# Dokumentation des Projektes

# [ Browsergame Castarma ]

SOMMERSEMESTER 2006 - HOCHSCHULE DER MEDIEN

STUDIENGANG MEDIENINFORMATIK

Teilnehmer:

Matthias Bitsch [16131]
Alfred Brose [15701]
Jörg Ingelfinger [15695]
Stefan Nolte [15710]

betreuender Professor:

Prof. Dr. Martin Goik

# Vorwort

Die vorliegende Dokumentation beschreibt die Planung, Organisation, Konzeption und Entwicklung des Projektes „Browsergame Castarma“.

Dieses Dokument soll einen möglichst umfassenden Überblick über die Vorgehensweise der Projektgruppe, bestehend aus Matthias Bitsch, Alfred Brose, Jörg Ingelfinger und Stefan Nolte geben und ist Bestandteil der Benotungsgrundlage für den Betreuer des Projektes, Prof. Dr. Martin Goik.

Es hat uns allen großen Spaß gemacht, bei diesem Projekt mitwirken zu können und es war eine interessante Erfahrung, die erlernten Programmier-Kentnisse in der Praxis umsetzen zu können.

# Inhaltsverzeichnis


Vorwort ........ 2

Inhaltsverzeichnis ........ 3

1. Einleitung ........ 4

   1.1 Was ist ein Browsergame? ........ 4

   1.2 Die Browsergame Szene ........ 5

2. Planung ........ 5

   2.1 Motivation für eigenes Browsergame ........ 5

   2.2 Projektziele ........ 6

   2.3 Projektorganisation ........ 6

3. Der Spielentwurf ........ 7

   3.1 Die Spielidee ........ 7

   3.2 Die Spielstory ........ 7

   3.3 Der Name ........ 7

4. Die Entwicklungsumgebung ........ 8

   4.1 Auswahl der Datenbank ........ 8

   4.2 Auswahl des Webservers ........ 9

   4.3 Auswahl der Frameworks ........ 9

      4.3.1 Hibernate Persistence Framework ........ 10

      4.3.2 Jakarta Velocity template engine ........ 11

      4.3.3 Direkt Web Remoting / AJAX ........ 12

5. Konzeption und Aufbau des Spiels ........ 16

   5.1 „UML"- Diagramm ........ 16

   5.2 Beschreibung und Erläuterung der wichtigsten Klassen ........ 16

      5.2.1 Player Klasse ........ 17

      5.2.2 Castle Klasse ........ 18

      5.2.3 Building / Trainable / Units / Research ........ 19

   5.3 Technologie-Tree ........ 21

   5.4 EventSystem ........ 21

   5.5 Das MessageSystem ........ 23

6. Probleme ........ 24

7. Aktueller Stand und Ausblick ........ 26

# 1. Einleitung

## 1.1 Was ist ein Browsergame?

Ein Browsergame ist ein Spiel, das auf einem oder mehreren Servern läuft und von den Nutzern mit einem Browser gespielt werden kann. Jeder Spieler hat dabei einen eigenen Account, über den er sich einloggen muss, um die Spielfunktionen nutzen zu können.

Browsergames laufen meist textbasiert ab, die Spielaktionen werden über Formularfelder getätigt. Viele Browsergames werden zudem noch durch Flash oder JavaScript aufgewertet, um die User-Navigation zu vereinfachen und zu veredeln.

Es gibt 2 Arten von Spielen, zeitlich unbegrenzte oder rundenbasierte. Ein rundenbasiertes Spiel läuft in einer durch eine Zeitvorgabe zeitlich begrenzte Runde ab, in der die Spieler ihre Aktionen planen und starten können. Dann werden die Aktionen gemeinsam ausgeführt und deren Konsequenzen berechnet. Im Anschluss wird eine neue Runde gestartet, wo die Spieler wieder Aktionen planen können. Die andere Variante, ein zeitlich unbegrenztes Browsergame, ist öfter vorzufinden. Hier läuft die Zeit kontinuierlich ab, die Aktionen werden sofort ausgeführt. Diese Variante erfordert mehr Aktivität des Spielers, um erfolgreich zu sein. Es ist üblich, sich nachts einen Wecker zu stellen, um weitere Aktionen starten zu können, um somit einen Vorteil gegenüber des Gegners zu haben.

Am meisten durchgesetzt hat sich eine Kombination aus Aufbau- und Strategiespiel. Mit Aufbau ist zum Beispiel das Bauen von Gebäuden oder das Trainieren einer Fußballmannschaft gemeint. Der Strategieteil ist in dem Fall das koordinierte Angreifen anderer Städte, die diplomatischen Beziehungen zu anderen Spielern, oder ähnliches. Auch Castarma ist ein Aufbau- und Strategiespiel, wie später näher erläutert wird.

Der Sinn von Browsergames liegt im Messen mit den anderen Spielern, dieses Messen kann verschiedene Formen annehmen, zum Beispiel kann man sich gegenseitig angreifen oder seine Fußballmannschaften gegeneinander spielen lassen. Das Abschneiden der Spieler wird in Ranglisten festgehalten, je nach Erfolg werden Punkte verteilt.

Man hat zudem in Browsergames meistens die Möglichkeit, sich in Gemeinschaften zusammenzufinden, je nach Spielcharakter werden diese Gemeinschaften Allianzen, Clans oder Gilden genannt.

Bekannte Browsergames sind Galaxy Wars, OGame und Inselkampf.

## 1.2 Die Browsergame Szene

Es gibt mittlerweile eine große Vielfalt von Browsergames, seien es Sport-, Kriegs-, Mafia-, Weltraum- oder sogar Alltags-Simulationen, wie zum Beispiel *Schoolwars*.

Wie schon angesprochen, finden sich die Spieler meist in Gemeinschaften zusammen. Jede etwas professionellere Gemeinschaft hat ein eigenes Forum oder sogar eine eigene Webseite. Dort bilden sich dann öfters Communities, die auch neben dem Spielen ihres Browsergames zusammen Dinge unternehmen, wie zum Beispiel Allianztreffen oder LAN-Parties.

Browsergames werden oft von den traditionellen Computer-Spielern belächelt, weil es keine gescheite Grafik und Interaktion gibt, wie dies eben bei normalen Spielen der Fall ist. Doch besonders in Deutschland ist eine große Szene herangewachsen, wo bei vielen Spielern das jeweilige Spiel den Alltag maßgeblich beeinflusst. Zum Beispiel, dass man, wie schon angesprochen, sich nachts einen Wecker stellt, um Aktionen zu starten, oder andere Tätigkeiten im RL (Abkürzung für Real-Life, d.h. das „echte" Leben) ausfallen lässt, um noch erfolgreicher und besser zu werden.

# 2. Planung

## 2.1 Motivation für eigenes Browsergame

Da jeder aus dem Projektteam schon Erfahrungen mit dem Spielen von Browsergames machen konnte, war der Reiz und das Interesse natürlich gross, ein eigenes Spiel zu entwickeln und umzusetzen. Wenn man sich mit einem anderem Spiel beschäftigt, hat man oft Ideen, wie man es verbessern könnte, doch diese Ideen lassen sich meist nicht umsetzen, da man keinen Kontakt zu den Spiel-Entwicklern hat. Auch die grosse Verantwortung, die man durch die Entwicklung und Veröffentlichung von einem solchen Spiel hat, und das Potenzial, was solch ein Browsergame am Markt hat, hat zu unserer Entscheidung für das Projekt beigetragen.

Anhand des Projektes wollten wir natürlich auch viel Erfahrung mit der Organisation und dem Management von professionelleren Software-Projekten sammeln, Spiele-Entwicklung ist schliesslich nichts anderes als Software-Entwicklung.

Auch hatten wir durch das Projekt die Chance, uns mit neuen Technologien wie AJAX oder Hibernate auseinanderzusetzen und diese kennenzulernen.

## 2.2 Projektziele

Wir haben uns folgende Projektziele bis zum Ende des Semesters vorgenommen:

Schaffen einer funktionierenden Basis
Erweiterbarkeit des Codes gewährleisten
MVC-Architektur einhalten

## 2.3 Projektorganisation

Wir haben versucht, eine Struktur in das Projekt zu bringen, indem wir feste Meetings vereinbart haben und eine gemeinsame Kommunikationsplattform genutzt haben.

Wir haben uns jeden Mittwoch-Nachmittag in der HdM getroffen, um Probleme zu diskutieren, Ideen zu sammeln und die weitere Vorgehensweise festzulegen. Falls einer der Projekt-Teilnehmer Probleme hatte, haben wir versucht, diese gemeinsam zu lösen. Ebenfalls haben wir die verschiedenen Aufgaben, je nach Interesse und Qualifikation an die Team-Mitglieder verteilt und dabei darauf geachtet, dass jeder der Teilnehmer einen umfassenden Überblick über das Projekt bekommt und nicht nur einen Teil davon kennenlernt. Diese Aufgabenverteilung haben wir in einem eigens für das Projekt aufgesetzten Wiki-System festgehalten, ebenso wie Recherche-Ergebnisse, Probleme, Ideen und Organisatorisches.

Die Konzeption haben wir zum größten Teil im Team erledigt. Dies geschah entweder auf den wöchentlichen Meetings, oder, falls das zu konzipierende Modul etwas grösser war, auf Programmier-Sessions. Dort haben wir uns jeweils ca. 2 Tage und Nächte bei einem der Teilnehmer getroffen, jeder hatte einen Laptop dabei und so konnten wir in Teamarbeit sehr effizient die verschiedenen Teilmodule des Projektes designen und implementieren. Diese Programmier-Sessions fanden ca. alle 2-3 Wochen statt. Die meiste Implementierungs-Arbeit fand jedoch getrennt statt, doch wir haben darauf geachtet, dass die zu implementierende Problematik so einfach war, dass wenig Probleme auftreten. Dies geschah durch eine gemeinsame intensive und genaue Absprache, wie jeder vorgehen soll. Als Kommunikationsmedium über das Wiki hinaus haben wir ICQ und Skype verwendet.

# 3. Der Spielentwurf

## 3.1 Die Spielidee

In den ersten Meetings haben wir besprochen, was für eine Art von Browsergame wir entwickeln wollen. Der erste Vorschlag war eine Charaktersimulation, bei der jeder Spieler einen eigenen Kämpfer steuert, den man trainieren und neue Kampftechniken erlernen lassen kann, Trainingshallen anmieten kann und sich eine gute Ausrüstung kaufen kann. Mit diesem Charakter sollte man dann an Turnieren teilnehmen und gegen andere Spieler antreten. Diese Idee haben wir relativ schnell verworfen, da wir in der Spielidee von Castarma mehr Erfolgschancen gesehen haben.

Jeder Spieler herrscht über eine eigene Burg. Anfangs kann man nur wenige Gebäude ausbauen. Je höher die Stufe eines Ressourcengebäudes ist, desto mehr Ressourcen bekommt der Spieler beispielsweise. Bei Erreichen einer bestimmten Stufe wird ein neues Gebäude oder eine Forschung freigeschalten. So kann man mit der Zeit immer mehr Gebäude bauen und nach einer bestimmten Zeit auch Einheiten ausbilden. Forschungen dienen dazu, die Gebäude/Einheiten zu verbessern oder neue Gebäude/Einheiten freizuschalten. Man hat ausserdem die Möglichkeit, andere Spieler anzugreifen, mit ihnen zu handeln oder ihnen Nachrichten zu schicken. Ziel ist es, die stärkste Burg mit der stärksten Armee zu besitzen und somit der mächtigste Spieler zu sein. Dies kann man durch eine hohe Aktivität, aber auch durch geschickte Diplomatie zu den anderen Spielern erreichen.

Die Spielidee wird noch erweitert, es werden weitere Features eingebaut werden, dies ist unter dem Punkt „Aktueller Stand und Ausblick“ nachzulesen.

## 3.2 Die Spielstory

Castarma spielt im Mittelalter, zur Zeit von Rittern und Hofdamen. Nach einem grossflächigen Krieg gibt es im Lande keine Herrschaft und somit keine Struktur mehr, es herrscht Anarchie. Jeder Spieler kann eine Armee aufbauen und auch die dunkle Seite der Macht, wie Hexenmeister, Dämonen und Bestien, beschwören - das ist jedoch nur in sehr weit entwickelten Burgen möglich, da dies einen hohen Wissensstand vorraussetzt, den man durch Forschungen erweitern kann. Die verschiedenen Burgherren haben somit die Chance, durch geschickte Kriegsführung und diplomatische Verhandlungen die Herrschaft zu erlangen und sich zu einer Gilde zu verbünden, um sich gegenseitig zu helfen.

## 3.3 Der Name

Castarma besteht aus den lateinischen Wörtern *castellum* (= die Festung, das Kastell) und *arma* (= die Waffen, der Krieg) und hat somit die Bedeutung „Krieg der Festungen“.

# 4. Die Entwicklungsumgebung

Zu Beginn des Projektes wurden die technischen Anforderungen, die eine online multiplayer Spielplattform stellt diskutiert. Ein besonderes Augenmerk lag dabei auf der Auswahl der Datenbank, des Webservers und der zu verwendenden Frameworks.

## 4.1 Auswahl der Datenbank

Die wichtigsten Kriterien bei der Auswahl der Datenbank, die betrachtet wurden, waren folgende:

- die DB sollte frei verfügbar sein (OpenSource)
- die DB sollte stabil laufen und eine hohe Verfügbarkeit haben
- die DB sollte bedienfreundlich sein (Admin-Tools, etc.)
- die DB sollte Transaktionen unterstützen

Diese Auswahlkriterien trafen auf zwei OpenSource Datenbanken zu – MySql und PostgreSQL. Beide haben hervorragende Referenzen, unterstützen Transaktionen und weisen eine hohe Performance auf. Die Tabelle unten zeigt den direkten Vergleich der Systeme.

| | POSTGRESQL | MYSQL |
|---|---|---|
| **ANSI SQL** | Sehr nahe am ANSI SQL Standard | Folgt nur einigen ANSI SQL Standards |
| **Performance** | Langsamer als MySql | Schneller als PostgreSQL |
| **Transaktionen** | Ja | Ja (nur bei Verwendung von InnoDB Tabellen) |
| **DB Replikationen** | Ja | Ja |
| **Foreign Key** | Ja | Nein |
| **Views** | Ja | Nein |
| **Stored Procedures** | Ja | Nein |
| **Trigger** | Ja | Nein |
| **Unions** | Ja | Nein |
| **Full joins** | Ja | Nein |
| **Constraints** | Ja | Nein |
| **Windows Unterstützung** | Ja | Ja |
| **ODBC** | Ja | Ja |
| **JDBC** | Ja | Ja |
| **Verschiedene Tabellen Typen** | Nein | Ja |

Vergleich MySql vs. PostgreSQL

Wie man unschwer erkennen kann, schneidet MySql in der Performance besser ab, was auch die hohe Verbreitung im Bereich der Web-Anwendungen erklärt. Unsere Entscheidung viel jedoch trotzdem auf die PostgreSQL Datenbank, da vor allem die Unterstützung von Fremdschlüsseln das Mapping der Java-Objekte in eine relationale Datenbank wesentlich erleichtert. Ein weiterer Punkt, der zu Gunsten der PostgreSQL Datenbank sprach, war die uneingeschränkte Unterstützung von Transaktionen.
Obwohl wir im späteren Projektverlauf doch recht wenig direkt mit der Datenbank gearbeitet haben, und das gesamte OR-Mapping von dem Hibernate Persistenz-Framework verwaltet wurde, war die Datenbankwahl eine der wichtigsten Entscheidungen, die im Laufe des Projekts zu treffen waren.

## 4.2 Auswahl des Webservers

Die Wahl des Webservers fiel mangels ernsthafter Konkurrenz eindeutig auf den *Apache Tomcat Webserver*. Es wurde die Version 5.5 eingesetzt. Dabei war zunächst unklar, ob wir die Servlet-Technologie verwenden sollen, oder doch die von Sun empfohlenen Java Server Pages zur sauberen Implementierung einer Model-View-Controller (MVC) Architektur. Da eine strikte Trennung der Logik von der Präsentationsschicht eines der internen Primäranforderungen an das Projekt war, jedoch die Handhabung der JSP's sich erfahrungsgemäß aus den Übungen zu der Vorlesung „Entwicklung von Medienanwendungen" schwierig erwies, entschieden wir uns doch für die reinen Servlets in Verbindung mit der template engine „Velocity" aus dem Hause Apache Jakarta.

## 4.3 Auswahl der Frameworks

Als die Wahl des Webservers und der Datenbank fest stand und die ersten Pseudo-UML-Diagramme die ungefähre Komplexität des Projektes veranschaulichten, war es relativ schnell klar, dass das Pensum im Rahmen eines Semesters nicht zu bewältigen wäre. Vor allem das OR-Mapping der Objekte mit allen Beziehungen und Vererbungen, sowie das Management der Datenbanktransaktionen einer Mehrbenutzeranwendung, würde den Großteil der zur Verfügung stehenden Zeit in Anspruch nehmen. Somit war klar, dass Hilfs-Frameworks zur persistenten Datenhaltung und Transaktionsmanagement und zur Realisierung der MVC Architektur unumgänglich sind. Daraus ergaben sich folgende Frameworks:

- Hibernate Persistence Framework zur persistenten Datenspeicherung
- Jakarta Velocity template engine
- Direkt Web Remoting (AJAX Framework)

Das Direkt Web Remoting (DWR) Framework war zu Beginn des Projektes ein kleines Experiment um den Funktionsumfang des DWR zu testen. Die Ergebnisse waren so überzeugend, dass wir uns spontan entschlossen haben, es im Projekt einzusetzen.

### 4.3.1 Hibernate Persistence Framework

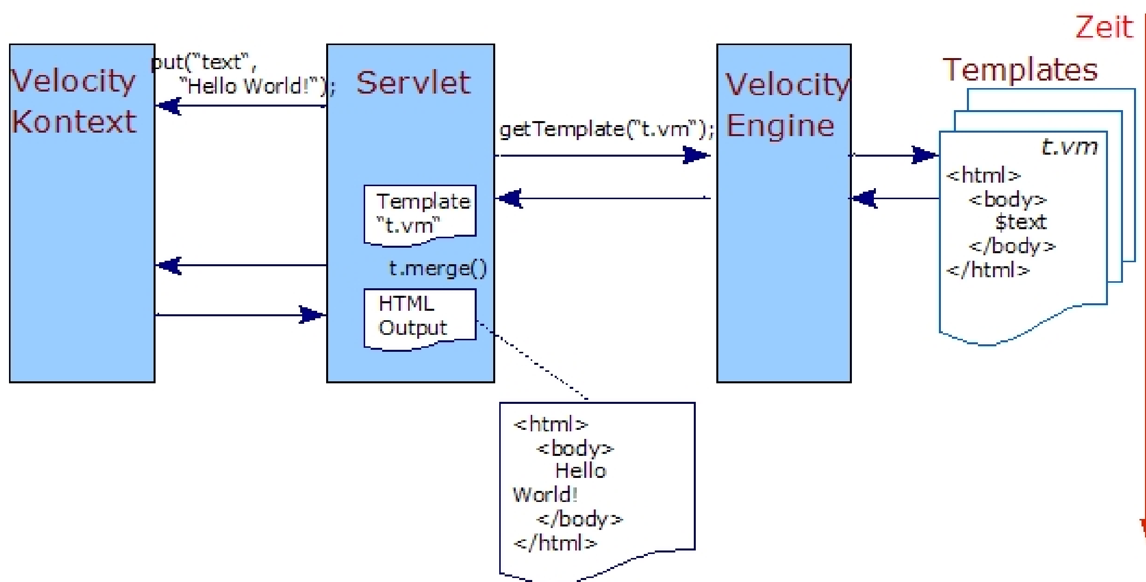
Hibernate ist ein sehr mächtiges Werkzeug um (Java)-Objekte in einer relationalen Datenbank zu speichern und wieder zu laden. Dabei bleiben alle Beziehungen und Vererbungen der Objekte erhalten und sind nach dem Laden wieder verfügbar. Hibernate kapselt die gesamte Datenbanklogik, sodass sich der Programmierer in der Regel gar nicht darum kümmern muss. Es wird lediglich eine xml-Konfigurationsdatei angelegt, in der der Pfad zur Datenbank, Nutzername und Password sowie einige weitere Parameter einzustellen sind. Desweiteren muss für jede Klasse, deren Objekte persistiert werden sollen ein sogenanntes Mapping-File angelegt werden. Das Mapping-File ist wie auch die Konfigurationsdatei eine xml-Datei und ist der Kern jeder Hibernate Anwendung. Darin wird festgelegt, wie ein Objekt in der Datenbank gehalten werden soll; Ob zum Beispiel alle Attribute mitgespeichert werden sollen oder nur die, die den Modifier *public* haben. Darüberhinaus enthält das Mapping-File auch Angaben zu den Beziehungen und deren Kardinalitäten, die abgebildet werden sollen. Unten sehen Sie ein typisches Mapping-File.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
	"-//Hibernate/Hibernate Mapping DTD 3.0 //EN"
		"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="castarma.main">
	<class name="Player" table="players">
		<id name="id" type="long" column="player_id">
			<generator class="increment"/>
		</id>
		<property name="name" column="name" unique="true"/>
		<property name="email" column="email" unique="true"/>
		<property name="password" column="password"/>
		<property name="registrationDateTS" column="registration_date_ts"/>
		<property name="lastLoginTS" column="last_login_ts"/>
		<property name="loggedIn" type="boolean" column="logged_in"/>

		<one-to-one name="mainCastle" class="Castle" column="castle_id"
cascade="all"/>
	</class>
</hibernate-mapping>
```
Ein typisches Hibernate Mapping-File

Dies ist ein Mapping-File für Objekte der Klasse Player. Darin werden den Attributen der Objekte Spaltennamen in der Tabelle *players* zugewiesen, so wird zum Beispiel das Attribut *loggedIn* in der Spalte *logged_in* gespeichert. Unter den property-Angaben wird bestimmt wie Beziehungen der Objekte abgebildet werden. Das *<one-to-one .. />* xml-Tag gibt an, dass Objekte der Klasse Player ein Objekt der Klasse Castle haben (Komposition), das in der Datenbank über die Spalte *castle_id* referenziert wird. Das Attribut *cascade=„all“* bestimmt, dass die Objekte, die in Beziehung zueinander stehen kaskadiert behandelt werden. So wird beim Laden des Player-Objektes aus der Datenbank das Castle-Objekt automatisch auch geladen.
Nach diesem Schema lassen sich beliebige Datentypen wie zum Beispiel Arrays oder Listen abbilden. Eine detaillierte Beschreibung des Hibernate Frameworks würde den Rahmen dieser Dokumentation sprengen, deswegen verweise ich an dieser Stelle auf das Standardwerk zum Thema Hibernate von Christian Bauer und Gavin King „Hibernate in Action“.

### 4.3.2 Jakarta Velocity template engine

Wie im Kapitel 4.2 schon angesprochen, war eine strikte Trennung der Logik (Java-Code) von der Darstellungsschicht (HTML-Code) eine wichtige Vorgabe, die wir unbedingt einhalten wollten und auch bis auf eine kleine Ausnahme (siehe MessageSystem) eingehalten haben. Wer schon mal mit Servlets gearbeitet hat, weiß wie unübersichtlich ein typisches Java-HTML-Gemisch in kürzester Zeit wird und wie mühselig es ist, wenn man mal „kurz" das Design ändern will. Die Alternative JSP ist in dieser Hinsicht zwar besser, aber das Herumhantieren mit den Tag-Libraries kann auch sehr schnell unübersichtlich werden. Eine template engine zu verwenden wurde daher schon sehr früh diskutiert, nicht zuletzt auf Grund positiver Erfahrungen mit template engines in Verbindung mit PHP. Wie es sich im Laufe des Projektes herausgestellt hat, ist das in unseren Augen die beste Lösung um eine MVC Architektur mit Java Servlets zu realisieren. Anhand des Schaubildes lässt sich die Funktionsweise der engine in groben Zügen erklären.




Jakarta Velocity Funktionsweise

Zu Beginn der Session werden im Servlet Instanzen von VelocityContext und VelocityEngine erzeugt. VelocityContext hält alle Daten, die im Template verfügbar sein sollen. In unserem Beispiel fügen wir mit dem Befehl *VelocityContext.put()* einen String *text* mit dem Inhalt *„Hello World!"* hinzu. Neben primitiven Datentypen wie Zeichenketten ist es auch möglich ganze Objekte dem Kontext hinzuzufügen. Nachdem wir dem Kontext einen String hinzugefügt haben, möchten wir auch ein Template parsen. Velocity Templates sind im Grunde nichts anderes, als gewöhnlicher HTML-Code vermischt mit Velocity Template Language (VTL). Die VTL bietet neben simplen Zugriffsmöglichkeiten auf Variablen und Objekte auch komplexere Kontrollstrukturen wie Schleifen um Arrays zu durchlaufen oder *if-else*-Verzweigungen.

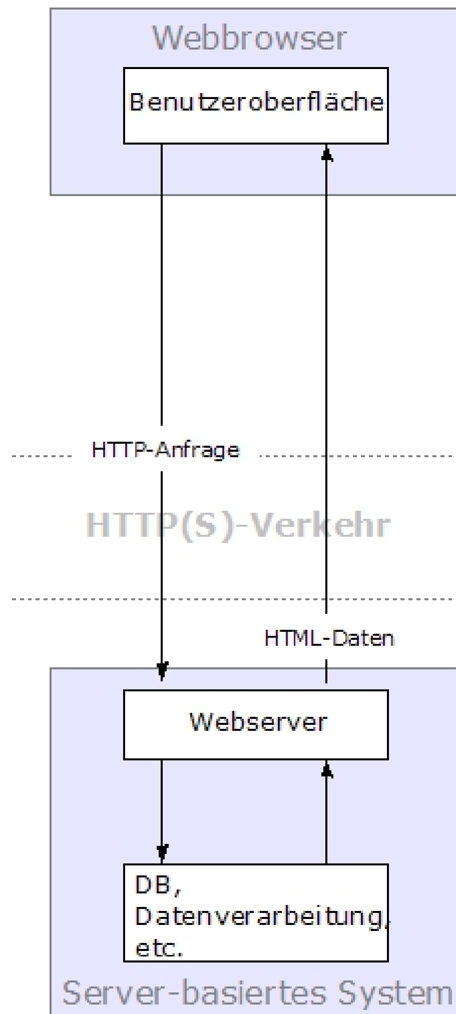
Um aus einem Template „fertigen“ HTML-Code zu erzeugen, laden wir das entsprechende Template (hier „t.vm“) mit dem Befehl *VelocityEngine.getTemplate(„t.vm“);* in das Servlet. Anschließend wird das geladene Template mit dem Velocity Kontext „gemerged“ - das heißt die VTL-Anweisungen im Template werden durch die Veloctity Engine auf den Daten im Kontext abgearbeitet und die entsprechenden Stellen mit den Ergebnissen aufgefüllt. Wie man auf dem Bild oben sieht, wird aus der VTL-Anweisung *$text* im Template, was in VTL-Syntax einen einfachen Zugriff auf die Variable mit dem Namen text bedeutet, ein „Hello World!“ im Output des Befehls *merge()*. Diesen Output kann man gleich ausgeben oder, wie in unserem Fall als return-Wert an ein anderes Servlet delegieren.

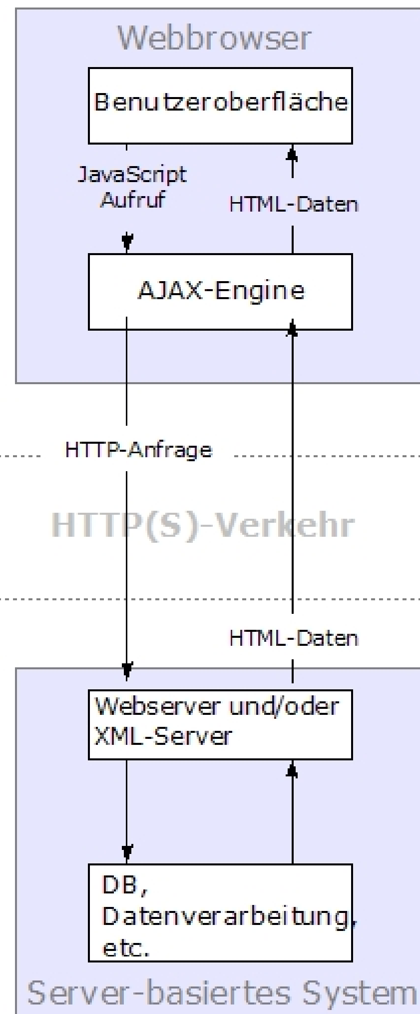
Die Velocity Engine lässt sich schon nach extrem kurzer Einarbeitungszeit sehr vielseitig einsetzen und bietet ein sehr hohes Maß an Flexibilität. Vor allem die Trennung zwischen Logik und Darstellung lässt sich sehr leicht und vor allem konsequent umsetzen. Eine umfangreiche Dokumentation finden Sie unter http://jakarta.apache.org/velocity/docs/.

### 4.3.3 Direkt Web Remoting / AJAX

Das Direkt Web Remoting (DWR) ist ein Java-basiertes Framework, das die AJAX-Funktionalitäten für Webanwendungen, die auf Servlets basieren anbietet. Bevor ich aber die Funktionsweise von DWR beschreibe, ist es notwendig erst die allgemeine Prinzipien der AJAX-Technologie zu erläutern.

AJAX ist eine Abkürzung für ***A****synchronous* ***J****avaScript* ***a****nd* ***X****ML*. Es bezeichnet ein Konzept der asynchronen Datenübertragung zwischen dem (Web)Server und dem Browser. Das heißt soviel, dass das klassische Neuladen der Seite bei jeder Aktion des Benutzers komplett entfällt. Stattdessen werden nur die auch wirklich geänderten Teile der Webseite zur Laufzeit dynamisch nachgeladen oder verändert. Die folgende Grafik verdeutlicht den wesentlichen Unterschied zwischen einer klassischen Webanwendung und einer AJAX-basierten.




Unterschied zwischen klassischen Webanwendungen und AJAX-Webanwendungen (Quelle: de.wikipedia.org)

Bei einer klassischen Webanwendung werden alle Aktionen des Benutzers wie zum Beispiel Klicks auf Buttons direkt an den Webserver gesendet. Zwangsläufig ist mit jeder Aktion ein HTTP-Request verbunden und damit auf ein Neuladen der gesamten Seite, da der Webserver immer nur komplette HTML-Seiten generiert. Bei einer AJAX-Anwendung werden alle Aktionen des Benutzers an die AJAX-Engine geleitet, die in einer Clientseitigen Scriptspache geschrieben ist. In der Regel wir JavaScript eingesetzt. Anhand dieser Aktionen erzeugt die AJAX-Engine einen asynchronen XMLHttp-Request, der kein Neuladen der Seite erfordert. Die XMLHttp-Requests werden Serverseitig abgearbeitet und statt HTML werden XML-Daten an den Client zurückgesendet. Diese XML-Daten werden von der AJAX-Engine entgegengenommen und entsprechend verarbeitet, zum Beispiel werden in ein div-Element aktualisierte Daten geschrieben oder eine JavaScript-Funktion aufgerufen.

Wie schon kurz angemerkt, basiert die AJAX-Technologie auf dem sogenanntem *XMLHttpRequest*. Die meisten Browser implementieren die JavaScript Klasse XMLHttpRequest, der Internet Explorer hat ein analoges AktiveX-Objekt. Die Instanzen dieser Klasse sind in der Lage asynchrone Requests an den Server zu schicken. Asynchron bedeutet, dass der Abarbeitungsfluss der Seite im Client nicht unterbrochen wird. Bei einer schnellen Internetverbindung merkt der

Nutzer nicht mal, dass ein Request an den Server gesendet wurde. Desweiteren ist es möglich eine sogenannte *callback*-Funktion festzulegen, die die Antwort des Servers nach Erhalt bearbeitet.

Die Antworten des Servers liegen in zwei Arten vor – als Textstring und als ein XMLDocument-Objekt. Um die volle Mächtigkeit von JavaScript und des XmlHttpRequests auszunutzen wird in der Regel das XMLDocument als Antwort verarbeitet, da es mittels JavaScript-nativen DOM-Methoden sehr einfach ist, durch die Struktur des XMLDocument zu navigieren. Eine umfangreiche AJAX-Einführung finden Sie unter http://developer.mozilla.org/de/docs/AJAX:Getting_Started.

Wie man leicht erkennen kann, hat man mit AJAX ein sehr mächtiges Werkzeug in der Hand um Webanwendungen zu entwerfen, die sich von herkömmlichen Desktop-Anwendungen kaum unterscheiden. Die größten Nachteile dieser Technik sind der nicht mehr funktionierende Zurück-Button des Browsers und die passive Rolle des Servers.

Das erste Problem ist das wichtigste Argument aller AJAX-Gegner. Durch den asynchronen Aufbau der Seite kriegt der Browser die Änderungen gar nicht mit – für ihn ist die Seite immer noch die, die er geladen hat. Viele AJAX-unerfahrene Benutzer versuchen immer wieder eine bestimmte Aktion rückgängig zu machen indem sie auf den Zurück-Button drücken. Der Ärger ist dann groß, wenn man anstatt eines Undo-Vorgangs auf der Seite landet, von der man zu der AJAX-Anwendung gelang. Zur Zeit gibt es noch keine Standardlösung dieses Problems, deswegen ist es unumgänglich dem Benutzer schon zu Beginn der Session einen Hinweis zu geben, dass der Zurück-Button nicht wie erwartet funktioniert. Eine andere Lösung, die auch oft angewendet wird ist das Öffnen der AJAX-Anwendung in einem neuen Fenster, sodass der Browser keine Back-Referenz besitzt und den Button deaktiviert.

Das zweite Problem betrifft weniger die Benutzer der AJAX-Anwendung, sondern die Entwickler. Oftmals ist es notwendig bestimmte Bereiche der Anwendung zyklisch zu aktualisieren. Da der Server selbst nicht in der Lage ist die aktuellen Daten an den Client zu schicken, muss der Client in regelmäßigen Zeitintabständen Requests an den Server schicken und prüfen, ob die Daten aktualisiert wurden. Dieses andauernde Polling erzeugt eine immense Last auf dem Server, die nicht mit der Auslastung herkömmlicher Anwendungen zu vergleichen ist. Es gibt zwar Lösungen um ein permanentes Polling zu vermeiden, jedoch leidet darunter die Skalierbarkeit einer AJAX-Anwendung.

Nachdem wir die AJAX-Technologie ausgiebig erörtert haben, können wir die Funktionalitäten des DWR Frameworks betrachten. Um DWR benutzen zu können, muss in der *web.xml* die DWR-engine (Servlet) auf einen eindeutigen Pfad gemappt werden. Desweiteren muss in der DWR-Konfiguration *dwr.xml* festgelegt werden, welche Klasse die HTTP-Requests entgegennimmt und verarbeitet. Unten sehen Sie, wie eine typische *dwr.xml* aussieht.

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
  <allow>
    <create creator="new" javascript="AjaxServer" scope="session">
      <param name="class" value="ajaxServer.AjaxServer"/>
    </create>
  </allow>
</dwr>
```

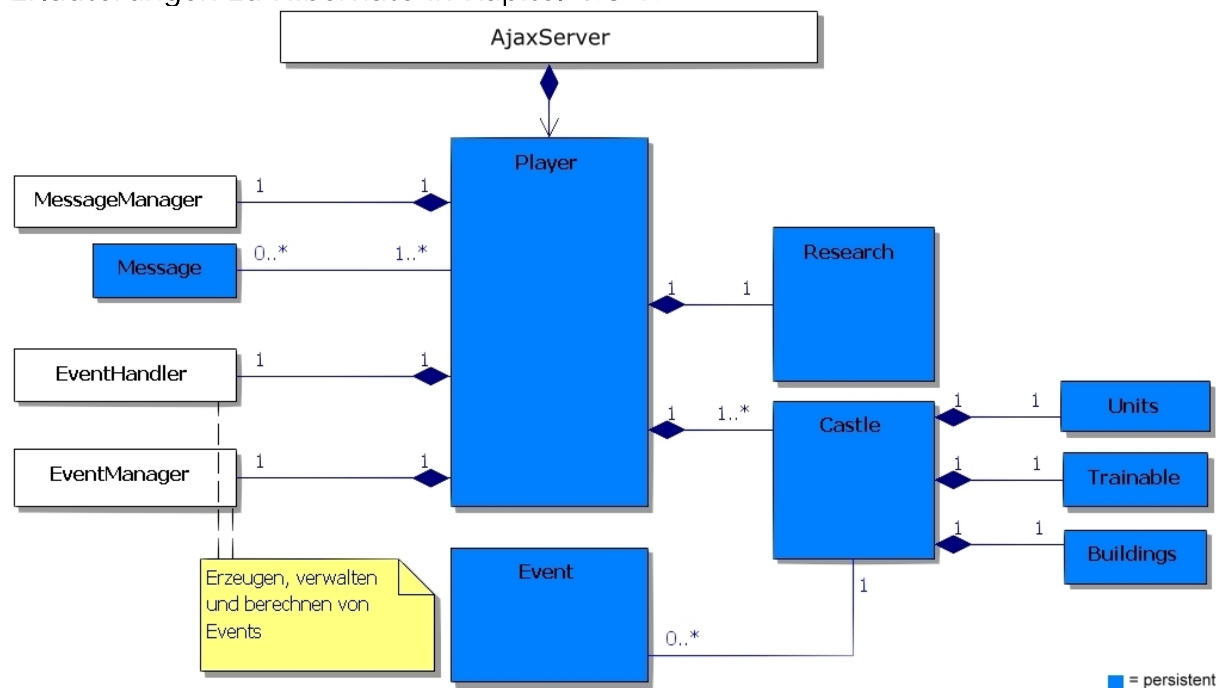
Eine typische DWR Konfiguration

Serverseitig besitzt DWR eine Funtion um die Server-Klasse, die in der *dwr.xml* angegeben wird zu analysieren und das JavaScript dazu zu erzeugen. Maßgeblich ist dabei der Reflection-Mechanismus. DWR erzeugt für jede *public* Methode der Server-Klasse eine JavaScript-Funktion, die die Klassenmethode aufrufen kann. Somit erlaubt es DWR aus dem Browser heraus mittels JavaScript auf Methoden der Klasse auf dem Webserver zuzugreifen.
Clientseitig kapselt DWR die Erzeugung, Versand, Empfang und Verarbeitung der Requests und Server-Antworten und stellt über das generierte JavaScript den Zugriff auf den Server zur Verfügung.
Neben der Übertragung von reinem Text oder HTML ist es mit DWR auch möglich Aufrufe von JavaScript-Funktionen in die XML-Antwort einzubetten. Diese Aufrufe werden auf dem Client nach Erhalt der XML-Antwort ausgeführt. Somit bietet die AJAX-Technologie im Zusammenspiel mit DWR die Möglichkeit höchst dynamische und vielseitige Webanwendungen zu realisieren, die sich in der Benutzerfreundlichkeit kaum von Desktop-Anwendungen unterscheiden.

# 5. Konzeption und Aufbau des Spiels

## 5.1 „UML"- Diagramm

Das nachstehende, um Funktionen und Variablen reduzierte Klassendiagramm zeigt die wichtigsten Klassen und ihre Zusammenhänge innerhalb unseres Projektes. Die blau gefärbten Klassen werden von Hibernate persistent gehalten, d.h. erzeugte Objekte werden dauerhaft in die Datenbank gespeichert. Siehe Erläuterungen zu Hibernate in Kapitel 4.3.1.




Die wichtigsten Klassen in Castarma

## 5.2 Beschreibung und Erläuterung der wichtigsten Klassen

In diesem Kapitel soll ein Überblick über den Sinn und Zweck einiger dieser persistenten Klassen gegeben werden und ggf. auf Besonderheiten der Implementierung eingegangen werden.

Alle Klassen die vom Persistenzframework Hibernate verwaltet werden sollen benötigen einige besondere Eigenschaften:

- leerer Standardkonstruktor,
- einen ObjektIdentifier (ID)
- vollständige Getters & Setters
- ein Mapping-File (XML mit Abbildungsvorschriften für die Persistenz)

Alle angesprochenen Klassen befinden sich im Package *castarma.main*.

### 5.2.1 Player Klasse

Objekte der Player-Klasse enthalten, wie der Name verrät, Informationen über den jeweiligen Spieler. Daten wie Name, E-Mail, Passwort (SHA-1-Hash-Wert), Zeitpunkt der Registrierung, letzter Login und aktueller Status (eingeloggt: ja/nein) sind hier hinterlegt und werden bei Änderungen aktualisiert.

Da jeder Account mindestens eine Burg besitzt, hat jeder Spieler ein Objekt vom Typ Castle, welches die Hauptburg (mainCastle) des Spielers meint. Die Möglichkeit mehrere Burgen zu besitzen ist vorgesehen aber noch nicht vollständig implementiert.

Um einmal getätigte Forschungen nicht in späteren Burgen erneut wiederholen zu müssen, haben wir uns entschlossen den Forschungsstand eines Spielers unabhängig von den Burgen zu machen. Deshalb wurde zentral und einmalig pro Spieler, eine Komposition auf ein Objekt des Typs Research (myResearches) eingerichtet. Genaueres zum Aufbau der Research Klasse entnehmen Sie bitte dem Kapitel 5.2.3.

Eine login() Methode setzt bei einem Login die „zuletzt eingeloggt“ Timestamp neu und lädt bestimmte Klassen, die einem Spieler nur zur Verfügung stehen müssen wenn er eingeloggt ist.

Lädt man ein Player Objekt, hat man den ganzen daran angebundenen Objektbaum dank Hibernate automatisch zur Verfügung und muss nicht jedes benötigte Objekt manuell laden.

Betrachten wir die Zeilen im Hibernate Mapping, die dies für die Player <-> Castle Beziehung ermöglichen:

Player-seitig:

```
<many-to-one name=„mainCastle“ class=„Castle“ column=„castle_id“ cascade=„all“/>
```

Castle-seitig:

```
<one-to-one name=„owner“ class=„Player“ property-ref=„mainCastle“/>
```

Außerdem kann man sehen dass auch ohne weiteres rückwärtig von Castle auf den dazugehörigen Player navigiert werden kann, da die Variable „owner“ wiederum auf ein Objekt der Player-Klasse zeigt.

Betrachten wir mittels eines Datenbank-Administrationstools (in unserem Fall PGAdmin) die Tabellen Player und Castle, erhalten wir z.B. folgende Ansichten:

Player-Tabelle:

| | player_id [PK] int8 | name varchar | email varchar | password varchar | istration_date int8 | last_login_ts int8 | logged_in bool | castle_id int8 | researches_id int8 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | ich | ich@castar | | 1150899820767 | 1151607015859 | FALSE | 1 | 1 |
| 2 | 2 | feind | feind@cas | | 1150899926880 | 1153165051352 | TRUE | 2 | 2 |
| 3 | 3 | sven | jkdnj@sjfk | | 1150926347321 | 1151510181187 | FALSE | 3 | 3 |
| 4 | 4 | du | roland.es@ | | 1150926487743 | 1150928235416 | FALSE | 4 | 4 |
| * | | | | | | | FALSE | | |

Castle-Tabelle:

| | castle_id [PK] int8 | castlename varchar | posx int4 | posy int4 | last_res_calc_ts int8 | res_wood float4 | res_food float4 | res_iron float4 | res_stone float4 | res_gold float4 | trainable_id int8 | buildings_id int8 | units_id int8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | meineburg | -1 | 0 | 1151607120129 | 737676 | 50000 | 472183 | 600000 | 75294 | 1 | 1 | 1 |
| 2 | 2 | feindburg | 5 | -4 | 1153165405651 | 40000 | 40000 | 833642 | 70000 | 201178 | 2 | 2 | 2 |
| 3 | 3 | wartburg | 4 | 1 | 1153165177934 | 447796 | 224106 | 224106 | 447796 | 522 | 3 | 3 | 3 |
| 4 | 4 | guttenberg | -2 | -4 | 1150928002962 | 9940 | 9940 | 9940 | 9940 | 9660 | 4 | 4 | 4 |

Zu beachten ist, dass es in der Castle-Tabelle keiner Spalte mit owner_id bedarf. Es genügen die Definition von „owner" im Klassencode und die Mapping Anweisungen um eine Navigierbarkeit zu garantieren.

### 5.2.2 Castle Klasse

Ein Castle ist die „ingame" - Heimat eines Spielers, d.h. der Ort an dem er Gebäude errichten und ausbauen, Einheiten ausbilden und Forschungen tätigen kann. Variablen wie der Name der Burg, kartesische Koordinaten für die Platzierung der Burg auf der Karte sowie die aktuell vorhandenen und verfügbaren Rohstoffmengen (Nahrung, Holz, Eisen, Stein, Gold) sind hier angesiedelt.

Jede User-Aktion die im Spiel vollzogen werden kann, beeinflusst in irgendeiner Form immer das Castle. Um dies zu veranschaulichen Betrachten wir die Menge aller User-Aktionen und die daraus resultierenden Änderungen.

| **Aktion** | **Veränderung in…** |
|---|---|
| Gebäudebau | Gebäudestufen, verfügbare Technologien, Ressourcen |
| Forschung | Forschungsstufen, verfügbare Technologien, Ressourcen |
| Truppenausbildung | Vorrätige Truppen, Ressourcen |
| Ausgehender eigener Angriff | Vorrätige Truppen |
| Eingehender fremder Angriff | Vorrätige Truppen, Ressourcen |
| Ausgehender Handel | Vorrätige Truppen, Ressourcen |
| Eingehender Handel | Ressourcen |

Hier wird die zentrale Stellung der Castle-Klasse deutlich. Sie benötigt zudem Zugriff auf alle betroffenen Bereiche. Das wird wie folgt bewerkstelligt:

Für die Speicherung der verschiedenen Gebäudestufe hat jedes Castle ein Objekt vom Typ *Building* (myBuildings). Zusätzlich hat jedes Festung ein Objekt vom Typ *Trainable* (myTrainable) in dem festgehalten wird welche der 18 Einheitentypen bereits durch Ausbau und Forschung freigeschaltet wurden und

somit ausgebildet werden können. Die Anzahl der tatsächlich fertigen und verfügbaren Truppen, die in der Burg lagern, sind in einem Objekt des Typs *Units* (myUnits) untergebracht.

Eine erwähnenswerte Methode der Castle-Klasse ist die Rohstoffberechnung (*CalculateResource()*), die abhängig von Gebäudestufen der ressourcenerzeugenden Gebäude, sowie verstrichener Zeit seit der letzten Berechnung, den Zuwachs an Rohstoffen für eine Burg berechnet. Die gesamte Berechnung basiert auf Differenzen zwischen Timestamps.

### 5.2.3 Building / Trainable / Units / Research

Die vier simplen Klassen *Building*, *Trainable*, *Units* und *Research* sind sehr ähnlich implementiert. Sie bestehen im Wesentlichen jeweils aus einem int-Array in dem die Stufen der Gebäude / Forschungen bzw. die Anzahl der Einheiten persistent gehalten werden.

Die einzelnen Felder des Array repräsentieren dabei immer genau eine Einheit, Forschung oder ein Gebäude.

Hier *Building* exemplarisch für alle vier Klassen:

- Das Array wird bei Erzeugung mit den Startzuständen für die Gebäude gefüllt

    - -1 = nicht verfügbar
    - 0 = kann gebaut werden
    - 1 = Stufe 1
    - 2 = Stufe 2
    - ... u.s.w.
    - -9 = unbenutztes Feld für Erweiterungen

- danach von Hibernate in die Datenbank gelegt.

- Wurde jetzt ein Gebäude um eine Stufe ausgebaut, inkrementiert man das Array im entsprechenden Feld um eins.

- Wird ein Gebäude neu verfügbar (z.B. durch Forschungen ) setzt man den Wert des Gebäudes von -1 auf 0 damit es dem Spieler im Baumenü im Frontend angezeigt wird.

Um Fehler zu unterbinden haben wir allen Gebäuden, Einheiten und Forschungen IDs vergeben, die sich durch die gesamte Implementierung ziehen. Die IDs entsprechen dabei immer genau dem Array-Index.

Das Haupthaus hat z.B. die buNr (buildingNumber) 1 d.h. im GebäudeArray ist es folglich unter Gebäude[1] zu finden.

| | belongs_to [PK] int8 | stage int4 | bunr [PK] int4 |
|---|---|---|---|
| 1 | 1 | -1 | 0 |
| 2 | 1 | 50 | 1 |
| 3 | 1 | 25 | 2 |
| 4 | 1 | 25 | 3 |
| 5 | 1 | 15 | 4 |
| 6 | 1 | 9 | 5 |
| 7 | 1 | 1 | 6 |
| 8 | 1 | 1 | 7 |
| 9 | 1 | 13 | 8 |
| 10 | 1 | 4 | 9 |
| 11 | 1 | 1 | 10 |
| 12 | 1 | 15 | 11 |
| 13 | 1 | 3 | 12 |
| 14 | 1 | 1 | 13 |
| 15 | 1 | 1 | 14 |
| 16 | 1 | -9 | 15 |
| 17 | 1 | -9 | 16 |
| 18 | 1 | -9 | 17 |
| 19 | 1 | -9 | 18 |
| 20 | 1 | -9 | 19 |

Die Spalte „belongs_to“ ist die Zuordnung eines Building-Arrays zu einem Castle, „bunr“ entspricht den Gebäude-IDs und „stage“ entspricht dem aktuellen Zustand des Gebäudes (-1 = nicht verfügbar,0 = verfügbar, 1,2,3,...n = Stufe, -9 = unbenutztes Feld).

Die Arrays wurden genügend groß definiert um im Falle einer Erweiterung ausreichend Platz für neue Gebäude, Einheiten und Forschungen zu haben. Zudem wäre es mit vertretbarem Aufwand möglich die bestehenden Arrays beliebig zu vergrößern.

Die Implementierung der Klassen Trainable, Unit und Research geschah analog.

Die Umsetzung der Datenhaltung in komfortablen iterierbaren Arrays war lustigerweise zunächst nicht unsere erste Wahl. Anfangs dachten wir an Variablen, d.h. pro Element, wie z.B. Haupthaus, eine Klassenvariable anzulegen, was wir jedoch schnell wieder verworfen, da z.B. keine einheitlichen Funktionen zur Gebäudestufenerhöhung machbar waren. Ebenso war keine Iteration über die Elemente denkbar und dies war unabkömmlich.

Auch eine komplette OO-Struktur bis auf jeden verschiedenen Gebäude- / Einheitentyp hatten wir kurz im Sinn. Das wäre aber unserer Meinung nach, dem bekannten Sprichwort „...mit Kanonen auf Spatzen geschossen...“ nicht sehr weit entfernt. Auch wenn eine OO-Lösung höchster Granularität den einen oder anderen Vorteil mit sich gebracht hätte, betrachten wir, unter Berücksichtigung des Aufwandes, die von uns erarbeitete Lösung als optimal für den angestrebten Verwendungszweck.

## 5.3 Technologie-Tree

Die Aufgabenstellung des Techtree ist es, alle Abhängigkeiten des Spiels abzubilden. Welche Voraussetzungen gibt es also für ein Gebäude? Was muss der Spieler gebaut oder erforscht haben, um eine Einheit ausbilden zu dürfen? Wann darf der Spieler mit seinen Forschungen beginnen?

Für unser Browsergame haben wir, in der für diesen Zweck gewohnten Art der Abbildung, ein Bild erstellt, das alle Abhängigkeiten mit Pfeilen darstellt.

Die Art der Implementierung, wie wir sie nun für unser Browsergame durchgeführt haben, ist komplett hart mit switch, case und If-Schleifen umgesetzt worden. Auch die Eigenschaften bzw. Daten von Einheiten oder Gebäuden, wie Geschwindigkeiten, Stärken, Fähigkeiten oder Hitpoints sind momentan über switch und case in eigens dafür vorgesehenen Klassen festgehalten. Während des Projektes haben wir uns hier für die Möglichkeiten entschieden, die wir für am schnellsten und einfachsten realisierbar hielten, auch wenn sie nicht optimal erweiterbar, oder schnell und einfach zu bearbeiten sind.

Für die Daten- und Eigenschaftshaltung sehen wir verschiedene Möglichkeiten als eine Alternative an. Zum Beispiel ein XML-Propertiesfile, das über einen Parser ins Spiel integriert wird, oder die neuen Enums, die seit Java 1.5 verfügbar sind. Diese würden eine Zusammenfassung oder Konzentration der Daten bedeuten, da man alle Eigenschaften einer Einheit in einer Enum halten könnte und nicht über mehrere switch und case Konstrukte abfragen müssten.

## 5.4 EventSystem

Das Event System ist das Herzstück unseres Browsergames - alles funktioniert eventbasiert. Wenn man beispielsweise ein Gebäude ausbaut, jemanden angreift, eine Forschung tätigt oder Einheiten ausbildet, wird ein Event gestartet. Somit hat das Event System eine hohe Priorität in der Entwicklung des Spiels.

Zusammensetzung:
Das Event System besteht aus mehreren Teilprodukten. Es besteht aus einem EventManager, einem EventHandler, einem MainEvent und aus den spezifischen Events. Im Folgenden werden alle Teilprodukte erklärt.

Der EventManager beschäftigt sich, wie der Name schon sagt, mit dem Management der Events. Baut beispielsweise ein Spieler eines seiner Gebäude aus, legt der EventManager ein BuildEvent an, insofern bestimmte Kriterien erfüllt sind. In diesem Beispiel kann nur ein BuildEvent angelegt werden, wenn der Spieler genug Ressourcen hat, sich kein Gebäude im Ausbau befindet, die maximale Stufe des Gebäudes noch nicht erreicht ist und das Gebäude vom Technologie-Baum schon frei geschaltet ist, also gebaut werden darf.

Auch bietet der EventManager weitere Funktionen, zum Beispiel liefert er die zu einem Spieler gehörigen Events und prüft, welche Events fertig sind und reicht diese zur Behandlung an den EventHandler weiter.

Eventerzeugung:
Wenn vom EventManager ein Event erzeugt wird, läuft dieses über die MainEvent Klasse. Diese Klasse hält alle Events (mit den jeweiligen EventTypes und den Besitzern) und referenziert die jeweiligen typspezifischen Events. Je nach Konstruktor-Parameter wird ein anderer Event erzeugt, der Konstruktor wird also überladen. Etwas genauer: Im EventManager wird ein MainEvent erzeugt, dieses MainEvent erzeugt wiederum in seiner Klasse, abhängig von den Parametern, den zugehörigen Event. Auf das obige Beispiel bezogen wird hier also ein BuildEvent erzeugt. Um MainEvent und BuildEvent miteinander zu verbinden, wird zuerst das BuildEvent gespeichert und dessen eindeutige ID dem zugehörigen MainEvent zugeordnet. Danach kann das MainEvent in der Datenbank abgelegt werden.

Der EventHandler behandelt die fertigen Events, die ihm von der EventManager Klasse übergeben wurden. Im Beispiel eines Gebäudebaus wird die Gebäudestufe inkrementiert, oder bei einem Angriff wird das AttackScript aufgerufen, um den Angriff zu berechnen. Bei der Berechnung der Events gibt es verschiedene Dinge die beachtet werden müssen. So ist der Zeitpunkt der Berechnung sehr relevant bzw. die Reihenfolge, wie die Events berechnet werden absolut wichtig. Besonders bei Angriffen ist die Reihenfolge entscheidend, dazu muss, bevor ein Angriff auf eine Burg berechnet wird, geprüft werden, ob es für diese Burg noch Events gibt.

Ein Beispiel soll diesen Ablauf etwas verdeutlichen: Spieler A greift Spieler B an. Sobald seine Truppen angekommen sind, stößt Spieler A (da er sich als erstes ins Spiel eingeloggt hat) die Berechnung des AttackEvents an. Nun wird zuerst geprüft, ob es noch fertige Events der Burg von Spieler B gibt, da er z.B. zu diesem Zeitpunkt Einheiten oder Verteidigungsanlagen fertig gestellt haben könnte.

Uns ist es sehr wichtig, dass das Event System komfortabel erweiterbar ist, für den Fall, dass ein neuer Eventtyp implementiert werden soll. Das Event System soll ein eigenständiges System sein, das vom Prinzip ebenso für andere Browsergame-Arten funktionieren soll.

## 5.5 Das MessageSystem

Das MessageSystem dient den Usern und Administratoren, sich untereinander Nachrichten, so genannte IngameMessages, zu schicken. Desweiteren werden darüber die EventMessages erstellt, d.h. wenn ein User angegriffen wurde, wird ihm der Ausgang des Angriffes als EventMessage angezeigt.

Das MessageSystem befindet sich vollständig im Paket castarma.message und besteht aus den Klassen MessageManager, Message und EventMessage. Diese werden im Folgenden erläutert.

Wenn eine IngameMessage oder EventMessage beliebigen Typs erstellt wird, läuft das über den MessageManger. Dieser stellt dafür verschiedene Methoden bereit, für die IngameMessages sind das:

- newMessage(...): falls ein User einem Anderen schreibt
- newAdminMessage(...): falls ein Administrator Nachrichten an alle User schreibt
- newGuildMessage(...): falls ein Gildenmitglied Nachrichten an alle User der Gilde schreiben will

Jede dieser Methoden erstellt ein Message-Objekt - anhand der Übergabeparameter kann entschieden werden, um was für eine Message es sich handelt - und speichert diese. Für die EventMessages wird analog dazu für jedes Event eine eigenes EventMessage-Objekt erstellt, dies wird vom EventHandler aufgerufen, wenn er das Event berechnet hat. Dies sind die Methoden:

- newBuildEventMessage(...)
- newResearchEventMessage(...)
- newUnittrainEventMessage(...)
- newAttackEventMessage(...)
- newTradeEventMessage(...)
- newSpyEventMessage(...).

Die MessageManager-Klasse bietet ausserdem noch Funktionalitäten, die vom Frontend aus aufgerufen werden:

- getNormalMessagesAsList(): Rückgabe der Ingame-Messages für den User
- getNormalUnreadMessagesAsList(): ungelesene Nachrichten für den User
- getWrittenNormalMessagesAsList(): selbst geschriebene Nachrichten (für den Postausgang)
- getMessage(messageId): Ausgabe eine bestimmten Message
- getEventMessage(messageId): Ausgabe einer bestimmten EventMessage
- deleteMessage(messageId): Löschen einer bestimmten Message
- deleteEventMessage(messageId): Löschen einer bestimmten EventMessage
- getAdminMessagesAsList():Rückgabe der Ingame-Messages für den User, die vom Admin kommen
- getGuildMessagesAsList(): Rückgabe der Ingame-Messages für den User, die von einem Gilden-Member als Rundmail kommen
- getEventMessagesAsList(): Rückgabe der EventMessages für den User

Die Klasse Message dient der Speicherung der verschiedenen Ingame-Messages, je nach Konstruktor-Aufruf (durch die MessageManager-Klasse) mit verschiedenen Parametern wird eine andere Message erstellt, eine normale Ingame-Message, eine Admin-Message oder eine Gilden-Messages. Diese Klasse wird von einer Hibernate-Mappingdatei in die Datenbank gemappt.

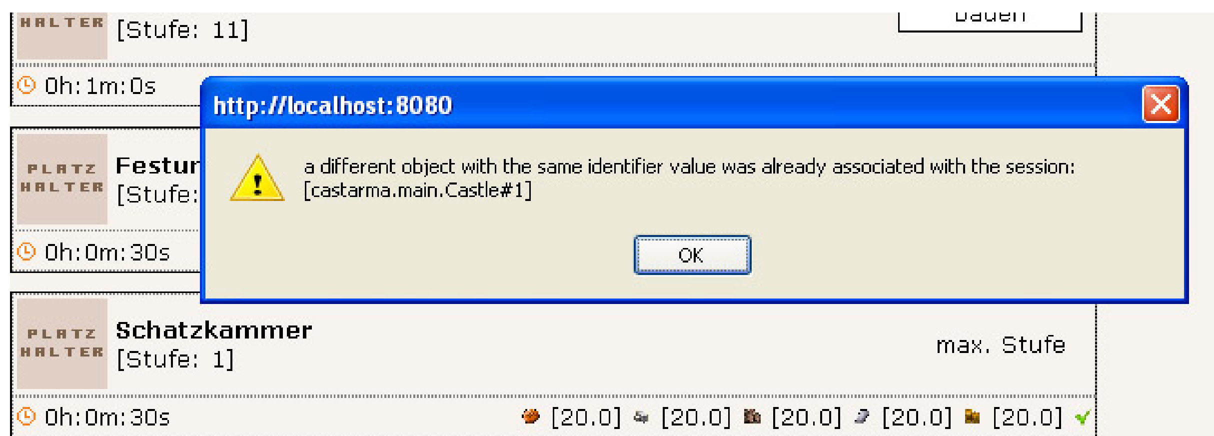
Ebenso wie die Message-Klasse speichert auch die EventMessage-Klasse die verschiedenen auftretenden EventMessages und wird ebenfalls von einer Hibernate-Mappingdatei in die Datenbank gemappt. Das Besondere an dieser Klasse ist, dass sie die EventMessages gleich als HTML-Code in einem String abspeichert. Dies hat den Grund, dass im Beispiel eines Angriffes die EventMessage sehr umfangreich wird - es müssen die Verluste des Angreifers und die des Verteidigers angezeigt werden. Aus Performancegründen ist es sicherlich sinnvoller, nur einen String zu speichern, als alle statistischen Daten, die in der EventMessage stehen werden, dies soll den (einzigen) Verstoß gegen die MVC-Architektur rechtfertigen.

# 6. Probleme

In diesem Kapitel werden die Probleme besprochen, die während des Projektes auftraten und uns längere Zeit beschäftigt haben.

Tomcat-Verzeichnis:
Trotz stundenlangen Versuchens schafften wir es nicht, unsere Bilder, die im Tomcat abgelegt wurden, im Browser anzeigen zu lassen. Problemförderer war in diesem Fall die schlechte Dokumentation des Tomcat Mappings, die in diesem Fall kaum hilfreich war. Abhilfe schaffte hier die Nachfrage bei unserem begleitenden Professor. Nun war es nichts Großes, die Mappings richtig anzupassen. Unsere Bilder sind nun unter „Tomcat\cas\gfx“ erreichbar, das Servlet auf „Tomcat\cas\ajaxserver“ erreichbar.

Hibernate-Sessionobjekte:
Während den Tests des Spiels trat im Frontend immer wieder derselbe Fehler auf. Hauptsächlich wenn der Button für ein neues Gebäude, oder für eine neue Forschung gedrückt wurde.



Wir interpretieren die Fehlermeldung so, dass ein Objekt doppelt im Speicher liegt und Hibernate so nicht genau wissen kann, welches der beiden identischen Objekte es nun zur Bearbeitung heranziehen soll. Allerdings war auch festzustellen, dass trotz der Fehlermeldung das Spiel immer korrekt weiterlief und alle Befehle richtig bearbeitet wurden, auch wenn bei jedem Klick auf einen Button die Fehlermeldung erneut ausgegeben wurde.

Die Ursache für diesen Fehler sehen wir in nicht geschlossenen Hibernate-Sessions. So befinden sich von dieser Session noch Objekte im Cache und bei einem erneuten Abruf desselben Objekts, befindet sich ein und dasselbe Objekt zweimal im Cache und der Fehler tritt auf. Da sich das gesamte Projekt-Team erst in Hibernate einarbeiten musste, haben sich kleine Fehler in die Behandlung mit den Hibernate-Sessions eingeschlichen. Da wir die meisten Fehler schon gefunden und beseitigt haben und deshalb die Anzahl auftretender Fehlermeldungen abgenommen hat, sehen wir dieses Problem als gelöst an.

Falls nach der Beseitigung der Programmierfehler des Teams die Fehlermeldung immer noch auftreten sollte, wird das Problem an Hibernate selbst liegen. In diesem Fall sehen wir auch das Problem im Cache von Hibernate, das dann entweder zu langsam im Löschen von Objekten ist, oder ein Objekt aus der Datenbank holt, obwohl es bereits vorhanden ist. Auch denkbar wäre, dass es am Transaktionsmanager liegt, der die Verbindungen von Hibernate zur Datenbank organisiert und ebenfalls Objekte zu lange hält. Eine Lösung in diesem Fall wäre, einem Player, der sich ins Spiel einloggt eine Hibernate-Session zuzuordnen und mit dieser Session alle Datenbank Aktionen auszuführen. Dies hätte allerdings weit reichende Änderungen zufolge, da der Player nicht in allen Bereichen unserer Klassen verfügbar ist, wo auch auf die Datenbank zugegriffen werden muss. Zusätzlich stellt sich hier die Frage, ob diese Lösung noch eine zufriedenstellende Performance ermöglicht. Im Allgemeinen ist ein Arbeiten mit Session-Pooling schneller und Ressourcen sparender, wobei in unserem Browsergame oft und schnell hintereinander auf die Datenbank zugegriffen wird und selten über einen langen Zeitraum hinweg.

# 7. Aktueller Stand und Ausblick

Nachdem wir nun den Aufbau und die Probleme des Browsergames betrachtet haben, ziehen wir in diesem Kapitel ein Fazit über den aktuellen Stand des Projektes und geben einen Ausblick auf die Zukunft des Spiels.

Unsere Projektziele, die wir uns zu Beginn gesetzt hatten, siehe Kapitel Planung, wurden schlussendlich erreicht, auch wenn in manchen Fällen noch die letzten Details fehlen. So ist das Kampfscript in seiner aktuellen Form noch nicht in der Lage, komplexe Angriffszenarien zu berechnen, wie dies in der Zukunft möglich sein soll. Das Messagesystem muss ebenfalls noch mal überarbeitet werden, da es mit dem Kampfscript nicht richtig abgestimmt ist und so inhaltlich falsche Nachrichten generiert werden, mehr dazu siehe Kapitel 5.5.

Auch die Abstimmung von Ressourcen und Kosten, das sogenannte Balancing kann erst in einem Beta-Test durchgeführt werden, da es sehr aufwendig ist herauszufinden, ob die Spieler im Laufe des Spiels zu viele Ressourcen erhalten und somit in der Lage sind, entgegen unserer Spielidee, zu viele Einheiten zu bauen bzw. ihre Burg zu schnell auszubauen.

Auf das Erstellen von eigenen Grafiken wurde aus Zeitmangel und mangelnden Fachkenntnissen im Bereich der Computergrafik vollständig verzichtet. Diese werden von einer externen Person erstellt. Auch das Design des Frontends ist in seiner aktuellen Form rein zweckmäßig und zum Test aller Funktionen vorhanden.

Als ein Hauptziel für die Zukunft, haben wir uns gesetzt, bis zum 1.1.2007 mit unserem Browsergame online zu gehen. Da 3 von 4 Teammitgliedern nun ins Praxissemester gehen und außerhalb der Arbeitszeiten vermutlich genügend Zeit haben werden, ist dieses Ziel als realistisch zu betrachten.

Zusätzlich gibt es noch verschiedene Dinge die in das Spiel integriert werden sollen:

- Spielern soll es möglich sein sich in Gilden zu verbünden.
- Im Laufe des Spiels soll es den Spielern möglich sein, andere Burgen einzunehmen oder Nebenburgen aufbauen zu können.
- Charakterentwicklung: Jeder Spieler erhält zum Spielbeginn einen Helden, den er trainieren, strategisch im Krieg einsetzen und mit dem er Aufgaben lösen kann.
- Landkarte: Momentan besitzen die Burgen zwar bereits schon eindeutige Koordinaten auf einer Karte, allerdings gibt es noch keine Möglichkeit die Karte bzw. Teile davon auszugeben. Es soll auch möglich sein, auf der Karte nicht nur die Burgen darzustellen die es bereits gibt, sondern auch Berge, Flüsse, Höhlen etc.
- Ressourcenhandel über einen zentralen Markt: Bis jetzt sind die Spieler nur in der Lage direkt Ressourcen zu verschicken. Es sollte auch möglich gemacht werden, auf einem Markt Ressourcen anzubieten, die dann ein anderer Spieler kaufen kann.
- Die Möglichkeiten, die AJAX bietet, sollen intensiver im Frontend umgesetzt werden (z.B. Drag&Drop).