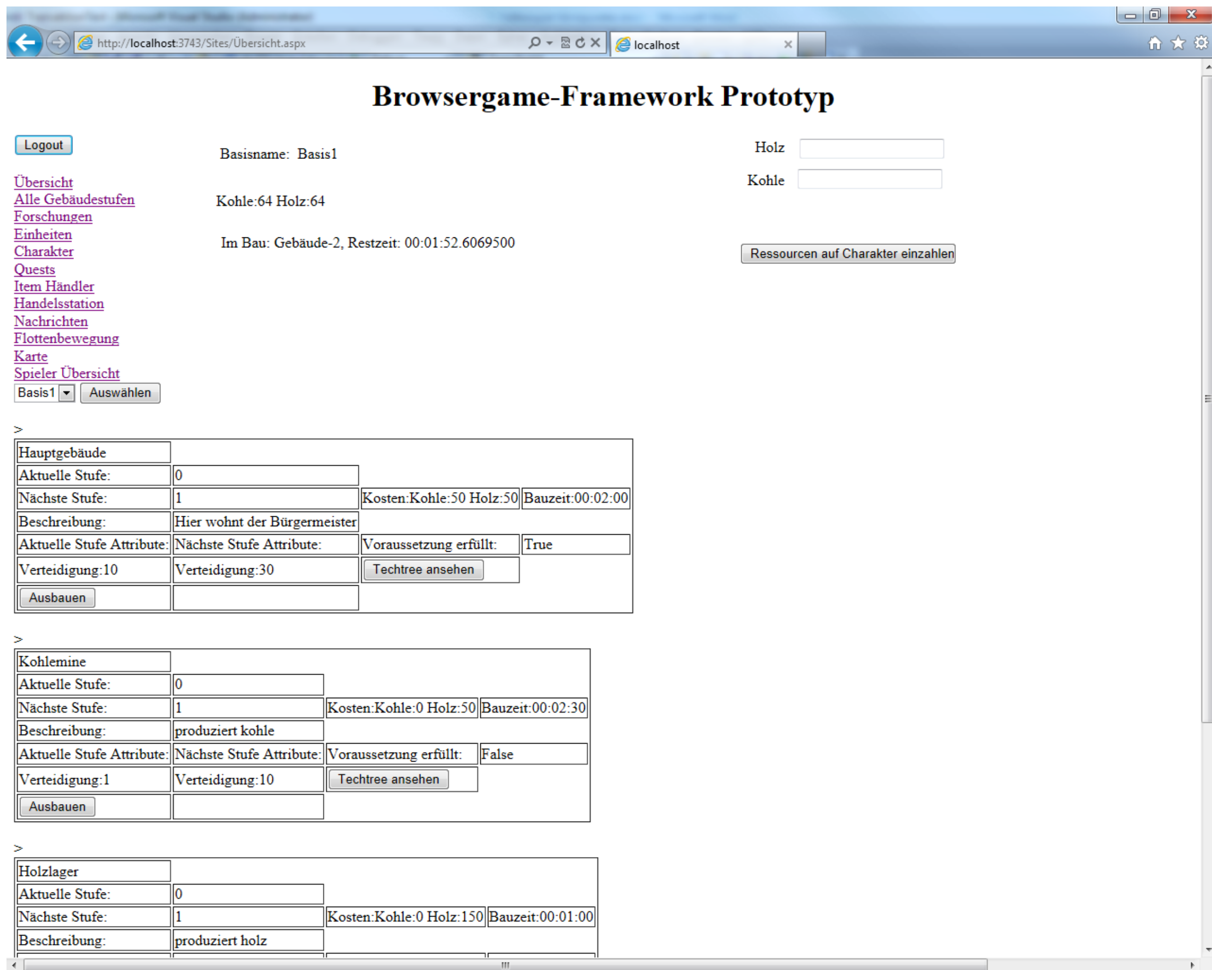

Abbildung10: Prototyp der Webseite

Zuletzt muss die ASP.NET-Anwendung mithilfe des IIS7 gehostet werden.

8. Zusammenfassung und Ausblick

8.1. Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Framework für online basierte Aufbau- und Rollenspiele entwickelt. Der realisierte Prototyp stellt Funktionalität für umfangreiche Spiele bereit. Um sinnvolle Anforderungen für die Features aufzustellen, wurden einige Browsergames aus dem Aufbau- und Rollenspiel-Genre dahingehend untersucht.
Das Framework wurde als alleinstehender Server konzipiert, der einen Dienst bereitstellt über den jegliche Funktionalität zur Verfügung gestellt wird. Um konkrete Spieleideen zu verwirklichen muss man allerdings noch Clients entwickeln, welche als Windows-Forms-, WPF- oder ASP.NET-Anwendungen realisiert werden können. Die Clients benutzen einen Proxy, der die gesamte Kommunikation mit dem Framework verbirgt und sich automatisch generieren lässt, wodurch die Entwicklung der Clients vereinfacht wird.
Die Architektur des Systems wurde nach Quasar entworfen. Der Anwendungskern, der die komplette fachliche Logik enthält, ist somit in mehrere Komponenten unterteilt und leicht erweiterbar. Technische Komponenten sind vollständig von den fachlichen Komponenten getrennt, wodurch Abhängigkeiten reduziert werden und die Wiederverwendbarkeit erleichtert wird. Durch Lambda-Ausdrücke und eine Konfigurationsdatei ist es an einigen Stellen möglich das Framework zu individualisieren.

8.3. Resümee

Zeitverhalten: Da alle von Benutzern angefragten Objekte zwischen WCF-Dienst und Proxy des Clients serialisiert werden, könnte dies bei vielen gleichzeitigen Benutzern eventuell zum Engpass führen. Die neu berechneten Basis-Objekte werden oft benötigt. Da die Basis, Listen von allen Gebäuden und Kampfeinheiten speichert, welche ziemlich umfangreich ausfallen können, könnte gerade dieses Objekt zum Problem führen. Eine mögliche Lösung wäre es, die Clients intelligenter zu machen, sodass sie beispielsweise nur nach dem Login das komplette Basis-Objekt anfordern und im Laufe einer Sitzung, nur Änderungen gesondert anfragen. Das derzeitige Problem, dass Transaktionen nicht parallel ablaufen, wirkt sich stark auf die Antwortzeiten aus.
Die Antwortzeiten wurden für einen Aufruf zurzeit gemessen. Das Testsystem war ein Desktop-PC mit Windows 7 in der 64 Bit Edition, einem vierkern-Prozessor mit 3,4 GHZ, 3,75 GB Arbeitsspeicher und als Datenbank der Microsoft SQL-Server 2005. Gemessen wurde die Zeitspanne vom Absetzen einer Anfrage vom Client an den WCF-Dienst, bis zum Eintreffen der Antwort. Alles lief lokal auf dem System, sodass die Übertragungszeit zwischen Client und Server vernachlässigt werden konnte. Die Aufrufe waren einmal das Login und das Anfragen des Basis-Objekts. Beim Anfragen des Basis-Objekts, berechnet das Framework ein komplettes Update für den Benutzer. Für den Test wurde die Datenbank mit den Daten aus dem Fallbeispiel gefüllt.

Registrierte Benutzer	∅ ms für Login	∅ ms für Basis abfragen
100	15	109
250	17	160
1000	27	314

Tabelle3: Zeitverhalten

Die Zeiten steigen mit der Anzahl der registrierten Benutzer deshalb, weil für jeden Benutzer alle Gebäude erstellt werden. Bei 12 Gebäude-Objekten aus dem Fallbeispiel, umfasst die Gebäude-Tabelle bei 1000 registrierten Benutzern bereits 12.000 Datensätze.

Zuverlässigkeit: Da das Framework noch keinem Langzeittest unterzogen wurde, lässt sich die Zuverlässigkeit nicht konkret beschreiben. Die Sicherheitsfassade fängt aber im Falle einer Ausnahme innerhalb des Anwendungskerns diese ab und gewährleistet somit den fortlaufenden Betrieb des Systems. Die Änderungen die innerhalb des fehlerhaften Anwendungsfalles durchgeführt wurden, werden durch das Zurückrollen der Transaktion rückgängig gemacht, wodurch inkonsistente Datenbestände vermieden werden. Der WCF-Dienst selber könnte eventuell zu Abstürzen führen, da WCF aber ein von Microsoft entwickeltes Framework ist, darf man davon ausgehen, dass es sehr robust ist.

Skalierbarkeit: Das System wurde bisher noch nicht mit einer großen Anzahl von Benutzern getestet, da aber zurzeit keine parallelen Transaktionen möglich sind, wären die Antwortzeiten bei vielen Benutzern sehr lang. Für den Fall, dass die Kommunikation zwischen Web-Browser und Web-Server den Flaschenhals darstellt, ist es keine Schwierigkeit mehrere Web-Server mit dem Framework kommunizieren zu lassen und somit die Last auf diese zu verteilen. Falls der Engpass auf Seiten des Frameworks liegt, ist es allerdings problematischer. Es ist nicht ohne weiteres möglich mehrere Instanzen des Frameworks parallel arbeiten zu lassen oder die Datenbank zu partitionieren.

Sicherheit: Die entwickelte Rechteverwaltung stellt sicher, dass Benutzer nur Aktionen für sich selber ausführen können. Außerdem wird sichergestellt, dass Benutzer nicht betrügen können, das heißt, sie sind nicht in der Lage die Spielstände zu ihren Gunsten zu manipulieren, da alle Berechnungen serverseitig durchgeführt werden.

Korrektheit: Der Großteil der fachlichen Logik des Frameworks wurde mittels Komponenten-Tests auf seine Korrektheit hin überprüft. Dadurch kann gewährleistet werden, dass sich das Framework zum Großteil korrekt verhält und keine falschen Werte produziert.

Erweiterbarkeit: Um die Erweiterbarkeit prüfen zu können, müsste man ein Szenario für eine beispielhafte Erweiterung durchspielen. Durch den modularen Aufbau des Anwendungskerns lässt sich zumindest leichter bestimmen, an welche Stelle neue Funktionalität hingehört. Dadurch lässt sich das Framework leichter erweitern als es bei einem monolithischen System der Fall wäre.

Benutzbarkeit: Die Benutzbarkeit lässt sich ohne Feldversuch nur sehr schwierig bestimmen. Jedoch benötigt man kein Wissen über die Interna des Frameworks um ein Spiel zu erstellen. Man erzeugt lediglich die benötigten Daten und kann sich vollständig auf die

Entwicklung eines Clients konzentrieren. Das Framework ist in dem Sinne vollständig, dass man lauffähige Spiele veröffentlichen kann.

Wiederverwendbarkeit: Durch den Komponenten-orientierten Aufbau des Frameworks und der Trennung von technischen und fachlichen Komponenten, was die Abhängigkeiten reduziert, lassen sich einzelne Komponenten leichter wiederverwenden. Die Einheiten-Komponente allerdings hat zu viele Abhängigkeiten zu anderen fachlichen Komponenten, wodurch sie nur schwer wiederzuverwenden ist.

Flexibilität: Da sich die untersuchten Spiele aus Kapitel 2 zum Großteil nur durch die Arten der Ressourcen, Attribute, Kampfeinheiten, Gebäudetypen und Forschungen unterscheiden, lassen sie sich mithilfe des entwickelten Frameworks, welches an diesen Stellen konfigurierbar ist, zu einem großen Teil nachbauen. Außerdem lassen sich Daten zur Laufzeit hinzufügen. Man kann beispielsweise neue Items, Quests oder Forschungen ohne Neustart des Systems in die Datenbank einpflegen.

Test: Die Komponenten-Tests waren sehr aufwändig, weil die benötigten Komponenten-Objekte manuell als Dummy nachgebaut wurden. Der Einsatz eines Mock-Frameworks wie beispielsweise MOQ[31] für das .NET-Framework, hätte das Testen eventuell beschleunigen können.

8.3. Ausblick

Eine sinnvolle Erweiterung wäre es Punkte und Ranglisten einzuführen, damit sich Spieler miteinander Messen können. Beispielsweise könnte man Benutzern für neu erstellte Basen, dem Ausbau von Gebäuden und dem Stufenaufstieg des Charakters, Punkte geben, welche sie in Ranglisten aller Benutzer einsehen können. Des Weiteren ist das Datenmodell der Gilden-Komponente bereits fertig und alle benötigten Anwendungsfälle sind in der Fassade definiert, weshalb es mit nur sehr geringem Aufwand möglich ist die Gilden-Komponenten fertig zu implementieren. Eine weitere sinnvolle Erweiterung wäre es administrative Methoden hinzuzufügen, damit man neue Daten beispielsweise direkt über den Client hinzufügen kann. Die Rechteverwaltung ist bereits dafür ausgelegt, dass es Methoden gibt, die nur von Administratoren aufgerufen werden können.
Damit das Framework allerdings auch mit mehreren Benutzern gleichzeitig performant läuft, sollte eine Lösung für die Problematik mit parallelen Transaktionen entwickelt werden.

[31] http://code.google.com/p/moq/

Literaturverzeichnis

Bigpoint GmbH: URL http://de.bigpoint.com [Zugriff: 09.08.2011]

Butrynowski, C.: Entwicklung eines Systems zur Clusteranalyse von Benutzerprofilen Online PDF: URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/studien/butrynowski.pdf [Zugriff: 11.08.2011]

Codeplex Open Source Community, Browsergame Framework.NET (BGF): URL http://bgframework.codeplex.com [Zugriff: 11.08.2011]

Cwalina, K., Abrams, B.: *Richtlinien für das Framework-Design: Konventionen, Ausdrücke und Muster für wiederverwendbare .NET-Bibliotheken*. München [u.a.] 2007.

Erik Scholtz & Richard Grotz GbR, Kartellwar: URL http://www.kartellwar.de [Zugriff: 14.08.2011]

Eilebrecht, K., Starke, G.: *Patterns Kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 3. Auflage. Heidelberg 2010.

Farbflut Entertainment GmbH, Pennergame: URL http://www.pennergame.de [Zugriff: 14.08.2011]

Gamma, E. [u.a.]: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 6. Auflage. München [u.a.] 2011.

Gameforge Productions GmbH, OGame: URL http://www.ogame.de [Zugriff: 14.08.2011]

Google Inc., MOQ: URL: http://code.google.com/p/moq [Zugriff: 13.08.2011]

HPM Kommunikation GmbH, Browsergame-Engine: URL http://www.mybrowsergame.com [Zugriff: 11.08.2011]

InnoGames GmbH, Die Stämme: URL http://die-stämme.de [Zugriff: 14.08.2011]

Inqnet GmbH, Gondal: URL http://www.gondal.de [Zugriff: 14.08.2011]

JBoss Community, Hibernate: URL http://www.hibernate.org [Zugriff: 11.08.2011]

JBoss Community, Hibernate: URL http://community.jboss.org/wiki/WhoUsesHibernate [Zugriff: 13.08.2011]

Kohana, PHP5 Framework: URL http://kohanaframework.org [Zugriff: 11.08.2011]

Link, J., Adler, F.: *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*. 2. überarbeitete und erweiterte Auflage. Heidelberg 2005.

Microsoft Deutschland GmbH, WCF: URL http://msdn.microsoft.com/de-de/library/ms731082.aspx [Zugriff: 11.08.2011]

Microsoft Deutschland GmbH, LINQ: URL http://msdn.microsoft.com/de-de/library/bb397676.aspx [Zugriff: 13.08.2011]

Microsoft Deutschlang GmbH, SvcUtil.exe: URL http://msdn.microsoft.com/de-de/library/aa347733.aspx [Zugriff: 13.08.2011]

Miskovic, D.: URL http://www.browsergamesliste.com [Zugriff: 11.08.2011]

Modular Gaming: URL http://modulargaming.com [Zugriff: 11.08.2011]

Playa Games GmbH, Shakes & Fidget: URL http://www.sfgame.de [Zugriff: 14.08.2011]

RedMoon Studios GmbH & Co. KG, Knight Fight: URL http://www.knightfight.de [Zugriff: 14.08.2011]

Siedersleben, J.: *Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar*. Heidelberg 2005.

Siedersleben, J.: *Quasar: Die sd&m Standardarchitektur Teil 1*. Online PDF: URL https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-_Quasar_1__sd_m_Brosch_re_.pdf [Zugriff: 11.08.2011]

SpringSource, Inc., Grails: URL http://grails.org [Zugriff: 11.08.2011]

Stargate-Galaxys: URL http://www.stargate-galaxys.com [Zugriff: 14.08.2011]

Travian Games GmbH, Travian: URL http://www.travian.de [Zugriff: 14.08.2011]

Wiarda, I., Little Goblin: URL http://dewarim.de [Zugriff: 11.08.2011]

Wikimedia Foundation Inc.: URL http://de.wikipedia.org/wiki/Browsergame [Zugriff: 11.08.2011]

XS Software JSCo, Zaren Kriege: URL http://www.zarenkriege.de [Zugriff: 14.08.2011]

Tabellenverzeichnis

Tabelle1: Feature-Matrix S.8

Tabelle2: Codeabdeckung.. S.48

Tabelle3: Zeitverhalten.. S.55

Abbildungsverzeichnis

Abbildung1: Siedersleben, J.: *Quasar: Die sd&m Standardarchitektur Teil 1*. Online PDF S.16: URL https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-_Quasar_1__sd_m_Brosch_re_.pdf [Zugriff: 11.08.2011] S.11

Abbildung2: Kompontenschnitt S.20

Abbildung3: Kompositionsstrukturdiagramm der Attribut-Komponente S.21

Abbildung4: Fachliches Datenmodell der Einheiten-Komponente S.25

Abbildung5: Technologiebaum Beispiel S.28

Abbildung6: Login-Prozess S.32

Abbildung7: Gesamtsystem-Architektur S.34

Abbildung8: Architektur der technischen Infrastruktur S.35

Abbildung9: Technik-Architektur S.36

Abbildung10: Prototyp der Webseite S.53

Glossar

Caching: Ein Verfahren um beispielsweise Daten aus einer Datenbank zwischenzuspeichern, um oft verwendete Daten schneller zurückgeben zu können.

Lazy loading: Ein Verfahren, dass unter anderem von Hibernate verwendet wird, um Objekte nicht sofort vollständig aus der Datenbank zu laden. Eigenschaften von Objekten werden beim ersten Zugriff nachgeladen.

Transaktion: Eine Menge von Operationen, bei der entweder alle, oder keine ausgeführt werden.

API (Application Programming Interface): Eine öffentliche Schnittstelle einer Programmbibliothek, um auf die Funktionalität dieser zuzugreifen.

Daemon: Ein Prozess der im Hintergrund abläuft.

Serialisierung: Eine Technik, um Objekte in eine sequenzielle Darstellungsform zu konvertieren. In dieser Form können Objekte beispielsweise über einen Datenstrom versendet werden.

Annotation: Ein Programmiersprachmittel um beispielsweise Member einer Klasse zu markieren, damit sie von WCF serialisiert werden.

Delegat: Ein Funktionszeiger in C#, um beispielsweise Methoden als Parameter übergeben zu können.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 15.08.2011____ ________________________