Abschlussbericht
PG 561
„Process-Cloud for Business“

Teilnehmer:
Sascha Becker
Miguel Büscher
Ulrich Gabor
Zani Sarkisyan
Marcus Wübbeling

12. November 2012

Institution

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 5 für Programmiersysteme

Betreuer

Dipl.-Inform. Markus Doedt
Dipl.-Inform. Maik Merten

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Was ist eine Projektgruppe? 1
1.2. Thema dieser Projektgruppe 1
1.3. Aufbau dieses Dokuments 2

2. Organisation und Ablauf **3**
2.1. Ablauf dieser Projektgruppe 3
2.1.1. Seminarphase 3
2.1.2. Semester 1 3
2.1.3. Semester 2 3
2.2. Wöchentliche Treffen 4
2.3. Verwendete Fremdsoftware 4

3. Grundlagen (Seminarphase u. 1. Semester) **7**
3.1. Business Process Model and Notation 7
3.1.1. Einleitung 7
3.1.2. Notation 8
3.1.3. Weitere Anmerkungen 9
3.2. Maven 11
3.2.1. `pom.xml` 11
3.2.2. Verzeichnisstruktur 13
3.2.3. Lebenszyklus 13
3.2.4. Projekte erstellen und packen 14
3.3. Hibernate / JPA 15
3.3.1. Hibernate 15
3.3.2. JPA 15
3.3.3. JPA und GAE 15
3.3.4. JPA 2.0 Features 16
3.3.5. Gemeinsamkeiten von Hibernate und JPA 17
3.3.6. Hibernate Features 18
3.4. jBPM 19
3.4.1. Was ist jBPM? 19

3.4.2. Features [Oum11] 19
3.4.3. Architektur 20
3.5. Activiti 24
3.5.1. Was ist Activiti 24
3.5.2. Aufbau von Activiti 25
3.5.3. Die Activiti Engine 25
3.5.4. Beispiel 27
3.6. Google App Engine 30
3.6.1. Restriktionen: Der Datastore / Memcache 31
3.6.2. Vorteile: Das Eclipse Toolkit 33
3.6.3. Vorteile: Kosten und Management 33
3.7. Windows Azure 34
3.7.1. Was ist Windows Azure? 34
3.7.2. Kosten 35
3.7.3. Windows Azure (Betriebssystem) 35
3.7.4. SQL Azure 37
3.7.5. Programmiermodell 38
3.7.6. Interoperabilität 39
3.8. Heroku 41
3.8.1. Beschreibung von Heroku 41
3.8.2. Architektur 42
3.8.3. Preismodell 43
3.8.4. Heroku für Java 43
3.8.5. Vorgehen zum Veröffentlichen von Java-Applikationen . 43
3.8.6. Datenbank 44
3.9. VMForce 44
3.10. JBoss Cloud – StormGrind 45

4. Grundlegende Erkenntnisse (Semester 1) 47
4.1. Google App Engine 47
4.1.1. Activiti 47
4.1.2. jBPM5 48
4.1.3. jBPM4 49
4.2. Windows Azure 51
4.2.1. Activiti 51
4.2.2. jBPM 54
4.3. Heroku 54
4.3.1. Allgemeines 54
4.3.2. jBPM 55
4.3.3. Activiti 55

4.4. Entscheidungen fürs 2. Semester 59

5. Proof-of-concept – Monopoly in der Cloud (Semester 2) 61
5.1. Anforderungen 61
5.2. Projektplan 61
5.3. Clientseite (Graphical User Interface) 63
5.3.1. Aufbau 63
5.3.2. Basis-HTML 63
5.3.3. Coffeescript 64
5.3.4. LESS 68
5.4. Serverseite 68
5.4.1. Aufbau 68
5.4.2. Projektkonfiguration 71
5.4.3. Datenmodell 74
5.4.4. Spiellogik 75
5.4.5. Delegates 79
5.4.6. Web-Services 80
5.4.7. Nachrichten 81
5.4.8. KI 83
5.5. Testing 85
5.5.1. Prozesse 85
5.5.2. Services und Nachrichten 88
5.5.3. GUI 90
5.6. Resümee 96

6. Zusammenfassung 99

A. Anhang 103

Abbildungsverzeichnis 115

Listings 117

Literaturverzeichnis 119

1. Einleitung

1.1. Was ist eine Projektgruppe?

Als zentraler Bestandteil des Informatik-Studiums sollen Studierende in Projektgruppen (PGs) in kleinen Untergruppen praktische Problemstellungen selbstständig bearbeiten. Dabei soll der Einsatz von Hilfsmitteln und das generelle Vorgehen selbst geplant, implementiert und in einem Zwischen- und einem Endbericht dokumentiert werden. Jedes Semester werden zu diesem Zweck thematisch unterschiedliche Projektgruppen angeboten, welche jeweils eine Dauer von zwei Semestern umfassen. Jede Projektgruppe wird dabei von zwei Projektgruppenbetreuern unterstützt.

1.2. Thema dieser Projektgruppe

Die Aufgabenstellung der PG 561, deren Endbericht gerade vorliegt, lautet „aktuelle Technologien im Cloud-Computing-Umfeld als mögliche Grundlage für ein 'rechenzentrumfreies' Geschäftsprozessmanagement insbesondere hinsichtlich Skarlierbarkeit und Steuerbarkeit zu evaluieren“ [LS5]. Das Erreichen folgender Minimalziele ist dabei unbedingt erforderlich:

- Einarbeitung in grundlegende Theorien und Formalismen von serviceorientierten Architekturen und zur Modellierung von Geschäftsprozessen.

- Eingehende Recherche im Bereich Cloud-Computing und Einarbeitung in aktuelle Technologien aus diesem Bereich.

- Erfolgreiche Ausarbeitung eines Geschäftsprozesses unter Zuhilfenahme der erwähnten Technologien, mit anschließend erfolgtem Deployment auf einer kommerziellen Cloud-Plattform.

- Systematischer Vergleich der implementierten Lösungen.

1.3. Aufbau dieses Dokuments

Im Folgenden Kapitel 2 werden zuerst die Rahmenbedingungen der Projektgruppe abgesteckt. Unter anderem der generelle Ablauf, allgemeine Entscheidungen und die zur Organisation genutzte Software werden vorgestellt. Danach folgt in Kapitel 3, Seite 7, eine Einführung in die Grundlagen aller Themen, welche für die Projektgruppe relevant waren, also z. B. die untersuchten Plattformen und Technologien. Dies war der Einstieg ins Thema noch vor dem ersten Semester. Im ersten Semester sollten dann Grundlagen erarbeitet werden und mögliche Lösungen verglichen werden. Die Ergebnisse finden sich in Kapitel 4, Seite 47. In Kapitel 5, Seite 61, wird dann die beispielhafte Applikation vorgestellt, die im Rahmen der Projektgruppe entwickelt werden sollte.

2. Organisation und Ablauf

2.1. Ablauf dieser Projektgruppe

2.1.1. Seminarphase

Die Projektgruppe startete mit einer Seminarphase. Jeder der Teilnehmer erhielt ein Thema zugewiesen, welches er selbstständig erarbeiten sollte. Im Rahmen einer Blockveranstaltung erfolgte dann ein Kennenlernen und Vorträge zu den jeweiligen Themen. Folgend musste jeder Teilnehmer seine Erkenntnisse noch in einem Aufsatz zusammenfassen, sodass dies als Basiswissen während der Zeit der PG zur Verfügung steht.

Die Ergebnisse der Seminarphase finden sich größtenteils in Kapitel 3, Seite 7.

2.1.2. Semester 1

Im ersten Semester haben sich die Teilnehmer mit den Thematiken vertraut gemacht. Es wurden unterschiedliche Prozess-Engines hinsichtlich ihrer Cloud-Kompatibilität untersucht. Diesbezüglich wurde sich zuerst vor allem auf JBPM und GAE (GoogleAppEngine) konzentriert, später verschob sich der Fokus mehr Richtung Activiti und Azure bzw. Heroku. Die Gründe werden in Kapitel 4, Seite 47, näher erläutert.

2.1.3. Semester 2

Im Rahmen des zweiten Semesters sollten die erworbenen Kenntnisse an einer Software umgesetzt werden, um exemplarisch die Verwendung von Business-Prozessen in der Cloud zu zeigen. Dazu fungierten die PG-Betreuer als „Kunde“ und gaben ein prozessbasiertes Spiel in Auftrag. Es sollte das Spiel Monopoly basierend auf Geschäftsprozessen und als Webanwendung umgesetzt werden.

Die Details der Implementierung werden in Kapitel 5, Seite 61, vorgestellt.

2.2. Wöchentliche Treffen

Die Projektgruppenteilnehmer trafen sich einmal wöchentlich real, zusammen mit den beiden Projektgruppenbetreuern, um sich über den Fortschritt zu informieren und die Aufgaben für die jeweils nächste Woche festzulegen. Von diesen Treffen wurde jeweils ein Ergebnisprotokoll angefertigt, um nachhalten zu können, was besprochen und entschieden wurde. Der Protokollführer wechselte dabei von Woche zu Woche reihum (der Sitzreihenfolge nach).

Darüber hinaus wurde entschieden, dass es einen dedizierten Moderator geben soll, welcher die Treffen führt und dafür sorgt, dass sowohl die Rahmenbedingungen eingehalten werden, aber auch, dass jeder weiß, wo er steht und wie er weiter machen soll. Diesbezüglich wurde sich darauf geeinigt, dass jeder einmal für den Zeitraum von 5 aufeinanderfolgenden Wochen moderiert. Die Reihenfolge wurde hierbei nicht näher festgelegt.

2.3. Verwendete Fremdsoftware

Zur Dokumentverwaltung wurde das System BSCW[1] ausgewählt. Dort liegen vor allem die Ergebnisprotokolle der wöchentlichen Treffen und die Ausarbeitungen und Präsentationen der Seminarphase. Zur Verwaltung von Informationen und als Art GroupWare wurde Origo[2] verwendet. Dort wurden im Wiki u. a. Informationen zur Einrichtung von Entwicklungsumgebungen vor allem in Bezug auf Cloud-Lösungen gesammelt.

Das System Origo wurde zum Mai 2012 abgeschaltet. Ein Ersatz wurde nicht benötigt und auch die im Vorfeld dort gesammelten Informationen wurden im zweiten Semester nicht gebraucht, sodass der Einsatz von Origo ersatzlos gestrichen wurde.

Für die Entwicklungsumgebung nutzen wir vor allem Eclipse[3] in Kombination mit dem Maven-Plugin[4]. Als Repository für Quellcode kamen vor allem Git-Repositories zum Einsatz. (Beide Technologien werden im Laufe des Endberichts noch genauer beschrieben.) Einerseits wurden diese bei Github[5] gehostet, auf der anderen Seite wurde im Verlauf des ersten Semesters verstärkt mit Heroku[6] gearbeitet, welches wiederum vollständig auf eine Kombination aus Maven und

[1] https://bscw.uamr.de/
[2] http://www.origo.ethz.ch/, PG: http://pg-pcb.origo.ethz.ch/
[3] http://www.eclipse.org/
[4] http://maven.apache.org/eclipse-plugin.html
[5] https://github.com/
[6] http://www.heroku.com/

Git setzt (zumindest für Java-Quellcode), sodass das dort abgelegte Repository als zentrales genutzt wurde.

3. Grundlagen (Seminarphase u. 1. Semester)

Im aktuellen Kapitel werden die noch vor dem ersten Semester erarbeiteten Erkenntnisse vorgestellt. Sie dienten als allgemeine Grundlage und sollten jedem den Einstieg ins Thema ermöglichen.

Untersucht wurden dabei die folgenden Themen, welche der Reihe nach vorgestellt werden:

- Business Process Model and Notation
- Maven
- Hibernate / JPA
- jBPM
- Activiti
- Google App Engine
- Windows Azure
- Heroku
- VMForce
- JBoss Cloud – StormGrind

Die Abschnitte 3.2, 3.3 und 3.10 stammen von Kada Benadjemia, einem ehemaligen Mitglied der Projektgruppe.

3.1. Business Process Model and Notation

3.1.1. Einleitung

Die explizite Fassung von Geschäftsprozessen stellt in Unternehmen praktisch eine Notwendigkeit dar. Sie erfolgt meist mit Hilfe grafischer Hilfsmittel oder

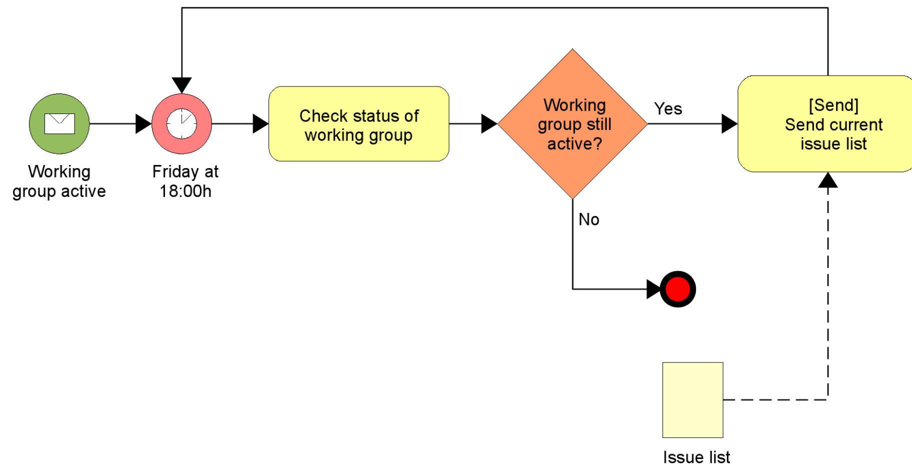

Abbildung 3.1.: Beispiel einer Arbeitsgruppe. Quelle: http://en.wikipedia.org/wiki/Business_Process_Model_and_Notation.

in Form von (Excel-)Tabellen. BPMN („*Business Process Model and Notation*“ seit Version 2.0, vorher „*Business Process Modeling Notation*“) ist eine Notation für Geschäftsprozesse, welche dem grafischen Ansatz folgt. Abläufe werden als Graph-Struktur abgebildet. Dafür stehen innerhalb des BPMN-Standards verschiedene Diagramm-Typen und -Elemente zur Verfügung. Seit der Version 2.0 existiert außerdem ein Metamodell, um standardkonforme Diagramme automatisiert verwerten zu können.

Die Einsatzgebiete von BPMN sind sehr vielfältig ([All09], Seite 13). Es können sowohl fachliche Diagramme, will heißen vor allem Modelle von „groben“ Zusammenhängen, aber auch deutlich detailliertere und dann auch ausführbare Diagramme erstellt werden.

Getragen wird dieser Standard dabei auch von Größen der Industrie, wie IBM [Whi], aber auch Borland, Fujitsu, Oracle, SAP [OMG].

3.1.2. Notation

Wir wollen hier die Notation von BPMN nur kurz erläutern und beschränken uns deshalb auf einen groben Umriss der Notation. Für stichhaltige Erklärungen der Notation sei z. B. auf [All09] verwiesen.

Die Abbildung 3.1 enthält ein einfaches Beispiel eines Prozesses. Es lassen

sich grob drei Arten von Elementen unterscheiden,

Kreise repräsentieren Ereignisse,

abgerundete Rechtecke sind Vorgänge und

Rauten stellen Verzweigungen im Prozess dar.

In der vorliegenden Abbildung 3.1 beginnt der Prozess also z. B. mit einer Nachricht darüber, dass die Arbeitsgruppe aktiv ist. Dann wird auf Freitag 18 Uhr gewartet. Danach folgt ein Prozess, das Überprüfen des Status der Gruppe. Die Raute danach wertet die im Prozess ermittelten Daten aus und verzweigt den Prozess entsprechend. An den Kanten stehen die Bedingungen. Der rechte Prozess verwendet noch Daten, in diesem Fall die „*Issue list*“. Der runde Kreis in der Mitte ist ein End-Ereignis.

Die vorgestellten Basiselemente sind im BPMN-Standard noch deutlich detaillierter definiert. Unter anderem gibt es verschiedene Start- und Endereignis-Typen, aber auch verschiedene Rautentypen.

Darüber hinaus lassen sich auch Kollaborationen zwischen verschiedenen Partnern darstellen. Partner kommunizieren über Nachrichten miteinander (ein anderer Partner hätte also z. B. im vorangegangenen Beispiel den Prozess mit einer Nachricht starten können). Die Abbildung 3.2 enthält ein einfaches Beispiel, welches die Kommunikation zwischen einer Bäckerei und einem Kunden darstellt. Sie kommunizieren via Nachricht miteinander. Die Bäckerei wiederum lässt sich in den Verkäufer und den Bäcker unterteilen. Innerhalb der Bäckerei können die Elemente direkt miteinander verbunden werden. Zwischen den Partnern ist diese direkte Verbindung nicht möglich. Jeder Prozess muss eine Nachricht abschicken, diese muss auf der anderen Seite empfangen werden.

3.1.3. Weitere Anmerkungen

Sowohl syntaktisch als auch semantisch lässt sich derselbe Prozess auf verschiedene Weisen repräsentieren. Auch die Art der Komplexität ist variabel. So bietet es sich an für komplexe Prozesse mehrere Diagramme zu erstellen, die dann auf unterschiedlichen Abstraktionsleveln angesiedelt sind. Ausführbare Diagramme müssen hinreichend detailliert sein, damit diese auch wirklich ausführbar sind. Für einen groben Überblick enthält diese Darstellung jedoch meist viel zu viele Informationen. Ein deutlich abstrakteres zweites Diagramm kann Abhilfe schaffen.

Von Version 1 zu Version 2.0 wurde vieles im BPMN-Standard überarbeitet. Abgesehen von dem bereits genannten Metamodell, welches Modelle automatisiert verwertbar macht, wurde auch der Im- und Export standardisiert. Darüber

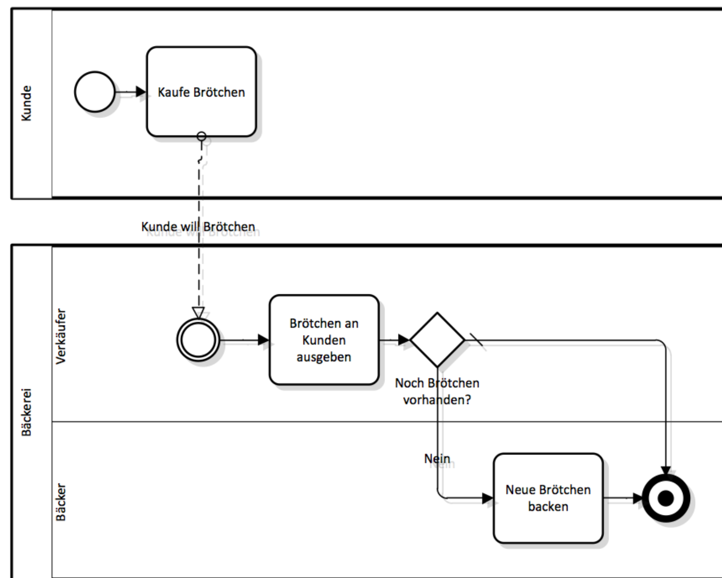

Abbildung 3.2.: Ein Beispiel eines Nachrichtenaustauschs mit Unterteilung eines Partners. Abgeänderte Version eines Beispiels aus [All09].

hinaus kamen neue Diagrammtypen hinzu, zum Beispiel Choreographiediagramme, welche dediziert den Nachrichtenaustausch von Partnern zeigen, oder Konversationsdiagramme, welche konkret zeigen welche Beteiligten an einem Prozess direkt miteinander interagieren. Sogenannte „*Conformance Level*“ sorgen außerdem für eine Vergleichbarkeit der auf dem Markt verfügbaren Tools.

3.2. Maven

Maven ist ein automatisiertes Build-Management-System, also ein System zum automatisierten Erzeugen eines Anwendungsprogramms, unter anderem auf der Spezifikation von Abhängigkeiten. Maven bietet Funktionen für das Projektaufbaumanagement und das automatische Auflösen von Projektabhängigkeiten. Maven ist ein etabliertes System und tauchte im Laufe der Recherchephase immer wieder auf. Viele der Projekte werden mit Maven verwaltet, sodass wir um die Verwendung von Maven nicht herum kamen. Auch im zweiten Semester kam es zum Einsatz, da Heroku intern auf eine Kombination aus Git und Maven aufbaut [hera].

Konvention vor Konfiguration Das „Konvention vor Konfiguration“-Konzept ermöglicht das Reduzieren von notwendigen Konfigurationseinstellungen über das Einhalten von Konventionen. Einige Angaben müssen dennoch über Konfigurationsdateien geschehen.

In Maven sind Projekte in Artefakte aufgeteilt. Teile des Projekts, so wie das Projekt als Ganzes, werden als Artefakte deklariert. Ein Artefakt wird in der `pom.xml` angegeben.

3.2.1. `pom.xml`

Die zentrale Konfigurationsdatei ist die `pom.xml`. In ihr werden Projektaufbau und -Abhängigkeiten deklariert. Die `pom.xml` wird verwendet um Abhängigkeiten, Projektumgebung und Projektbeziehungen zu speichern. Hier beispielhaft der mögliche Grundaufbau einer `pom.xml`:

```
<project>
 <modelVersion>1.x.x</modelVersion>
 <groupId>de.maven.beispiele</groupId>
 <artifactId>beispiel-artefakt</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>
 ...
```

```
</project>
```

Hier wird das Projekt `beispiel-artefakt` angelegt. Das Projekt soll der Projektgruppe/Projektdomäne `de.maven.beispiel` angehören. Die Version ist `1.0-SNAPSHOT`. Die Angaben `groupId`, `artifactId`, `version` dienen unter anderem zu einer eindeutigen Lokalisierung im Maven-Repository. In diesem Fall:

```
$M2_REPO/de/maven/beispiele/beispiel-artefakt/1.0-SNAPSHOT
```

Das Beispielprojekt soll als Jar-Datei(`<packaging>`) kompiliert werden.

Verwaltung von Abhängigkeiten Maven ermöglicht das automatische Auflösen von Abhängigkeiten. Abhängigkeiten lassen sich wie folgt als Artefakte in der `pom.xml` auf folgende Weise deklarieren:

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-core</artifactId>
 <version>1.x.x</version>
 <scope>compile</scope>
</dependency>
```

Die Abhängigkeit ist die Core-Komponente der Spring-API. Die Angaben `groupId`, `artifactId`, `version` sind Pflicht. Wie oben beschrieben dienen sie der Lokalisierung im Maven-Repository. Die Abhängigkeit soll für den `compile`-Vorgang verwendet werden (`<scope>`).

Projektbeziehungen Damit Projektbeziehungen von Maven korrekt verwalten werden, müssen die Module in der `pom.xml` in der Sektion `<modules>` angeben werden. Die Deklaration sieht so aus:

```
<modules>
 <module>native-layer</module>
 <module>utils</module>
 ...
<modules>
```

Es gibt die zwei Unterprojekte native-layer und utils, die über das Schlüsselwort `<module>` aggregiert werden.

3.2.2. Verzeichnisstruktur

Dem Konzept des „Konvention vor Konfiguration“ folgt die Standard-Verzeichnisstruktur von Maven. Die `pom.xml` kann durch das Einhalten der Standard-Verzeichnisstruktur stark vereinfacht werden.

Hier die wichtigsten Verzeichnisse nach http://maven.apache.org/:

- src: alle Eingabedateien
  - src/main: Eingabedateien für die Erstellung des eigentlichen Produkts
    - src/main/java: Java-Quelltext
    - src/main/resources: Andere Dateien, die für die Übersetzung oder zur Laufzeit benötigt werden, beispielsweise Java-Properties-Dateien
  - src/test: Eingabedateien, die für automatisierte Testläufe benötigt werden
    - src/test/java: JUnit-Testfälle für automatisierte Tests
- target: alle erzeugten Dateien
  - target/classes: Kompilierte Java-Klassen

3.2.3. Lebenszyklus

Die Entwickler von Maven gehen von einem wiederkehrenden Lebenszyklus in Projekten aus und definieren einen Standardzyklus, der aus einer festen Abfolgen von Phasen besteht. Die Liste der Phasen lässt sich hier wie folgt ablesen:

archetype Erstellt ein Template für ein Archetypprojekt.

validate Prüft, ob die Projektstruktur gültig ist.

compile Kompiliert den Quellcode im Verzeichnis src/main/java sofern nichts anderes spezifiziert worden ist.

test Hier wird der kompilierte Code Tests unterzogen.

package Das Kompilat und alle essentiellen Dateien werden in einer Jar-Datei verpackt.

integration-test Das Paket wird in ein anderes Verzeichnis, einen entfernten Rechner oder Server geladen und geprüft.

verify Prüft, ob die vorgegebenen Strukturen übereinstimmen.

install Installiert das Paket in das lokale Maven-Repository. Das Projekt kann jetzt als Abhängigkeit oder Modul verwendet werden.

deploy Installiert das Paket in ein entferntes Maven-Repository (optional das globale), sodass andere Entwickler das Projekt als Abhängigkeit laden können.

Nicht alle Phasen müssen ausgeführt werden, dennoch gilt: Wird eine Phase ausgeführt, so werden alle ihr vorhergehenden Phasen ausgeführt.

3.2.4. Projekte erstellen und packen

Um ein Projekt zu erstellen, sollte man zunächst die Verzeichnisstruktur aus Abschnitt 3.2.2 einhalten, sodass zur Konfiguration nur noch eine `pom.xml` wie oben beschrieben erstellt werden muss. Ein Maven Webprojekt-Archetyp kann durch den Aufruf

```
mvn archetype:create -DgroupId=de.maven.beispiele \
 -DartifactId=webapp-beispiel \
 -DarchetypeArtifactId=maven-archetype-webapp
```

geschehen. Dadurch besteht bereits die Verzeichnisstruktur und eine Beispiel-`pom.xml`, die aufgegriffen und angepasst werden kann.

Compile

Mit dem Aufruf

```
mvn compile
```

wird der Quellcode im Verzeichnis `src/main/java` in den Zielordner `target/classes` kompiliert.

Package

Der Aufruf

```
mvn package
```

erzeugt ein JAR-Archiv im Ordner `target`, welches nun mit dem Befehl `install` ins lokale oder globale Repository installiert werden kann.

Install

Der Befehl

```
mvn install
```

sorgt dafür, dass das Paket in das lokale Maven-Repository installiert wird, um das Projekt als Abhängigkeit oder Modul wiederverwenden zu können.

3.3. Hibernate / JPA

Der Versuch jBPM auf GAE ans Laufen zu bringen, zwang uns zur Auseinandersetzung mit der Konvertierung von Hibernate zu JPA, da GAE nur JPA unterstützt. Es ging im Wesentlichen darum die Unterschiede zwischen den beiden Technologien zu erörtern, um einen Überblick über den Aufwand gewinnen zu können. Speziell dafür hat sich die Projektgruppe in beiden Technologien eingearbeitet.

Grundsätzlich leisten beide Technologien ähnliche Arbeit und dienen zur automatischen Persistierung und Abbildung objektorientierter Daten auf eine relationale Datenbank. Hibernate besitzt alle Funktionalitäten die JPA verlangt und somit adressiert JPA eine Untermenge an Funktionalitäten die Hibernate zur Verfügung stellt.

3.3.1. Hibernate

Hibernate ist ein objektorientiertes Persistenz- und ORM-Framework und stellt eine API zur Verfügung, welche POJOs (Plain Old Java Object(s)) auf relationale Datenbanken abgebildet und so persistiert. Hibernate setzt primär auf Konfigurationsdateien, um das Abbilden auf Datenbanken automatisieren zu können.

3.3.2. JPA

JPA hingegen ist in erster Linie eine Spezifikation zur Abbildung von POJOs auf eine relationale Datenbank. JPA kann Hibernate im Hintergrund verwenden, sodass sich Vorteile aus beiden Technologien vermischen lassen.

Der Vorteil JPAs ist der „Konvention vor Konfiguration“-Ansatz und die damit verbundene Verwendung von Entity-Beans. Entity-Beans sind annotierte Java-Objekte, die einer strikten Konvention folgen, wodurch in vielen Fällen Konfigurationsdateien entfallen. JPA stellt Entitäts-, Assoziations-, Veerbungs- und Persistence-Listener-Annotationen zur Verfügung.

3.3.3. JPA und GAE

Bei der Ersetzung Hibernates durch JPA kam es gleichzeitig darauf an, welche Funktionen JPAs auf der GAE zur Verfügung gestellt wurden. GAE stellte nur eine eigene Implementierung JPAs zur Verfügung. Sie entsprach in weiten Teilen der JPA 1 Implementierung, weswegen bei der Betrachtung von JPAs weiter zwischen JPA 2.0 und JPA 1 Features unterschieden werden musste.

3.3.4. JPA 2.0 Features

Im Folgenden eine Auflistung von JPA 2.0 spezifischer Funktionalität, auf die in der GAE verzichtet werden muss.

Element Collections sind dafür zuständig, dass erweiterte Datensätze wie Maps und Ähnliches auf die relationale Datenbank abgebildet werden können. Annotationen wie @ElementCollection, @MapKeyColomn, @CollectionTable, @AttributeOverride sind JPA2.0 spezifisch.

Pessimistisches Sperren (Locks) Für den Fall, dass verschiedene Benutzer an den selben Daten arbeiten, lassen sich Daten mit einem Lese- oder Schreib-Lock versehen. JPA 2 unterstützt optimistische und pessimistische Locks [jpa]. Mit dem Befehl

```
em.lock(employee, LockModeType.PESSIMISTIC_WRITE);
```

wird der Lock-Modus gesetzt. Es existieren folgende Lock-Modi [jpa]:

- PESSIMISTIC_READ
- PESSIMISTIC_WRITE
- PESSIMISTIC_FORCE_INCREMENT

Des Weiteren lässt sich ein Lock-Timeout setzen, sodass eine gesperrte Datenbankoperation erst einmal einen Timeout abwartet, bevor eine LockTimoutException geschmissen wird [jpa]. Um den Lock-Timeout für alle Persistierungen zu setzen, lässt sich die `persistence.xml` unter der Sektion `<properties>` wie folgt anpassen:

```
<property name="javax.persistence.lock.timeout" value="2000"/>
```

oder per Befehl:

```
properties.put("javax.persistence.lock.timeout", 4000);
```

Um den Lock-Timeout für spezielle Entitäten zu setzen, verwendet man den Befehl

```
em.setProperty("javax.persistence.lock.timeout", 10000);
```

Die Befehle für den Entity-Manager `lock`, `find` und `refresh` besitzen ebenfalls die Möglichkeit Locks zu setzen [jpa].

Hibernate	JPA
SessionFactory.openSession()	EntityManagerFactory.createEntityManager()
Session.getNamedQuery()	EntityManagerFactory.createNameQuery()
Query.list()	Query.getResultList()
Query.setString()	Query.setParameter()
Query.uniqueResult()	Query.getSingleResult()
Session.saveOrUpdate()	EntityManager.persist()
Query.setParameterList()	Query.setParameter()
Session.createCriteria()	EntityManager.createQuery()
Session.createSQLQuery()	Session.createNativeQuery()
Session.updaten)	EntityManager.merge()

Tabelle 3.1.: Hibernate-Befehle, die in JPA-Befehle übertragen werden müssen.

orphanRemoval -Attribut (Entfernen von verwaisten Datenbank Einträgen in Parent-Child-Relationen) wurde zur @OneToMany-Annotationsfunktionalität hinzugefügt. Dieses Attribut ist JPA2.0-Funktionalität.

Datenabfragen nach konkreter Entity ist JPA2.0-Funktionalität. Ein Beispiel wäre der Aufruf:

```
select e from Ist_Entwickler e where type(e) in (Ist_Entwickler)
```

In JPA 2.0 ist es möglich mit `type(...)` die konkrete Entität (den Typ) eines Elements abzufragen.

Criteria-API , eine Schnittstelle die das typisierte Abfragen von Datensätzen zulässt, hätte ausgelassen werden müssen. Einstellungen zum Caching in JQL-Anfragen wurden in JPA2.0 ebenfalls angepasst.

Weitere Annotationen: @Access-Annotationen (bestimmt, ob Zugriff auf Entities/Attribute Field-based oder Property-based ist), @OrderColumn (bestimmt die Sortierung von Objekte im Arbeitsspeicher).

3.3.5. Gemeinsamkeiten von Hibernate und JPA

Hibernate sowie JPA besitzen eine Art Persistence Factory, die erst einmal initialisiert werden muss. Beide Technologien besitzen ziemlich ähnliche Funktionen zum Aktualisieren der Persistenz sowie zum Abfragen, die bis auf die Funktionsnamen weitgehend identisch sind und funktionieren. Anfragen in der

nativen Query-Sprache Hibernates (HQL) sind identisch mit der JPAs (JQL). Eine Ersetzung wäre relativ gradlinig vonstatten gegangen, wie sich an der Tabelle 3.1 ablesen lässt. Die Konfigurationsdatei könnte wegfallen und die zu persistierenden POJOs könnten direkt annotiert werden.

3.3.6. Hibernate Features

Hibernate besitzt eine Menge von Funktionen, die es nicht in die JPA-Spezifikation geschafft haben. Hier ein kleiner Überblick über die Funktionalitäten, die man in Bezug auf GAE hätte entfernen müssen.

Generische Id-Generatoren: Spezifizieren von generischen Id-Generatoren, die für eine eigenes oder spezielles Id-Vergabeverfahren zuständig sind. Klassen, die vom Interface IdentifierGenerator abgeleitet wurden, sowie die @GenericGenerators-Annotation hätten entfernt werden müssen.

Attributvalidatoren: Validatoren sorgen dafür, dass Datensätze bestimmten Regeln entsprechen. Validatoren werden mit Beans Annotiert. Hier die Annotationen, nach denen gesucht hätte werden müssen: @Length, @Max, @Min, @NotNull, @NotEmpty, @Past, @Future, @Pattern, @Range, @Size, @Email, @CreditCardNumber, @Valid

Sorting oder Comparer: Sorting und Comparer sind Annotationen, die zur Bestimmung einer gewisse Sortierung in der Datenbank zuständig sind. Aufzufinden wäre die Annotation @Sort.

Filters: Filters filtern Datensätze, sodass bei Anfragen nur gewisse Daten aus der Datenbank geladen und verarbeitet werden. Filter werden in der Session-Klasse, die aus der SessionFactory Hibernates übergeben wird, angegeben und somit muss das Schlüsselwort

```
enableFilter(...);
```

und die Annotation @Filter aufgefunden werden.

Fetch-Strategien: Fetch-Strategien spezifizieren, auf welche Art und wie viele Datensätze geladen beziehungsweise vorgeladen werden. Das Schlüsselwort FetchMode und die Annotations @org.hibernate.annotations.Fetch und @BatchSize.

@org.hibernate: @org.hibernate sind generell Annotations, nach denen man bei der Ersetzung Ausschau halten kann.

Die Ersetzung Hibernates durch JPA fand letztendlich nicht statt, da wir mit der Zeit erkennen mussten, dass das Ersetzen oder Deployen von Drools (siehe Abschnitt 3.4.2) der weitaus größere Faktor im Einsatz von JBPMN 5 auf der GAE gewesen wäre.

3.4. jBPM

Der folgende Abschnitt basiert zu großen Teilen auf der Seminararbeit von Moussa Oumarou [Oum11], an welchem nur geringfügig Änderungen vorgenommen wurden.

3.4.1. Was ist jBPM?

jBPM ist eine Open-Source-Software für Business Process Management und stellt eine Plattform zum Entwickeln und Entwerfen von Business-Prozessen zur Verfügung. Es wird ein Tool für die grafische Modellierung von Geschäftsprozessen bereitgestellt, sodass Business-Prozesse nicht nur von IT-Spezialisten, sondern auch von Business-Analysten entworfen, entwickelt und verbessert werden können. jBPM bietet eine zentralisierte Plattform an, die die Definition, die Ausführung und die Administration von Arbeitsablauf-Prozessen vereinigt, um die Interaktion zwischen Benutzern und Systemen zu verwalten. Mithilfe der API können Software-Entwickler mit Business-Analysten in der gemeinsamen Plattform kommunizieren. Bis jBPM 4 wurde die Prozessbeschreibungssprache jPDL verwendet. Mit der Version 5 wurde jBPM um BPMN2.0 erweitert. BPMN ist in der Version 2.0 zusammen mit anderen Verbesserungen vor allem um die Ausführbarkeit erweitert worden.

3.4.2. Features [Oum11]

- Es wird eine flexible und skalierbare Prozessmaschine (Process Engine) zur Verfügung gestellt.
- Eine JPA-basierte Persistenz-Verwaltung wird zwar in den Features angegeben, der Bericht wird aber im Laufe der Zeit darstellen, dass dieses Feature nicht vollständig umgesetzt wurde.
- jBPM kann mit anderen Technologien integriert werden, insbesondere sind dies
  - Seam, ein Web-Framework von JBOSS.
  - Spring, ebenfalls ein Web-Framework.

  - Drools, eine Businesslogik Integrations-Plattform für Regeln, Arbeitsabläufe und Ereignisverarbeitung[jbp]. Diese Technologie ist der Projektgruppe beim Deployen auf der GAE zu einer größeren Hürde geworden, was später im Bericht dargelegt wird.

- Die Architektur ist erweiterbar und individuell anpassbar.
- jBPM bietet ein optionales Prozess-Repository, um Prozesse auszuführen.

3.4.3. Architektur

Version 4

Das Herzstück jBPM4s ist die Process Virtual Machine (PVM). Die Process Virtual Machine folgt dem Ansatz eine einheitliche Art und Weise zu bieten, um viele verschiedene Prozesssprachen in eine Konzeptsprache abbilden zu können. In der PVM wurden Grundkonzepte eines ausführbaren Graphen abstrahiert. Andere Business Process Sprachen können darauf abgebildet werden. Der Vorteil ist schlicht und einfach, dass man eine einheitliche Engine besitzt um viele verschiedene Business Process „Dialekte“ umsetzen zu können. Die PVM selbst lässt sich über eine API verwenden.

Definition [Oum11]. Um mehrere Prozesssprachen zu unterstützen, basiert jBPM 4 auf der PVM [TB]. Die PVM ist ein Framework, das dazu dient ausführbare Graphen, die Prozesse abbilden, zu spezifizieren, zu verwalten und auszuführen. Die PVM bestimmt die geeignete Sprache für eine bestimmte Situation, da es mehrere Prozesssprachen geben kann, die zusammen existieren. Eine Prozessdefinition stellt einen ausführbaren Fluss dar und hat eine Struktur, die als Diagramm grafisch dargestellt werden kann. Die PVM trennt die Struktur einer Prozessdefinition von dem Aktivitätsverhalten. Weiterhin übernimmt sie die Ausführung eines Prozesses von einer Aktivität zu einer anderen und übergibt das Verhalten der Aktivitäten an bestimmten Java-Klassen. Es gibt eine API (ActivityBehaviour), die als Schnittstelle zwischen der PVM und dem Aktivitätsverhalten-Code dient. Prozesssprachen wie jPDL werden einfach durch eine Reihe von ActivityBehaviour-Implementierungen und einen Parser bereitgestellt [jbp04].

Vorteile der PVM Die Vorteile der PVM sind, dass man den Laufzeit-Zustand der Prozessausführung in der Datenbank abspeichern kann. Darüber hinaus wird ein Ausführungskontext für Prozesslaufzeitdaten zur Verfügung gestellt. Die Prozesse können dementsprechend auch parallel ausgeführt werden. Die

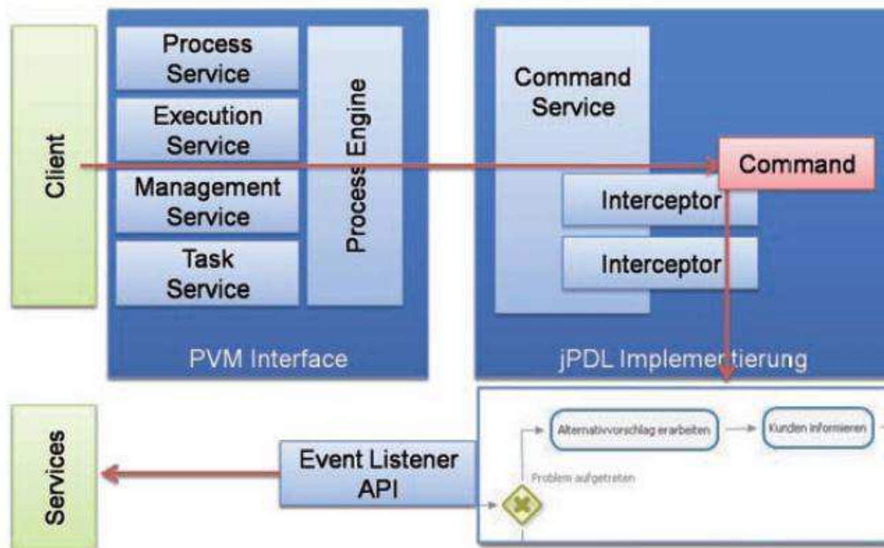

Abbildung 3.3.: jBPM 4-Architektur. Quelle: [BF09]

PVM bietet ebenfalls Unterstützung für Prozesse, die menschliche Eingaben erfordern.

ProcessEngine [Oum11]. Die Abbildung 3.3 stellt eine Übersicht der Architektur von jBPM bis Version 4.x dar. Die PVM bietet verschiedene Schnittstellen zur Verwaltung von Prozessen an. Alle Interaktionen mit jBPM finden mithilfe dieser Schnittstellen statt. Alle Dienst-Interfaces kann man von der ProcessEngine bekommen, die wiederum von einer Configuration zurückgegeben wird. Die ProcessEngine stellt die eigentliche Umgebung dar. Hinter den Schnittstellen steckt eine CommandService, die die Ausführung aller Kommandos übernimmt. An der CommandService können sehr einfach Interceptors zur Transaktionsverwaltung oder Logging angebracht werden [BF09].

Version 5 [Oum11]

Die Abbildung 3.4 gibt einen Überblick über die Komponenten, aus denen jBPMN5 besteht. Die Core Process Engine ist der Kern dieser Architektur. Sie ist unerlässlich, um einen Prozess auszuführen. Alle anderen Komponenten sind optional. Die Applikationsdienste sprechen die Core Engine an, wenn sie sie brauchen. Ein anderer optionaler Dienst ist das History Log. Er speichert alle Informationen über den aktuellen und den vorherigen Zustand aller Pro-

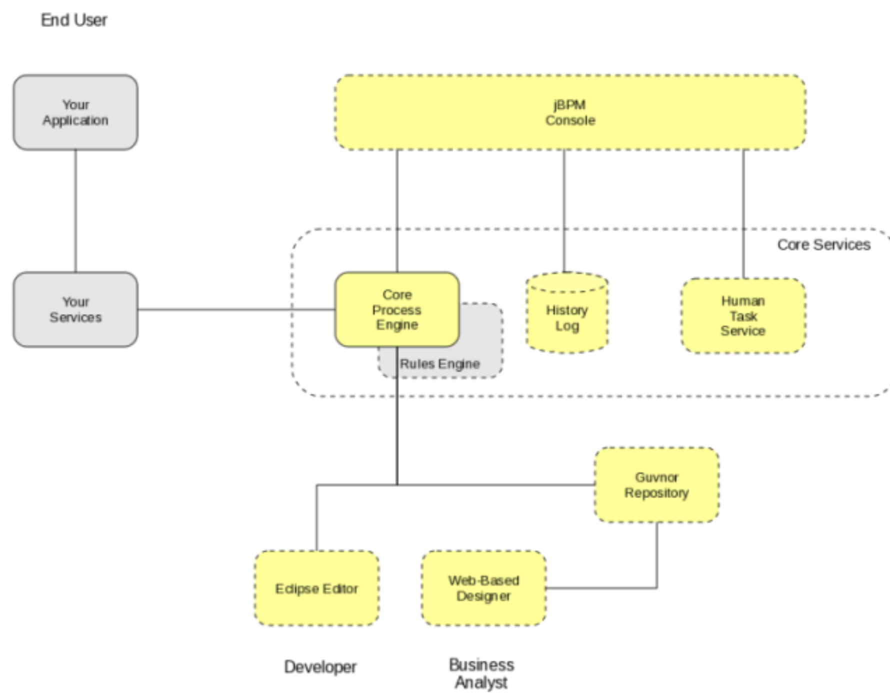

Abbildung 3.4.: jBPM 5 - Architektur. Quelle: [jbp]

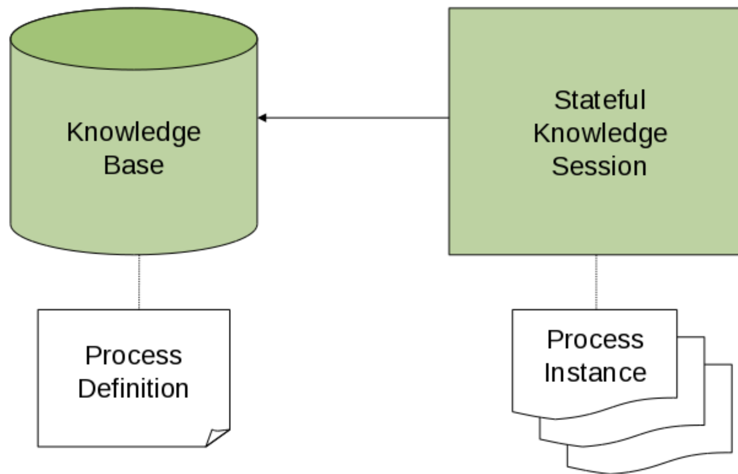

Abbildung 3.5.: jBPM 5 - Core Engine. Quelle: [jbp]

zessinstanzen. Die Human Task Service ist auch optional. Sie verwaltet die Benutzerinteraktionen, wenn Personen an dem Prozess teilnehmen.

Core Engine [Oum11]. Die Core-Engine ist das Herz von jBPM. Sie ist eine leichtgewichtige Workflow-Maschine, die die Geschäftsprozesse ausführt. Sie kann als Teil einer Anwendung eingebettet oder als ein Dienst (zum Beispiel in der Cloud) aufgespielt werden. Die Abbildung 3.5 zeigt eine grafische Darstellung der Core-Engine-Architektur in jBPM 5.x. Um mit der ProcessEngine zu kommunizieren, legt man zuerst eine Session an. Diese Session ist dann zuständig für die Kommunikation mit der ProcessEngine. Eine Session muss eine Wissensbasis-Referenz haben. In dieser Wissensbasis werden die Referenzen aller Prozessdefinitionen gespeichert. Um eine Session anzulegen, braucht man erst eine Wissensbasis, in die alle Prozessdefinitionen geladen werden. Nachdem eine Session angelegt wurde, kann man anfangen, Prozesse auszuführen. Mehrere Session-Objekte können die gleiche Wissensbasis benutzen. Die Wissensbasis wird im Allgemeinen einmal beim Starten der Anwendung angelegt. Man kann einer Wissensbasis Prozesse zur Laufzeit hinzufügen oder entfernen.

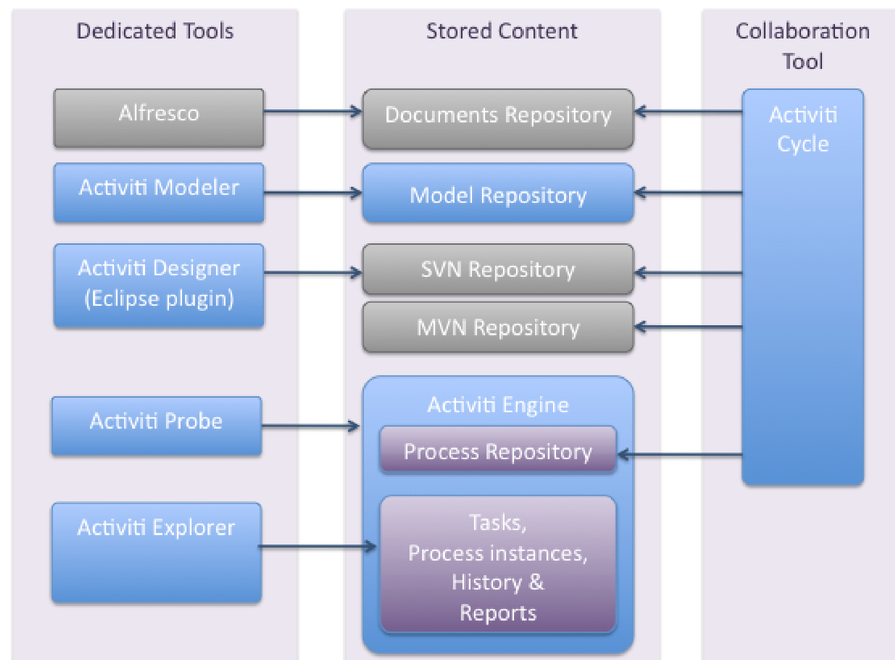

Abbildung 3.6.: Die Komponenten von Activiti. Quelle: http://www.activiti.org

3.5. Activiti

3.5.1. Was ist Activiti

Bei Activiti handelt es sich um eine quelloffene Software für das Business Process Management. Activiti läuft unter der Apache Lizenz und wurde das erste Mal im Dezember 2010 von Alfresco und Spring Source veröffentlicht. Alfresco warb dafür eigens den Chefentwickler des jBPM-Projektes von Red Hat ab. Da Activiti als Weiterentwicklung von der damals in Version 4.0 erhältlichen jBPM Software gesehen wird, erschien es in der ersten Version als 5.0, um dies zu verdeutlichen. Aktuell ist Activiti in der Version 5.9 erhältlich. Activiti setzt auf den oben beschriebenen BPMN 2.0 Standard, welcher die direkte Ausführung von Geschäftsprozessen ermöglicht.

3.5.2. Aufbau von Activiti

Die Abbildung 3.6 gibt einen Überblick über die einzelnen Activiti Komponenten und deren Struktur. Zunächst gibt es den Activiti Modeler und den Activiti Designer, zwei Werkzeuge zur grafischen Erstellung von Geschäftsprozessen. Der Activiti Designer ist umfangreicher und ermöglicht z. B. das Hinzufügen von technischen Details wie Java Service Tasks und Execution Listeners. Activiti Probe/Explorer sind in den neuesten Versionen zum Activiti Explorer vereint worden und stellen eine Webanwendung dar, welche den Zugriff auf die Activiti Engine während der Laufzeit und für alle Benutzer ermöglicht. Somit lassen sich eigene Instanzen von Prozessen erstellen, bearbeiten und zuweisen, Fehlerquellen untersuchen oder auch ein umfangreiches Task Management mit grafischer Benutzeroberfläche durchführen. Die benötigten Daten für die Prozesse, Benutzer etc. werden alle in den entsprechenden Repositories gespeichert. Activiti Cycle ist eine Webanwendung welche einen BPM Round Trip ermöglichen soll. Für unsere Zwecke ist Cycle aber nicht weiter relevant.

3.5.3. Die Activiti Engine

Der zentrale und wichtigste Teil von Activiti ist die Activiti Engine, auf welche wir uns in unserer Arbeit fokussieren werden. Bei ihr handelt es sich um eine Java Process Engine, die BPMN 2.0 Prozesse unterstützt. Sie läuft im Hintergrund, lädt die Daten aus den Repositories und ermöglicht den Zugriff auf diese via eigener API. Ausführen lässt sie sich in jeder Java-Umgebung und ermöglicht es Benutzerupdates und Prozessupdates in einem Durchgang zu bearbeiten, sprich alle Aktionen, die im Explorer via GUI durchgeführt werden können, können auch mit Hilfe der API direkt mit der Engine kommuniziert werden. Die Engine ermöglicht es weiterhin Prozessausführungen isoliert zu testen, Timer zu Transaktionen hinzuzufügen, unterstützt verborgene Event Listener um Prozesse mit eigenem Java-Code zu verknüpfen und diese Details vom geschäftlichen Teil des Prozesses zu trennen. Ebenfalls lassen sich mittels BPMN Shortcuts Bereiche, die im BPMN 2.0 Standard relativ umfangreich ausfallen, durch eigene Activiti-Attribute ersetzen bzw. verkürzen.

Wie in Abbildung 3.7 zu sehen, wird die Activiti Engine mit Hilfe einer ProcessConfiguration erstellt, welche in der XML-Datei Namens `activiti.cfg.xml` beschrieben ist. Sie kann nach Erstellung direkt auf die abgebildeten Services zugreifen, welche die Workflow/BPM-Methoden enthalten.

Mit dem Quellcode aus Listing 3.1 kann eine Engine mit allen Services erstellt werden. In der ersten Zeile wird nun eine neue Instanz der Engine erstellt, in den darauffolgenden Zeilen Instanzen der einzelnen Services, auf die man dann

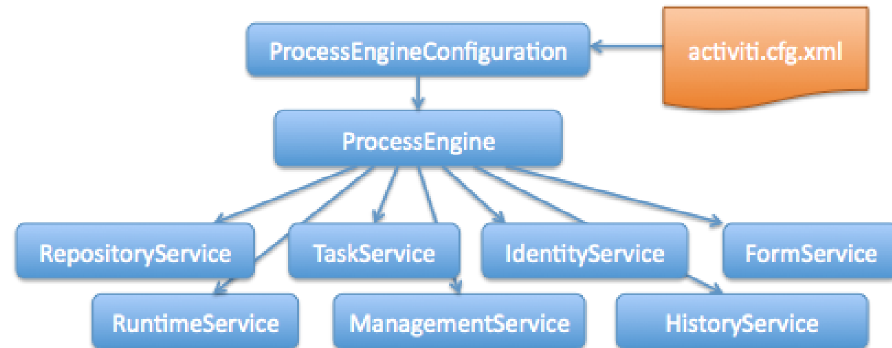

Abbildung 3.7.: Activiti Engine. Quelle: http://www.activiti.org

```
ProcessEngine processEngine = ProcessEngines.
    getDefaultProcessEngine();

RuntimeService runtimeService = processEngine.getRuntimeService();
RepositoryService repositoryService = processEngine.
    getRepositoryService();
TaskService taskService = processEngine.getTaskService();
ManagementService managementService = processEngine.
    getManagementService();
IdentityService identityService = processEngine.getIdentityService
    ();
HistoryService historyService = processEngine.getHistoryService();
FormService formService = processEngine.getFormService();
```

Listing 3.1: Erstellen einer Engine mit allen Services.

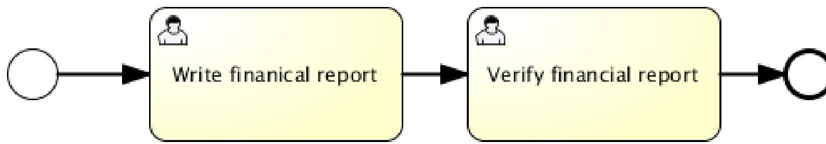

Abbildung 3.8.: Beispielprozess Financial Report. Quelle: http://www.activiti.org

von seinen Prozessen aus zugreifen kann. Dafür speichert man den Business Process nach BPMN 2.0 Standard in einer Datei mit Endung bmpn20.xml oder lässt diese Automatisch vom Modeler oder Designer erzeugen, nachdem man seinen Prozess grafisch erstellt hat.

3.5.4. Beispiel

Zur Erläuterung wie ein BPMN-Prozess in Activiti in der entsprechenden `bpmn20.xml`-Datei repräsentiert wird und anschließend über eine Java-Klasse gesteuert wird, nehmen wir hier das Beispiel eines Financial Reports (siehe Abbildung 3.8), der von einem Anwender geschrieben werden muss, um anschließend vom Chef verifiziert werden zu können.

Die Definition des Activiti-Prozesses als XML-Datei findet sich in Listing A.3 auf Seite 109 im Anhang.

Der Prozess erhält eine eindeutige ID, über die er später abgerufen werden kann. Danach werden die 4 Events im Prozess definiert, in diesem Fall der StartEvent und EndEvent sowie unsere zwei Usertasks writeReportTask und verifyReportTask. Die Usertasks erhalten eine Beschreibung (`<documentation>`) und eine Einschränkung für die Ausführbarkeit. Hier dürfen nur User der Gruppe accountancy bzw. management den Task durchführen.

Der Workflow, also der Ablauf des Prozesses, wird durch die `<sequenceFlow>` Abschnitte definiert, welche jeweils eine Quelle und ein Ziel haben. Abschließend folgt im TAG `<bpmndi:BPMNDiagram>` noch eine Beschreibung zur optischen Darstellung des Prozesses. Diese lässt sich automatisch generieren.

Den so erstellten Prozess kann man nun entweder mit dem Activiti Explorer deployen, um ihn den Nutzern der grafischen Umgebung zur Verfügung zu stellen, oder man spricht den Prozess direkt über die API der Engine an. Mit letzterer Methode wollen wir hier einen kompletten Durchlauf des Prozesses simulieren.

```
1 ProcessEngine processEngine = ProcessEngineConfiguration
2   .createStandaloneProcessEngineConfiguration()
3   .buildProcessEngine();
```

Listing 3.2: Prozessengine erstellen.

Wie weiter oben beschrieben, wird hier also zunächst einmal eine neue Prozessengine mit der gegebenen Konfiguration erstellt. Mit dieser Instanz der Engine kann man nun arbeiten.

```
1 RepositoryService repositoryService = processEngine.
     getRepositoryService();
2 RuntimeService runtimeService = processEngine.getRuntimeService();
```

Listing 3.3: Repositories laden.

Man lädt die benötigten Repositories. Da es sich hier um einen einfachen Usertask handelt, lädt man lediglich das `Service-Repository` und das `Runtime-Repository` zum ausführen des Prozesses.

```
1 repositoryService.createDeployment()
2   .addClasspathResource("FinancialReportProcess.bpmn20.xml")
3   .deploy();
```

Listing 3.4: Deployen der Ressourcen.

Anschließend fügt man nun alle benötigten Ressourcen hinzu, in unserem Fall den als `*.bpmn20.xml` gespeicherten Prozess, und deployt diese ins entsprechende Repository.

```
1 String procId = runtimeService.startProcessInstanceByKey("
     financialReport").getId();
```

Listing 3.5: Neue Instanz vom Prozess.

Mit Hilfe des `runtimeService` startet man eine neue Instanz des soeben ausgeführten Prozesses, indem man ihn über seine eindeutige ID aufruft.

```
1 TaskService taskService = processEngine.getTaskService();
2 List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup
     ("accountancy").list();
3 for (Task task : tasks) {
4   System.out.println("Following task is available for accountancy 
       group: " + task.getName());
5
6   // Weise ihn dem User fozzie der Accountancy-Gruppe zu
7   taskService.claim(task.getId(), "fozzie");
8 }
```

Listing 3.6: User-Zuweisung für Task 1.

Nun kann man sich über eine `TaskQuery` eine Liste der Usertasks, die für eine bestimmte Benutzergruppe bestimmt sind, ausgeben lassen.

```
1 tasks = taskService.createTaskQuery().taskAssignee("fozzie").list()
    ;
2 for (Task task : tasks) {
3   System.out.println("Task for fozzie: " + task.getName());
4   taskService.complete(task.getId());
5 }
```

Listing 3.7: User zugewiesene Tasks ausgeben.

Der User kann im Activiti Explorer tasks "claimen", sprich signalisieren, dass er einen Task bearbeiten will. Dies wird hier durch eine Zuweisung des `taskAssignee` simuliert. Der Task wird also nun dem Benutzer fozzie zugewiesen.

```
1 tasks = taskService.createTaskQuery().taskCandidateGroup("
    management").list();
2 for (Task task : tasks) {
3   System.out.println("Following task is available for accountancy
      group: " + task.getName());
4   taskService.claim(task.getId(), "kermit");
5 }
```

Listing 3.8: User-Zuweisung für Task 2.

Das Gleiche führt man nun für den zweiten Usertask durch, welcher für die Benutzergruppe `management` bestimmt ist. Daher wird hier nun ein User aus dieser Gruppe zugeordnet, in unserem Fall kermit.

```
1 for (Task task : tasks) {
2   taskService.complete(task.getId());
3 }
```

Listing 3.9: Tasks als komplett bearbeitet setzen.

Zum Schluss wird für jeden Tasks die `complete()-Funktion` aufgerufen, welche eine vollständige Bearbeitung der Tasks simuliert. Da unser Prozess nur aus diesen beiden Usertasks besteht, wird er nach dem erfolgreichen abarbeiten beider Tasks beendet.

Somit haben wir einen einfachen Prozess komplett über die Engine API angesteuert.

Der vollständige Quellcode des hier erarbeiteten Beispiels ist in Listing A.4 auf Seite 110 im Anhang zu finden.

3.6. Google App Engine

Die Google App Engine (GAE) ist eine Cloud Computing Plattform, die 2008 von Google als „Platform as a Service“ der Öffentlichkeit zur Verfügung gestellt wurde. Die Plattform soll es Benutzern ermöglichen, ihre Webapplikationen innerhalb der Google-Infrastuktur auszuführen und zu hosten. Diese Applikationen sind dabei einfach zu erstellen, pflegen und skalieren, wann immer mehr Datenverkehr und Datenspeicher benötigt wird. Mit der Benutzung der Google App Engine muss man sich nicht selbst um die Wartung der Server kümmern und somit auch keine Administratoren oder andere Angestellte für diesen Zweck beschäftigen, wie man sie bei eigener Infrastuktur benötigen würde.

Die App Engine bietet drei unterschiedliche „Features“ des Cloud Computings für den Endbenutzer:

- Eine PaaS, welche Entwickler und Organisationen zum Entwickeln von öffentlichen oder eigenen Webanwendungen benutzen können.
- Anwendungen, die für die Google App Engine entwickelt wurden, und als SaaS durch den Endbenutzer über den Webbrowser ausgeführt werden.
- Die Möglichkeit Third Party Webservices von anderen Plattformen über das Web zu integrieren oder zu benutzen.

Seit der Veröffentlichung im Jahre 2008 wird die Programmiersprache Python unterstützt. Knapp ein Jahr später wurde für die Entwickler der Java-Sprachsupport zur Verfügung gestellt. Seit Ende 2009 wird auch Googles eigene Programmiersprache Go unterstützt.

Beim Erstellen einer Webanwendung mit Java muss man bei der Google App Engine beachten, dass diese mit Restriktionen kommt und nicht alle Java EE Funktionen unterstützt werden. Entfernte Methodenaufrufe mittels RMI und das *Java Naming and Directory Interface* (JNDI), wie auch *Enterprise Java Beans* (EJB), welche über das JNDI nachgeschlagen werden, sind nur einige dieser Restriktionen. Auch Datenbankverbindungen über JDBC stellen ein Problem dar und sind nicht möglich. Dank der Highlevel Schnittstellen JPA oder JDO lassen sich Daten modellieren und in Google’s BigTable Datenbank persistieren. Als Schnittstelle zum Webbrowser können die JSP-, JSF- und Servlet-Technologien eingesetzt werden, die in einem modifizierten Jetty Servlet-Container zum Einsatz kommen. Jetty[1] ist ein Opensource Webserver Service, welcher einen HTTP-Server, HTTP-Client und einen Servlet-Container bereitstellt und von

[1] http://www.mortbay.org/jetty/

Google aufgrund der Größe, der Flexibilität, dem Speicherbedarf und der Erweiterbarkeit gewählt wurde.

Eine weitere Restriktion an die sich die PG gewöhnen musste war, dass auf das lokale Datensystem nicht geschrieben werden kann. Eine Alternative dazu ist der Datastore, um Daten persistent abzuspeichern, oder der Memcache, um Daten flüchtig abzulegen. Auf diese Restriktion wird im folgenden Unterabschnitt näher eingegangen.

Die vermutlich entscheidendste Restriktion der Google App Engine – neben dem Verzicht auf ein relationales Datenbanksystem – ist, dass nicht alle JRE Java-Klassen unterstützt werden. So können im Moment nur ca. 40% der JRE Klassen verwendet werden. Neben den meisten Klassen aus dem java.awt Package, zum Erstellen von User Inferfaces, fehlen auch die Klassen aus dem rmi Package, um entfernte Methodenaufrufe durchzuführen.

Google hat eine Whitelist[2] für die App Engine erstellt, in der man alle Klassen findet, die aktuell von der App Engine erlaubt werden. Besonders die Persistenzrestriktion stellte die PG vor Probleme, da die Google App Engine kein Hibernate unterstützt, sondern nur JPA. Auf dieses Problem wird im vorliegenden Bericht genauer im Kapitel 3.6 eingegangen, wo es um die Google App Engine im Zusammenhang mit JBPM geht.

3.6.1. Restriktionen: Der Datastore / Memcache

Die App Engine setzt nicht, wie vielleicht gewünscht und erhofft, auf SQL zur Speicherung von Daten aller Anwendungen, sondern auf BigTable. BigTable ist dabei eine Datenbank, die von Google selbst für schemalose, strukturierte Daten geschrieben wurde – schemalos, da BigTable nicht zu den relationalen Datenbanken gehört und somit eher vergleichbar zu einer Hashmap ist, die verteilt, sortiert und geschachtelt ist. Allen Daten von Anwendungen werden in einer BigTable-Tabelle gespeichert, die in sogenannte Tablets partitioniert und auf die Server verteilt wird. Ein Server ist dabei für ca. 100 Tablets von einer Größe von ca. 100 - 200 MB zuständig. Bei steigender Last können zusätzliche Server eingesetzt werden, weshalb man von einer horizontalen Skalierung bei BigTable spricht. Wie in Abbildung 3.9 zu sehen, kommuniziert eine Google App Anwendung über sprachspezifische Schnittstellen mit BigTable. Im Falle von Java wären dies die standardisierten Schnittstellen JDO und JPA, bei Python setzt der Datastore als Schnittstelle auf BigTable.

Im Gegensatz zum Datastore ist der Memcache ein temporärer Speicher, der von jeder Instanz einer Anwendung geteilt wird (Abbildung 3.10). Anhand des

[2] http://code.google.com/intl/de-DE/appengine/docs/java/jrewhitelist.html

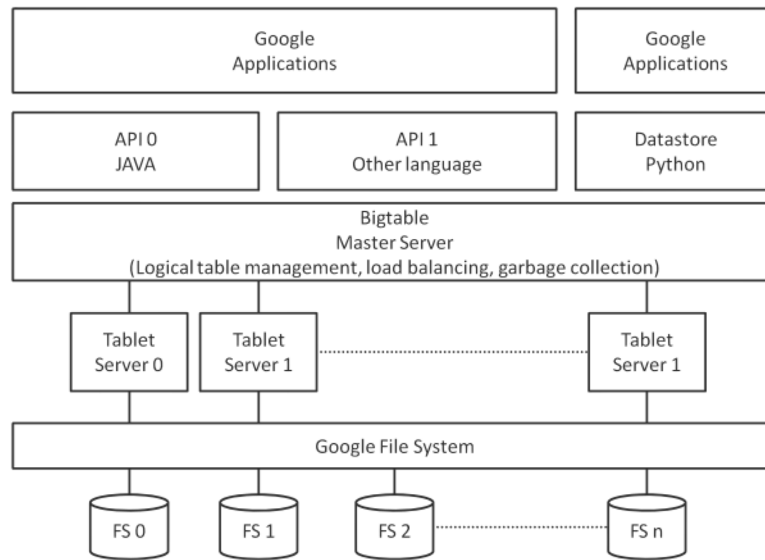

Abbildung 3.9.: Die Datastore Architektur. [Ciu09]

Namens ist bereits erkennbar, dass dieser Speicher sich im Arbeitsspeicher des Servers befindet. Die Zugriffszeiten sind somit deutlich schneller, da nichts von der Festplatte gelesen werden muss. Ein weiterer Unterschied zum Datastore liegt darin, dass jeder Eintrag wie bei einer HashMap mit einem eindeutigen Schlüssel abgelegt wird und nur auf 1MB beschränkt ist. Es ist möglich eine Verfallszeit in Sekunden anzugeben, um den Eintrag aus dem Memcache zu entfernen, jedoch kann man damit rechnen, dass der Eintrag vor Ablauf der Frist aufgrund von Ressourcenmangel gelöscht wird. Daher eignet sich der Memcache hauptsächlich nur für Daten, die man auf andere Weise wieder bekommen kann, wie z.B durch das Ablegen im langsameren Datastore.

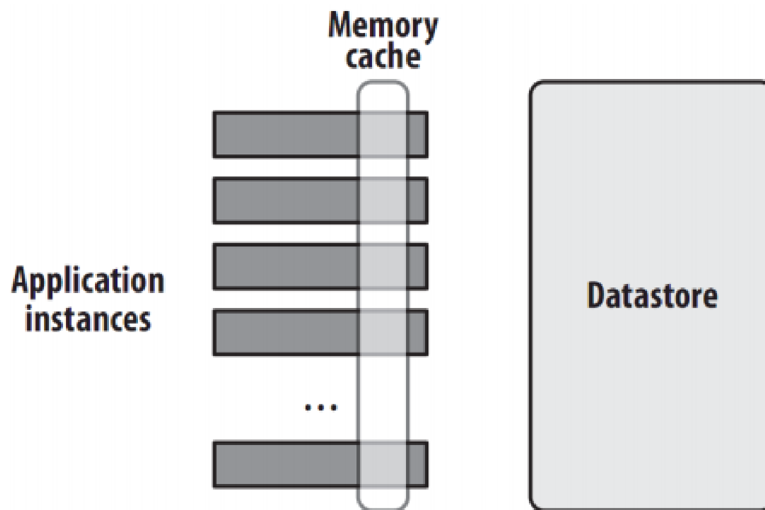

Abbildung 3.10.: Der Memcache. [Sev09]

3.6.2. Vorteile: Das Eclipse Toolkit

Google gibt den Entwicklern, die mit der App Engine arbeiten wollen, unterschiedliche Werkzeuge mit, um die Arbeit mit der App Engine zu erleichtern. Darunter befindet sich das Eclipse Toolkit.

Das Eclipse Toolkit[3] ist ein Eclipse Plugin, womit sich über einen Wizard Webapplikations-Projekte erstellen lassen. Das Plugin erstellt dabei eine Beispielanwendung mit allen nötigen Konfigurationsdateien, um die Anwendung auch direkt auf die App Engine deployen zu können.

3.6.3. Vorteile: Kosten und Management

Wichtig für die PG war, dass der Clouddienst mit dem man arbeitet, natürlich möglichst kostenlos ist. Die Google App Engine ist dabei zu einem bestimmten Grad kostenlos. So stellt Google den Entwicklern eine sogenannte *Free Quota*, die man ausschöpfen kann. Erst für ressourcenhungrige Applikationen kann die App Engine kostenpflichtig werden.

Das aktuelle Preismodell der AppEngine sieht wie folgt aus [Goo]:

- 0,10 US-Dollar pro CPU-Kern und Stunde.
- 0,10 US-Dollar pro GByte eingehenden Traffic.

[3] http://code.google.com/intl/de-DE/eclipse/

- 0,12 US-Dollar pro GByte ausgehenden Traffic.
- 0,15 US-Dollar pro GByte Speicherplatz.
- 0,0001 US-Dollar für jeden E-Mail-Empfänger.

Das Freikontingent sieht dabei folgendermaßen aus:

- 6,5 CPU-Stunden und einem Datenvolumen (eingehend und ausgehend) von einem GByte pro Tag.
- Die Seiten können unbegrenzt aufgerufen werden und hängen u. a. von den noch verfügbaren CPU-Stunden ab.
- Jeder Benutzer darf 10 Anwendungen erstellen.
- Mit dem Mail API dürfen höchstens 100 Mails pro Tag verschickt werden. (Früher noch 2000).

Der Ressourcenverbrauch lässt sich benutzerdefiniert limitieren und ist flexibel erweiterbar, sodass Entwickler ihre maximalen Ausgaben sicher planen können. Dabei können Nutzer tägliche Grenzen für CPU, Bandbreite, Speicherplatz und E-Mails festlegen.

Für die PG bot sich dieses Preismodell an, da man ohne Zeitlimit, anders als z. B. bei Windows Azure, problemlos mit der Google App Engine arbeiten konnte.

3.7. Windows Azure

3.7.1. Was ist Windows Azure?

Windows Azure ist die Cloud Computing Plattform von Microsoft. Angekündigt und zu Testzwecken in Betrieb genommen wurde die Plattform im Oktober 2008, seit dem 1. Februar 2010 steht sie offiziell und zum produktiven Betrieb zur Verfügung. Windows Azure lässt sich als Platform as a Service (PaaS) einordnen, einzelne Teile können auch als Infrastructure as a Service (IaaS) klassifiziert werden.

Mit Windows Azure lassen sich Anwendungen und Daten bereitstellen, wobei Skalierbarkeit, Verfügbarkeit, Wartbarkeit, Ausfallsicherheit und Sicherheit dieser Anwendungen durch die Plattform gewährleistet wird. Von der zu Grunde liegenden Infrastruktur (Hardware, Betriebssystem, Load Balancer, Firewall, etc.) wird abstrahiert, sodass der Administrationsaufwand für den Kunden gering ist. Der Kunde kann seine Aufgaben über ein Management-Portal oder

mit der zugehörigen API durchführen und muss z. B. für die Skalierung der Anwendung lediglich die Anzahl der gehosteten Instanzen seiner Anwendung erhöhen.

Es können große Mengen Daten sicher und von überall verfügbar gespeichert werden, wobei sämtliche Daten in mehrfacher Ausführung gespeichert werden, um Datenverlust zu vermeiden. Zur Verbesserung der Performanz können Daten in mehreren Rechenzentren repliziert werden.

Für eine leichtere Entwicklung stellt die Azure Plattform verschiedene Services zur Verfügung, welche Aufgaben wie Kommunikation zwischen Komponenten, Authentifizierung sowie Autorisierung oder das Caching von Daten übernehmen.

3.7.2. Kosten

Das Kostenmodell von Windows Azure ist verbrauchsorientiert, d. h. es wird nur für in Anspruch genommene Dienste bezahlt. Die Abrechnung findet monatlich statt und es werden neben den normalen Preisen auch spezielle Abonnements für längere Laufzeiten angeboten. Einige Beispiele für Preise im August 2011 [Micd] sind:

- Computing: $0.24 pro Medium Instanz pro Stunde
- Storage: $0.15 pro GB pro Monat, $0.01 pro 10000 Transaktionen
- SQL Azure: $9.99 pro Datenbank (max. 1 GB) pro Monat
- Access Control: $1.99 per 100000 Transaktionen
- Service Bus: $9.95 für ein Pack aus 5 Verbindungen
- Caching: $45.00 für 128 MB Cache
- Data Transfer: $0.15 pro GB aus der Cloud heraus (interne Transfers sind kostenfrei)

3.7.3. Windows Azure (Betriebssystem)

Windows Azure ist das Betriebssystem der Cloud Plattform. Die Komponenten von Windows Azure sind Compute, Storage, der Fabric Controller sowie Connect und das Content Delivery Network (CDN) (Abb. 3.11). Mithilfe des Fabric Controller wird die Infrastruktur kontrolliert. Mit Compute und Storage werden elementare Dienste wie Rechenkapazität und Speicherplatz zur Verfügung gestellt. Connect ermöglicht es, mittels IPsec eine direkte Netzwerkverbindung

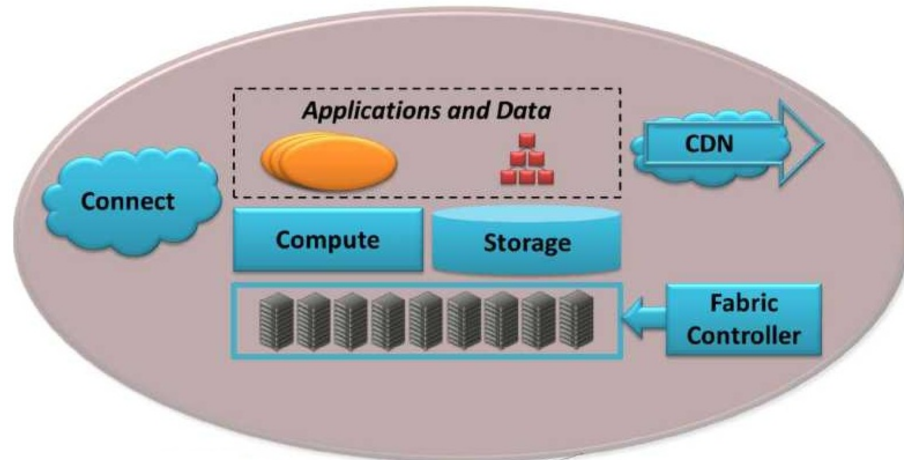

Abbildung 3.11.: Die Komponenten von Windows Azure. Quelle: [Chab]

in die Cloud zu erstellen. Das CDN ermöglicht es, Kopien von Daten über die ganze Welt zu verteilen.

Im Folgenden werden die Komponenten Compute und Storage genauer betrachtet.

Compute

Der Compute-Service von Windows Azure ermöglicht das Ausführen von Anwendungen in einer virtuellen Maschine [Chab]. Für verschiedene Arten von Anwendungen gibt es verschiedene Roles, von denen mindestens eine gewählt werden muss. Die momentan verfügbaren Roles sind:

- Web Role,
- Worker Role und
- VM Role.

Durch die Wahl einer Role wird das System in der virtuellen Maschine beeinflusst und lässt sich genauer an den Anwendungszweck anpassen.

Die **Web Role** wird vornehmlich für Web-basierte Anwendungen verwendet. In ihr befindet sich eine voreingestellte Instanz der Internet Information Services (IIS) 7, sodass sich z. B. ASP.NET-Anwendungen oder WCF-Services problemlos zur Verfügung stellen lassen. Auch native Anwendungen, welche nicht auf dem .NET-Framework basieren, können in einer Web Role betrieben werden.

Die **Worker Role** eignet sich für länger andauernde Aufgaben wie Simulationen oder Berechnungen im Hintergrund einer Webanwendung. Im Vergleich zur Web Role wird bei einer Worker Role auf ein vorinstalliertes und konfiguriertes IIS verzichtet.

Die **VM Role** ermöglicht das Laden der virtuellen Maschine mit einem von Nutzer bereitgestellten Windows Server 2008 R2 Image. Durch die VM Role wird also eine genauere Konfiguration der virtuellen Maschine ermöglicht.

Die virtuelle Maschine gibt es in verschiedenen Größen mit variierenden Rechen-, Speicher- und Festplattenkapazitäten. Momentan bewegen sich diese zwischen 1x 1 GHz CPU, 768 MB Speicher, 20 GB Festplatte und 8x 1.6 Ghz CPU, 14 GB Speicher und 2040 GB Festplatte.

Für jede Role kann die Anzahl der parallel laufenden Instanzen eingestellt werden, wobei die minimale Anzahl für jede Role bei zwei liegt. Die zur Verfügung stehenden Anwendungen können mittels HTTP, HTTPS oder TCP angesprochen werden. Jede Anfrage durchläuft auf ihrem Weg einen Load Balancer, der eine beliebige Instanz der entsprechenden Rolle auswählt und die Anfrage an diese weiterleitet. Da es weder Garantien noch Einflussmöglichkeiten auf das Verhalten des Load Balancers gibt, müssen die Anwendungen zustandslos sein (siehe Abschnitt 3.7.5).

Storage

Der Storage-Service von Windows Azure ermöglicht die Speicherung und den Zugriff auf persistente Daten sowie die asynchrone Kommunikation zwischen verschiedenen Roles [Chaa]. Es gibt drei grundlegende Strukturen, welche verschiedene Vor- und Nachteile mit sich bringen:

- Blobs,
- Tables und
- Queues.

Alle Datenstrukturen haben gemeinsam, dass sie in dreifacher Ausführung gespeichert werden, um Datenverlust zu vermeiden. Dabei wird insbesondere Konsistenz garantiert. Des Weiteren ist die Speicherung und der Zugriff auf die Daten durch REST (Representational State Transfer) realisiert, wobei jede Datenstruktur ihre eigene API hat.

3.7.4. SQL Azure

Mit SQL Azure wird es ermöglicht, relationale Datenbanken in der Cloud zu betreiben [Chaa]. Obwohl die Datenstrukturen in Windows Azure Stora-

ge effizienter sein können, sind relationale Datenbanken aufgrund der weiten Verbreitung ein wichtiges Element der Azure Plattform. Neben den für viele Entwickler vertrauten Techniken soll es so einfacher werden, bestehende Systeme zu integrieren. Die Basis für SQL Azure sind die SQL Server Produkte von Microsoft, Funktionen wie z. B. Volltextsuche sind allerdings momentan nicht verfügbar. SQL Azure besteht aus den Komponenten Database, Reporting und Data Sync. Hier wird ausschließlich auf die Database Komponente eingegangen.

Für Anwendungen ist die Nutzung von SQL Azure Database analog zu der eines lokalen SQL-Servers. Die unterstützten Protokolle sind TDS (Tabular Data Stream [Syb]) und OData, sodass bereits vorhandene Werkzeuge problemlos nutzbar sind. Eine Einschränkung ist bei der Laufzeit einer Anfrage zu beachten – sie darf nicht länger als eine bestimmte Zeitspanne benötigen. So soll sichergestellt werden, dass Anfragen anderer Nutzer zeitnah durchgeführt werden können.

Jeder SQL Azure Account kann mehrere logische Server nutzen, welche ihrerseits mehrere Datenbanken haben können. Einzelne Datenbanken dürfen maximal 50 GB groß sein. Wie auch bei Windows Azure Storage werden sämtliche Daten in dreifacher Ausführung gespeichert und permanent in einem konsistenten Zustand gehalten.

3.7.5. Programmiermodell

Die Windows Azure Plattform stellt einige Anforderungen an Anwendungen, damit die von Microsoft gegebenen Garantien eingehalten werden können. Auf diese Anforderungen, sowie deren Bedeutung für die Entwicklung, soll in diesem Abschnitt eingegangen werden.

Anforderungen

Um die Vorteile der Windows Azure Plattform nutzen zu können, müssen Anwendungen drei Anforderungen erfüllen [Chac]:

- Die Anwendung besteht aus einer oder mehr Roles.
- Es laufen mehrere Instanzen der Anwendung gleichzeitig.
- Die Anwendung verhält sich korrekt, wenn es Fehler in Instanzen einer Role gibt.

Die erste Anforderung ergibt sich allein aus der Methode, mit der Anwendungen in Windows Azure ausgeführt werden. Es muss eine der verfügbaren Roles gewählt werden, die Wahl ist abhängig vom Zweck der Anwendung.

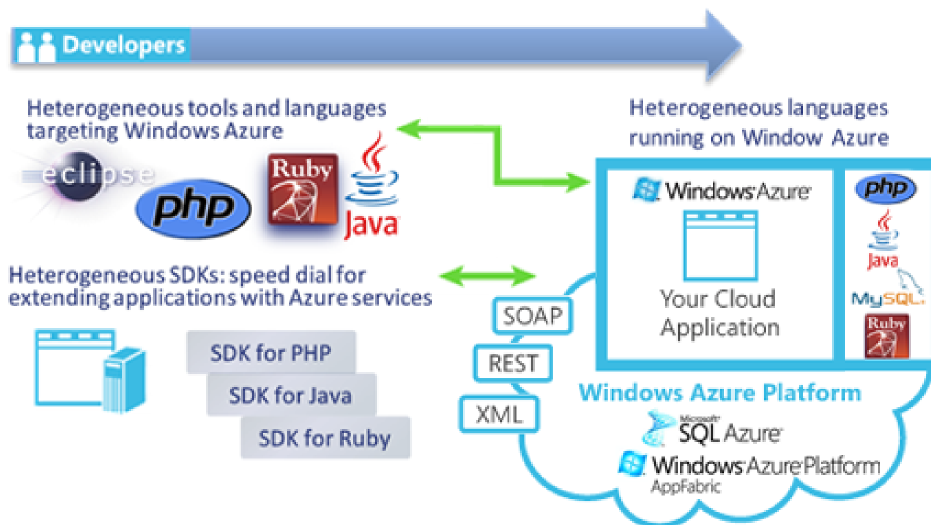

Abbildung 3.12.: Interoperabilität mit der Windows Azure Plattform. Quelle: [Micb]

Die zweite und dritte Anforderung sind vorwiegend für die Skalier- und Verfügbarkeit wichtig. Um diese zu erfüllen ist es nötig, dass die Anwendungen zustandslos arbeiten, d. h. das jede Anfrage unabhängig von vorherigen Anfragen stattfinden kann. Dies resultiert daraus, dass weder konfigurierbar noch vorhersagbar ist, welche der Instanzen einer Role für eine Anfrage gewählt wird.

Die Nutzung des Programmiermodells ist interessant, da nur für Anwendungen, die dem Programmiermodell gerecht werden, das Service Level Agreement (SLA) gilt, d. h. Garantien wie z. B. Erreichbarkeit in 99,95% der Zeit gewährleistet werden. Generell sind Anwendungen jedoch nicht gezwungen, das Modell zu erfüllen.

3.7.6. Interoperabilität

Windows Azure ist laut Microsoft [Micc] eine offene Plattform, die Entwicklern Möglichkeiten bietet, mit anderen Programmiersprachen und Werkzeugen als den hauseigenen zu arbeiten (Abb. 3.12). Als Sprachen werden .NET, PHP, Ruby, Python und Java angegeben und als alternatives Werkzeug Eclipse. Durch die in Windows Azure verwendeten Standards (u. a. HTTP, XML, Rest und Soap) soll außerdem die Möglichkeit für den Datenaustausch zwischen verschiedenen Technologien ermöglicht werden.

Als Werkzeuge stehen die Community-basierten SDKs für Java, PHP und

Ruby, sowie ein Plug-in für Eclipse zur Verfügung (siehe nächster Abschnitt). Informationen für Entwickler stellt Microsoft auf ihrem Interoperabilitätsportal [Micb] bereit.

Windows Azure Plug-in for Eclipse

Das Plug-in für Eclipse [Saw] soll Entwicklern die Arbeit mit Windows Azure erleichtern. Die Funktionalität beinhaltet

- Projekt-Templates,
- Beispiele für die Startup-Datei (siehe Abschnitt 3.7.6),
- Funktionen für leichteres lokales Testen,
- einen auf Ant basierenden Project-Builder,
- eine GUI zur Role-Konfiguration (Instance count, VM Size, Endpoints, ...),
- eine GUI für die Konfiguration von Remote Access und
- Schema-Validierung und Auto-Complete für Konfigurationsdateien.

Entwicklung mit Java

Die Entwicklung für Windows Azure mit Java unterscheidet sich grundlegend von der Entwicklung mit Microsoft-Technologien, da keinerlei Services zur Ausführung von Java zur Verfügung stehen. Windows Azure ist daher eher IaaS (Infrastructure as a Service) als PaaS (Platform as a Service). Der beschriebene Weg basiert auf einem Vortrag von Bert Ertman [Ert] auf den DevDays 2011.

Für die Ausführung einer Anwendung wird eine Worker-Role gewählt, welche nach jedem Bootvorgang das Java Runtime Environment (JRE/JDK) und einen Application-Server oder Servlet Container installiert. Dazu werden diese entweder im approot-Ordner einer Anwendung plaziert oder aus dem Windows Azure Storage geladen. Mit der Startup.cmd-Datei, welche automatisch nach dem Start der VM ausgeführt wird, werden diese entpackt und der Application-Server bzw. Servlet Container gestartet. Als letztes müssen die Endpoints der VM an die entsprechenden Endpoints des Application-Server weitergeleitet werden. Dies stellt nun die Plattform für Java-Anwendungen bereit.

Eine Anwendung kann mit Eclipse und dem Windows Azure Plug-in (siehe Abschnitt 3.7.6) zum Service-Paket verarbeitet werden. Danach wird sie in das Projekt mit dem Application-Server und der Runtime hinzugefügt. Die

Abbildung 3.13.: Das Zusammenspiel von Java SDK und Windows Azure. Quelle: [Soy]

Startup.cmd wird genutzt, um benutzte Bibliotheken zur Verfügung zu stellen und Konfigurationen am Application-Server vorzunehmen (z. B. Datenbanktreiber und Datenquellen). Das so erstellte Projekt wird anschließend zu einem Service-Paket verarbeitet und kann in Windows Azure bereitgestellt werden.

Nutzung von Windows Azure Services Durch die SDKs für Java (Windows Azure, AppFabric) lassen sich einige der Services von Windows Azure in Java verwenden (Abb. 3.13). Java APIs existieren für die Storage Services, den Service Bus und Access Control. SQL Azure ist in vollem Umfang mit einem JDBC-Treiber (Java Database Connectivity) von Microsoft [Mica] nutzbar.

3.8. Heroku

3.8.1. Beschreibung von Heroku

Heroku bietet ein Geschäftsmodell der Platform-as-a-Service-Kategorie an. Applikationen unterschiedlicher Programmiersprachen können ausgeführt werden, z. B. Ruby, Node.js, Java, Python, u. a. Der Service wurde um Git herum implementiert, d. h. Git-Repositories können über ein dediziertes Command-Line Interface (kurz CLI) direkt auf Heroku veröffentlicht werden. Vor allem heißt es jedoch auch, dass man seinen eigenen Quellcode hochladen muss, und keine bereits kompilierten JAR-Dateien (im Fall von Java) ausreichen.

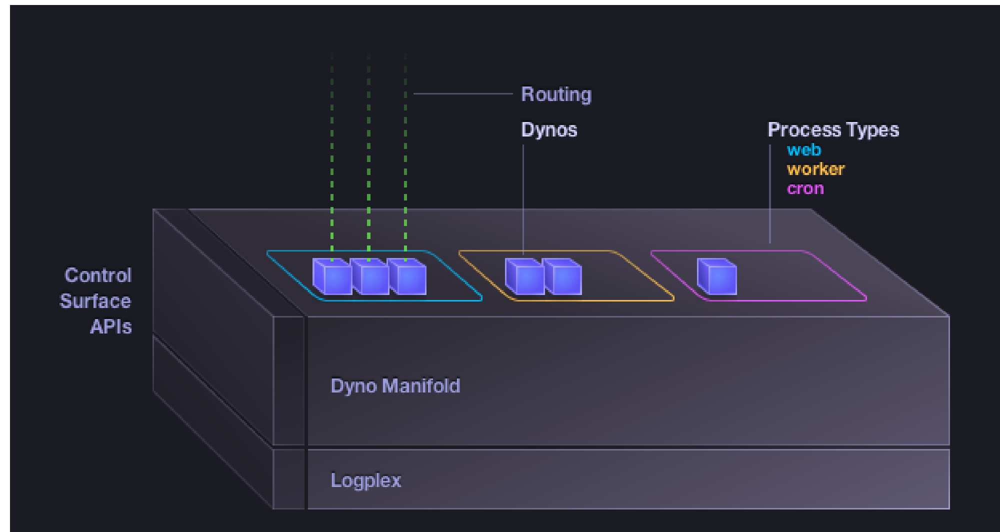

Abbildung 3.14.: Bild der Heroku-Architektur. Quelle: [Herb]

3.8.2. Architektur

Die Heroku-Architektur ist wie in Abbildung 3.14 aufgebaut und dabei vollständig prozessorientiert. Die Prozesse nennen sich bei Heroku „Dynos“. Wie in der Abbildung zu sehen gibt es drei grundlegende Typen:

- web,
- worker und
- cron.

Die **Web-Dynos** verarbeiten Anfragen an den Server. Das Routing der Anfragen an die ggf. mehreren verfügbaren Web-Dynos übernimmt die Heroku-Plattform. Die **Worker-Dynos** sind dazu gedacht länger andauernde Aktivitäten (z. B. Bild- oder Video-Konvertierungen) in den Hintergrund schieben zu können. In der Heroku-Mentalität arbeiten sie eine Queue ab. Die meisten Frameworks bieten entsprechende Funktionalität an (Ruby: Clockwork[4], Java: Quartz und RabbitMQ[5]). Die **Cron-Dynos** dienen dazu regelmäßig Aktivitäten zu *planen*. Sprich sie starten und legen einen Hinweis über die zu erledigende Aktivität in die Queue. Diese wird dann allerdings von einem Worker-Dyno abgearbeitet.

[4] https://devcenter.heroku.com/articles/clock-processes-ruby
[5] https://devcenter.heroku.com/articles/scheduled-jobs-custom-clock-processes-java-quartz-rabb

In der eigenen Anwendung definiert man innerhalb der Datei Namens „Procfile" jeweils ein Kommando für jeden dieser Prozesstypen. Über die Konsole kann man die Dynos dann parallelisieren. Sie alle laufen unabhängig voneinander und um die genaue Ausführungsreihenfolge kümmert sich die Heroku-Plattform. Die Ausgaben landen in einem zentralen Log.

Darüber hinaus bietet Heroku ein CLI zum Deployen von Anwendungen, zum Einsehen des Logs u. a. an. Das alles wird unter dem Überbegriff „Control Surface APIs" zusammengefasst

3.8.3. Preismodell

Die Bezahlung ist in unterschiedliche Schichten aufgeteilt. Auf der einen Seite gibt es die unterschiedlichen Dyno-Typen. Web-Dynos unterliegen bestimmten Beschränkungen (z. B. maximal 30 Sekunden Ausführungszeit) während Worker-Dynos für Arbeiten im Hintergrund (z. B. Konvertierung von Bildern) gedacht sind. Die Kunden können jederzeit je nach Bedarf eine bestimmte Anzahl der beiden Dyno-Arten buchen, dabei entspricht ein Dyno 720 Prozessstunden pro Monat und 750 Prozessstunden sind frei. Einfache Anwendungen, die mit einem Frontendprozess auskommen, sind dementsprechend kostenlos.

Auf der anderen Seite lassen sich dedizierte Datenbanken und so genannte Add-Ons buchen. Eine auf 10000 Zeilen begrenzte PostgreSQL Datenbank ist kostenlos, für entsprechenden Aufpreis gibt es auch ganze dedizierte Datenbankserver. Unter den Add-Ons finden sich zusätzlich viele nützliche Erweiterungen, z. B. Softwarelösungen zur Rechnungsstellung, zum Verarbeiten von eingehenden E-Mails, zum Ausführen von Cronjobs, aber auch Unterstützung für Websockets, erweiterte Logging-Mechanismen, und vieles mehr.

3.8.4. Heroku für Java

Heroku für Java basiert zum jetzigen Zeitpunkt auf OpenJDK Version 6 sowie Maven 3 und unterstützt die meisten J2EE Technologien. In Java 6 geschriebene Webanwendungen können auf der Heroku-Plattform ausgeführt werden, wenn die nötigen Abhängigkeiten, sowie ein Servlet Container (z. B. Tomcat oder Jetty) zur Ausführung der Anwendung, in der POM-Datei angegeben sind und der Pfad zu den auszuführenden Dateien im Procfile definiert ist [Ada11]. Dies ist notwendig, da auf Heroku kein Java-Servlet Container vorgegeben ist.

3.8.5. Vorgehen zum Veröffentlichen von Java-Applikationen

Die Heroku Plattform unterstützt das Angeben von Abhängigkeiten über POM-Dateien. Von jBPM musste also nicht die vollständige Bibliothek hochgeladen

werden, stattdessen ist es ausreichend, die eigene Anwendung zu entwickeln und die Abhängigkeiten in der POM-Datei, wie in der Entwicklung mit Maven, anzugeben. (Genau genommen führt Heroku nach dem Upload des Git-Repositories eben gerade Maven aus.)

Nach dem Herunterladen der Abhängigkeiten wird die Applikation auf Heroku kompiliert und danach gestartet. Während des Kompiliervorgangs erhält man die Ausgaben direkt im Anschluss an die PUSH-Operation von Git. Während die Applikation läuft, kann man sich die Ausgabe über die von Heroku bereitgestellte CLI-Anwendung selbst anschauen.

3.8.6. Datenbank

Die Nutzer der Heroku-Umgebung können eine Datenbank benutzen, indem sie die dazugehörigen Bibliotheken in der POM-Datei angeben und die Datenbankparameter konfigurieren. Falls der Nutzer sich für die PostgreSQL Datenbank entscheidet, ist ein Lesezugriff auf die Umgebungsvariablen der Heroku-Plattform notwendig (siehe Abschnitt 4.3.3).

3.9. VMForce

VMForce war ein Partnerschaftsprojekt zwischen VMware und Salesforce.com, das im April 2010 angekündigt wurde. Ziel der Kooperation war es, eine Cloud-Strategie zu entwickeln, so dass Java-Webanwendungen auf einer Cloud-Plattform hochgeladen werden können.

Eine VMForce-Anwendung sollte auf dem Spring-Framework entwickelt werden und auf dem SpringSource-tc-Server, der Enterprise-Version von Apache-Tomcat-Server, laufen. Als Cloud-Plattform kam Force.com von Salesforce zum Einsatz. Außerdem hatte Force.com geplant, eine relationale Datenbank und verschiedene Dienste zur Verfügung zu stellen, wie z. B. die Chatter-Services von Force.com, die VMware-vCloud-Technologie, eine API zur Integration von Web-Services und spezielle Tools für mobile Anwendungen [VMF].

Das VMForce-Projekt wurde jedoch im August 2011 gestoppt, nachdem Salesforce.com die konkurrierende Cloud-Plattform Heroku für 212 Mio. Dollar gekauft hat [Nat11]. Entsprechend hat sich die Projektgruppe nicht weiter mit dieser Plattform auseinandergesetzt.

3.10. JBoss Cloud – StormGrind

StormGrind ist ein Oberbegriff für JBoss-Cloud-Lösungen. Die Projekte BoxGrinder, Steam Cannon und CirrAS sollen zusammen einen umfassenden Cloud-Service bereitstellen. Die einzelnen Projekte lassen sich den in der Einleitung vorgestellten Cloud-Ebenen wie folgt zuordnen:

BoxGrinder Mit BoxGrinder werden virtuelle Maschinen erstellt, die bei IaaS-Anbietern wie Amazon, ausgeführt werden können.

SteamCannon SteamCannon dient zur Einrichtung des IaaS, baut also auf der PaaS-Umgebung auf, und dient als PaaS-Vermittler. Mit SteamCannon lassen sich PaaS-Umgebungen individuell einrichten und konfigurieren und Webanwendungen aufsetzen.

CirrAS CirrAS ermöglicht ein einfaches Application-Deployment und schließt damit die Brücke, um Anwendungen auf der von SteamCannon bereitgestellten PaaS-Umgebung ausführen zu lassen.

Das Problem der JBoss-Cloud-Lösungen war der Work-in-Progress-Zustand. Die Dokumentation ist lückenhaft und beschränkt sich scheinbar auf Standard- und Demofälle. Ganze Ausfälle von StormGrind-Unterprojekten, wie der eigenen SteamCannon-Website, waren auch zu beobachten. Die wenigen dokumentierten Szenarien waren für die Projektgruppe ungeeignet.

Da vor allem der Ausfall der SteamCannon-Website zum Zeitpunkt der Recherche sehr gravierend war, und die PG somit nicht die notwendigen Informationen sammeln konnte, wurde entschlossen diese Lösung nicht weiter zu untersuchen.

4. Grundlegende Erkenntnisse (Semester 1)

4.1. Google App Engine

Zusammen mit Windows Azure fiel relativ früh die Wahl als „Clouddienst“ für die PG auf die Google App Engine. Zum einen aufgrund des Supports für Java und dem dazugehörigen sehr komfortablen Eclipse Plugin, zum anderen wegen des kostenlosen Kontingents, welches die Google App Engine für unser Projekt anbot.

Nach der Einarbeitung in die Google App Engine stellte sich die Frage, welche Prozessumgebung die PG benutzen soll. Die Wahl fiel dabei zunächst auf Activiti und jBPM5. Später wurde auch mit jBPM4 gearbeitet, da aufgrund von Persistenzproblemen jBPM5 nicht zu GAE kompatibel war.

Zuerst wird auf die GAE in Bezug auf Activiti eingegangen. Das Hauptaugenmerk dieses Kapitels richtet sich auf die GAE zusammen mit jBPM5 und jBPM4, da die PG sich in dem ersten Semester am längsten mit diesen beiden Prozessumgebungen befasst hat.

Der Abschnitt 4.1.3 stammt von Kada Benadjemia.

4.1.1. Activiti

Activiti setzt auf MyBatis[1] anstatt auf Hibernate bzw. JDBC. Bei MyBatis handelt sich um ein Open-Source-Persistenz-Framework für Java und .NET. Der Vorteil von MyBatis liegt in der Trennung von Datenbankzugriffscode vom restlichen Applikationscode. Hierfür werden der Applikation Data-Access-Objects zur Verfügung gestellt und die SQL-Statements in XML-Dateien, auch SQL-Maps genannt, geschrieben. Da es sich aber nicht um ein Object-Relational-Mapping-Framework handelt, muss der Entwickler sich selbst sowohl um das Schreiben der SQL-Statements als auch um das Mapping von objektorientierten Klassen und relationalen Tabellen kümmern.

Versucht man MyBatis in der GAE auszuführen, erhält man eine Access Control Exception, weil MyBatis versucht den Classloader zu benutzen (ja-

[1] http://de.wikipedia.org/wiki/Mybatis/

va.lang.RuntimePermission getClassLoader). Aufgrund der Einschränkungen der Javafunktionalität durch die GAE ist das also nicht möglich.

Dieser Umstand und der Fakt, dass aktuell keine SQL-Datenbank unterstützt wird, verhindern den Einsatz von MyBatis und damit auch von Activiti in der Google App Engine. Dieser Weg wurde daher nicht vertieft und Activiti auf der GAE nicht weiter betrachtet.

4.1.2. jBPM5

Unser erstes Ziel war es, die derzeit aktuelle Version jBPM 5.0 auf der Google App Engine zu deployen und zu testen.

Nach Einarbeitung in den Userguide und dem Betrachten des Quellcodes stellte sich heraus, dass jBPM5 auf Maven setzt. Für die Persistenz setzt jBPM auf Hibernate, eine Implementierung der JPA-Spezifikation, und H2[2] als Datenbank. Bei der H2 Datenbank handelt es sich um eine in Java geschriebene SQL-Datenbank. H2 ist als Open-Source-Software für die Java-Plattform verfügbar. Bei der Entwicklung des Systems steht ein schlanker Aufbau sowie eine im Vergleich zu anderen Datenbanksystemen hohe Verarbeitungsgeschwindigkeit im Vordergrund. Sie verwendet die JDBC API und unterstützt im Vergleich zu MySQL zusätzlich „Pure Java“ und verschlüsselte Datenbanken.

jBPM5 und die Persistenz

Eine der Hauptschwierigkeiten beim Deployen in der Cloud scheint auch hier die Persistenz zu sein. Dies gilt besonders für die Google App Engine, da diese noch keine relationalen Datenbanken unterstützt. Momentan gibt es zwar eine SQL-API zu Testzwecken, für diese muss man sich aber bewerben. Für den Bewerbungsantrag sind genaue Angaben über Größe und Umfang der Datenbank nötig, Informationen, die uns zum aktuellen Zeitpunkt nicht zur Verfügung stehen. Somit muss jBPM den von Google stattdessen verwendeten Datastore als Datenbank nutzen. Der Zugriff auf diesen Datenspeicher ist nur mit reinem JPA möglich. Hibernate wird hingegen nicht unterstützt.

Nach genauerem Betrachten des jBPM5-Source-Codes bleibt die Erkenntnis, dass sich die Hibernate-Queries nicht einfach aus dem Code entfernen lassen, da sie in essenziellen Projektordner wie „ jBPM_db“ oder „ jBPM_persistence“ vorkommen (ca. 70 Hibernate Queries). Somit bleibt nur die Möglichkeit, eben jene Queries manuell durch JPA-konforme Queries zu ersetzen. Grundsätzlich sind die Query-Sprachen Hibernates und JPAs identisch. Dennoch fehlen Funktionalitäten wie beispielsweise Datenabfragen nach konkreten Entities in der

[2] http://www.h2database.com/

JPA 1.0 Spezifikation (siehe 3.3.4). In Abschnitt 3.3.3 lassen sich alle Funktionalitäten und Annotationen ablesen, die derzeit nicht in der GAE verfügbar sind. Nach Aussage des jBPM5 Entwicklers Kris Verlaenen werden bis auf eine Stelle im gesamten Code nur JPA konforme Hibernate-Queries verwendet. Die in Abschnitt 3.3.3 befindliche Tabelle 3.1, Seite 17, zeigt, dass eine Ersetzung relativ gradlinig hätte verlaufen können. Tatsächlich ließe sich somit also der komplette Code von jBPM5 entsprechend der Vorgaben der GAE anpassen, wäre da nicht die erwähnte eine Stelle die eine plain Hibernate-Query benutzt. Die besagte Stelle[3] benutzt dabei in Zeile 66 die hibernate-spezifische Annotation „@CollectionOfElements“. „@CollectionOfElements“ gilt als deprecated und lässt sich durch das JPA-konforme @ElementCollection ersetzen. Dies ist aber erst ab Spezifikation Version 2.0 von JPA möglich. Die Google App Engine unterstützt bisher nur JPA 1.0, weshalb eine Anpassung des Codes aktuell nicht möglich wäre.

jBPM5 und Drools

jBPM setzt auf Drools, einer Business-Rules-Engine, auf. Sie ermöglicht die Trennung vom Design der Prozesse von den einzuhaltenden Business-Regeln des Unternehmens, mit dem Ziel den Designprozess zu vereinfachen.

Drools benötigt eine Konfigurationsdatei, deren Existenz es jedes Mal prüft bevor es sich überhaupt die Business-Rules-Files anschaut. Da jeglicher Dateizugriff in der GAE untersagt wird, wird also selbst bei funktionierender `File.exists()` Abfrage für die Konfigurationsdatei spätestens beim Versuch die DLR-Dateien mit `getClass().getResourceAsStream` zu lesen eine *SecurityException* geworfen.

Mit dieser Erkenntnis – eine Suche via Google und in diversen Foren scheint dies zu bestätigen – ist eine Ausführung von Drools in der GAE aktuell nicht möglich.

Aufgrund dieser Schwierigkeiten mit Persistenz und Drools wurde ebenfalls jBPM4 untersucht, in der Hoffnung Drools und plain Hibernate-Queries würden in dieser Version nicht existieren. In dem nächsten Kapitel wird daher genauer auf die Arbeit und Ergebnisse der PG mit jBPM4 und der GAE eingegangen.

4.1.3. jBPM4

Das Ziel der Gruppe, die sich mit der GAE im Zusammenhang mit jBPM4 beschäftigte, war eine funktionierende jBPM Version auf die GAE zu deployen.

[3] https://github.com/droolsjbpm/jbpm/blob/master/jbpm-persistence-jpa/src/main/java/org/jbpm/persistence/processinstance/ProcessInstanceInfo.java

Es wurde dabei die Version 4.4 benutzt.

Im Vergleich zu jBPM 5 und der Hauptgrund für den Wechsel der Prozessumgebung war, dass das Drools-Paket in jBPM 4.4 so gut wie nicht genutzt wird. Drools tauchte nur in drei Dateien auf, die nicht zwingend notwendig für die Prozessausführung waren. Somit konnte man sich auf das eigentliche Problem konzentrieren, was daraus bestand jBPM 4 kompatibel zu JPA 1 zu machen, um es auf der GAE deployen zu können. Ähnlich wie mit jBPM5 stellte sich hier die Frage, welche Anpassungen von Nöten sind, um das Projekt für die GAE kompatibel zu machen.

Die ersten Probleme tauchten bereits bei dem Projektimport mit Maven auf. So waren einige Abhängigkeiten in der `pom.xml` nicht mehr aktuell und mussten manuell angepasst werden. Die PG entschied sich für die Vorgehensweise nicht direkt das komplette Projekt mit allen Abhängigkeiten zu importieren, sondern zuerst zu schauen, welche Projekte relevant für einen kleinen Testprozess sind, der auf die GAE deployed werden sollte.

jBPM4 und die Persistenz

Nach Behebung der Importfehler konnte jBPM4 auf das Nötigste reduziert werden und auf die GAE deployed werden, jedoch ohne ein lauffähiges Testprojekt. Das hieß, dass die Persistenz noch weiterhin angepasst werden musste, so dass der nächste Schritt das genaue Anpassen der Hibernate-Stellen war. Für die Projektgruppe relevante Stellen, die Hibernate beinhalten, sind in der untersuchten Version vor allem in den beiden Unterprojekten `JBPM-db` und `JBPM-pvm`. Besonders `JBPM-pvm` wies eine Menge pure Hibernate-Funktionsaufrufe auf. Hierbei handelte es sich um die zentrale Process Virtual Machine, die den Kern der Prozessausführung darstellte. Es ließen sich 30 Dateien mit Quellcode ausfindig machen, die Hibernate-spezifischen Code aufwiesen. Die Dateien selbst waren meist klein. Problematisch hingegeben war die Verflechtung Hibernates mit jBPM 4, sodass wir von unserem Ersetzungsvorhaben abgesehen haben.

Aufgrund der Probleme mit der Persistenz im Zusammenhang mit der aktuellen JBPM5 Version und der älteren JBPM4 Version entschied sich die PG dafür, die GAE als Cloud Plattform nicht weiter zu verfolgen. So war der Aufwand, um kompatible jBPM-Prozesse für die GAE zu entwerfen, einfach zu groß.

Die GAE sollte in Zukunft jedoch, besonders im Hinblick auf die Persistenz, weiter verfolgt werden, da Google bereits erste APIs und Unterstützungen liefert, um mit relationalen Datenbanken zu arbeiten.

Erwähnenswert ist die im vorherigen Unterkapitel angesprochene SQL API. Experimentell testet Google außerdem die relationale Datenbank Google Cloud

SQL[4], welche eine weitere mögliche Lösung des Persistenzproblems sein könnte.

4.2. Windows Azure

4.2.1. Activiti

Generell lässt sich sagen, dass Activiti auf Windows Azure ohne größere Probleme lauffähig ist. Bis zur erfolgreichen Ausführung sind jedoch einige Schritte nötig. Zum Testen haben wir ein Java-Servlet geschrieben, welches einige elementare Funktionen von Activiti aufruft und u. a. den Datenbankzugriff testet.

Um das Activiti-Servlet zu testen, wird mit Hilfe des Azure-Eclipse-Plugins ein Windows-Azure-Projekt erstellt und die Abhängigkeiten sowie unser Servlet in Form einer `war`-Datei hinzugefügt. Weiterhin enthält das Projekt die Ordner `deploy` und `WorkerRole1`. Ersterer enthält nach dem Kompilieren des Projekts die Dateien `ServiceConfiguration.cscfg` und `WindozsAzurePackage.cspkg`, welche zum Deployen auf Windows Azure benötigt werden. Der `WorkerRole1`-Ordner enthält Dateien, auf die wir später zur Laufzeit zugreifen wollen. Wir konfigurieren im Projekt, dass wir eine kleine Compute-Instanz verwenden wollen und das es als Workerrole ausgeführt werden soll. Weiterhin haben wir einige Abhängigkeiten aus dem Projekt entfernt und als `zip`-Dateien in den Windows Azure Storage Blob geladen, damit wir nicht bei jedem Deployvorgang wieder alles hochladen müssen.

Beim Starten einer Role-Instanz wird automatisch die im Projekt befindliche Datei `startup.cmd` ausgeführt. Der Inhalt dieser Datei wird in Listing 4.1 gezeigt. In dieser Datei wird die Arbeit verrichtet, die nötig ist, um unser Servlet zur Ausführung zu bringen.

Im ersten Abschnitt werden alle nötigen `zip`-Dateien aus dem Blob geladen. Zu nennen sind hier der Apache Tomcat, das Java Runtime Environment (JRE), für Activiti benötigte Libraries und zugehörige Webanwendungen sowie `SQLJDBC4.jar`, der Datenbanktreiber für SQL Azure unter Java. Der Tomcat-Server und das JRE wird zur Ausführung der Webanwendung innerhalb der Role benötigt. Die Activiti zugehörigen Webanwendungen sind einerseits der Activiti Explorer und andererseits die REST-Schnittstelle.

Da in Azure bei jedem Start einer Role-Instanz ein neuen Server erstellt wird, müssen jedes Mal alle Dateien heruntergeladen, entpackt und an die entsprechenden Orte kopiert werden. Im einzelnen werden die benötigten Bibliotheken in das `lib`-Verzeichnis sowie die Webanwendungen in das `webapps`-Verzeichnis des Tomcat-Servers kopiert.

[4] https://developers.google.com/cloud-sql/

```
1  @REM download resources
2  cscript util\download.vbs "http://pgpcb2.blob.core.windows.
       net/util/apache-tomcat-7.0.22-windows-x64.zip"
3  cscript util\download.vbs "http://pgpcb2.blob.core.windows.
       net/util/jre.zip"
4  cscript util\download.vbs "http://pgpcb2.blob.core.windows.
       net/util/activity_sharedlibs.zip"
5  cscript util\download.vbs "http://pgpcb2.blob.core.windows.
       net/util/activity_webapps.zip"
6  cscript util\download.vbs "http://pgpcb2.blob.core.windows.
       net/util/sqljdbc4.jar"
7
8  @REM unzip Tomcat
9  cscript /B /Nologo util\unzip.vbs apache-tomcat-7.0.22-windows-x64.
       zip "%ROLEROOT%\approot"
10
11 @REM unzip library files
12 cscript /B /Nologo util\unzip.vbs activity_sharedlibs.zip "
       %ROLEROOT%\approot\apache-tomcat-7.0.22\lib\"
13 copy sqljdbc4.jar "%ROLEROOT%\approot\apache-tomcat-7.0.22\lib\"
14
15 @REM unzip JRE
16 cscript /B /Nologo util\unzip.vbs jre.zip "%ROLEROOT%\approot"
17
18 @REM unzip webapps
19 cscript /B /Nologo util\unzip.vbs activity_webapps.zip "%ROLEROOT%\
       approot\apache-tomcat-7.0.22\webapps\"
20
21 @REM copy project files to server
22 copy tutorial1.war "%ROLEROOT%\approot\apache-tomcat-7.0.22\webapps
       \"
23
24 @REM start the server
25 cd "%ROLEROOT%\approot\apache-tomcat-7.0.22\bin"
26 set JRE_HOME=%ROLEROOT%\approot\jre
27 startup.bat
```

Listing 4.1: Unsere startup.cmd.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <property name="databaseSchemaUpdate" value="true"/>

    <property name="jdbcUrl" value="jdbc:sqlserver://<database_identifier>.database.windows.net:1433;Database=<database_name>" />
    <property name="jdbcDriver" value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
    <property name="jdbcUsername" value="<username>@<database_identifier>" />
    <property name="jdbcPassword" value="<password>" />

    <property name="jobExecutorActivate" value="true" />
  </bean>

</beans>
```

Listing 4.2: Unsere activiti.cfg.xml für das Activiti Projekt.

Sobald alle benötigten Dateien vorhanden sind, wird die JRE_HOME Variable gesetzt und im bin-Verzeichnis des Tomcat die startup.bat ausgeführt, was den Server startet.

Bei richtiger Konfiguration der Endpunkte der Role lassen sich nun die Webanwendungen durchs Internet aufrufen. Weiterhin lässt sich mittels Remote-Desktop-Verbindung auf den Server zugreifen, um Anpassungen vorzunehmen.

Die benötigte SQL-Azure-Datenbank muss einmalig in der Konfigurationsoberfläche von Windows Azure erstellt werden. Danach wird der Zugriff auf die Datenbank in Activiti in der activiti.cfg.xml, also der Konfigurationsdatei für die Activiti-Engine, konfiguriert. Unsere Datei ist in Listing 4.2 dargestellt.

Der Eintrag jdbcUrl gibt die Adresse der Datenbank sowie die den Datenbankname an und über jdbcDriver wird der Datenbanktreiber für SQL Azure angegeben. Die Werte <database_identifier>, <database_name>, <username> und <password> sind Platzhalter und hängen von der Konfiguration der Datenbank ab. Mit den passenden Werten für diese Einstellungen funktionierte

Activiti problemlos unter Nutzung der SQL-Azure-Datenbank.

4.2.2. jBPM

Die Verwendung von jBPM5 in Kombination mit Windows Azure stellte sich als ein schwieriger Fall heraus. Zwar lässt sich die grundlegende Funktionalität von jBPM schnell zur Verfügung stellen, die erfolgreiche Verwendung einer Azure SQL Database konnte jedoch bis zum Ende nicht erreicht werden. Bei der Verwendung der Persistenz wird zur Laufzeit reproduzierbar eine Ausnahme geworfen, welche durch das Scheitern einer Transaktion durch einen Null-Pointer in den Funktionen der jBPM-Bibliothek entsteht. Eine Suche nach möglichen Ursachen und Lösungen ergab nach mehreren Wochen keine weiteren Erkenntnisse. Der Aufbau des Projektes und die Konfiguration der Persistenz entspricht der Beschreibung in Abschnitt 4.2.1.

Durch das in Windows Azure genutzte Programmiermodell (siehe Abschnitt 3.7.5, Seite 38), insbesondere der Zustandslosigkeit der Anwendungen und der Notwendigkeit einer Datenspeicherung außerhalb der Role, ist die Verwendung der Persistenz essentiell für die sinnvolle Nutzung von jBPM. Wir haben uns deswegen aus Mangel an Lösungen dazu entschlossen die Arbeit an jBPM, zumindest in Azure, einzustellen und Activiti als Prozessengine zu nutzen.

4.3. Heroku

4.3.1. Allgemeines

Die Cloud-Plattform Heroku wirbt mit einem einfachen Command-Line Interface (kurz CLI), welches sehr bequemen Zugang zu der Funktionalität von Heroku bereitstellt. Leider erwies sich die Einrichtung des Clients auf Windows als kompliziert. Das normale Prozedere umfasst die Installation von Git (msysgit[5]) und dem eigentlichen Heroku-Client. Der Heroku-Client wiederum nutzt SSH und baut auf dem Git-Protokoll auf.

Bei den ersten Versuchen brachen viele der Heroku-Operationen, welche auf dem Git-Protokoll aufsetzen, jedoch ab. Dies sorgte dafür, dass man Fehlermeldungen von Heroku nicht lesen konnte. Auch der Heroku-Support konnte bei dieser Problematik nur bedingt weiterhelfen. Im Endeffekt stellte sich die Kombination mit dem SSH-Client PuTTY[6] als nicht unterstützt heraus. Ein Wechsel auf OpenSSH[7] (welcher z. B. der msysgit-Installation beiliegt) löste

[5] http://msysgit.github.com/
[6] http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
[7] http://www.openssh.org/

diese Probleme. Dass der Heroku-Client in Kombination mit PuTTY einfach nicht funktioniert, war jedoch scheinbar auch dem Heroku-Support bzw. den Heroku-Dokumentatoren nicht bewusst. Es findet sich zum jetzigen Zeitpunkt zumindest kein entsprechender Hinweis in der Dokumentation. Wir haben den Heroku-Support dort um Nachbesserung gebeten.

4.3.2. jBPM

Das größte Problem mit jBPM auf Heroku war das Herausfinden aller richtigen Abhängigkeiten und Angabe dieser in der `pom.xml`. Außerdem kristallisierten sich hierbei all die Probleme des vorigen Abschnitts heraus, sodass sich das Ausprobieren auf Heroku auf mehrere Wochen verteilte.

jBPM ohne Persistenz bzw. mit der H2-Datenbank als Persistenz lief irgendwann jedoch einwandfrei. Die PostgreSQL-Datenbank von Heroku für jBPM konnte im Rahmen der PG jedoch nicht vollständig initialisiert werden. Es gab immer wieder kleinere Komplikationen, z. B. dass die Einstellungen in einer XML-Datei vorgenommen werden, Heroku die URL zur Datenbank jedoch als Umgebungsvariable bereitstellt. Das Ermitteln der korrekten Datenbanktreiber verlief in der Theorie zwar erfolgreich, aber funktioniert hat auch das nicht richtig.

Da zu diesem Zeitpunkt bereits absehbar war, dass wir im zweiten Semester Acitivi nutzen werden (allein schon, weil die Einrichtung der Persistenz dort deutlich problemloser verlief) und die Fortschritte bezüglich jBPM und Persistenz sehr schleppend verliefen, hat sich die PG entschlossen dies nicht weiter zu verfolgen.

4.3.3. Activiti

Um eine Activiti-Prozessinstanz auf der Cloud-Plattform Heroku (siehe Abschnitt 3.8) ausführen zu können, haben wir das bereits bekannte Beispiel aus Abbildung 3.8, Seite 27, gewählt. Die zugehörige Acitivi-Prozessdefinition in XML-Form findet sich in Listing A.3, Seite 109. Die ausgewählte Prozessdefinition enthält eine Sequenz von zwei User-Tasks und lässt sich auf der Heroku Plattform ausführen.

Prinzipiell kann eine Activiti-Prozessdefinition auf einem lokalen Rechner mithilfe der Activiti-Engine ausgeführt werden. Die Engine benutzt eine vorkonfigurierte H2-Datenbank, die auf dem Rechner gestartet wird. Da aber Heroku eine PostgreSQL-Datenbank (siehe Abschnitt 3.8.6) zur Verfügung stellt, wurde die Datenbank umgestellt, indem die von der Prozess Engine gelesene Datei mit dem Namen `activiti.cfg.xml` (siehe Listing 4.3) erstellt wurde und die

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5         xsi:schemaLocation="http://www.springframework.org/schema/
              beans   http://www.springframework.org/schema/beans/
              spring-beans.xsd">
6
7    <bean id="processEngineConfiguration" class="org.activiti.engine.
         impl.cfg.StandaloneInMemProcessEngineConfiguration">
8
9   <property name="jdbcUrl" value="jdbc:postgresql://localhost:5432/
        template1/activiti;DB_CLOSE_DELAY=1000" />
10     <property name="jdbcDriver" value="org.postgresql.Driver" />
11     <property name="jdbcUsername" value="postgres" />
12     <property name="jdbcPassword" value="asdf" />
13
14     <!-- Database configurations -->
15     <property name="databaseSchemaUpdate" value="true" />
16
17     <!-- job executor configurations -->
18     <property name="jobExecutorActivate" value="false" />
19
20     <!-- mail server configurations -->
21     <property name="mailServerPort" value="5025" />
22   </bean>
23
24 </beans>
```

Listing 4.3: activiti.cfg.xml-Datei für PostgreSQL.

Datenbank-Parameter in der Datei angepasst wurden.

Die Prozess-Engine lässt sich dann mit dem Aufruf der Methode, die in Listing 4.4 angegeben ist erzeugen.

Die ausgewählte Prozessdefinition wurde mit dem in Abschnitt 3.5.4, Seite 27 (siehe vollständiger Quellcode in Listing A.4, Seite 110), vorgestellten Codeabschnitt getestet, wobei wir das Erzeugen der Process Engines (die Zeilen 3 bis 6) durch die von uns erstellten Methoden (siehe Codeabschnitt 4.5) ersetzt haben.

Um die Prozess-Engine auf Heroku instantiieren zu können, muss die Anwendung erst die Umgebungsvariable von Heroku einlesen und damit die Datenbank-Parameter setzen. Ein Beispiel, wie auf die Variable zugegriffen und eine Prozess-Engine instantiiert werden kann, ist im Listing 4.6 zu finden. Die Probleme, mit denen wir konfrontiert wurden, wurden vor allem durch die Erstellung der

```
1  // Erstellen einer Prozess-Engine mit der activiti.cfg.xml-Datei
2
3  public void buildProcessEngineLocal() {
4    this.setProcessEngine(ProcessEngineConfiguration
5        .createProcessEngineConfigurationFromResource("activiti.cfg
           .xml")
6        .buildProcessEngine());
7  }
```

Listing 4.4: Erstellen eines Process-Engine.

```
1  BuildProcessEngine db = new BuildProcessEngine();
2
3  /*
4   * Erstellen der Activiti Process Engine mit postgreSQL
5   * auf dem lokalen Rechner wird die Methode -
6   * db.buildProcessEngineLocal aufgerufen
7   */
8
9  db.buildProcessEngineOnHeroku();
10 processEngine = db.getProcessEngine();
```

Listing 4.5: Erstellen einer Process-Engineinstanz.

```
1  // Create Activiti process engine on Heroku
2    public void buildProcessEngineOnHeroku() {
3      URI dbUri = null;
4      try {
5        dbUri = new URI(System.getenv("DATABASE_URL"));
6      } catch (URISyntaxException e) {
7        e.printStackTrace();
8      }
9
10         String username = dbUri.getUserInfo().split(":")[0];
11         String password = dbUri.getUserInfo().split(":")[1];
12         String dbUrl = "jdbc:postgresql://" + dbUri.getHost() +
               dbUri.getPath();
13
14         this.setProcessEngine(ProcessEngineConfiguration.
               createStandaloneProcessEngineConfiguration()
15                 .setJdbcDriver("org.postgresql.Driver")
16                 .setJdbcUsername(username)
17                 .setJdbcPassword(password)
18                 .setJdbcUrl(dbUrl)
19                 .setDatabaseSchemaUpdate("true")
20                 .setJobExecutorActivate(true)
21                 .setMailServerPort(5025)
22                 .buildProcessEngine());
23   }
```

Listing 4.6: Erstellen einer Process-Engine auf Heroku.

POM-Datei verursacht. Die activiti-engine 5.8-Abhängigkeit, die von Maven heruntergeladen wurde, war beschädigt. Das hatte zur Folge, dass die Activiti-Klassen nicht gefunden werden konnten. Daher haben wir in der POM-Datei die Version 5.8 der activiti-engine-Angabe auf 5.7 heruntergesetzt. In der neuen Version 5.9, die seit 01.03.2012 veröffentlicht ist, ist dieses Problem behoben worden.

4.4. Entscheidungen fürs 2. Semester

Die Projektgruppe musste sich zum Ende des ersten Semesters hin die Frage stellen, in welche Richtung die Projektplanung gehen soll. Die Erkenntnis, dass die Google App Engine aufgrund der fehlenden Persistenz für relationale Datenbanken – zumindest zu dem Zeitpunkt –, nicht in Frage kam, beschränkte die Auswahl auf zwei mögliche Clouddienste, auf die das Projekt im Laufe des zweiten Semesters aufgespielt werden konnte.

Windows Azure bot kaum Einschränkungen, war jedoch aufgrund der nötigen Kreditkarte, die man für die freien sechs Monate brauchte, nicht die ideale Wahl. Heroku sah als relativ neuer Clouddienst vielversprechend aus, da das freie Kontingent zum Testen des Projektes ausreichte und kaum Einschränkungen vorhanden waren.

Daher gehörte im zweiten Semester unter anderem das konkrete Aufspielen des umzusetzenden Projektes auf Heroku zur Projektplanung. Neben der Wahl des Clouddienstes war zum Ende des ersten Semesters klar, dass die Projektgruppe als Prozessumgebung Activiti verwendet und dieses somit im zweiten Semester auf Heroku aufsetzen wird. JBPM in seinen unterschiedlichen Versionen war aufgrund der schlechten Erfahrungen zusammen mit der Google App Engine und aufgrund der sich stark unterscheidenden Versionen für die Projektgruppe keine wirkliche Alternative mehr. Diese beiden, aus den Erkenntnissen des ersten Semesters gewonnenen, Entscheidungen bildeten die Grundlage für das weitere vorgehen im zweiten Semester, um das von den Projektbetreuern am Anfang eben jenes Semesters ausgewählte Projekt umzusetzen.

5. Proof-of-concept – Monopoly in der Cloud (Semester 2)

5.1. Anforderungen

Nachdem sich die Projektgruppe im ersten Semester hauptsächlich mit der Recherche unterschiedlicher Prozessumgebungen und Cloudsystemen befasst hat, war als Inhalt des zweiten Semesters die Umsetzung der gewonnenen Erkenntnisse in Form einer, von den Projektgruppenbetreuern ausgewählten, Anwendung vorgesehen. Dies sollte auf einem oder im besseren Fall zwei Cloudsystemen lauffähig sein.

Die Aufgabe der Projektgruppe bezog sich auf die Umsetzung eines auf Prozessen basierenden Projekts, welches in der Cloud von mehreren Benutzern ausführbar sein sollte. Am Ende des ersten Semesters wurde dieses von den Projektgruppenbetreuern ausführlicher spezifiziert. Umgesetzt werden sollte das Spiel Monopoly. Da das Spiel jedem bekannt war und der rundenbasierte Spielaufbau gut strukturierbar ist, bot es sich für einen prozessorientierten Ansatz an.

Die Minimalanforderungen waren dabei ein lauffähiges Spiel zu entwickeln, welches auf Prozessen basiert und zumindest von einem Spieler komplett fehlerfrei gespielt werden kann. Dazu konnte die PG entscheiden, ob sie eine einfache KI entwickelt, die gegen den Spieler spielt und einfache Aktionen ausführt, oder ob man eine Variante umsetzt, bei der weitere menschliche Gegenspieler am Spielgeschehen teilnehmen können.

Aus welchen Gründen sich die PG bei der Umsetzung für eine dieser Möglichkeiten entschieden hat, wird im folgenden Unterabschnitt erläutert.

5.2. Projektplan

Aufgrund der Erkenntnisse aus dem ersten Semester war klar, dass lediglich Heroku und Windows Azure für unsere prozessgesteuerte Anwendung in Frage kommen. Aufgrund der notwendigen Kreditkarte bei Azure konzentrierte sich die Projektgruppe auf die gezielte Umsetzung des Projekts auf dem Clouddienst Heroku.

Zu Anfang wurde die Projektgruppe in drei Gruppen unterteilt, die sich mit unterschiedlichen Aufgaben der Umsetzung auf Heroku beschäftigten sollten. Dabei haben sich während des Semesters folgende Planungsbereiche herauskristallisiert:

- Erstellen der GUI,
- Umsetzung eines Datenmodell,
- Prozessumsetzung,
- Entwickeln nötiger Javaklassen,
- Nachrichten- und Kommunikationsumsetzung,
- Umsetzung von unterschiedlichen Tests (JUnit, Selenium) und
- Umsetzung der KI.

Die Aufzählung gibt dabei die zeitliche Reihenfolge der Planungsaufgaben wieder.

Neben einer GUI-Gruppe, die sich mit der Umsetzung des Spielfelds auseinandersetzte, beschäftigten sich die beiden anderen Gruppen mit dem Datenmodell sowie der Einarbeitung und Umsetzung erster Prozesse mit Hilfe der Prozessumgebung „Activiti“. Da die GUI-Gruppe fast immer auf die Arbeit der anderen Gruppen angewiesen war, arbeitete diese bis zum Ende des Semesters an Verbesserungen, Anpassungen und neuer Funktionalität der GUI.

Nach der Fertigstellung des Datenmodells begann die entsprechende Gruppe mit dem Erstellen der benötigten Javaklassen. Die Gruppe war ebenfalls dafür zuständig, die während des Semesters auftauchenden Fehler in den Klassen zu beheben sowie weitere Anpassungen, Änderungen und Features umzusetzen.

Die Gruppe, welche für die Erstellung der Prozesse zuständig war, teilte sich ein weiteres Mal auf, da sich nach dem Erstellen der ersten Prozesse die Frage nach der Kommunikation zwischen Prozessen und Server stellte. Hierunter fielen Aufgaben wie die Kommunikation über JSON-Nachrichten und die Antwort der Prozesse auf Anfragen der GUI. In Abschnitt 5.4, Seite 68, wird näher auf die Umsetzung der Serverseite eingegangen.

Zum Ende des Semesters wurde der Wunsch der Projektgruppenbetreuer geäußert Tests durchzuführen, da diese unter anderen Gegebenheiten essentiell für die Entwicklung von Prozessen und Webanwendungen sind. Dabei führte die Projektgruppe neben Unittests der Java-Klassen und der Activiti-Prozesse auch Tests der GUI mit Hilfe des Testframeworks Selenium durch, worauf in Abschnitt 5.5, Seite 85, näher eingegangen wird.

Während das Spiel zunächst nur einen Einzelspielermodus besessen hat, dort aber zunehmend fehlerfrei und besser zu spielen war, stellte sich der Projektgruppe am Ende des Semesters die Frage, welche Anforderung sie erfüllen sollte. Sollte nun ein KI-Spieler erstellt werden oder sollte man die Möglichkeit des Spielens mit anderen Spielern umsetzen.

Die PG entschied sich für die Umsetzung eines KI Spielers, worauf näher in Abschnitt 5.4.8 eingegangen wird.

5.3. Clientseite (Graphical User Interface)

Die GUI umfasst alles, was der Benutzer sieht und was er anklicken kann. Zwischen der GUI und der serverseitigen Software werden Nachrichten über eine REST-Schnittstelle ausgetauscht, was in Abschnitt 5.4.1, Seite 68, näher erläutert wird.

5.3.1. Aufbau

Die GUI besteht aus vielen Komponenten, die wir auf Basis der verwendeten Technologien gliedern:

- Basis-HTML,
- Coffeescript und
- LESS.

5.3.2. Basis-HTML

Der HTML-Code enthält das grundlegende Layout und initialisiert die Browser-Umgebung, z. B. indem die benötigten Dateien geladen werden. Hier wird auf mehrere Frameworks gesetzt.

Das HTML5-Grundgerüst entspricht der HTML5Boilerplate[1]-Vorlage. Diese ermöglicht die Verwendung aktueller HTML5-Tags in vielen Browsern und bringt von Haus aus jQuery[2] (ein Javascript-Framework) mit. jQuery okkupiert den Funktions-Shortcut *$* und erlaubt darüber eine Browser-übergreifende DOM-Manipulation. Die Funktionalität integriert sich nahtlos in Coffeescript (siehe folgender Abschnitt 5.3.3), sodass auf das Beispiel in Listing 5.3, Seite 66, für die exemplarische Verwendung von jQuery verwiesen wird. Darüber hinaus

[1] http://html5boilerplate.com/
[2] http://jquery.com/

wird noch jQuery UI[3] und der Twitter Bootstrapper[4] eingesetzt. Beides definiert viele grafische Elemente (z. B. Hinweis-Fenster, Tabs, Hinweis-Meldungen, kleine Icons, . . .) vor, die man dann ohne viel Aufwand verwenden kann.

Das HTML-Grundgerüst enthält bereits die Spielerinformationen auf der linken Seite, da diese beim Laden dann automatisch durch den Twitter Bootstrapper initialisiert werden. Auf viele nachträgliche Änderungen wird bei den Spielerinformationen verzichtet, da das sogenannte Accordion dann erneut initialisiert werden müsste.

5.3.3. Coffeescript

Coffeescript[5] ist eine Programmiersprache, die einen vereinfachten Umgang mit Javascript ermöglicht. Unter anderem bringt Coffeescript eine objektorientierte Sicht mit, die den meisten Programmierern vermutlich geläufiger ist als die Javascript-eigene Prototyp-basierte Sicht. Auch lässt sich der meiste Javascript-Code kompakter formulieren, da Einrückungen Semantik vorgeben und Klammern weggelassen werden können. Es werden einige Hilfsmethoden definiert und Coffeescript kümmert sich um das Scoping von Variablen. Coffeescript-Dateien lassen sich zu Javascript-Dateien kompilieren und sind somit von jedem Browser ausführbar.

Ein Beispiel für die semantische Bedeutung von Einrückungen gibt Listing 5.1. Dort wird ein verschachteltes Array/Objekt mithilfe von Einrückungen definiert. Zu Javascript kompiliert ergibt das Quellcode wie in Listing 5.2 dargelegt.

In Listing 5.3 wird exemplarisch die Logger-Klasse (die im Folgenden noch vorgestellt wird) gezeigt. Man sieht exemplarisch die Klassen-Definition mit Konstruktur, die abkürzende Schreibweise @variable für Attribute, die nahtlose Integration mit der jQuery-Funktion *$*, sowie die Semantik der Einrückung bei Klassen, Funktionen und Kontrollflussoperatoren.

MVC-Architektur

Die Coffeescript-Schicht unterteilt sich in unterschiedliche Klassen, welche in Abbildung 5.1 gezeigt werden. Zusammen mit dem Basis-HTML, lässt sich ein grobes MVC-Pattern erkennen.

Einerseits gibt es die Klasse *GUI*, welche die direkte Interaktion mit dem HTML-Quellcode übernimmt. Zum einen sind dort Methoden zum Neuzeichnen von Straßen und Spielfiguren enthalten (*Redraw*-Methoden), zum anderen aber

[3] http://jqueryui.com/
[4] http://twitter.github.com/bootstrap/index.html
[5] http://coffeescript.org/

```
vorgabe =
  'verbunde':
    'lila': [1, 3]
    'lightblue': [6, 8, 9]
    'violet': [11, 13, 14]
    'orange': [16, 18, 19]
    'red': [21, 23, 24]
    'yellow': [26, 27, 29]
    'green': [31, 32, 34]
    'blue': [37, 39]
  'feldeigenschaften':
    0:
      name: 'Los'
    1:
      name: 'Badstraße'
      preis: 60
```

Listing 5.1: Array/Objekt-Definition in Coffeescript.

```
  vorgabe = {
    'verbunde': {
      'lila': [1, 3],
      'lightblue': [6, 8, 9],
      'violet': [11, 13, 14],
      'orange': [16, 18, 19],
      'red': [21, 23, 24],
      'yellow': [26, 27, 29],
      'green': [31, 32, 34],
      'blue': [37, 39]
    },
    'feldeigenschaften': {
      0: {
        name: 'Los'
      },
      1: {
        name: 'Badstraße',
        preis: 60
      }
    }
  };
```

Listing 5.2: Compilat von Listing 5.1.

```
class Logger
  constructor: () ->
    @queue = []
    ###
    zwei Timer, einmal der Standard-Poll und einmal 3 Sekunden
    warten, damit man eine Log-Message auch lesen kann.
    ###
    @wait = false
    @timer = setTimeout (=> @output()), (1*1000)


  logMessage: (msg) ->
    $('#logger').append(msg + '<br>')
    $('.navbar-inner').effect('highlight', {color: "#0088CC"}, 3000)
    @queue.push msg
    if !@wait
      clearTimeout(@timer)
      @output()
  logError: (msg) ->
    @logMessage '<span style="color:red; font-weight: bold;">' + msg + '</span>'

  output: () ->
    @wait = false
    @timer = false

    if @queue.length
      $('#logbar').html @queue.shift()
      @wait = setTimeout (=> @output()), (3*1000)
    else
      @trigger()
      @timer = setTimeout (=> @output()), (1*1000)


  bind : (fn) ->
    @_events ||= []
    @_events.push(fn)

    if !@queue.length
      @trigger()

  trigger: (args...) ->
    if @_events
      for callback in @_events
        callback.apply(this, args)
      @_events = []
```

Listing 5.3: Beispielhafte Logger-Klasse in Coffeescript.

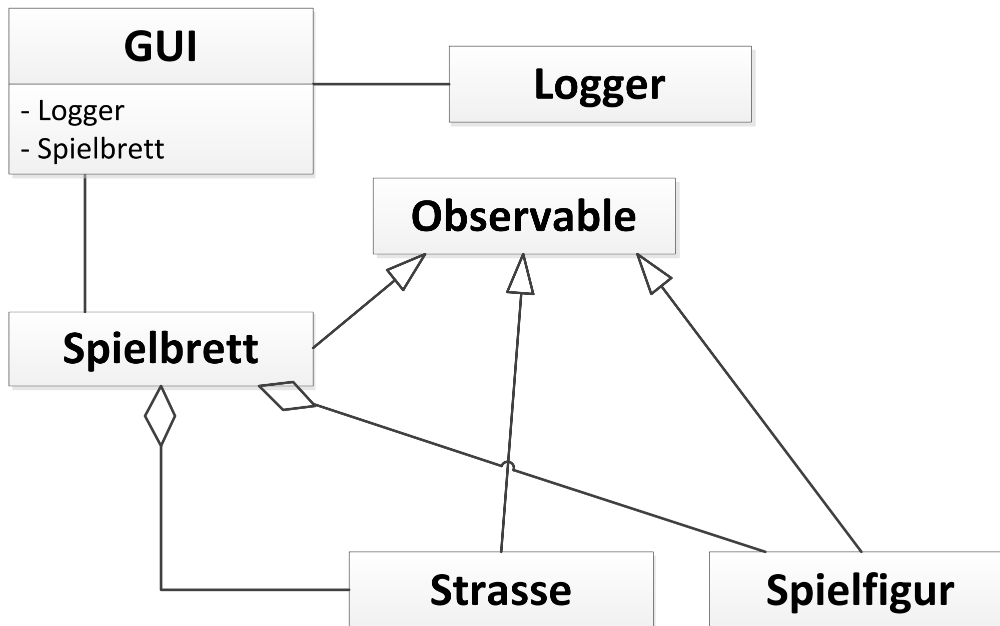

Abbildung 5.1.: Bild der GUI-Architektur.

auch Event-Handler und die Interaktion mit dem Server. Die GUI kommuniziert mit dem Server über eine REST-Schnittstelle. Dafür schickt sie regelmäßig AJAX-Aufrufe an den Server, der mithilfe von JSON-Nachrichten antwortet, was der Spieler als nächstes zu tun hat (siehe Abschnitt 5.4.1, Seite 68). Diese Nachrichten werden von der GUI-Klasse ausgewertet.

Andererseits gibt es reine Datenhaltungsklassen (*Spielbrett*, *Strasse*, *Spielfigur*). Sie enthalten eine Kopie der Spieldaten vom Server. Diese sind mithilfe des Observer-Patterns an *Redraw*-Methoden der GUI-Klasse angebunden – sobald Attribute der Datenhaltungsklassen verändert werden, wird eine Aktualisierung des HTML-Codes angestoßen.

Die *Logger*-Klasse zeigt Nachrichten an den Benutzer an. Da in schneller Abfolge mehrere Log-Nachrichten vom Server abgerufen werden können, der Benutzer jedoch ein paar Sekunden braucht um eine neue Nachricht zu lesen, werden diese innerhalb der *Logger*-Klasse in einer Schlange verwaltet, und mit kurzen Pausen dazwischen ausgegeben.

5.3.4. LESS

LESS[6] erweitert CSS um variables Verhalten. Unter anderem werden Variablen, Mix-Ins und ein einfaches Funktions-Sprachkonstrukt angeboten. Sehr entgegen kommt dem Entwickler auch die Möglichkeit CSS zu verschachteln. LESS-Dateien lassen sich zu normalen CSS-Dateien kompilieren, sodass auch diese von jedem Browser genutzt werden können.

Exemplarisch sei auf Listing 5.4 verwiesen. Man sieht einerseits ganz oben die Definition von Variablen, welche im Folgenden verwendet werden, andererseits die Verschachtelung von Selektoren: Entweder als Vater-Kind-Beziehung (normal verschachtelt) oder als Verfeinerung der bereits selektierten Teilmenge (der Selektor wird mit einem & voran gesondert markiert), z. B. in Zeile 22.

Das Compilat ist in Listing 5.5 zu sehen.

5.4. Serverseite

5.4.1. Aufbau

Der Aufbau der Serverseite setzt sich aus einer Vielzahl von Komponenten zusammen. Die Grundlage des Programms ist das Java Runtime Environment (JRE) sowie der HTTP-Server bzw. Servlet Container Jetty, welcher letztenendes die Anwendung ausführt. Weiterhin wird für die Persistenz eine SQL-Datenbank verwendet. Als Prozessengine wird Activiti[7] genutzt, welche die Basis für sämtliche Prozesse sowie deren Ausführung und Persistenz ist. Zur Transformation von Java-Klassen ins JSON-Format sowie die umgekehrte Richtung wird XStream[8] genutzt. XStream ist eine freie Bibliothek zur (De)Serialisierung von Java-Objekten zu XML und JSON, wobei die Ausgabe mittels Annotationen konfigurierbar ist. Für die Kommunikation zwischen Client und Server werden RESTful-Web-Services mittels Jersey (JAX-RS) verwendet. Diese Auf die Funktionsweise der Kommunikation wird in den Abschnitten 5.4.6 und 5.4.7 im Detail eingegangen.

Das Programm setzt sich aus mehreren Komponenten zusammen. Die Basis des Spiels bilden das Datenmodell und die Spiellogik. Das Datenmodell wird ausschließlich für die Struktur genutzt und besitzt keine nennenswerte Funktionalität. Diese wird stattdessen im Spiellogik-Modul umgesetzt, welches seinerseits auf dem Datenmodell arbeitet. Weiterhin gibt es ein Modul, welches Funktionalität im Zusammenhang mit den Activiti-Prozessen, d. h. in Form von

[6] http://lesscss.org/
[7] http://activiti.org/
[8] http://xstream.codehaus.org/

```
@spielfeldBreite: 41px;
@spielfeldHoehe: 91px;

.feldLinks {
  .spielfeld {
    left: 0px;
  }
  .streetcolor {
    right: 3px;
    top: 0;
  }
  .houses {
    right: 0;
    top: 0;
    width: 2px;
    height: @spielfeldHoehe;
    border-left: 1px solid white;
  }
  .houses div {
    width: 2px;
    height: @spielfeldBreite / 4;
    &.eins {
      right: 0;
      top: 0;
    }
    &.zwei {
      top: @spielfeldBreite / 4;
    }
    &.drei {
      top: @spielfeldBreite / 4 * 2;
    }
    &.vier {
      top: @spielfeldBreite / 4 * 3;
    }
  }
}
```

Listing 5.4: Beispieldefinition in LESS.

```
.feldLinks .spielfeld {
  left: 0px;
}
.feldLinks .streetcolor {
  right: 3px;
  top: 0;
}
.feldLinks .houses {
  right: 0;
  top: 0;
  width: 2px;
  height: 91px;
  border-left: 1px solid white;
}
.feldLinks .houses div {
  width: 2px;
  height: 10.25px;
}
.feldLinks .houses div.eins {
  right: 0;
  top: 0;
}
.feldLinks .houses div.zwei {
  top: 10.25px;
}
.feldLinks .houses div.drei {
  top: 20.5px;
}
.feldLinks .houses div.vier {
  top: 30.75px;
}
```

Listing 5.5: Compilat von Listing 5.4.

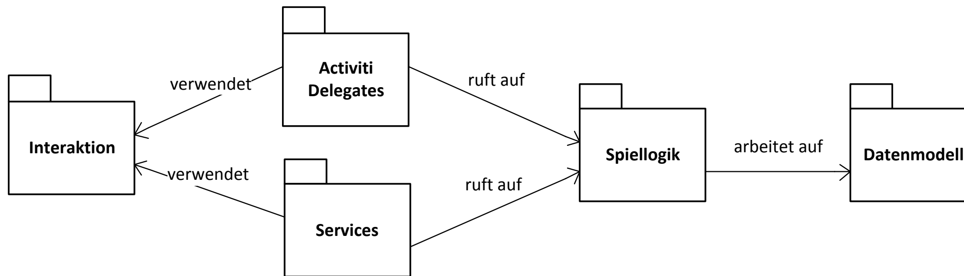

Abbildung 5.2.: Schematische Darstellung der Module des Programms. Aktionen im Programm werden von den Activiti Delegates sowie der Services durch Aufrufe der Spiellogik initiiert. Ebenso werden von dort Nachrichten für Clients in die Messagequeue gelegt. Die Spiellogik operiert auf dem Datenmodell.

Delegates, umsetzt. Die für die Kommunikation verantwortlichen Module sind das Interaktions- und das Service-Modul. Während ersteres eine Messagequeue und eine Menge an Nachrichten beinhaltet, die zwischen Client und Server ausgetauscht werden, beinhaltet letzteres Methoden, die über Web-Services erreicht werden können.

Die Verbindung zwischen den einzelnen Modulen wird schematisch in Abbildung 5.2 dargestellt.

5.4.2. Projektkonfiguration

Dank der ausführlichen Dokumentation von Heroku für Java [Ada11], an der wir uns während der Programmierphase orientiert haben, lies sich unsere Java-Webanwendung ohne größere Schwierigkeiten auf Heroku ausführen. Bevor dies jedoch möglich ist, sind einige Konfigurationsschritte nötig:

Nachdem ein Heroku-Account angelegt und der Heroku-Toolbelt (enthält CLI-Client, Foreman und Git) herunterladen und installiert wurden, wird die gewünschte Ordnerstruktur des auf Java und Maven basierenden Web-Projektes erstellt. Alle zur Ausführung des Projektes notwendigen Java-Bibliotheken werden als Abhängigkeiten zur pom.xml-Datei von Maven (siehe Abschnitt 3.2, Seite 11) hinzugefügt. In der sogenannten Procfile-Datei, die in dem Grundverzeichnis des Projektes angelegt wird, werden die gewünschten Einstellungen an der Ausführungsumgebung festgelegt. In unserem Fall sieht die Procfile-Datei wie in Listing 5.6 aus.

Das erste Teil des Befehls weist auf den gewünschten Container hin, in diesem

```
web: java $JAVA_OPTS -jar target/dependency/jetty-runner.jar --port
    $PORT target/*.war
```

Listing 5.6: Der Inhalt der Procfile-Datei.

```
<bean class="java.net.URI" id="dbUrl">
  <constructor-arg value="${DATABASE_URL}" />
</bean>
<bean class="org.apache.commons.dbcp.BasicDataSource" destroy-
    method="close" id="dataSource">
  <property name="driverClassName" value="org.postgresql.Driver"
      />
  <property name="url" value="#{ 'jdbc:postgresql://' + @dbUrl.
      getHost() + @dbUrl.getPath() }" />
  <property name="username" value="#{ @dbUrl.getUserInfo().split
      (':')[0] }" />
  <property name="password" value="#{ @dbUrl.getUserInfo().split
      (':')[1] }" />
  <property name="defaultAutoCommit" value="false" />
</bean>
```

Listing 5.7: Ein Auschnitt aus der Kontextkonfiguration-Datei zum Einlesen der Datenbankparameter von Heroku.

Fall ist es ein eingebetteter Jetty-Servlet-Container. Der zweite Teil zeigt auf die Projektdataien, die auf dem Container auszuführen sind, wobei *$JAVA_OPTS* und *$PORT* Systemvariablen der Heroku-Plattform sind.

Ein weiterer wichtiger Schritt ist das Laden der Prozess-Engine, die eine vorkonfigurierte Datenbank benötigt. Um die von Heroku kostenfrei zur Verfügung gestellte PostgreSQL-Datenbank nutzen zu können, wird zuerst das Heroku Postgres-Add-on[9] zur Anwendung hinzugefügt. Dann werden von der Anwendung die entsprechenden Datenbankparameter, die von der Heroku-Umgebung zu Verfügung gestellt wurden, angesprochen und gesetzt (siehe Abschnitt 4.3.3, Seite 55). In unserem Projekt werden die Engine, Datenbankkonfiguration und der Ladevorgang der Prozessdefinitionen in die Datenbank mit Hilfe des Kontextes der Webanwendung ausgeführt (siehe die vollständige `applicationContext.wml`-Datei in Listing A.5, Seite 111).

Der Auschnitt aus der Kontextkonfigurations-Datei in Listing 5.7 liest die Datenbankparameter von den Variablen der Heroku-Umgebung aus, während Listing 5.8 den Zugriff auf die Datenbank ermöglicht und die Prozess-Engine

[9] https://devcenter.heroku.com/articles/heroku-postgresql

```
1  <bean id="transactionManager"
2    class="org.springframework.jdbc.datasource.
        DataSourceTransactionManager">
3    <property name="dataSource" ref="dataSource" />
4  </bean>
5
6  <bean id="processEngineConfiguration" class="org.activiti.spring.
      SpringProcessEngineConfiguration">
7    <property name="dataSource" ref="dataSource" />
8    <property name="transactionManager" ref="transactionManager" />
9    <property name="databaseSchemaUpdate" value="true" />
10   <property name="jobExecutorActivate" value="true" />
11 </bean>
12
13 <bean id="processEngine" class="org.activiti.spring.
      ProcessEngineFactoryBean"
14   destroy-method="destroy">
15   <property name="processEngineConfiguration" ref="
        processEngineConfiguration" />
16 </bean>
```

Listing 5.8: Ein Auschnitt aus der Kontextkonfiguration-Datei, der den Zugriff auf die Datenbank erlaubt und die Prozess-Engine der Anwendung zur Verfügung stellt.

```
<bean id="activitiRule" class="org.activiti.engine.test.
    ActivitiRule">
  <property name="processEngine" ref="processEngine" />
</bean>
```

Listing 5.9: Ein Auschnitt aus der Kontextkonfiguration-Datei, der das Testen von Prozessen erlaubt.

sowohl instanziiert als auch für die Anwendung konfiguriert.

Weiterhin kann man mit Hilfe des Codes in Listing 5.9 die Activiti-Prozesse testen, was detaillierter in Abschnitt 5.5.1, Seite 85, erläutert wird.

5.4.3. Datenmodell

Da die Stammdaten eines Projektes ein wichtiger Baustein dessen Entwicklung sind, wird in diesem Kapitel auf das Datenmodell des Monopoly-Spieles eingegangen und es kurz erläutert. In dem Klassendiagramm (siehe Abbildung A.1, Seite 114) ist zu sehen, welche Daten in der Datenbank dauerhaft gespeichert werden und wie diese miteinander in Verbindung stehen.

Das Spiel besteht (siehe die *Game* Klasse in der Abbildung A.1, Seite 114) aus einem Spielbrett (siehe *GameBoard* Klasse), das Felder (siehe die *Space* Klasse) enthält. Die Felder können Grundstücke, Straßen, Werke oder Bahnhöfe sein.

Weiterhin gibt es wie beim klassischen Spielfeld Eckfelder („LOS"-Feld, „Gefängnis"-Feld, „Gehe ins Gefängnis"-Feld und „Frei Parken"-Feld), Steuerfelder sowie Kartenfelder, bei denen der Spieler eine Karte aus dem entsprechenden Kartenstapel ziehen und die darauf angewiesene Aktion ausführen muss.

Außerdem enthält das Spiel zwei Kartenstapel, die die gerade beschriebenen Aktionskarten für die Felder bereitstellen. Dabei gibt es verschiedene Ereignisse und Aktionstypen, wie sie von Monopoly bekannt sind.

Beispielsweise kann der Spieler dazu angehalten werden alle seine Häuser und Hotels zu renovieren (siehe *RenovationCard* Klasse), bekommt eine Gefängnisfreikarte (siehe *JailFreeCard* Klasse) oder muss sich auf ein anderes Feld bewegen.

Besonders hervorzuheben an unserem Datenmodell ist die Erzeugung der Spielfelder und Spielkarten. Während der Initialisierung eines Spieles werden diese durch die in einer `*.properties`-Datei gespeicherten Informationen erzeugt. Dadurch wird die Erweiterbarkeit des Spieles gewährleistet. So ist sowohl das Erzeugen einer unterschiedlichen Anzahl von Feldern und Karten, als auch die Unterstützung verschiedener Sprachen, Namen, Beschreibungen etc. möglich.

5.4.4. Spiellogik

Zwar enthält das Datenmodell einige grundlegende Funktionen, aber die gesamte Logik des Spieles wird durch die Activiti-Prozesse gesteuert. Wie in der Abbildung 5.3 zu sehen ist, besteht der Hauptprozess aus einer Sequenz von Service Tasks, Signalen, Unterprozessen und Verzweigungen, wobei jeder Service Task eine Schnittstelle zu einer Delegate-Klasse (siehe Abschnitt 5.4.5) und jedes Signal eine Schnittstelle zu einem Web Service (siehe Abschnitt 5.4.6) ist.

Am Anfang des Prozesses (siehe in der Abbildung 5.3 das erste Task) werden zuerst die nötigen Variablen, die Anzahl der Spieler, die Reihenfolge in der sie spielen werden, der Name des realen Spieler, so wie der Geldbetrag, den die Spieler am Anfang des Spieles zur Verfügung haben etc. , initialisiert. Anschließend wird in dem nächsten Service Task, welcher auf dem Bild dargestellt wird, das Spiel gestartet, indem die Spieler auf das erste Spielfeld gesetzt werden und der aktuelle Spieler würfeln darf. Die entsprechende Nachricht wird in dem „Roll Dice“- Service Task generiert. Nachdem der aktueller Spieler gewürfelt hat, geht der Prozessablauf weiter. Anhand des Würfelergebnisses, wird entschieden ob er sich bewegen darf, die entsprechende Nachricht generiert und an den Spieler geschickt. Wenn der Spieler sich bewegen darf, wird er durch die Logik in dem „Move Player“-Service Task bewegt und anschließend wird überprüft, ob er über das „LOS“-Spielfeld gekommen ist, damit er den „Über LOS“-Betrag gutgeschrieben bekommt. Wenn der Spieler im Gefängnis war und keinen Pasch gewürfelt hat und noch nicht drei Runden im Gefängnis stand, darf er sich nach dem Würfeln nicht mehr bewegen. In diesem Fall geht der Prozessablauf durch die „player cannot move“- Sequenz. Hier wird geprüft, ob es aktuell einen Gewinner gibt und falls dies nicht der Fall ist, wird der nächste Spieler als aktueller Spieler gesetzt.

Durch die sogenannten Call-Activities (siehe die fettgedruckten Vierecksymbole in der Abbildung 5.3) werden die Unterprozesse instantiiert und aufgerufen. Hier müssen sämtliche Prozessvariablen, welche im Unterprozess verwendet werden sollen, in den Properties angegeben werden. Ansonsten werden die Werte nicht an den Unterprozess weitergereicht. Nachdem ein Unterprozess ausgeführt wurde, wird die entsprechende Instanz wieder gelöscht.

Der „playerAction“-Unterprozess (siehe Abbildung 5.4) enthält Service-Tasks für jeden möglichen Spielfeldtyp, welcher in der Spielfeldlogik implementiert ist. So ist z. B. die Logik der Kartenfelder, also ein Feld, auf dem der Spieler aus einer der zwei Kartenstapeln des Spieles eine Karte ziehen muss, in der Abbildung 5.6 dargestellt. Nachdem die Logik des Feldes, auf dem der Spieler sich befindet, ausgeführt ist, stehen dem Spieler durch den „regularAction“-Unterprozess (siehe Abbildung 5.5) verschiedene, sich jede Runde wiederholende,

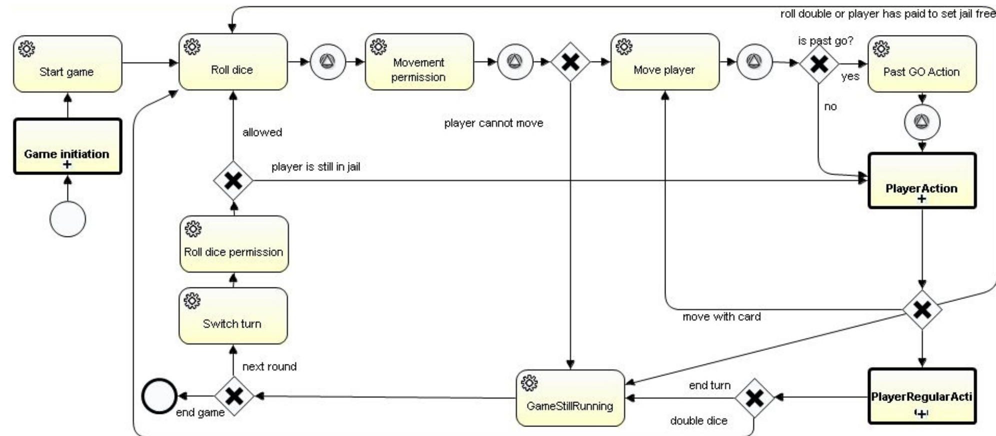

Abbildung 5.3.: Der Monopoly-Spiel Hauptprozess.

optionale Aktionen zur Auswahl. Der Spieler kann seinen Zug beenden, oder falls möglich Grundstücke tauschen, Gebäuden kaufen oder verkaufen, sowie Hypotheken aufnehmen oder abbezahlen.

Im Allgemeinen, nachdem der Benutzer zum ersten Mal über die entsprechende URI das Spiel aufgerufen hat, wird eine Instanz des dargestellten Prozesses erzeugt und diese solange schrittweise ausgeführt, bis der Prozess einen Signal-Zustand erreicht hat. In diesem Fall wurde eine Nachricht (siehe Abschnitt 5.4.7) über einen entsprechenden Web Service an den Spieler geschickt und der Process wartet auf eine Antwort. Sobald der Spieler geantwortet hat (also die GUI), geht der Prozessablauf weiter und die GUI im Falle von Veränderung des game-Objektes entsprechend aktualisiert. Falls der aktuelle Spieler ein KI-Spieler (siehe Abschnitt 5.4.8) ist, zeigt die GUI dem realen Spieler die aktuelle Nachricht an, die der von der KI ausgeführten Aktion entspricht.

Während der Modellierung der Prozesse wurde die Erfahrung gemacht, dass die Konstruktion eines auf Prozessen basierten Spielablaufs relativ komplex ist. Einerseits wird für jede Spielfunktion oder jeden Zugriff auf Prozess-Variablen ein Aufruf der entsprechenden Delegate-Klasse benötigt, andererseits ist die Fehlererkennung und -behandlung sehr mühsam, da die Fehler in der Prozesslogik erst nach dem Ausführen der Prozesse erkannt werden. Während einer Prozessausführung können Variablen gesetzt und gelesen werden, wobei sie nur in dem entsprechenden Prozess und seinen Unterprozesse sichtbar sind (siehe oben). Wenn eine Variable in einem Prozess benötigt wird und sie nicht richtig gesetzt ist, wird das erst erkannt, nachdem der Prozess versucht hat, auf sie zuzugreifen.

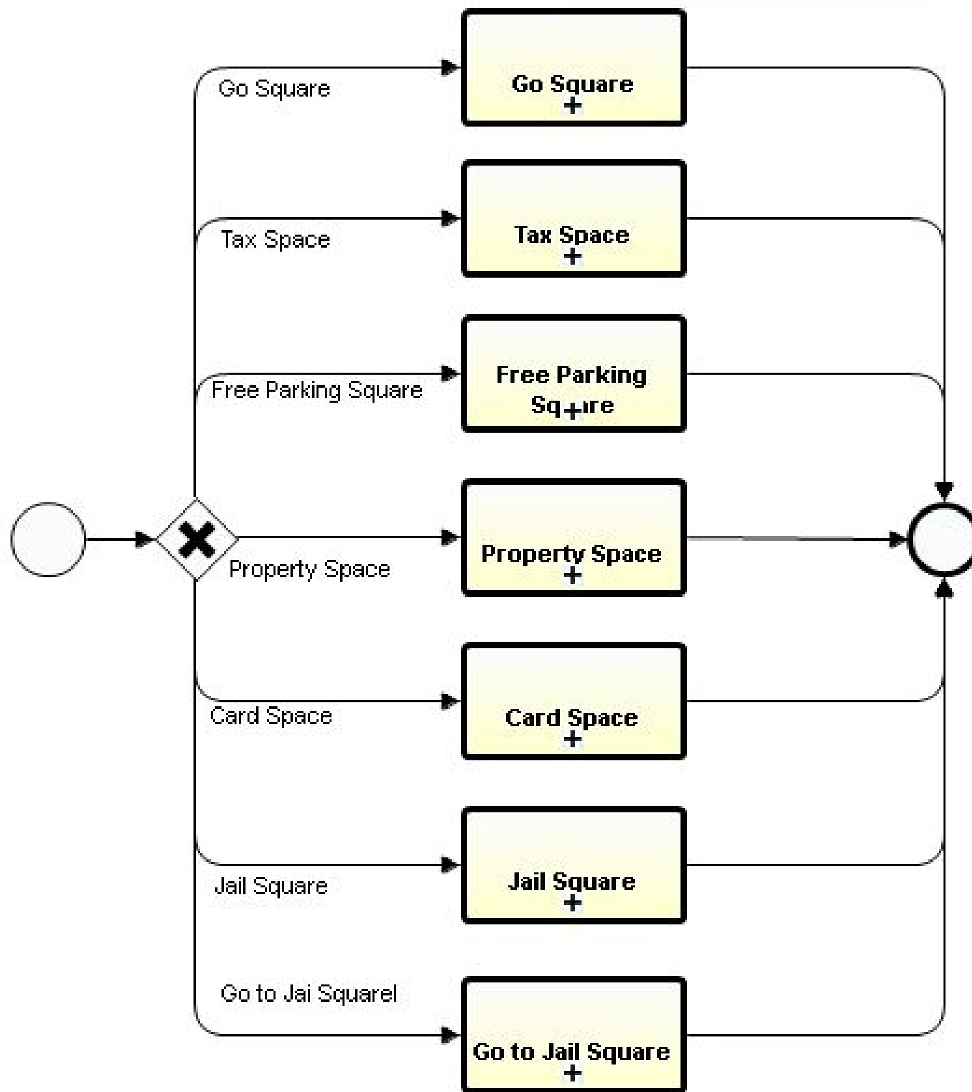

Abbildung 5.4.: Der playerAction-Unterprozess des Hauptprozesses.

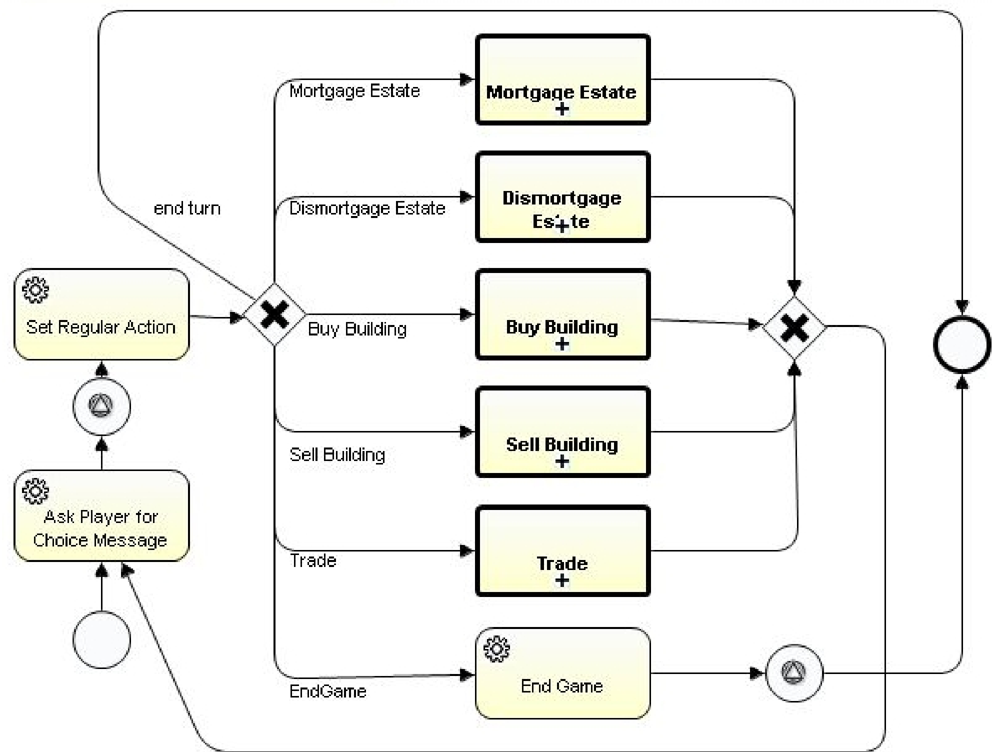

Abbildung 5.5.: Der regularAction-Unterprozess des Hauptprozesses.

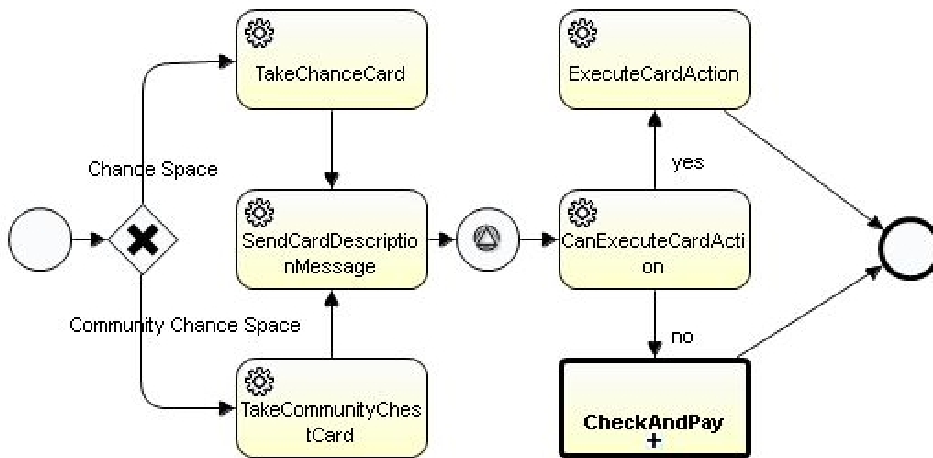

Abbildung 5.6.: Der Card Space-Unterprozess des regularAction-Prozesses.

Zum Abschluss lässt sich sagen, dass sich einige Schwierigkeit gezeigt haben, obwohl die Dokumentation [Act12a] und die Beispiele [Act12b] von Activiti sehr hilfreich waren. Sowohl Tippfehler in den Prozessdefinitionen, als auch Fehler in der Logik der Prozesse, konnten erst zur Ausführungszeit gefunden werden. Weiterhin ist deren Behandlung sehr zeitaufwändig gewesen, was sich für uns als ein Nachteil der prozessbasierten Programmierung erwiesen hat.

5.4.5. Delegates

Delegates sind Java-Klassen, die Entwicklern die Möglichkeit geben eigene Logik innerhalb von Prozessen ausführen zu können. In unseren Activiti-Prozessen befinden sich auf jedem Pfad Service Tasks, welche jeweils eine Delegate-Klasse aufrufen.

Die Klassen erhalten ihre Funktionalität durch die Implementierung des Interface JavaDelegate. Diese bestehen dann aus der Methode execute, welche eine Execution (also eine Ausführung des Prozesses) von der Process-Engine übergeben bekommt und in die die eigentliche Logik geschrieben wird.

Zunächst hat man die Möglichkeit Logs auszugeben um später kontrollieren zu können, was wo in der Logik passiert ist (siehe Listing 5.10).

In jeder Delegate-Klasse lädt man zunächst das Game-Objekt aus der Prozessvariablen game. Anschließend werden, falls nötig, weitere Prozessvariablen geladen, welche im aktuellen (Unter-)Prozess definiert und gesetzt sein müssen.

```
java.util.logging.Logger.getLogger(
    this.getClass().getCanonicalName()
  ).info("Starting");
```

Listing 5.10: Logging in Delegates.

```
Game game = ((Game) (execution.getVariable("game")));
Boolean hasPaid = (Boolean)execution.getVariable("PlayerHasPaid");
```

Listing 5.11: Laden der Prozessvariablen.

Dies kann einerseits durch das Deklarieren direkt im Prozess oder bei Aufrufen von Unterprozessen durch Call-Activities, über das Setzen der benötigten Variablen als Input- bzw. Outputparameter (siehe Listing 5.11).

Nun kann man Methoden auf dem Game-Objekt ausführen und die erhaltenen Daten verarbeiten, ändern und/oder über MessageCentral Nachrichten für die GUI vorbereiten. Wichtig ist hierbei, dass alle Variablen, die verändert wurden, am Ende der Delegate-Klasse auch wieder in dem Prozess gespeichert werden. Das Game-Objekt ist hier besonders wichtig, da es sozusagen das komplette Spiel darstellt, auf dem gearbeitet wird und die vollzogenen Änderungen nicht verloren gehen dürfen (siehe Listing 5.12).

5.4.6. Web-Services

Zur Abwicklung der Kommunikation zwischen Client und Server werden Web-Services auf der Serverseite zur Verfügung gestellt. Zur Umsetzung der Services nutzen wir Jersey, die Referenzimplementierung von JAX-RS (Java API for

```
if (hasPaid) {
  java.util.logging.Logger.getLogger(
      this.getClass().getCanonicalName()
    ).info("Spieler hat Grundstück bezahlt");
  game.getCurrentPlayer().buyProperty();
  MessageCentral mc = game.getMessageCentral();
  mc.addMessageForPlayer(game.getCurrentPlayer().getSessionID(),
    InteractionMessageFactory.createNotificationPropertyBought(game
        ));
}
```

Listing 5.12: Aktionen in der Delegate.

RESTful Web Services)[10]. JAX-RS verfolgt das Ziel, Entwicklern eine Abstraktion für die Details des HTTP-Protokolls zu bieten und somit eine einfache und robuste Methode zur deklarativen Definition von Web-Services zu bieten. Diese Web-Services arbeiten, wie bereits der Name erahnen lässt, nach den Prinzipien des Representational State Transfer (REST)[11].

Der Zugriff auf die Services erfolgt über ein URI (Unified Resource Identifier) in Kombination mit einer HTTP-Request-Methode (z. B. GET, PUT, POST). Um eine Java-Methode mittels Jersey als Service bereitzustellen, müssen daher der Pfad des Service und die zugeordnete HTTP-Methode über Annotationen festgelegt werden. Wird also beispielsweise eine Methode mit den Annotationen `@GET`, `@Path("/example/id")` sowie `@Produces(application/json)` versehen, so ist der Service bei Benutzung der GET-Methode über

`http://exampleserver.com/example/1234`

erreichbar. Weiterhin ist deklariert, dass das Format der Ausgabe JSON ist und es wird dem Service „1234“ an Stelle des Platzhalters `id` übergeben. An diesem Beispiel sieht man, dass die Bereitstellung eines Service über die Annotationen einfach und flexibel durchzuführen ist.

Die Services bilden die Schnittstelle zwischen Client und Server in der Anwendung, d. h. für jede Interaktion die stattfindet wird ein Service genutzt. Im Normalfall ruft der Client einen Service auf und gibt die vom Service benötigten Informationen mit der Anfrage weiter. Der Server verarbeitet die Anfrage durch Auswertung der Informationen und Ausführen der relevanten Aktionen und gibt danach eine Antwort an den Client zurück. Diese Antwort beinhaltet einen HTTP-Statuscode und je nach Service eine Menge an Daten im JSON-Format. Die gesendeten JSON-Daten enthalten z. B. Daten über den Zustand des Spiels oder Nachrichten, die eine Antwort des Clients und damit den Aufruf eines weiteren Service erfordern. Der HTTP-Statuscode wird dazu verwendet, dem Client mitzuteilen ob und wie die Verarbeitung seiner Anfrage verlaufen ist. Im Falle einer erfolgreichen Anfrage wird der Statuscode „200 OK“ zurückgegeben. Sollte die Anfrage z. B. weniger Daten enthalten als nötig, so würde ein „400 Bad Request“ zurückgegeben werden und der Client könnte darauf reagieren und versuchen die Anfrage mit den vollständigen Daten erneut durchzuführen.

5.4.7. Nachrichten

Die Kommunikation zwischen Client und Server findet mittels Nachrichten im JSON-Format statt. Services und Activiti-Delegates erstellen und verarbeiten

[10] http://jcp.org/en/jsr/detail?id=311
[11] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Nachrichten und stellen sie mittels einer MessageQueue für den Client bereit. Der Client ruft die für ihn vorhandenen Nachrichten per Anfrage an einen Service ab. Die Nachrichten entsprechen einer Java-Klasse, welche mittels XStream zu JSON serialisiert bzw. aus JSON zu einem Nachrichtenobjekt deserialisiert werden. Es gibt verschiedene Arten von Nachrichten, welche abhängig vom Zweck der Nachricht verwendet werden. Beispiele für Nachrichtentypen sind:

- Notification: Eine Mitteilung an den Spieler, die nur bestätigt werden muss.
- Choice: Auswahl, die der Spieler mittels Knopfdruck trifft (z. B. Ja/Nein-Dialog).
- Selection: Auswahl, die mittels Wahlmöglichkeit sowie Knopfdruck gewählt wird (z. B. Radiobutton-Dialog).
- Input: Texteingabe (z. B. Name eingeben).
- Update: Der Zustand des Spiels hat sich geändert und muss abgerufen werden.
- GameData: Zustand des Spiels mit relevanten Daten für die GUI.

Zu beachten ist, dass die beiden letzten Nachrichtentypen nicht für die Kommunikation mit dem Spieler, sondern ausschließlich für den Client gedacht sind. Die Nachrichten werden also sowohl für technische als auch für Nachrichten an den Spieler genutzt.

Die Nachrichten haben ein Kopf- und Inhaltsfeld, eine Menge an Aktionen, eine Menge an Auswahlmöglichkeiten sowie eine Menge an Status- und Ergebnisvariablen. Kopf- und Inhaltsfeld dienen der textuellen Darstellung für den Spieler (z. B. in Form eines Dialogfensters), Aktionen korrespondieren zu Knöpfen, Auswahlmöglichkeiten zu Radiobuttons und die Ergebnisvariablen werden auf Clientseite entsprechend der Wahl des Spielers gesetzt. Die Statusvariablen ermöglichen zusätzlich den Transport von beliebigen Informationen. Zusätzlich zu den genannten Feldern hat eine Nachricht noch ein Antwort-URI, welche den URI des Service enthält, an den vom Client die mit Antwortvariablen gefüllte Nachricht gesendet werden soll. Die meisten der Felder in Nachrichten sind optional zu füllen, d. h. wenn der Nachrichtentyp ein Feld nicht braucht, so kann es einfach ungenutzt (und damit leer) bleiben.

Der Ablauf der Kommunikation besteht neben dem Spielbeitritt im wesentlichen aus zwei möglichen sich wiederholenden Abläufen (siehe Abbildung 5.7). Initiiert wird die Kommunikation durch den Client mit einer Anfrage nach

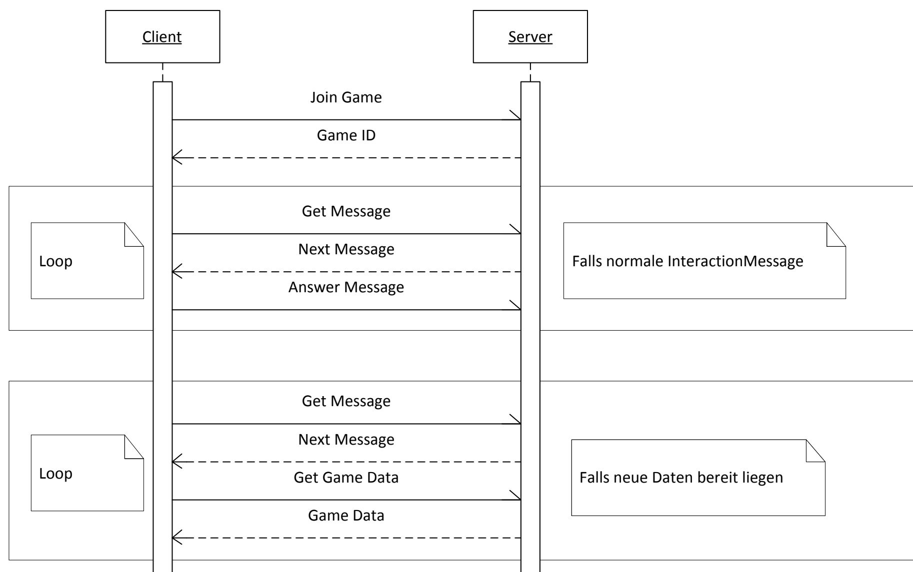

Abbildung 5.7.: Darstellung des Ablauf der Kommunikation zwischen Client und Server. Abgesehen von dem Spielbeitritt gibt es zwei unterschiedliche sich wiederholende Abläufe, welche abhängig von der vom Server erhaltenen Nachricht ist.

neuen Nachrichten. Im Fall von vorhandenen Nachrichten wird die nächste Nachricht als Antwort vom Server gesendet. Abhängig vom Nachrichtentyp (normale InteractionMessage oder Update) wird dann der Verlauf der weiteren Kommunikation bestimmt. Im Falle einer InteractionMessage wird die erhaltene Nachricht von der GUI umgesetzt und durch Interaktion des Nutzers entsprechend beantwortet. Im Gegensatz dazu wird bei einer Update-Nachricht eine weitere Anfrage an einen anderen Service auf der Serverseite gestellt, welche mit dem aktuellen Stand der Spieldaten beantwortet wird. Nachdem die Daten empfangen und verarbeitet wurden, wird der Ablauf durch den Abruf der Nachrichten erneut begonnen.

5.4.8. KI

Um die KI funktionsfähig umzusetzen, haben wir uns letztendlich für eine Variante entschieden, für welche die eigentlichen Aktiviti-Prozesse nun nicht

```
HashMap<String, String> resultVariables = msg.getResultVariables();
ArrayList<String> actions = msg.getActions();
ArrayList<String> choices = msg.getChoices();

String answer = null;
if (msg.getMessageID() != null) {
switch (msg.getMessageID()) {
```

Listing 5.13: Extrahieren der Variablen aus der Message.

```
case StreetToBuildOn:
answer = ""+ choices.get((int) (Math.random()*choices.size()));
break;
```

Listing 5.14: Generierung der Antwort bei Auswahl der zu bebauenden Straße.

verändert werden müssen, weil die KI direkt mit dem Server kommuniziert. Im Datenmodell gibt es die Klasse Player für reale Spieler. Diese enthält zunächst alle Informationen über einen Spieler wie Name, Position etc. und gleichzeitig Methoden um alle Aktionen, die der Spieler als Reaktion auf seine Position auf dem Spielfeld und seine Interaktion ausführen kann, umzusetzen.

Die Klasse für den Computerspieler, welche die KI abbildet, erbt nun von dieser Klasse Player und enthält zusätzlich eine Methode handleInteraction(Game game, InteractionMessage msg), welche die in den Delegates erstellte Nachricht an den Spieler verarbeitet. Geht die Nachricht im Falle eines normalen Spielers direkt an die GUI, wird sie hier sozusagen nicht abgeschickt und auf dem Server verarbeitet. Dafür haben wir den Nachrichten zunächst eine ID zugewiesen. Anhand dieser ID kann die Methode nun per switch-case Abfrage die Nachrichten unterscheiden, und entsprechend der Vorgaben im Code standardisierte Antworten auf die Nachrichten geben, welche dann den Aktionen in der GUI beim normalen Spieler entsprechen.

In Listing 5.13 sieht man zunächst wie dafür Result-, Action- und Choice-Variablen aus der Message extrahiert werden. Anschließend wird die MessageID überprüft. In den Folgenden Listings werden beispielhaft zwei Fälle erklärt.

Im Falle der Auswahl der zu bebauenden Straße ergibt sich die Antwort aus einer zufällig ausgewählten möglichen Antwort aus den vom Server generierten Choices (siehe Listing 5.14).

Geht es um den Kauf eines Grundstückes, wird über die von Player geerbte Methode canBuyProperty() zunächst abgefragt, ob der Spieler dieses Grundstück kaufen kann, und anschließend je nach Möglichkeit die Antwort ausgewählt

```
case BuyProperty:
if (this.canBuyProperty()) {
answer = "0";
} else {
answer = "1";
}
break;
```

Listing 5.15: Generierung der Antwort beim Grundstückkaufs.

```
resultVariables.put(answer, answer);
```

Listing 5.16: Antwort als ResultVariable zurückgeben.

(siehe Listing 5.15).

Am Ende wird der ResultVariable die Antwort hinzugefügt (siehe Listing 5.16) und der Server behandelt die Antwort wie eine normale, von der GUI erhaltene Antwort auf eine Nachricht.

Die Komplexität wurde hier bewusst niedrig gehalten, da es nicht darum ging eine möglichst aufwendige KI zu basteln, sondern einen vernünftigen Spielablauf mit mehreren Spielern zu gewährleisten. Durch die KI war es uns dann auch möglich beim Spielen selbst viele Testcases abzudecken und die resultierenden Fehler zu beheben.

5.5. Testing

5.5.1. Prozesse

Da Geschäftsprozesse ein wichtiger Bestandteil des Projekts sind, werden die einzelnen Module eines Prozesses auf mögliche Fehler getestet, um eine bessere Produktqualität zu erreichen. Geprüft werden die Prozesse anhand der Dokumentation [Act12c] von Activiti und mithilfe der JUnit Tests.

Um die Ausführungszeit zu verkürzen werden die Tests containerlos auf dem lokalen Rechner durchgeführt, was durch das Nutzen der Spring Bibliothek von Activiti ermöglicht wird. Zunächst bindet man eine Test-Engine in den Kontext der Anwendung ein (siehe die vollständige `applicationContext.xml`-Datei A.5, Seite 111), so dass diese dadurch dem Programm bekannt wird. Der Ausschnitt aus der Kontextkonfigurations-Datei (siehe Listing 5.9, Seite 74) zeigt, welche Test-Bibliotheken vor der Ausführung des Programms bekannt sind. Die Datenbankkonfiguration erfolgt so, wie in Kapitel 5.4.2, Seite 71,

```
<bean id="activitiRule" class="org.activiti.engine.test.
    ActivitiRule">
  <property name="processEngine" ref="processEngine" />
</bean>
```

Listing 5.17: Einbinden der Test Engine.

```
import org.activiti.engine.RuntimeService;
import org.activiti.engine.runtime.Execution;
import org.activiti.engine.test.ActivitiRule;
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
    SpringJUnit4ClassRunner;
```

Listing 5.18: Importieren der Bibliotheken, die zum Testen der Prozesse nötig sind.

beschrieben.

Die zum Testen der Prozesse notwendigen Bibliotheken sind in Listing 5.18 zu finden. Bevor mit dem Testen begonnen werden kann, gibt man, wie in Listing 5.19 zu sehen ist, noch vor der Klassendefinition an, dass die Tests mithilfe der SpringJUnit4ClassRunner Klasse auszuführen sind und gibt dafür den Pfad zu dem verwendeten Kontext an, damit man Zugriff auf die dort angegebene Datenbankkonfiguration etc. erhält. Die Tests werden dann durch die Annotations von JUnit durchgeführt.

Das Prinzip des Testens sieht nun so aus, dass man ein Objekt oder eine Variable von der Process Engine generieren lässt und anschließend prüft, ob es richtig erstellt wurde bzw. die Variable den richtigen Wert hat.

Die Testmethode startMonopolyPlayGameProcessTest() z. B. dient zum Überprüfen, ob der Hauptprozess des Spieles korrekt gestartet und richtig initialisiert wurde. Dafür verwendet man die Methoden aus der von uns geschriebenen *ProcessController*-Klasse, die mithilfe der Activiti-Process-Engine den Programmablauf steuert. In dem gezeigten Testfall werden die Anzahl der Prozessinstanzen, der Executions und der aktuellen Processexecutions beim Starten des Spieles (siehe Listing 5.20) überprüft.

Ein weitere Testmethode 5.21 überprüft, ob das *Game*-Objekt von dem Prozess erzeugt wurde.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:META-INF/spring/applicationContext
    .xml")
public class ProcessControllerTest {
@Autowired
private RuntimeService runtimeService;
@Autowired
@Rule
public ActivitiRule activitiSpringRule;
private static ProcessController ProcessController;
private static String GAMEID;
private static String PLAYERID = "payeridTest";
private static String PLAYER_NAME = "testPlayerName";
```

Listing 5.19: Annotationen der Klasse und den Attributen zum Testen der Prozesse.

```
@Test
public void startMonopolyPlayGameProcessTest() {
  ProcessController = new ProcessController();
  // start new PlayGame process instance
  GAMEID = ProcessController.startProcessPlayGame(PLAYERID);
  //3 Process Instances for this gameid expected
  assertEquals(3, ProcessController.getAllProcessInstances(GAMEID).
      size());
  //6 process executions for this gameid expected
  assertEquals(6, ProcessController.getAllProcessExecutions(GAMEID)
      .size());
  //2 current process executions with this gameid expected
  assertEquals(2, ProcessController.getCurrentProcessExecutions(
      GAMEID).size());
}
```

Listing 5.20: Testmethode zum Überprüfen , ob der Hauptprozess richtig initiiert und gestartet wurde.

```
@Test
public void gameObjectInitTest() {
  Game game = ProcessController.getGame(GAMEID);
  //expected the same gameid and the game object for this gameid
      not null
  assertNotNull(game);
  assertEquals(GAMEID, game.getGameUUID());
}
```

Listing 5.21: Testmethode zum Überprüfen, ob die *Game* Instanz erzeugt wurde.

```
1  @Test
2  public void currentExecutionTest() {
3    Game game = ProcessController.getGame(GAMEID);
4    mc = game.getMessageCentral().getMessage(PLAYERID);
5    String signal = mc.getSignal();
6    HashMap<String, Execution> executionWaitingForSignal =
         ProcessController
7      .getExecutionWaitingForSignal(GAMEID, signal);
8    // Expected equels executions waiting for signal
9    assertEquals(executionWaitingForSignal.get(signal).
         getProcessInstanceId(),
10   ProcessController.getCurrentProcessExecution(GAMEID).
         getProcessInstanceId());
11 }
```

Listing 5.22: Überprügung des aktuellen Prozesszustandes.

Nach diesem ersten Test können wir beliebig weitere Prozesse testen und Testfälle erstellen. Wir zeigen dieses anhand eines Beispiels. Mit der Methode *setPlayerNameTest()* holen wir uns zuerst den Namen des Spielers von der aktuellen Nachricht und testen danach, ob eben dieser richtig gesetzt wurde. Man holt sich dazu (wie in Listing 5.20 gezeigt wird) das Game-Objekt (siehe Kapitel 5.4.3, Seite 74) von dem *ProcessController*-Objekt und liest davon den durch die Nachricht gesetzten Namen des aktuellen Spieles ein. Das Testen aller weiteren Prozessvariablen erfolgt auf ähnliche Weise.

Um zu überprüfen, ob sich der Prozess in dem gewünschten, auf ein Signal wartenden, Zustand befindet, dient die Methode in Listing 5.22. In diesem Fall wird geprüft, ob das Signal in der aktuellen Nachricht mit dem aktuellen Signalzustand in dem Prozess übereinstimmt.

Weiterhin wird getestet, wie in Listing 5.23 zu sehen, ob nach der Signalbestätigung der Prozess weiter ausgeführt wird und sich dann in einem neuen Signalzustand befindet.

5.5.2. Services und Nachrichten

Die Kommunikationsnachrichten wurde im Projekt mittels JUnit getestet. Die Menge der Tests ist jedoch recht klein, da die Komplexität nicht sehr hoch ist und bereits auf ausgiebig getestete Datenstrukturen von Java zurückgreift. Im einzelnen getestet wurde die MessageQueue und die MessageCentral. Die MessageCentral stellt das Interface für Services zu den Nachrichten der einzelnen Spieler dar. Verwaltet werden die Nachrichten separat für jeden Spieler in einer eigenen MessageQueue. Die MessageQueue selbst hält eine Menge an Nachrichten

```
1  public void signalReseivedTest() {
2    String signal = mc.getSignal();
3    ProcessController.getMessageForPlayer(GAMEID, PLAYERID);
4    ProcessController.doAction(mc, GAMEID, PLAYERID);
5    //expected 0 executions waiting for this signal
6    assertEquals(0, ProcessController.getExecutionWaitingForSignal(
        GAMEID, signal).size());
7    Game game = ProcessController.getGame(GAMEID);
8    mc = game.getMessageCentral().getMessage(PLAYERID);
9    //expected a signal but not the same
10   assertNotSame(signal, mc.getSignal());
11 }
```

Listing 5.23: Überprügung des aktuellen Prozesszustandes nach der Signalbestätigung.

nach FIFO-Prinzip. Den Code zu den Tests findet sich im Anhang (A.1 bzw. A.2 ab Seite 103).

Die Funktionsweise der MessageQueue besteht daraus, die elementaren Funktionen einer Queue in dieser Umsetzung zu prüfen. Diese sind addMessage, removeMessage und getNextMessage.

Um die Funktionalität der MessageCentral zu prüfen, wird vor jedem Test eine Game-Instanz angelegt, welche mit zwei Spielern initialisiert wird. Dies ist nötig, da die MessageCentral Informationen von Spielern nutzt, um die einzelnen MessageQueues zuzuordnen. Eine vollständig ausgeprägte Instanz der Game-Klasse wird jedoch nicht benötigt. Nachdem die Initialisierung abgeschlossen ist, kann dann der eigentliche Test stattfinden. Dazu werden die Methoden hasMessages(), getMessage() und getMessageAsJson() auf ihr Verhalten geprüft. Es werden z. B. Nachrichten hinzugefügt oder entnommen, geprüft ob die Reihenfolge stimmt, die richtige Anzahl von Nachrichten übrig ist oder ob das Verhalten im Falle von Fehleingaben wie vorgesehen ist.

Das Testen der Jersey Services kann ebenfalls mittels JUnit durchgeführt werden. Das aktive Testen der Services erschien uns jedoch wenig sinnvoll, da die Services selbst fast ausschließlich die Activiti- bzw. Spiellogik anstoßen und sonst allenfalls Nachrichten annehmen und weiterreichen. Gleichzeitig ist der In- und Output vom Spielzustand abhängig und damit so gut wie nicht unabhängig zu testen. Da Activiti und die Spiellogik separate Tests haben, wurde also auf die Service-Tests verzichtet.

5.5.3. GUI

Auch die GUI einer Webapplikation muss getestet werden, da sie die Schnittstelle zum Nutzer ist.

Da ein manueller GUI-Test sehr zeitaufwändig ist, gibt es Werkzeuge wie das Test-Framework Selenium[12], welches ein automatisiertes Testen von Webanwendungen ermöglicht. Bei den Testfällen von Selenium handelt es sich um „record / replay“ Testfälle.

Durch die Selenium IDE wird es also ermöglicht Interaktionen mit einer Webanwendungen aufzunehmen und diese Testfälle dann automatisiert und beliebig oft zu wiederholen. Selenium ist in Java programmiert und kann daher auf jedem System mit einem aktuellen JRE ausgeführt werden.

Für die Erstellung der Tests gibt es zwei mögliche Vorgehensweisen. Die erste Möglichkeit stellt die bereits erwähnte Selenium IDE, ein Plugin für Firefox, da. Mit ihr lassen sich Browserinteraktionen aufnehmen und abspielen sowie Testfälle visualisieren.

Abhängig vom Aufbau und der Komplexität der Website funktioniert die Aufnahme entsprechender Kommandos mehr oder weniger gut. Generell ist die IDE allerdings sehr gut dazu geeignet, schnell und einfach ein Grundgerüst für einen komplexeren Test zu erstellen.

Die zweite Möglichkeit Selenium zum Testen zu verwenden ist durch die Selenium API, auch Selenium Webdriver API genannt, gegeben. Selenium WebDriver ist der Nachfolger von Selenium Remote Control und ist das neue Feature der aktuellen Selenium 2 Version. Im Gegensatz zu Selenium Remote Control, wo mittels Javascript mit dem Browser interagiert wird, kann mit der Implementierung der WebDriver API der Browser nativ bzw. durch spezielle Browsererweiterungen angesprochen werden. Selenium 2 umfasst zusätzlich auch den Selenium-Server, welcher optional benutzt werden kann. Mit diesem lassen sich z. B. Tests via Selenium Grid auf verschiedene Systeme verteilen.

Die Selenium API ist in mehreren Programmiersprachen verfügbar. Derzeit existieren Bibliotheken für

- C#,
- Java,
- Perl,
- PHP,
- Python und

[12] http://seleniumhq.org/

- Ruby.

Um einen Test für Selenium zu schreiben, muss nur das Package bzw. Modul in das Projekt importiert werden. Zusätzlich lassen sich per HTTP-Request Kommandos an den Selenium-Server übermitteln, der in der Selenium API integriert ist. Somit ist es möglich mit jeder Programmiersprache, die HTTP-Requests beherrscht, den Server und somit die Tests zu steuern.

Die Selenium API ist besonders für nicht standardmäßig ablaufende Testfälle geeignet, benötigt aber eine gewisse Einarbeitungszeit. Aus zeitlichen Gründen hat sich die Projektgruppe hauptsächlich mit der Selenium IDE beschäftigt, auch wenn bei der Komplexität unserer Webanwendung die Selenium API sicherlich geeigneter gewesen wäre.

In dem folgendem Unterkapitel wird der erstellte Testfall für die Monopoly Webanwendungen näher erläutert. Weiterhin wird auf die Erfahrungen eingegangen, die die Projektgruppe mit dem GUI-Testen anhand von Selenium sammeln konnte.

Der Selenium IDE Monopoly Test-Case

Das Erstellen eines funktionierenden Test-Case für unsere Webanwendung gestaltete sich anfangs schwierig. Zum Einen war die Arbeit mit Selenium für alle Teilnehmer eine neue Erfahrung, zum Anderen stellte die Komplexität der Monopoly-GUI eine Hürde für das Arbeiten mit Selenium dar.

Der Grund dafür war die Menge an unterschiedlichen Spielfeldern, die für den Spieler unterschiedliche Aktionsmöglichkeiten mit sich brachten. So musste der Testfall nicht nur einfaches Würfeln beherrschen, sondern auch zu jedem Spielfeld eine passende Antwort liefern. Für den Fall, dass das Feld vom Typ Straße war, gab es zum Beispiel die Möglichkeit das Grundstück zu kaufen oder es nicht zu tun. Dazu kamen weitere Möglichkeiten wie das setzen von Häusern, Hotels, Hypotheken oder das Reagieren auf Ereignis- und Gemeinschaftskarten. Die Erstellung des Testfalls war somit im Vergleich zu normalen Seiten mit einfachen Registrationssystemen weitaus komplexer.

Als Lösung auf die unterschiedlichen Entscheidungen, die ein Spieler je nach Feld haben kann, wurde für jeden Fall ein einzelner Test-Case erstellt, der in die komplette Test-Suite beliebig oft eingebaut werden konnte.

So war es nicht nötig, immer wieder per Hand jeden Testprozess neu zu schreiben. Selenium speichert dabei jeden Testprozess als `html`-Datei ab, die in die Test-Suite geladen werden kann. Abbildung 5.8 zeigt dabei die Selenium IDE mit den benutzten Testprozessen.

Kommandos in Selenium, wie in Abbildung 5.8 zu sehen, werden auch „Selenese“ genannt. Selenese sind eine Menge von Kommandos die für ein Test

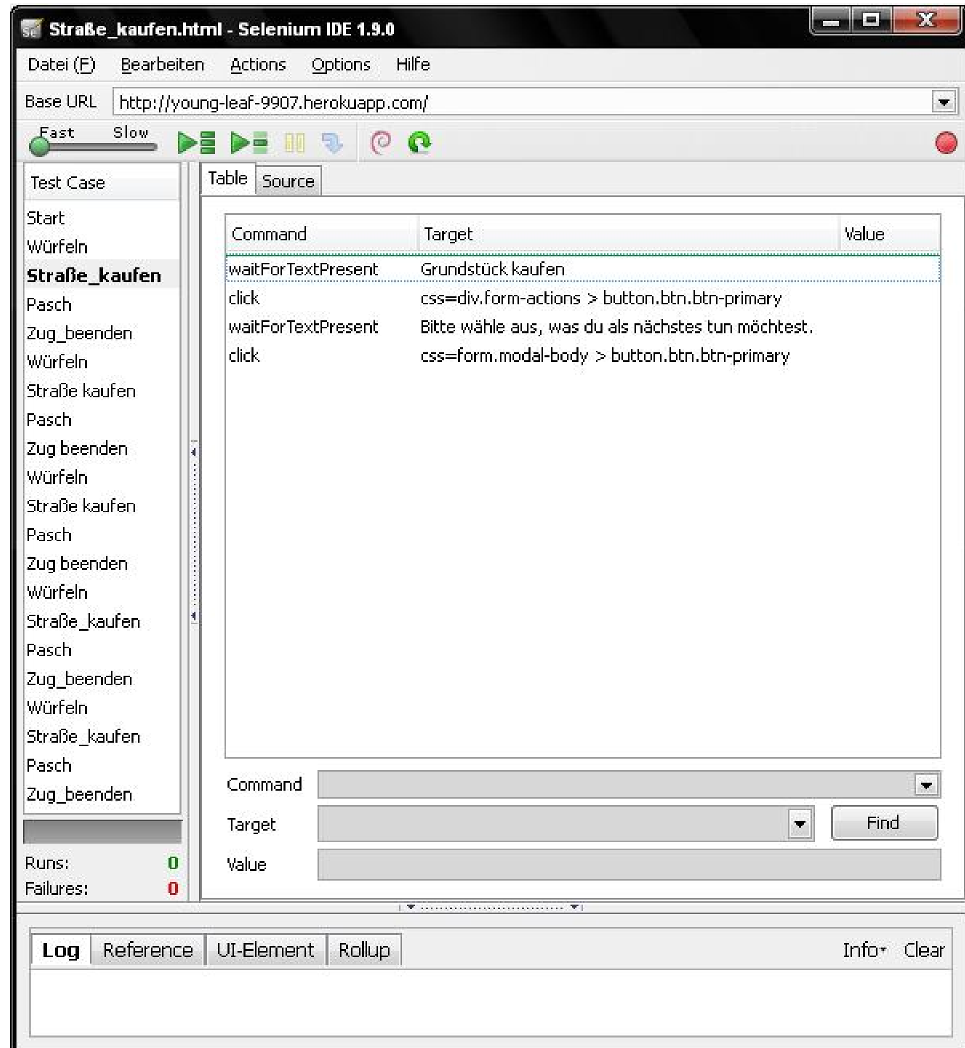

Abbildung 5.8.: Die Selenium Test-Suite mit Einbindung und Duplikationen unterschiedlicher Test-Cases.

benötigt werden. Ein Nachteil der zum Ende der Projektgruppe auffiel ist, dass Selenese keine Datenflusssteuerung und somit kein Iterieren von Test-Cases unterstützt. Wenn ein Test ausgeführt wird, wird dieser als Sequenz ausgeführt, d. h. es wird ein Kommando nach dem Anderen ausgeführt. Es gibt keine Möglichkeit festzulegen, dass Test-Cases beliebig oft durchgeführt werden sollen, da Bedingungs- und Iterationsstatements nicht unterstützt werden. Stattdessen musste jeder Test-Case in die Test-Suite kopiert werden, damit er ausgeführt wurde. Daher bestand die Test-Suite hauptsächlich aus Duplikaten der einzelnen Testprozesse, was in Abbildung 5.8 gut zu erkennen ist. Schlug ein Test-Case fehl, wurde der nächste abgearbeitet. Es gab somit keinerlei Möglichkeit, auf vorher durchlaufene und ggf. fehlgeschlagene Test-Cases zurückzuspringen.

Dadurch wurde es umso problematischer, einen Test zu erstellen, der das Spiel soweit spielen konnte, bis z. B. der Spieler alle Straßen einer Farbe besaß und somit in der Lage war, seine ersten Häuser zu kaufen.

Es war aufgrund der vielen Möglichkeiten nicht abzusehen, nach wie vielen Runden der Spieler diese Situation erreichen würde und da die Test-Suite stets nur so viele Iterationen durchlief, wie vorher Tests eingefügt wurden, war es nicht möglich eine große Anzahl an Testfällen der Anwendung abzudecken.

Erst zum Ende hin entdeckte die Gruppe das „Flow“-Addon[13], welches ein Opensource Addon der Selenium IDE ist und GOTO Sprunganweisungen als Kommandozeilen erlaubt. Mit Hilfe dieses Addons wurde die Selenium Selenese mit gerade den wichtigen Kommandos erweitert, um ein Datenflussrelationen zu ermöglichen. Anhand von drei Listings wird eine Zusammenfassung des Lösungsansatzes mit Hilfe von „Flow“ erläutert.

In Listing 5.24 wurden in dem Beispiel zunächst zwei Variablen im Hauptlabel „Prozess“ gespeichert. Bei diesen Variablen handelt es sich um boolsche Werte, die angeben, ob es sich um eine Straße oder um eine „Zug Beenden“ Situation handelt.

```
1  <tr>
2    <td>label</td>
3    <td>Prozess</td>
4    <td></td>
5  </tr>
6  <tr>
7    <td>store</td>
8    <td>false</td>
9    <td>s</td>
10 </tr>
11 <tr>
12   <td>store</td>
```

[13] http://51elliot.blogspot.de/2008/02/selenium-ide-goto.html

```
13   <td>false</td>
14   <td>z</td>
15 </tr>
```

Listing 5.24: Im Hauptlabel werden zunächst wichtige Variablen abgespeichert.

Als nächsten Schritt sieht man in Listing 5.25, wie eine Straßenabfrage genau aussieht und wie dabei die neuen Kommandos gotoLabel und gotoIf verwendet werden, welche eine Sprunganweisung auf ein definiertes Label erlauben. Findet man mithilfe des Kommandos storeTextPresent einen gezielten Text, der auf der Seite erscheint, so wird true in die zugehörige, vorher definierte Variable, gespeichert, ansonsten false.

```
1  <tr>
2    <td>storeTextPresent</td>
3    <td>Grundstück kaufen</td>
4    <td>s</td>
5  </tr>
6  <tr>
7    <td>gotoIf</td>
8    <td>${s}</td>
9    <td>Straße</td>
10 </tr>
11 <tr>
12   <td>gotoLabel</td>
13   <td>Beende</td>
14   <td></td>
15 </tr>
16 <tr>
17   <td>label</td>
18   <td>Straße</td>
19   <td></td>
20 </tr>
21 <tr>
22   <td>click</td>
23   <td>css=div.form-actions &gt; button.btn.btn-primary</td>
24   <td></td>
25 </tr>
```

Listing 5.25: Die Straßenabfrage mit den dazu gehörenden Sprunganweisungen.

Das gleiche Vorgehen sieht man in Listing 5.26, wo der Zug jeweils beendet wird und man die eigentliche Iteration sieht. Denn wird der Zug beendet, springt man mit dem Kommando gotoLabel zurück auf das Label „Prozess“, wodurch man den kompletten Test-Case erneut ausführt.

```
1  <tr>
2    <td>label</td>
```

```
    <td>Beende</td>
    <td></td>
</tr>
<tr>
    <td>storeTextPresent</td>
    <td>Zug beenden</td>
    <td>z</td>
</tr>
<tr>
    <td>gotoIf</td>
    <td>${z}</td>
    <td>Beende Zug</td>
</tr>
<tr>
    <td>label</td>
    <td>Beende Zug</td>
    <td></td>
</tr>
<tr>
    <td>click</td>
    <td>css=button.btn.btn-success</td>
    <td></td>
</tr>
<tr>
    <td>gotoLabel</td>
    <td>Prozess</td>
    <td></td>
</tr>
```

Listing 5.26: Eine Abfrage ob der Zug beendet werden soll und die Sprunganweisung auf das Hauptlabel.

Mit dem „Flow“-Addon war es also doch möglich, ein Test-Case beliebig oft zu wiederholen, ohne wie zuvor die Test-Cases manuell immer wieder duplizieren zu müssen. Außerdem war es möglich alle Fälle abzudecken, die während eines Zugs auftreten können.

Es sei dabei angemerkt, dass es sich hier in den drei Beispielen um eine gekürzte Version der eigentlichen Test-Cases handelt. Die Test-Suite beinhaltet weitaus mehr Abfragen wie z. B. Antwortverhalten auf Tausch- und Würfelanfragen, um das Spiel erfolgreich spielen zu können.

Fazit

Rückblickend auf die letzten Wochen war die Arbeit mit dem Testframework Selenium eher ein Verstehen und Einarbeiten in ein Testframework für Webanwendungen, als die Benutzung eines Testframeworks, um möglichst viele und

schwer zu findende Fehler in der Webanwendung automatisiert zu finden.

Dies hat vor allem zwei Gründe: Zum einen ermöglichte die Selenium IDE anfangs keine optimale Abdeckung aller Testfälle, die zum erfolgreichen Testen unserer Webanwendung, vor allem aufgrund der fehlenden Iteration, von Nöten waren. Das Benutzen der Selenium IDE war ohne Addons nicht für unsere Webanwendung geeignet und für die Einarbeitung in die komplizierte Selenium API fehlte der Projektgruppe zum Ende des Semesters die Zeit.

Zum anderen wurde das „Flow“-Addon zu spät gefunden, so dass man die automatisierten Tests, die beliebig viele Iterationen ausführen konnten, zu spät erstellen konnte.

Die Mitglieder der Projektgruppe hatten bis zur Erstellung der ausführlichen Tests bereits meist schon manuell viele Fehler durch mehrmaliges Spielen gefunden, so dass das Testen mit dem fertigen Selenium-Test eher ein „Nice-to-have“ war.

Dennoch bot die Selenium IDE während der Entwicklungszeit eine gute und schnelle Möglichkeit zumindest einige Testfälle, die man nach einer abzählbaren Anzahl von Runden erreichen konnte, zum Anfang der Testphase abzudecken. Ein früheres Beginnen der Testphase im Semester und mehr Teilnehmer der Projektgruppe, die sich mit der gezielten Einarbeitung von Selenium und der Selenium API beschäftigt hätten, hätte sicherlich weitaus schnellere Test-Cases und somit mehr Vorteile für die Projektgruppe gebracht.

5.6. Resümee

Die Umsetzung von Monopoly mit Prozessen hat sich am Ende doch komplexer als erwartet herausgestellt. Einig ist sich die PG darüber, dass eine Umsetzung ohne Prozesse wesentlich einfacher gewesen wäre. Dies ergibt sich aus dem Fakt, dass unser Projekt zwar relativ viele mögliche Prozesspfade hat, der Gesamtumfang aber deutlich geringer ausfällt als z. B. beim Rechnungswesen von Unternehmen. Durch den modularen Aufbau und die vielen variierenden Auswahlmöglichkeiten pro Zug sorgten für einen erhöhten Trial and Error Ansatz. Zunächst wurde das Datenmodell entworfen, danach die Kommunikation zwischen GUI und Server, die Prozesse und die GUI selbst. Das Bugfixing der Prozesse war aufgrund der in jedem Schritt vorhandenen vielen Entscheidungsmöglichkeiten ziemlich aufwendig. Zu viele Faktoren spielten hier eine Rolle als dass ein effektives Arbeiten möglich gewesen wäre. Die GUI-Arbeiten waren immer auf den Fortschritt und das Bugfixing der Prozesse angewiesen, weshalb auch hier kein effektives Arbeiten möglich war.

Innerhalb der GUI wurde auf Standard-Frameworks gesetzt, und der Code

von Anfang an strukturiert gehalten. Auch gravierende Änderungen – z. B. die Logger-Klasse kam erst sehr spät dazu – waren so ohne große Probleme möglich. Das Spiel war eine gute Möglichkeit das Zusammenspiel von vielen Frameworks in einer relativ komplexen HTML-Anwendung auszuprobieren, und unserer Einschätzung nach, ist uns das gelungen.

Viele Business-Prozesse wurden am Ende erweitert bzw. ergänzt, da viele Faktoren, die wir in der ursprünglichen Planung nicht berücksichtigt hatten, erst später zum Vorschein kamen. Probleme bereitete auch das Arbeiten mit festen Strings auf der Backend-Seite und in den Prozessen. Durch falsche Schreibweise, Vergessen von nachträglichen Anpassungen an den verschiedenen Stellen nach Variablenänderungen oder auch durch verborgene Leerzeichen in den Namen entstanden unzählige Fehler.

Dies ist einerseits durch unseren Ansatz der Umsetzung, andererseits aber auch durch den Aufbau der Prozesse und dem Arbeiten mit dem Activiti-Designer zu begründen. Viele der Erkenntnisse erlangten wir aber erst bei der Arbeit, da die Dokumentation viele Fragen offen ließ und die Erfahrungen der Gruppe auf dem Gebiet auch sehr gering waren.

6. Zusammenfassung

Nach zwei Semestern können wir als Teilnehmer sagen, dass wir sehr viele neue Erfahrungen sammeln konnten. Wir haben dabei in den beiden Semestern unterschiedliche Erfahrungen sammeln können, auf die wir näher eingehen wollen.

Die Projektgruppe wurde im ersten Semester damit konfrontiert, sich gezielt in unbekannte und neue Technologien einzuarbeiten und als Gruppe miteinander zu arbeiten und zu kommunizieren. Die Einarbeitung in die unterschiedlichen Clouddienste war dabei für alle Projektteilnehmer Neuland. Das gleiche galt für die Einarbeitung in die unterschiedlichen Prozessumgebungen wie Activiti und JBPM. Während viele Teilnehmer die Modellierungssprache BPMN bereits kannten, war das Arbeiten mit Prozessumgebungen hingegen allen Teilnehmern neu.

Vor allem im ersten Semester stellte die Projektgruppe fest, wie viel von den eigentlich geplanten und vorgestellten Technologien aufgrund von Inkompatibilitäten nicht verwendbar oder umsetzbar waren. Besonders die Erfahrung, sich mehrere Wochen mit einer Prozessumgebung und einem Clouddienst zu beschäftigen, um dann festzustellen, dass es zumindest zu diesem Zeitpunkt nicht möglich ist, ein von uns geplantes Projekt mit dieser Technologie umzusetzen, war ziemlich demotivierend. Im Laufe des ersten Semesters wurden viele Technologien ausprobiert und trotzdem endeten die meisten Ansätze damit festzustellen, dass die verschiedenen Ansätze für unser Projekt unmöglich einsetzbar oder ungeeignet waren. Am Ende griff man auf Heroku, die Anfangs nicht berücksichtige Platform, zurück. Das die verschiedenen Angebote alle noch ständig erweitert werden und viele Features wie z. B. SQL-Datenbanken zum Zeitpunkt der Projektumsetzung noch nicht verfügbar waren, erschwerte die Suche nach einer geeigneten Lösung ebenfalls.

Neben diesen Erfahrungen waren im ersten und zweiten Semester die Projektplanung und Kommunikation in der Gruppe selbst eine große Herausforderung. So musste die Projektgruppe lernen, ordentlich miteinander zu kommunizieren und Aufgaben richtig zwischen den Teilnehmern aufzuteilen. Während die Projektplanung und Aufgabenverteilung im zweiten Semester eine weitaus wichtigere Rolle annahm, zeigte sich bereits im ersten Semester, wie gut die Projektgruppe miteinander kommunizierte. Es gab jedoch auch immer mal wieder Probleme,

da die Teilnehmer nur spärlich die unterschiedlichen Nachrichtendienste wie Skype oder ICQ benutzten oder zu spät ihre Mails überprüften.

Das zweite Semester drehte sich dann zum größten Teil um die Planung und Umsetzung des von den Betreuern festgelegten Projekts. Die Projektplanung gestaltete sich sehr gradlinig und trotz der kleinen Anzahl an Teilnehmern wurden Aufgaben zumeist rechtzeitig und fair unter den Teilnehmern aufgeteilt. Die Umsetzung war jedoch immer wieder gespickt mit Tücken, welche zu Anfang nicht bedacht wurden. Das lag vermutlich daran, dass niemand in der Gruppe wirkliche Erfahrungen in der Arbeit mit prozessbasierten Anwendungen und Java-Webservices hatte. So wurden Probleme an der Projektstruktur erst später klar, als sich die Lösung nicht umsetzen ließ wie gedacht. So mussten kurz vor Ende des zweiten Semesters noch umfassende Änderungen durchgeführt werden, um das Spiel funktionsfähig zu machen. Der erste Ansatz der PG konnte nicht umgesetzt werden und so wurde die KI auf dem Server umgesetzt, die normale Messages zur Kommunikation mit der GUI nutzte. Die Umsetzung der GUI lief hingegen ohne Probleme ab, da die Teilnehmer in diesem Bereich schon deutlich mehr Erfahrung mitbrachten.

Die Umsetzung des Projektes und die Fehlerbeseitigung stellte sich schwieriger dar als erwartet. Die Signale und Prozessvariablen, welche in den Prozessen definiert wurden, mussten korrekt im Code verwendet werden und umgekehrt. Beim Testen ergab sich daraus ein Crash durch jeden noch so kleinen Rechtschreibfehler da es keine Verbindung von Code und Prozessen in Eclipse gab. Zusätzlich konnte an der GUI erst dann weitergearbeitet werden, wenn der nächste Schritt im Prozess oder Code fertiggestellt wurde oder der Fehler behoben wurde. Diese Tatsachen zusammen gestalteten sowohl die Programmierung als auch das Testen des Projektes als äußert müßig. Das gegen Ende des Semesters mit einbezogene Testen der GUI, der Prozesse und des Codes konnte das manuelle Testen ebenfalls nicht ersetzen.

Während sich die Meinungen der Teilnehmer unterscheiden, ob sie jemals wieder mit an prozessbasierten Anwendungen arbeiten wollen, lernten wir, wie man gezielt mit Hilfe von Prozessen eine Webanwendung in der Cloud modellieren und umsetzen kann. Dabei war nicht nur die Umsetzung des Projekts, sondern auch die Erfahrung mit dem Umgang von Problemen bei der Umsetzung interessant.

Letztendlich kann man sagen, dass die Teilnehmer viel gelernt haben. Sowohl im Bereich der Projektplanung, als auch in den Bereichen der unterschiedlichen Technologien, die sich die Projektgruppe aneignen musste. So bleibt am Ende der Dank an unsere Betreuer, die sich die Mühe gemacht haben, ihr geplantes Projekt auf eine über den Verlauf der PG schrumpfende Personenanzahl soweit anzupassen, dass die Gruppe dieses Projekt auch umsetzen konnte.

Wir bedanken uns bei den Betreuern und für die lehrreichen zwei Semester.

A. Anhang

```
1  package com.tudo.pgpcb.monopoly.test;
2
3  import static org.junit.Assert.*;
4
5  import java.util.UUID;
6
7  import org.junit.After;
8  import org.junit.Before;
9  import org.junit.BeforeClass;
10 import org.junit.Test;
11
12 import com.tudo.pgpcb.monopoly.game.Game;
13 import com.tudo.pgpcb.monopoly.interaction.InteractionMessage;
14 import com.tudo.pgpcb.monopoly.interaction.
       InteractionMessageFactory;
15 import com.tudo.pgpcb.monopoly.interaction.MessageQueue;
16 import com.tudo.pgpcb.monopoly.interaction.InteractionMessage.
       MessageType;
17
18 public class MessageQueueTest {
19
20   MessageQueue q;
21
22   public InteractionMessage createInteractionMessage(String header,
         String content){
23     InteractionMessage msg = new InteractionMessage();
24     msg.setHeader(header);
25     msg.setContent(content);
26     return msg;
27   }
28
29   @Before
30   public void before(){
31     q = new MessageQueue();
32   }
33
34   @Test
35   public void testAddMessage() {
36     assertEquals(0,q.getMessageCount());
37     for(int i = 0; i<1000; i++){
```

```
38         q.addMessage(createInteractionMessage(Integer.toString(i), "
               This␣is␣message␣number␣" + i));
39         assertEquals(i+1,q.getMessageCount());
40       }
41     }
42
43     @Test
44     public void testGetNextMessage(){
45       testAddMessage();//fill queue
46       assertEquals(1000,q.getMessageCount());
47       InteractionMessage msg = null;
48       for(int i = 0; i<1000; i++){
49         msg = q.getNextMessage();
50         assertTrue(msg.getHeader().equals(Integer.toString(i)));
51         assertEquals(1000−1−i,q.getMessageCount());
52       }
53     }
54
55     @Test
56     public void testRemoveMessage(){
57       testAddMessage();//fill queue
58       assertEquals(1000,q.getMessageCount());
59       q.removeMessage(null);//no message deleted
60       assertEquals(1000,q.getMessageCount());
61       InteractionMessage msg = createInteractionMessage("never␣used",
              "same␣here");
62       q.removeMessage(msg);//no message deleted
63       assertEquals(1000,q.getMessageCount());
64       msg = createInteractionMessage("1000", "OVER␣ONE␣THOUSAAAAND");
65       q.addMessage(msg);
66       assertEquals(1001,q.getMessageCount());
67       q.removeMessage(msg);//delete the recently added message
68       assertEquals(1000,q.getMessageCount());
69       q.removeMessage(msg);//delete the same message again, so not
             deleting a message as it is not found
70       assertEquals(1000,q.getMessageCount());
71     }
72
73
74
75     @After
76     public void after(){
77       q = null;
78     }
79
80   }
```

Listing A.1: Der MessageQueue-Test.

```
1  package com.tudo.pgpcb.monopoly.test;
2
3  import static org.junit.Assert.*;
4
5  import java.util.UUID;
6
7  import org.junit.After;
8  import org.junit.Before;
9  import org.junit.Test;
10
11 import com.tudo.pgpcb.monopoly.datamodel.Player;
12 import com.tudo.pgpcb.monopoly.game.Game;
13 import com.tudo.pgpcb.monopoly.interaction.InteractionMessage;
14 import com.tudo.pgpcb.monopoly.interaction.
       InteractionMessageUtility;
15 import com.tudo.pgpcb.monopoly.interaction.MessageCentral;
16
17 public class MessageCentralTest {
18
19   Game game = null;
20   MessageCentral c;
21   Player player1;
22   Player player2;
23
24   public InteractionMessage createInteractionMessage(String header,
         String content){
25     InteractionMessage msg = new InteractionMessage();
26     msg.setHeader(header);
27     msg.setContent(content);
28     return msg;
29   }
30
31   @Before
32   public void before(){
33     //As the queueCentral uses information that comes from the game
           , the game has to be initialized at least to a certain
           degree
34     game = new Game();
35     game.setGameUUID(UUID.randomUUID().toString());
36     player1 = new Player();
37     player1.setName("Player␣1");
38
39     player2 = new Player();
40     player2.setName("Player␣2");
41
42     //player.id will be set when added -> will be used for queue
           mapping
43     game.addNewPlayer(player1);;
```

```
44     game.addNewPlayer(player2);
45
46     c = new MessageCentral(game);
47   }
48
49   @Test
50   public void testAddMessageForPlayer() {
51     assertFalse(c.hasMessages(Integer.toString(player1.getId())));
         //verify that no message exists right now for player1
52     assertFalse(c.hasMessages(Integer.toString(player2.getId())));
         //verify that no message exists right now for player2
53     assertFalse(c.hasMessages("-asdfc14785"));//verify that no
         message exists right now for a non existent player
54
55     c.addMessageForPlayer(Integer.toString(player1.getId()),
         createInteractionMessage("0", "This is message number 0"));
          //add a message for player1
56
57     assertTrue(c.hasMessages(Integer.toString(player1.getId())));//
         check if there is a message now for player1
58     assertFalse(c.hasMessages(Integer.toString(player2.getId())));
         //check if there still is no message for player2
59     assertFalse(c.hasMessages("-asdfc14785"));//check if there
         still is no message for a non existent player
60
61     c.addMessageForPlayer(Integer.toString(player1.getId()),
         createInteractionMessage("1", "This is message number 1"));
          //add a message for player1
62
63     assertTrue(c.hasMessages(Integer.toString(player1.getId())));//
         check if there is a message now for player1
64     assertFalse(c.hasMessages(Integer.toString(player2.getId())));
         //check if there still is no message for player2
65     assertFalse(c.hasMessages("-asdfc14785"));//check if there
         still is no message for a non existent player
66
67     c.addMessageForPlayer(Integer.toString(player2.getId()),
         createInteractionMessage("0", "This is message number 0" +
         0)); //add a message for player1
68
69     assertTrue(c.hasMessages(Integer.toString(player1.getId())));//
         check if there is a message now for player1
70     assertTrue(c.hasMessages(Integer.toString(player2.getId())));//
         check if there is a message now for player2
71     assertFalse(c.hasMessages("-asdfc14785"));//check if there
         still is no message for a non existent player
72   }
73
74   @Test
```

```
75    public void testGetMessage() {
76      testAddMessageForPlayer(); //fill messages
77      assertTrue(c.hasMessages(Integer.toString(player1.getId())));
          //Verify message(s) are there for the player (2 messages)
78      assertTrue(c.hasMessages(Integer.toString(player2.getId())));
          //see above (1 message)
79
80      //Get two messages for player 1, check if they exist and have
          the right order. Afterwards try to get another message,
          although none exists.
81      InteractionMessage msg = c.getMessage(Integer.toString(player1.
          getId()));
82      assertNotNull(msg);
83      assertTrue(msg.getHeader().equals("0"));
84      assertTrue(c.hasMessages(Integer.toString(player1.getId())));
          //verify that another message exists for player 1
85      msg = c.getMessage(Integer.toString(player1.getId()));
86      assertNotNull(msg);
87      assertTrue(msg.getHeader().equals("1"));
88      assertFalse(c.hasMessages(Integer.toString(player1.getId())));
          //verify that no other message exists for player 1
89      msg = c.getMessage(Integer.toString(player1.getId()));
90      assertNull(msg);
91
92      //Same as above, but for player2 who has only one message
93      msg = c.getMessage(Integer.toString(player2.getId()));
94      assertNotNull(msg);
95      assertTrue(msg.getHeader().equals("0"));
96      assertFalse(c.hasMessages(Integer.toString(player2.getId())));
          //verify that no other message exists for player 2
97      msg = c.getMessage(Integer.toString(player2.getId()));
98      assertNull(msg);
99
100     assertFalse(c.hasMessages(Integer.toString(player1.getId())));
          //Verify no messages remain
101     assertFalse(c.hasMessages(Integer.toString(player2.getId())));
          //Verify no messages remain
102   }
103
104   @Test
105   public void testGetMessageAsJson() {
106     testAddMessageForPlayer(); //fill messages
107     assertTrue(c.hasMessages(Integer.toString(player1.getId())));
          //Verify message(s) are there for the player (2 messages)
108     assertTrue(c.hasMessages(Integer.toString(player2.getId())));
          //see above (1 message)
109
110     //Get two messages for player 1, check if they exist and have
          the right order. Afterwards try to get another message,
```

```
        although none exists.
111     String msgString = c.getMessageAsJSON(Integer.toString(player1.
            getId()));
112     InteractionMessageUtility util = new InteractionMessageUtility
            ();
113     InteractionMessage msg = util.createInteractionMessageFromJSON(
            msgString);
114     assertNotNull(msg);
115     assertTrue(msg.getHeader().equals("0"));
116     assertTrue(c.hasMessages(Integer.toString(player1.getId())));
            //verify that another message exists for player 1
117     msgString = c.getMessageAsJSON(Integer.toString(player1.getId()
            ));
118     msg = util.createInteractionMessageFromJSON(msgString);
119     assertNotNull(msg);
120     assertTrue(msg.getHeader().equals("1"));
121     assertFalse(c.hasMessages(Integer.toString(player1.getId())));
            //verify that no other message exists for player 1
122     msgString = c.getMessageAsJSON(Integer.toString(player1.getId()
            ));//get null as return value
123     assertNull(msgString);
124 
125     //Same as above, but for player2 who has only one message
126     msgString = c.getMessageAsJSON(Integer.toString(player2.getId()
            ));
127     msg = util.createInteractionMessageFromJSON(msgString);
128     assertNotNull(msg);
129     assertTrue(msg.getHeader().equals("0"));
130     assertFalse(c.hasMessages(Integer.toString(player2.getId())));
            //verify that no other message exists for player 2
131     msgString = c.getMessageAsJSON(Integer.toString(player2.getId()
            ));//get null as return value
132     assertNull(msgString);
133 
134     assertFalse(c.hasMessages(Integer.toString(player1.getId())));
            //Verify no messages remain
135     assertFalse(c.hasMessages(Integer.toString(player2.getId())));
            //Verify no messages remain
136   }
137 
138   @After
139   public void after(){
140 
141     c = null;
142     game = null;
143   }
144 
145 }
```

Listing A.2: Der MessageCentral-Test.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions id="definitions"
3    targetNamespace="http://activiti.org/bpmn20"
4    xmlns:activiti="http://activiti.org/bpmn"
5    xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL">
6
7          <process id="financialReport" name="Monthly␣financial␣
             report␣reminder␣process">
8
9            <startEvent id="theStart" />
10
11           <sequenceFlow id='flow1' sourceRef='theStart' targetRef='
               writeReportTask' />
12
13           <userTask id="writeReportTask" name="Write␣monthly␣
               financial␣report" >
14             <documentation>
15               Write monthly financial report for publication to
                   shareholders.
16             </documentation>
17             <potentialOwner>
18               <resourceAssignmentExpression>
19                 <formalExpression>accountancy</formalExpression>
20               </resourceAssignmentExpression>
21             </potentialOwner>
22           </userTask>
23
24           <sequenceFlow id='flow2' sourceRef='writeReportTask'
               targetRef='verifyReportTask' />
25
26           <userTask id="verifyReportTask" name="Verify␣monthly␣
               financial␣report" >
27             <documentation>
28               Verify monthly financial report composed by the
                   accountancy department.
29               This financial report is going to be sent to all the
                   company shareholders.
30             </documentation>
31             <potentialOwner>
32               <resourceAssignmentExpression>
33                 <formalExpression>management</formalExpression>
34               </resourceAssignmentExpression>
35             </potentialOwner>
36           </userTask>
37
38           <sequenceFlow id='flow3' sourceRef='verifyReportTask'
               targetRef='theEnd' />
```

```

      <endEvent id="theEnd" />

    </process>

</definitions>
```

Listing A.3: Darstellung des Financial Report Prozesses als XML in Activiti.

```java
public class TestFinancialReport {
  public static void main(String[] args) {
    ProcessEngine processEngine = ProcessEngineConfiguration
      .createStandaloneProcessEngineConfiguration()
      .buildProcessEngine();

    // Hole die beiden Activiti Services
    RepositoryService repositoryService = processEngine.
        getRepositoryService();
    RuntimeService runtimeService = processEngine.getRuntimeService
        ();

    // Deploy die Prozessdefinition
    repositoryService.createDeployment()
      .addClasspathResource("FinancialReportProcess.bpmn20.xml")
      .deploy();

    // Start eine neue Instanz des Prozesses
    String procId = runtimeService.startProcessInstanceByKey("
        financialReport").getId();

    // Hole den ersten Usertask
    TaskService taskService = processEngine.getTaskService();
    List<Task> tasks = taskService.createTaskQuery().
        taskCandidateGroup("accountancy").list();
    for (Task task : tasks) {
      System.out.println("Following␣task␣is␣available␣for␣
          accountancy␣group:␣" + task.getName());

      // Weise ihn dem User fozzie der Accountancy-Gruppe zu
      taskService.claim(task.getId(), "fozzie");
    }

    // Verifiziere das fozzie den Task erhalten kann
    tasks = taskService.createTaskQuery().taskAssignee("fozzie").
        list();
    for (Task task : tasks) {
      System.out.println("Task␣for␣fozzie:␣" + task.getName());

      // Simuliere Bearbeitung des Task
```

```
      taskService.complete(task.getId());
    }

    System.out.println("Number of tasks for fozzie: "
      + taskService.createTaskQuery().taskAssignee("fozzie").count());

    // Hole und weise den zweiten Task zu, diesmal kermit aus der management-Gruppe
    tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();
    for (Task task : tasks) {
      System.out.println("Following task is available for accountancy group: " + task.getName());
      taskService.claim(task.getId(), "kermit");
    }

    // Simuliere Bearbeitung des Task und Beendigung des Prozesses
    for (Task task : tasks) {
      taskService.complete(task.getId());
    }
  }
}
```

Listing A.4: Business-Prozess zum Erstellen eines Finanzreportes welcher von einem Manager genehmigt werden muss.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd         http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd         http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.0.xsd         http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

  <bean class="java.net.URI" id="dbUrl">
    <constructor-arg value="${DATABASE_URL}" />
  </bean>
  <bean class="org.apache.commons.dbcp.BasicDataSource" destroy-
```

```
        method="close" id="dataSource">
12    <property name="driverClassName" value="org.postgresql.Driver"
        />
13    <property name="url" value="#{ 'jdbc:postgresql://' + @dbUrl.
        getHost() + @dbUrl.getPath() }" />
14    <property name="username" value="#{ @dbUrl.getUserInfo().split
        (':')[0] }" />
15    <property name="password" value="#{ @dbUrl.getUserInfo().split
        (':')[1] }" />
16    <property name="defaultAutoCommit" value="false" />
17  </bean>
18
19  <bean id="transactionManager"
20    class="org.springframework.jdbc.datasource.
        DataSourceTransactionManager">
21    <property name="dataSource" ref="dataSource" />
22  </bean>
23
24  <bean id="processEngineConfiguration" class="org.activiti.spring.
      SpringProcessEngineConfiguration">
25    <property name="dataSource" ref="dataSource" />
26    <property name="transactionManager" ref="transactionManager" />
27    <property name="databaseSchemaUpdate" value="true" />
28    <property name="jobExecutorActivate" value="true" />
29  </bean>
30
31  <bean id="processEngine" class="org.activiti.spring.
      ProcessEngineFactoryBean"
32    destroy-method="destroy">
33    <property name="processEngineConfiguration" ref="
        processEngineConfiguration" />
34  </bean>
35
36  <bean id="repositoryService" factory-bean="processEngine" factory
      -method="getRepositoryService" />
37  <bean id="runtimeService" factory-bean="processEngine" factory-
      method="getRuntimeService" />
38  <bean id="taskService" factory-bean="processEngine" factory-
      method="getTaskService" />
39  <bean id="historyService" factory-bean="processEngine" factory-
      method="getHistoryService" />
40  <bean id="managementService" factory-bean="processEngine" factory
      -method="getManagementService" />
41
42  <bean id="demo" class="com.tudo.pgpcb.activiti.
      ActivitiProcessEngine" />
43  <bean id="activitiRule" class="org.activiti.engine.test.
      ActivitiRule">
44    <property name="processEngine" ref="processEngine" />
```

```
45     </bean>
46 </beans>
```

Listing A.5: Die applicationContext.xml-Datei.

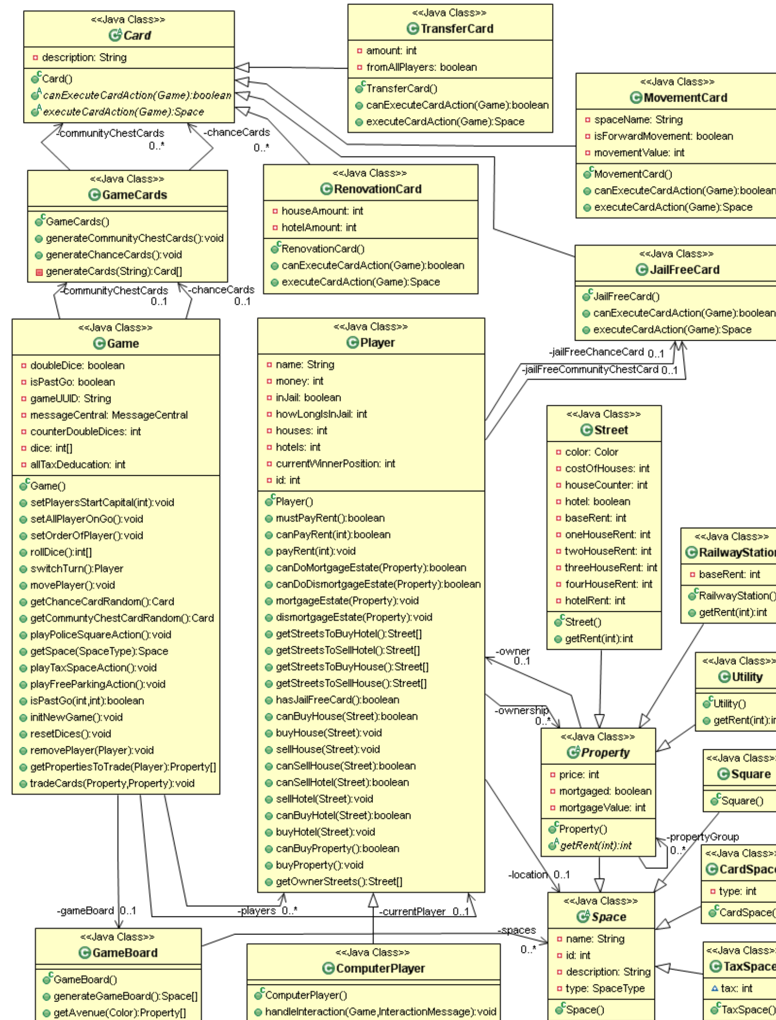

Abbildung A.1.: UML-Klassendiagramm des Monopoly-Spieles.

Abbildungsverzeichnis

3.1. Beispiel einer Arbeitsgruppe. Quelle: http://en.wikipedia.org/wiki/Business_Process_Model_and_Notation. 8
3.2. Ein Beispiel eines Nachrichtenaustauschs mit Unterteilung eines Partners. Abgeänderte Version eines Beispiels aus [All09]. . . . 10
3.3. jBPM 4-Architektur. Quelle: [BF09] 21
3.4. jBPM 5 - Architektur. Quelle: [jbp] 22
3.5. jBPM 5 - Core Engine. Quelle: [jbp] 23
3.6. Die Komponenten von Activiti. Quelle: http://www.activiti.org 24
3.7. Activiti Engine. Quelle: http://www.activiti.org 26
3.8. Beispielprozess Financial Report. Quelle: http://www.activiti.org 27
3.9. Die Datastore Architektur. [Ciu09] 32
3.10. Der Memcache. [Sev09] 33
3.11. Die Komponenten von Windows Azure. Quelle: [Chab] 36
3.12. Interoperabilität mit der Windows Azure Plattform. Quelle: [Micb] 39
3.13. Das Zusammenspiel von Java SDK und Windows Azure. Quelle: [Soy] 41
3.14. Bild der Heroku-Architektur. Quelle: [Herb] 42

5.1. Bild der GUI-Architektur. 67
5.2. Schematische Darstellung der Module des Programms. Aktionen im Programm werden von den Activiti Delegates sowie der Services durch Aufrufe der Spiellogik initiiert. Ebenso werden von dort Nachrichten für Clients in die Messagequeue gelegt. Die Spiellogik operiert auf dem Datenmodell. 71
5.3. Der Monopoly-Spiel Hauptprozess. 76
5.4. Der playerAction-Unterprozess des Hauptprozesses. 77
5.5. Der regularAction-Unterprozess des Hauptprozesses. 78
5.6. Der Card Space-Unterprozess des regularAction-Prozesses. . . . 79

5.7. Darstellung des Ablauf der Kommunikation zwischen Client und Server. Abgesehen von dem Spielbeitritt gibt es zwei unterschiedliche sich wiederholende Abläufe, welche abhängig von der vom Server erhaltenen Nachricht ist. 83
5.8. Die Selenium Test-Suite mit Einbindung und Duplikationen unterschiedlicher Test-Cases. 92

A.1. UML-Klassendiagramm des Monopoly-Spieles. 114

Listings

3.1. Erstellen einer Engine mit allen Services. 26
3.2. Prozessengine erstellen. 28
3.3. Repositories laden. 28
3.4. Deployen der Ressourcen. 28
3.5. Neue Instanz vom Prozess. 28
3.6. User-Zuweisung für Task 1. 28
3.7. User zugewiesene Tasks ausgeben. 29
3.8. User-Zuweisung für Task 2. 29
3.9. Tasks als komplett bearbeitet setzen. 29

4.1. Unsere startup.cmd. 52
4.2. Unsere activiti.cfg.xml für das Activiti Projekt. 53
4.3. activiti.cfg.xml-Datei für PostgreSQL. 56
4.4. Erstellen eines Process-Engine. 57
4.5. Erstellen einer Process-Engineinstanz. 57
4.6. Erstellen einer Process-Engine auf Heroku. 58

5.1. Array/Objekt-Definition in Coffeescript. 65
5.2. Compilat von Listing 5.1. 65
5.3. Beispielhafte Logger-Klasse in Coffeescript. 66
5.4. Beispieldefinition in LESS. 69
5.5. Compilat von Listing 5.4. 70
5.6. Der Inhalt der Procfile-Datei. 72
5.7. Ein Auschnitt aus der Kontextkonfiguration-Datei zum Einlesen der Datenbankparameter von Heroku. 72
5.8. Ein Auschnitt aus der Kontextkonfiguration-Datei, der den Zugriff auf die Datenbank erlaubt und die Prozess-Engine der Anwendung zur Verfügung stellt. 73
5.9. Ein Auschnitt aus der Kontextkonfiguration-Datei, der das Testen von Prozessen erlaubt. 74
5.10. Logging in Delegates. 80
5.11. Laden der Prozessvariablen. 80
5.12. Aktionen in der Delegate. 80

5.13. Extrahieren der Variablen aus der Message. 84
5.14. Generierung der Antwort bei Auswahl der zu bebauenden Straße. 84
5.15. Generierung der Antwort beim Grundstückkaufs. 85
5.16. Antwort als ResultVariable zurückgeben. 85
5.17. Einbinden der Test Engine. 86
5.18. Importieren der Bibliotheken, die zum Testen der Prozesse nötig sind. 86
5.19. Annotationen der Klasse und den Attributen zum Testen der Prozesse. 87
5.20. Testmethode zum Überprüfen , ob der Hauptprozess richtig initiiert und gestartet wurde. 87
5.21. Testmethode zum Überprüfen, ob die *Game* Instanz erzeugt wurde. 87
5.22. Überprügung des aktuellen Prozesszustandes. 88
5.23. Überprügung des aktuellen Prozesszustandes nach der Signalbestätigung. 89
5.24. Im Hauptlabel werden zunächst wichtige Variablen abgespeichert. 93
5.25. Die Straßenabfrage mit den dazu gehörenden Sprunganweisungen. 94
5.26. Eine Abfrage ob der Zug beendet werden soll und die Sprunganweisung auf das Hauptlabel. 94

A.1. Der MessageQueue-Test. 103
A.2. Der MessageCentral-Test. 105
A.3. Darstellung des Financial Report Prozesses als XML in Activiti. 109
A.4. Business-Prozess zum Erstellen eines Finanzreportes welcher von einem Manager genehmigt werden muss. 110
A.5. Die applicationContext.xml-Datei. 111

Literaturverzeichnis

[Act12a] *Activiti 5.10 User Guide*, http://activiti.org/userguide/#N1028F, 2012.

[Act12b] *Activiti 5.10 User Guide - Examples*, http://www.activiti.org/userguide/index.html#examples, 2012.

[Act12c] *Activiti 5.10 User Guide - Spring integration*, http://activiti.org/userguide/#apiUnitTesting, 2012.

[Ada11] Adam, *Heroku for Java*, http://blog.heroku.com/archives/2011/08/25/java/, 2011.

[All09] T. Allweyer, *BPMN 2.0 - Business Process Model and Notation: Einführung in den Standard für die Geschäftsprozessmodellierung*, 2. ed., 2009.

[BF09] Rücker B. and Menge F., *JBoss jBPM 4*, 2009.

[Chaa] David Chappell, *Introducing the windows azure platform*, http://go.microsoft.com/?linkid=9752185.

[Chab] ______, *Introducing windows azure*, http://go.microsoft.com/?linkid=9682907.

[Chac] ______, *The windows azure programming model*, http://go.microsoft.com/?linkid=9751501.

[Ciu09] Eugene Ciurana, *Developing with google app engine*, first ed., Apress, 2009 (English).

[Ert] Bert Ertman, *Developing java based cloud solutions using windows azure plugin for eclipse*, http://channel9.msdn.com/Events/DevDays/DevDays-2011-Netherlands/Devdays116.

[Goo] Google, *http://code.google.com/intl/de-DE/appengine/docs/billing.html*.

[hera] *https://devcenter.heroku.com/articles/java*.

[Herb] Heroku, *http://www.heroku.com/how/connect*.

[jbp] *JBoss jBPM 5.1 User Guide*, http://docs.jboss.org/jbpm/v5.1/userguide/.

[jbp04] *jBPM White Paper*, http://www.jboss.com/pdf/jbpm(underscore)whitepaper.pdf pp 2-9, 2004.

[jpa] *Locking in JPA*, http://www.objectdb.com/java/jpa/persistence/lock.

[LS5] LS5, *http://ls5-www.cs.tu-dortmund.de/files/Lehre/PGS/PG_PCB.pdf*.

[Mica] Microsoft, *Microsoft jdbc driver 3.0 for sql server and sql azure*, http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=19847.

[Micb] ———, *Microsoft platform and java interoperability*, http://java.interoperabilitybridges.com/cloud.

[Micc] ———, *Windows azure platform and interoperability*, https://www.windowsazure.com/en-us/develop/java/fundamentals/intro-to-windows-azure/.

[Micd] ———, *Windows azure platform offers*, http://www.windowsazure.com/en-us/pricing/details/.

[Nat11] Yefim Natis, *VMForce*, http://blogs.gartner.com/yefim_natis/2011/08/31/there-will-not-be-a-vmforce/, 2011.

[OMG] OMG, *http://www.omg.org/bpmn/BPMN_Supporters.htm*.

[Oum11] Moussa Oumarou, *Seminararbeit PG 561 - JBOSS jBPM*, 2011.

[Saw] Martin Sawicki, *Windows azure plugin for eclipse with java*, http://java.interoperabilitybridges.com/articles/windows-azure-plugin-for-eclipse-with-java.

[Sev09] Charles Severance, *Using google app engine*, first ed., O'Reilly Media, may 2009 (English).

[Soy] Soyatec, *Windows azure sdk for java developers*, http://www.windowsazure4j.org/.

[Syb] Sybase, *Tds 5.0 functional specification*, http://www.sybase.com/content/1040983/Sybase-tds38-102306.pdf.

[TB] Miguel Valdes Faura Tom Baeyens, *The Process Virtual Machine*, http://docs.jboss.com/jbpm/pvm/article/.

[VMF] *VMware News Releases*, http://www.vmware.com/company/news/releases/vmforce.html.

[Whi] Stephen A. White, *Introduction to BPMN*.