Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Philip Rose, Vassilos Stavrou

Plattform für kontextsensitive mobile Multiplayer-Games

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Philip Rose, Vassilos Stavrou

Plattform für kontextsensitive mobile Multiplayer-Games

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Kai von Luck
Zweitgutachter: Prof. Dr. Zhen Ru Dai

Eingereicht am: 29. August 2013

Philip Rose, Vassilos Stavrou

Thema der Arbeit
Plattform für kontextsensitive mobile Multiplayer-Games

Stichworte
Generisch, Pervasive Games, Rule Engine, DSL, mobile Games, Multiplayer

Kurzzusammenfassung
Diese Arbeit befasst sich mit der Entwicklung einer Plattform für kontextsensitive Multiplayer mobile Games, mit deren Hilfe es möglich sein soll, schnell und unkompliziert Spiele des besagten Typus zu erschaffen. Dabei sollen auch insbesondere Nicht-Programmierer mit der Plattform arbeiten können.

Philip Rose, Vassilos Stavrou

Title of the paper
Platform for context-sensitive mobile multiplayer games

Keywords
Generic, pervasive games, rule engine, dsl, mobile games, multiplayer

Abstract
This thesis is concerned with the development from a platform regarding multiplayer mobile games. With this platform it should be possible to create games of this type fast and easily. Especially people without developing knowledge should be able to create games with it.

Danksagung

An dieser Stelle möchten wir uns bei Prof. Dr. Kai von Luck für seine geduldige und intensive Betreuung bedanken. Er stand uns während der kompletten Betreuungszeit mit Rat und Tat zur Seite. Ein weiterer besonderer Dank geht an Viktoria Becker, die ihre wertvolle und knappe Zeit mit dem Korrekturlesen unserer Arbeit verbracht hat. Außerdem hat sie uns zu besseren und wertvolleren Menschen reifen lassen. Abschließend danken wir allen Waschbären auf dieser Welt für ihre Existenz und grüßen des Weiteren noch unsere Familien.

Inhaltsverzeichnis

1 Einleitung

Aus unserer heutigen Zeit sind Smartphones und andere mobile Geräte, wie z.B. Tablets nicht mehr wegzudenken und erfreuen sich bei allen Generationen immer größerer Beliebtheit. Inzwischen besitzen rund 30 Millionen Deutsche ein solches Gerät [HealthOn, 2013], Tendenz steigend. Neben der Verwendung von Office-Applikationen stehen die Spiele bei den Anwendern besonders hoch im Kurs. Jeder dritte Spieler benutzt sein Smartphone zum Spielen und zieht dieses somit dem heimischen PC oder der Spielekonsole vor [BITKOM, 2013].Wirft man einen Blick in den App-Store des Marktführers Google (800 Tausend Apps Stand April 2013 [AndroidPit - Swetlana Soschnikow, 2013], so erkennt man schnell, dass Spiele-Apps zurzeit den Markt dominieren. Egal ob man nach den am häufigsten heruntergeladenen oder den am besten bewertetesten Apps sucht, Spiele sind immer vorne mit dabei. Die Gründe hierfür liegen im wahrsten Sinne des Wortes auf der Hand. Ein Smartphone ist in der Regel klein und kompakt, so dass es überall mit hingenommen werden kann. Dadurch hat der Benutzer die Möglichkeit zu jeder Zeit und überall seine Lieblingsspiele zu spielen und muss dafür nicht vor dem heimischen Bildschirm verweilen. Weitere Eigenschaften, die das Spielen auf einem mobilen Gerät interessant gestalten, sind die verschiedenen Sensoren, wie z.B. Neigungssensoren oder Standortermittlung, welche deutlich mehr Abwechslung und Flexibilität in die Spiele hinein bringen.

Die Möglichkeit der eigenen Standortermittlung sorgt in Kombination mit dem mobilen Internet, welches inzwischen in jedem aktuellen mobilen Gerät fest integriert ist, für eine neue Art von Spielen, die es in dieser Form zuvor noch nicht gegeben hat. Die Rede ist hierbei von sogenannten "Location-based Games", bei denen der Spielverlauf durch die Einbeziehung des Standortes verändert werden kann. Spiele wie "Geocaching"[Groundspeak, Inc., 2013] gehören zu dieser Kategorie und sind spätestens seit Foursquare [The Foursquare team, 2013] bei Smartphone Benutzern in aller Munde. Einschränkungen hierbei gibt es eigentlich nur durch die Spieler selbst, da eine gewisse Spielerdichte wünschenswert ist. Ansonsten gestaltet sich eine Mitspielersuche als schwierig, sofern die Spiele nicht für Einzelspieler ausgelegt sind. Aus diesem Grunde finden Location-based Games häufig in urbanen Gegenden wie Großstädten statt, welche dabei zu gemeinsamen Spielfeldern werden. Durch die Einbeziehung

dieser potenziellen Räume bieten diese Spieletypen jede Menge Abwechslung und jedes mal einen völlig neuen Spielverlauf, wenn man an einem neuen Ort spielt. Außerdem können diese Spiele den Benutzern eine veränderte Wahrnehmung auf bereits bekannte Gegenden bieten, da man zum Beispiel in Spielen wie Foursquare beliebte Locations viel eher findet, als auf einer gewöhnlichen Karte. Betrachtet man diese Kategorien von Spielen einmal genauer, so fällt auf, dass die meisten von ihnen untereinander recht ähnlich sind. Abgesehen von den grafischen Oberflächen unterscheiden sie sich in der Regel nur in der Art, wie gespielt wird, also in ihren Regeln, während die Spiele immer mit kontextbezogenen Daten arbeiten.
Eine Möglichkeit, die besagten Gemeinsamkeiten von Location-based Games zusammenzufassen und bei der Kreierung eines solchen Spieles nur einen geringen Teil an Änderungen vornehmen zu müssen, ist das Ziel dieser Arbeit. Dabei geht es insbesondere um die komfortable und einfache Erzeugung und Erweiterung der Spielregeln, welche ein jedes solcher Spiele besitzt.

2 Analyse

In diesem Kapitel werden die notwendigen Anforderungen an die hier thematisierte Software gestellt, sowie mögliche Anwendungsfälle eines Referenzbeispieles durchgespielt. Des Weiteren wird die Machbarkeit des Projekts analysiert und vergleichbare Projekte betrachtet. Abschließend wird das Kapitel in einem Fazit zusammengefasst.

2.1 Zielsetzung

Das Ziel dieser Arbeit ist es, eine Architektur für mobile Spiele im Bereich „Context-based Services“ zu konstruieren. Darunter versteht man die Einbeziehung von kontextsensitiven Daten aus der realen Welt des Benutzers. Hierzu zählen zum Beispiel Daten wie die aktuelle Position, Uhrzeit, Umgebungstemperatur oder auch die Entfernung zu anderen Benutzern. Insbesondere Positions-/Lokationsdaten sind für die hier thematisierte Arbeit interessant. Mithilfe dieser Architektur soll es möglich sein, mit wenig Aufwand, ein eigenständiges mobiles Spiel zu erstellen, welches sich lediglich an einige Rahmenbedingungen zu halten hat. Die so erzeugten Spiele arbeiten dabei mit den kontextbezogenen Daten ihrer Mitspieler wie beispielsweise Lokationen, Uhrzeit, soziale Interaktionen etc. Die Spieler spielen dabei mit- oder gegeneinander. Um dabei den programmatischen Aufwand beim Erzeugen eines neuen Spieles so gering wie möglich zu halten, gilt es dabei, die Architektur weitestgehend generisch aufzubauen, um die notwendigen Änderungen klein und übersichtlich zu halten. Damit eine solche Umsetzung von einem Spiel möglichst unkompliziert funktioniert, werden folgende Rahmenbedingungen an den Entwickler gestellt:
Das Spiel sollte von und mit realen Personen gespielt werden. Der Bereich der künstlichen Intelligenz wird von der Plattform nicht abgedeckt und müsste eigenständig implementiert werden. Auf Grundlage dieser Rahmenbedingungen soll es für den Entwickler problemlos und unkompliziert möglich sein, mobile kontextbasierte Spiele zu entwickeln. Darüber hinaus wäre es wünschenswert, wenn auch Personen mit keinen oder nur geringen Programmierkenntnissen die Möglichkeit bekämen, ein solches Spiel zu gestalten. Der Schwerpunkt liegt hierbei auf der Entwicklung von sogenannten Pervasive Games.

2.2 Pervasive Gaming

Die Aussage *„doing things for real“* [Jonsson und Waern, 2008, S. 225] kann als das zentrale Merkmal von Pervasive Games angesehen werden. Bei dieser Art von Spielen wird die reale Welt als eine gigantische, sich permanent verändernde Spielfläche angesehen, in welcher alles Erdenkliche für das Spielgeschehen verwendet werden kann. Ein konkretes Spiel findet für gewöhnlich außerhalb von Gebäuden auf vordefinierten Plätzen bzw. Spielflächen statt. Diese Spielflächen sind üblicherweise nicht für ein spezifisches Spiel ausgelegt. Spiele können zwischen und teilweise mit Personen stattfinden, welche nicht bewusst an diesem teilhaben und eher die Rolle eines Passanten einnehmen.

Heutzutage werden viele tägliche Aktivitäten, die in der realen Welt stattfinden, mit digitalen Bereichen wie Games, Chats, Web, etc. verbunden. Pervasive Games versuchen hierbei spielerische und tägliche reale Aktivitäten miteinander zu vermischen. Auch die Grenze zwischen Spielern und nicht Spielern wird verschleiert. Pervasive Games sind nicht von einer spezifischen Technologie abhängig, jedoch bietet die Verwendung von mobilen Geräten deutlich mehr Möglichkeiten und vereinfacht die Organisation solcher Spiele. Zusammenfassend kann gesagt werden, dass unter den Begriff Pervasive Games, alle Arten von Spielen fallen, welche in der realen Welt stattfinden wie z.B. Virtual Reality Games, Augmented Reality, Mobile Games oder Location-based Games, wobei der Fokus dieser Arbeit auf der zuletzt genannten Art liegt [Jonsson und Waern, 2008].
Um das Ganze zu verdeutlichen, findet sich nachfolgend ein Beispiel einer konkreten Spielrealisierung.

2.3 Fallbeispiel

Räuber und Gendarm

Das Grundprinzip basiert auf dem beliebten Kinderspiel „Räuber und Gendarm“. Dabei spielen zwei Spieler gegeneinander und nehmen dabei jeweils die Rolle des Jägers bzw. des Gejagten ein. Welcher Spieler welche Rolle einnimmt, wird dabei zufällig ausgewählt. Ein Spiel startet, wenn zwei Spieler sich lokationsmäßig in unmittelbarer Nähe zueinander befinden. Die Spieler haben jetzt, je nachdem welche Rolle sie erhalten haben, unterschiedliche Aufgaben zu bewältigen.

Spielziel des Gejagten: Der Gejagte muss dem Jäger so lange wie möglich entkommen. Dabei weiß er nicht, welche Person ihn jagt, sondern nur, dass er gejagt wird. Um es dem Gejagten nicht zu leicht zu machen, hat er verschiedene Bedingungen zu erfüllen. Beispielsweise muss er sich in einem begrenzten Bereich aufhalten. Diesen darf er während der Spielzeit nicht verlassen. Weiterhin bekommt er innerhalb eines zufälligen Zeitintervalls kleine Aufgaben vom Server zugewiesen. Beispielsweise muss er einen nahegelegen Ort (Point of Interest) erreichen. Der Gejagte erhält Punkte je länger er auf der Flucht ist.
Spielziel des Jägers: Der Jäger muss den Gejagten vor Ablauf einer bestimmten Zeit erwischen. Ein Spieler gilt als gefangen, wenn sein Jäger sich für eine bestimmt Zeit in seiner unmittelbaren Nähe (wenige Meter) befindet. Startet ein Spiel, so weiß der Jäger zunächst nichts über sein Ziel. Erst im Verlauf des Spiels werden ihm Informationen mitgeteilt. Als Informationen werden ihm beispielsweise die Entfernung sowie Richtung zum Ziel oder die vom Gejagten besuchten Orte (Point of Interest) gesendet. Um den Spielern während einer Spielsitzung die Möglichkeit einer unerwarteten Wendung zu geben, erhalten beide Seiten ausgewählte Fähigkeiten, mit welchen sie das Geschehen zu ihren Gunsten verändern können. So kann der Jäger z.B. für einen kurzen Zeitraum die Position seines Ziels auf der Karte sehen. Der Gejagte kann wiederum in brenzligen Situationen eine virtuelle Rauchbombe werfen, wodurch der Jäger in dieser Zeit keine Informationen erhält.

Die hier thematisierte Arbeit verwendet das zuvor gezeigte Location-based Game „Räuber und Gendarm“ als Referenzbeispiel und wird anhand von diesem die Machbarkeit der Plattform unter Beweis stellen.

2.4 Anforderungen an die Plattform

Im folgenden Kapitel werden die notwenigen Anforderungen an die Architektur genauer erläutert.

2.4.1 Architektur

Um ein passendes Designkonzept für die Plattform zu ermitteln, gilt es zunächst, die notwendigen Anforderungen an die Architektur zu analysieren. Die Spiele selbst sollen auf mobilen Geräten gespielt werden. Eine elementare Position sollte hierbei die Plattform einnehmen, welche als zentrale Einheit fungiert. Eine direkte Kommunikation zwischen den mobilen

Endgeräten ist nicht gewünscht da dies die bewusste Manipulation (Cheating) eines Spieles deutlich erhöht. Des Weiteren soll die Plattform über alle Ereignisse und Veränderungen der Endgeräte informiert werden, um diese behandeln zu können, wodurch eine direkte Kommunikation zwischen Plattform und den dezentralen Endgeräten vonnöten ist. Auf zentraler Ebene ist außerdem eine Datenpersistierung gewünscht, so dass Spielerattribute, wie Accountdaten oder Fortschritt dauerhaft gespeichert werden. Je nach Art des Spieles kann es darüber hinaus wichtig sein, gewisse Spielstände und Situationen zu persistieren.

In der Plattform selbst ist eine klare Trennung zwischen der verarbeitenden Logik-Ebene und der Persistierung erwünscht, damit bei einem möglichen Technologiewechsel keine Abhängigkeiten entstehen. Bei der Persistierung selbst ist es von großer Wichtigkeit, dass die Daten auch bei unvorhersehbaren Ereignissen, wie dem temporären Ausfall der Plattform, weiterhin dauerhaft gespeichert sind. Außerdem sollte gewährleistet sein, dass die Speicherung oder Aktualisierung von Daten entweder vollständig oder gar nicht ausgeführt wird, um so Fehlerfälle zu vermeiden.

2.4.2 Kommunikation

Wie bereits im vorigen Abschnitt „Architektur“ erwähnt wurde, soll die Kommunikation ausschließlich zwischen der zentralen und den dezentralen Komponenten stattfinden. Da die Übermittlung von Nachrichten in beide Richtungen eine der wichtigsten Rollen in solch einer Architektur einnimmt, ist es von größter Wichtigkeit hierfür eine optimale Technologie zu finden. Folgende Bedingungen sollte diese mindestens erfüllen:

- Bei Endgeräten mit einer mobilen Internetverbindung darf nicht davon ausgegangen werden, dass diese zur jeder Zeit erreichbar sind. Temporäre Verbindungsprobleme sind durchaus üblich, weshalb gewährleistet werden muss, dass die gesendeten Nachrichten bei einem solchen Szenario nicht verloren gehen, sondern so lange vorgehalten werden, bis sichergestellt ist, dass das Endgerät die Nachricht abrufen kann.

- Umgekehrt kann es bei einer hohen Anzahl an aktiven Endgeräten zu einer kurzzeitigen Überlastung der Plattform kommen, wodurch nicht alle Nachrichten unmittelbar von dieser verarbeiten und behandelt werden können. Hierfür müssen die Nachrichten ebenfalls vorgehalten werden, so dass sie von der Plattform nacheinander in der Reihenfolge, in welcher sie angekommen sind, abgearbeitet werden können.

- Ferner ist es wichtig, die eigentliche Kommunikation zu verschlüsseln, damit die Nachrichten nicht von Dritten gelesen werden können. Ebenso muss die Möglichkeit einer Autorisierung gegeben sein, damit eine genaue Spieleridentifizierung vorgenommen werden kann und sichergestellt ist, dass die Plattform ihre Daten an die richtige Person versendet.
- Es wäre ebenfalls von Vorteil, wenn die Nachrichten persistiert werden können, damit diese bei einer Spielanalyse für Statistiken und sonstige Auswertungen auch im Nachhinein verwendbar sind.

2.4.3 Die Spielelogik

Angesichts der Tatsache, dass die Plattform auch von Game-Designern, welche keine fundierten Kenntnisse im Bereich der Softwareentwicklung vorweisen können, genutzt werden soll, ist ein äußerst wichtiger Punkt die einfache und intuitive Erstellung von Spielelogiken bzw. Spielregeln auch von Nicht-Programmierern. Ein weiterer, ebenfalls nicht außer Acht zu lassender Punkt ist die komfortable Erweiterung sowie Pflege der bestehenden Spielregeln für die Nicht-Entwickler. Es gelten folgende Bedingungen, welche die Spielelogik erfüllen sollte.

Anforderungen

- Basierend auf Regeln
  - Die Spielelogik soll anhand von Regeln aufgebaut werden. Diese Regeln sind in der Form „Wenn ..., dann ...“ zu formulieren.
- Erweiterbarkeit
  - Ein Ziel der Spielelogik ist es, diese problemlos um weitere Regeln erweitern zu können.
- Flexibilität
  - Die Anpassung und Entfernung von bestehenden Regeln sind weitere Funktionen, für die die Spielelogik verantwortlich ist.
- Austauschbarkeit

  - Um die Möglichkeit einer bequemen Spieleerzeugung gewährleisten zu können, soll die Spielelogik innerhalb der Architektur komfortabel durch alternative Spielelogiken, welche auf anderen/neuen Spielregeln basieren, ersetzt werden können. Dafür muss die Architektur so gestaltet werden, dass die Spielelogik leicht und unkompliziert ausgetauscht werden kann.

2.4.4 Nichtfunktionale Anforderung

In diesem Kapitel werden die wichtigsten nicht funktionalen Anforderungen an die Architektur gestellt und in Beziehung zur Zielsetzung gesetzt. Unter nicht funktionale Anforderungen sind Merkmale zur Qualität eines Softwareprodukts zu verstehen [Balzert, 2009, S. 463-471].

- Zuverlässigkeit: Fehlertoleranz
  - Eine fehlertolerante Software strebt die Bewahrung ihres Leistungsstands auch bei Softwarefehlern oder bei Nichterfüllung der Schnittstellenspezifikation an. (Aufrechterhaltung der Leistung trotz Fehlfunktion/-verhalten).
  - Die Architektur sollte mit verloren gegangenen Endgeräten umgehen können und bestenfalls diese unbemerkbar wieder aufnehmen.
- Effizienz: Zeitverhalten
  - Dieses Qualitätsmerkmal wird von einer Software erfüllt, wenn sie unter den vereinbarten Bedingungen gerechtfertigte Antwort- und Verarbeitungszeiten, wie auch einen angemessenen Durchsatz einhält bzw. erbringt.
  - Die Kommunikation zwischen den Endgeräten und der Architektur findet hauptsächlich über das mobile Internet statt, daher besteht eine relativ hohe Wahrscheinlichkeit, dass die Endgeräte für eine gewisse Dauer oder permanent nicht erreichbar sind. Für diese Problematik sind auch Lösungen zu finden oder zu erstellen.
- Wartbarkeit: Änderbarkeit
  - Ein Softwareprodukt erzielt das Qualitätsmerkmal „Änderbarkeit“, indem es sich auf den erforderlichen Aufwand der Implementierung/Durchführung von Fehlerkorrekturen, Anpassungen und Verbesserungen bezieht.

  - Die Architektur sollte die Möglichkeit bieten, einfach und schnell der Spielelogik neue Regel hinzuzufügen, sowie bestehende Regeln zu entfernen oder anzupassen.

- Portabilität: Austauschbarkeit

  - Das Qualitätsmerkmal „Austauschbarkeit“ erlangt ein Softwareprodukt dadurch, dass es die Möglichkeit zur Verfügung stellt, seine Software durch andere spezifizierte Softwares mit demselben Ziel im gleichen Umfeld zu ersetzen.

  - Eine wichtige Anforderung an die Architektur ist die Austauschbarkeit der Spielelogik. Diese wird gefordert, da durch den alleinigen Austausch der Spielelogik ein neues Spiel entstehen sollte. Somit besteht die Option weitere Spiele lediglich durch den Austausch der Spielelogik zu entwickeln.

2.5 Anwendungsfälle/Beispiele

Abbildung 2.1 zeigt die Plattform für das Referenzbeispiel „Räuber und Gendarm“ in der Form eines Use-Case Diagramms in der gängigen UML-Notation. Der Akteur ist in diesem Fall ein normaler Spieler, welcher mit seinem mobilen Endgerät ein Spiel spielen möchte. Der Systemkontext wird durch den gesamten Server repräsentiert. Der erste Anwendungsfall ist Starten des Spiels starten. Dieser importiert zwei weitere Fälle. Der Erste ist die Anmeldung, da sich ein Spieler zunächst anmelden/authentifizieren muss, um ein Spiel spielen zu können. Anschließend wird nach einem passenden Mitspieler gesucht. Wurde ein geeigneter Partner gefunden, so kann das eigentliche Spiel starten. In dem Diagramm ist dieser Anwendungsfall unter dem Namen „Spielen“ zu finden. Auch dieser Fall importiert wiederum zwei weitere Fälle. Der Spieler sendet kontinuierlich seine aktuelle Position an den Server, damit dieser anhand der Position ein mögliches Spielende ermitteln kann. Wenn dieser Anwendungsfall eintritt, informiert der Server den Spieler darüber, dass das Spiel beendet wurde.

2.6 Hard- und Software Anforderungen

Dieses Kapitel befasst sich mit notwendigen Anforderungen, welche sowohl an die Plattform als auch an die Endgeräte gestellt werden. Hierbei liegt der Schwerpunkt insbesondere auf der Hard- und Software.

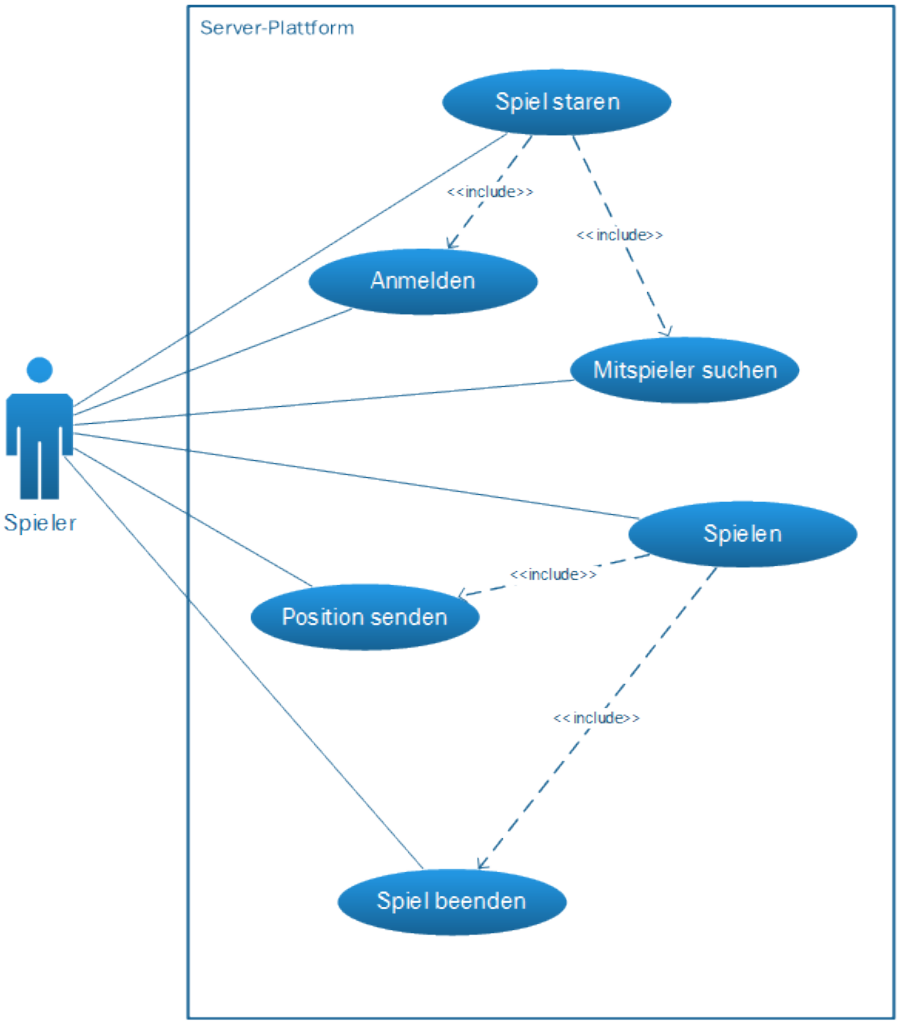


Abbildung 2.1: Plattform für das Referenzbeispiel als Use-Case Diagramm

2.6.1 Plattform

Wie bereits in den vorigen Kapiteln ausgiebig erläutert wurde, ist das Ziel, die Plattformarchitektur so generisch wie möglich zu halten. Dieser Ansatz soll auch im Bereich der Hardware umgesetzt werden, weswegen keine Festlegung auf einen spezifischen Plattformtypus besteht. Da der Ressourcenverbrauch je nach Art des Spieles stark variiert, kann eine generelle Aussage über die benötigte Hardware der Plattform ohnehin nicht getroffen werden, sondern muss für jede spezifische Lösung individuell analysiert und geprüft werden. Es sollten lediglich folgende Grundbedingungen erfüllt sein:

- Es wird eine Internetverbindung über einen Breitbandanschluss benötigt, um einen akzeptablen Datendurchsatz zu erzielen. Die genaue Geschwindigkeit ist allerdings auch hier wieder von der Art des Spieles abhängig.
- Ein modernes Betriebssystem wird vorausgesetzt, welches Software, die mittels Programmiersprachen höhere Abstraktionsebenen geschrieben wurden, ausführen kann.

Aufgrund der gestellten Anforderung bietet es sich an, für die Entwicklung eine Programmiersprache zu verwenden, die plattformunabhängig entwickelt und ausgeführt werden kann.

2.6.2 Engeräte

Die Entwicklung der Endgerät-Software geht in der Regel aus einer konkreten Spielidee hervor und folgt nicht oder nur bedingt dem generischen Ansatz. Folgende Minimalbedingungen sollten die Endgeräte allerdings erfüllen:

- Um ein Pervasive Game für ein mobiles Gerät zu entwickeln, ist es unabdingbar, dass dieses über einen GPS-Empfänger verfügt, welcher die aktuelle Geoposition ermittelt.
- Ebenso wird mobiles Internet vorausgesetzt, da die Verbindung zwischen der Plattform und den Endgeräten über das Internet realisiert wird.
- Da es bei Verbindungen über das mobile Netzwerk mitunter zu Latenzschwankungen kommen kann, muss eine Möglichkeit der Uhrensynchronisation der Geräte gewährleistet sein, damit die Plattform erkennen kann, welches Endgerät die Nachricht zu welchem Zeitpunkt versendet hat.
- Um eine Chancengleichheit der Spieler bei Verwendung unterschiedlicher Endgeräte zu gewährleisten, sollte sich auf ein einheitliches Bedienkonzept festgelegt werden. Bei Analyse der aktuell auf dem Markt verfügbaren Geräten, ist hier die Touchpadbedienung am naheliegendsten.

Da die Anbindung zwischen zentraler und dezentraler Komponente über eine neutrale Schnittstelle angeboten werden soll, ist die Wahl des der Gerätehardware nicht auf eine bestimmte Programmiersprache beschränkt, wodurch es möglich ist, für alle aktuellen Anbieter mobiler Geräten wie Android, iOS, oder auch Windows-Phone, Anwendungen zu entwickeln. Prinzipi-

ell ist jedes mobile Betriebssystem möglich, sofern die oben genannten Bedingungen erfüllt werden können.

2.7 Vergleichbare Plattformen

Die Recherchen haben ergeben, dass bereits viele Context-based Games für Smartphones auf dem Markt erhältlich sind. Eines der Erfolgreichsten ist das Spiel Foursquare, bei welchem die Benutzer mithilfe ihres mobilen Gerätes an einem realen Ort „einchecken“ (angeben, sich dort „real“ zu befinden) können und dafür Punkte erhalten. Bei der Suche nach einer generischen Serverarchitektur, wie sie in dieser Arbeit realisiert werden soll, kam das Framework „Photon“ den gestellten Anforderungen am nächsten. Photon ist ein Framework für Multiplayer-Spiele jeglicher Art und stellt dem Programmierer ein Grundgerüst für die Erzeugung eines Spieleservers zur Verfügung. Ebenfalls kümmert sich Photon um die Verbindung und den Nachrichtenaustausch zwischen dem Server und den Clients [Exit Games, 2013b]. Die Abbildung 2.2 zeigt die grobe Architektur von Photon:

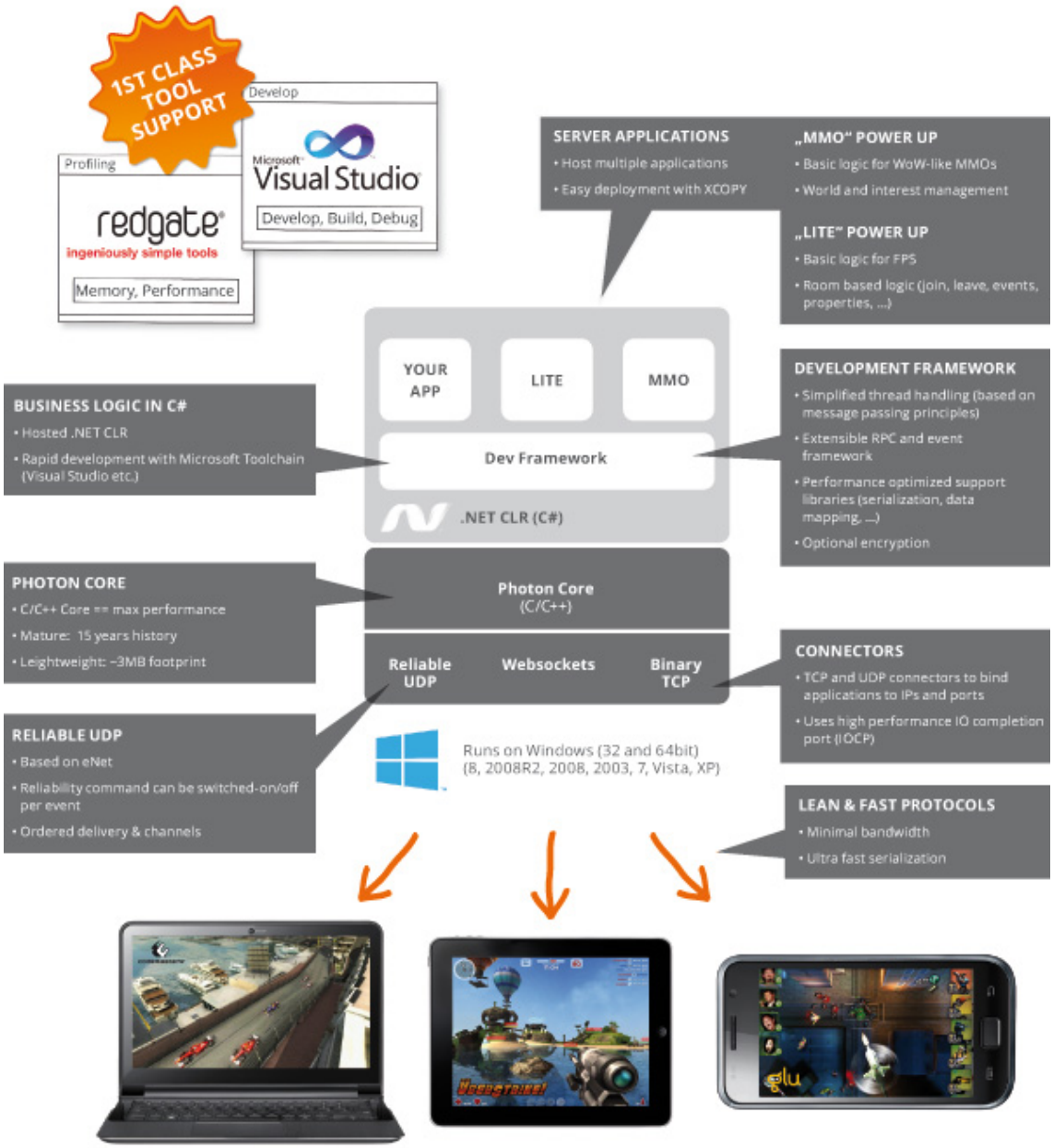


Abbildung 2.2: Photon-Architektur [Exit Games, 2013a]

Wie bei den anderen gefundenen Frameworks auch, ist es bei Photon allerdings notwendig, die Spielelogik mit einer konkreten Programmiersprache (bei Photon C#) umzusetzen. Ein generischer Ansatz mittels einem regelbasierten System, wie in der Analyse gefordert, ist hier nicht vorgesehen. Des Weiteren legt der Photon-Server seinen Schwerpunkt auf eine

hohe Skalierbarkeit, die für Spiele wie „Massively Multiplayer Online Games“ notwendig sein kann, allerdings für Context-based Games nur bedingt von Bedeutung ist. Es bleibt also festzuhalten, dass zwar bereits mehrere verfügbare Frameworks für die Entwicklung einer eigenen Game-Server existieren, jedoch keines den genauen Ansprüchen dieser Analyse gerecht wird.

2.8 Fazit

Nach der in diesem Kapitel durchgeführten Analyse wurde eine zentrale Architektur für die Plattform gewählt, mit welcher die dezentralen Endgeräte kommunizieren. Eine direkte Kommunikation zwischen den Geräten ist nicht vorgesehen. Die Anforderungen haben gezeigt, dass keine außergewöhnlichen Hardware-Anforderungen vonnöten sind und die eigentlichen Spiele für nahezu jede, heutzutage aktuelle, mobile Hardware entwickelt werden kann. Die Analyse vergleichbarer Plattformen hat gezeigt, dass zwar bereits ähnliche Projekte auf dem Markt existieren, jedoch keine von ihnen den Schwerpunkt des hier thematisierten generischen Ansatzes verfolgt. Das nachfolgende Kapitel wird sich mit der konkreten Umsetzung einer passenden Architektur befassen und dabei, erneut das Spiel „Räuber und Gendarm“ als Referenzbeispiel verwenden.

3 Design & Realisierung

Das folgende Kapitel befasst sich mit dem Design und der Realisierung von der zuvor analysierten Plattform. Einleitend wird mit der Erzeugung einer passenden Architektur begonnen, welche sich aus den zuvor ermittelten Anforderungen ergibt. Anschließend wird die erzeugte Architektur anhand des aus Kapitel 2.3 bekannten Referenzbeispieles in Form von Sequenzdiagrammen dargestellt. Des Weiteren wird darauffolgend näher auf die zu verwenden Technologien eingegangen.

3.1 Architektur

Dieser Abschnitt zeigt und beschreibt den Aufbau und die Struktur der geplanten Plattformarchitektur. Nach eingehender Betrachtung der Analyse wurde sich für die Wahl einer Client-Server-Architektur entschieden, da diese alle notwendigen Anforderungen erfüllt und sich für eine Kommunikation zwischen diversen Endgeräte mit der Plattform am besten anbietet. Wie in Abbildung 3.1 zu sehen ist, besteht das Serversystem (Plattform) aus vier Hauptkomponenten, welche abgesehen von dem Databasecontroller eigenständig agieren und jeweils auf unterschiedlichen Hardware-Geräten laufen können. Die Nachrichtenübertragung zwischen den Komponenten wird, wie auch zwischen weiteren externen Übertragungen, durch Message-Queues realisiert (siehe Kapitel 3.4). Bei der Wahl der Programmiersprache für die Plattformentwicklung wurde Java gewählt, da diese durch ihre Plattformunabhänigkeit den größtmöglichen Gestaltungsfreiraum bietet. So gibt es beispielsweise bei der Hardware- und Betriebssystemauswahl keinerlei Einschränkungen und es kann für Testzwecke bereits auf vorhandenes Material zurückgegriffen werden. Weitere Punkte, die für Java sprechen, sind ihre kostenlose Verfügbarkeit und die damit verbundene hohe Verbreitung.

Die **Authentication** Komponente bietet Clients die Möglichkeit an, einen Account zu erzeugen und sorgt für eine sichere Authentifizierung zwischen Server und Benutzer. Die anderen

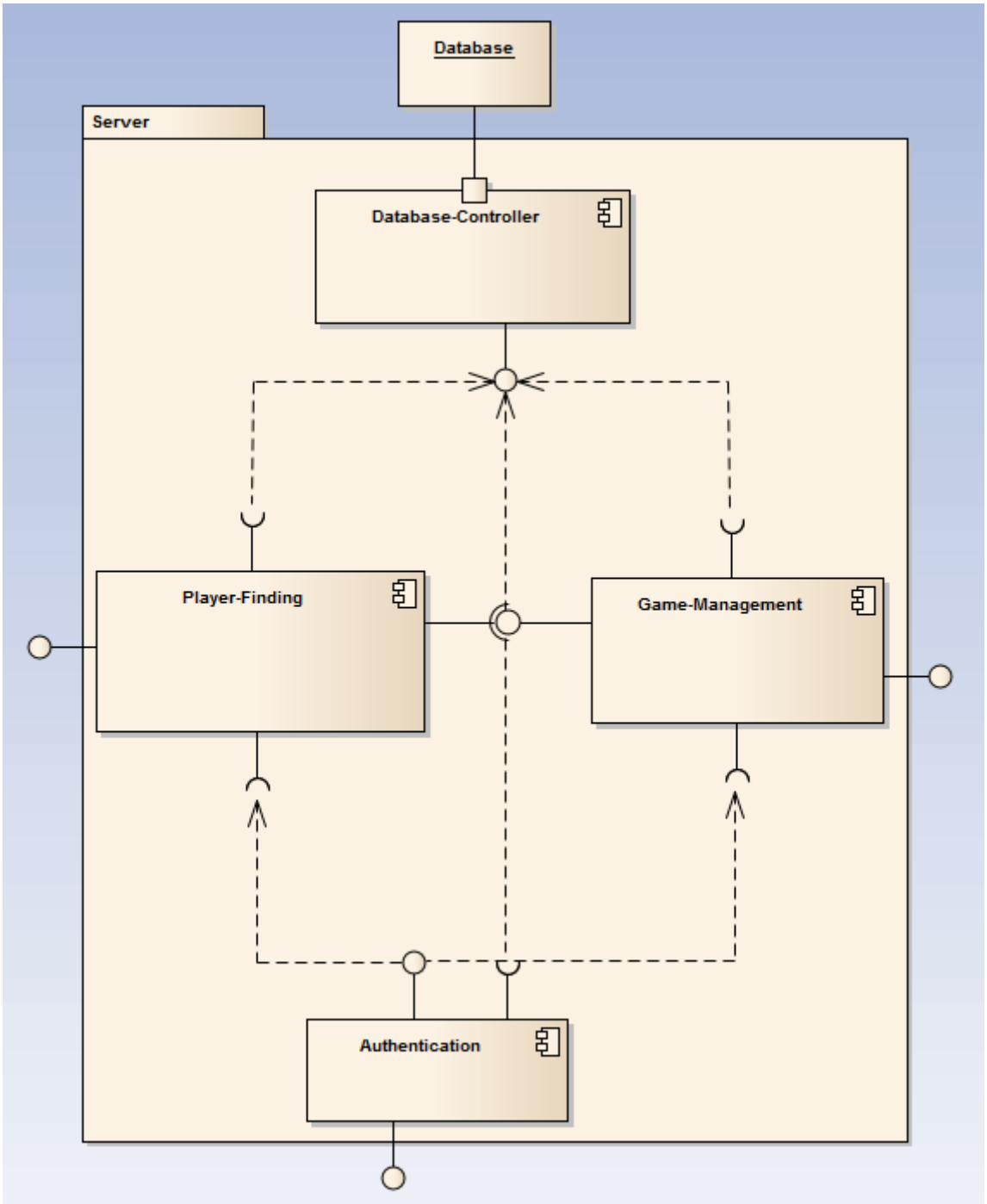


Abbildung 3.1: Plattformarchitektur

Komponenten haben die Möglichkeit, bei der Authentication die Rechte eines spezifischen Nutzers für ein Spiel zu erfragen.

Authentifizierte Clients senden ihre Bereitschaft zu spielen an die **Player-Finding** Komponente, die einen oder mehrere passende Mitspieler auswählt, sofern welche vorhanden sind. Wurden genügend Spieler gefunden, so übermittelt die Player-Finding Komponente eine Nachricht an das Game-Management, damit dort ein neues Spiel gestartet werden kann.

In der**Game-Management** Komponente findet das eigentliche Spiel statt. Sie arbeitet unabhängig von dem Player-Finding, so dass die Clients nur noch mit dem Game-Management kommunizieren, sobald sie sich in einem laufenden Spiel befinden.

Der **Database-Controller** wird von den anderen Komponenten für den Datenzugriff verwendet. Er wird als Plugin von diesen eingebunden, wodurch ein direkter Zugriff möglich ist.

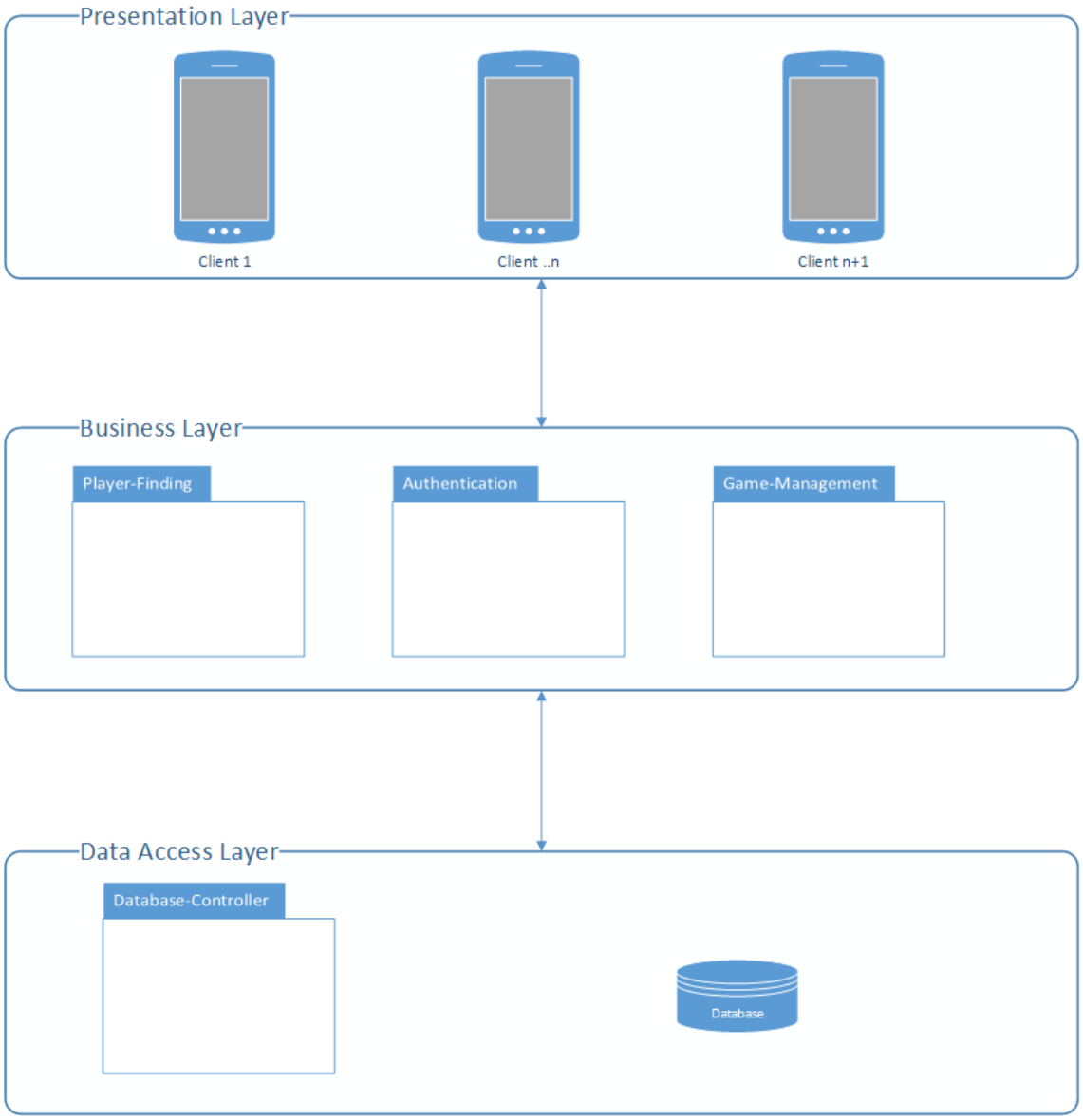


Abbildung 3.2: Architektur als Drei-Schichten-Modell

Die Abbildung 3.2 zeigt die hier dargestellte Architektur noch einmal in Form des Drei-Schichten-Modelles. Die Präsentationsschicht wird hierbei ausschließlich über die Endgeräte der Clients repräsentiert, da für den Server keine GUI-Komponente vorgesehen ist. Das Herzstück des Servers befindet sich in der Business-Schicht, welche alle Komponenten, bis auf

den Database-Controller beinhaltet. Da der Database-Controller für den direkten Zugriff auf die Datenbank zuständig ist, befindet er sich gemeinsam mit dieser in der Data-Access-Schicht.

3.1.1 Autentication Komponente

Im folgenden Abschnitt wird die Authentification Komponente näher erläutert. Die Authentification Komponente dient als zentrale Authentifikations- und Autorisationsverwaltung, sowie als zentrale Benutzerverwaltung der Architektur. Da diese Problematik nicht Schwerpunkt dieser Arbeit ist, wird ausschließlich auf die einzusetzende Technologie eingegangen und nicht auf die konkrete Umsetzung.

Bei der einzusetzenden Technologie wurde sich für LDAP (Lightweight Directory Access Protocol) entschieden. Zur konkreten Umsetzung käme die OpenSource Implementation „openLDAP“ zum Einsatz. Die Entscheidung fiel auf LDAP, da es sich in der Praxis für solche Problematiken etabliert hat. Außerdem weist LDAP bei der Authentifikation sowie bei der Autorisation seine Stärken und sein Hauptanwendungsschwerpunkt auf. Unter anderem bieten zahlreiche Technologien die Unterstützung des LDAP-Protokolls an [LDAP, 2013].

3.1.2 Player-Finding Komponente

In Abbildung 3.3 ist der Aufbau der Player-Finding Komponente, die aus drei weiteren Unterkomponenten besteht, zu sehen. Der zentrale Einstiegspunkt für die bereits authentifizierten Clients ist der **Dispatcher**. Daten, die nicht aus der Datenbank gelesen werden können und nur vom Client selbst bekannt sind, wie beispielsweise Lokationsdaten, werden dabei an den Server mitgesendet.

Der Dispatcher nimmt alle Clientanfragen bezüglich eines Spielwunsches entgegen und leitet diese an einen eigens dafür zuständigen Thread der **Pairing** Komponente weiter. In dieser Komponente werden als erstes die aktuellen, vom Client übermittelten Informationen in der Datenbank persistiert. Dies geschieht durch den als Plugin eingebundenen Database-Controller. Anschließend wird mithilfe eines sogenannten Feature-Vektors ein oder mehrere geeignete Mitspieler (Anhängig vom speziellen Spiel) ermittelt. Die Pairing-Komponente erhält hierbei die aktuellen Informationen über weitere, ebenfalls nach Mitspielern suchende, Clients von dem Database-Controller. Sollte für den aktuellen Client kein passender Mitspieler gefunden

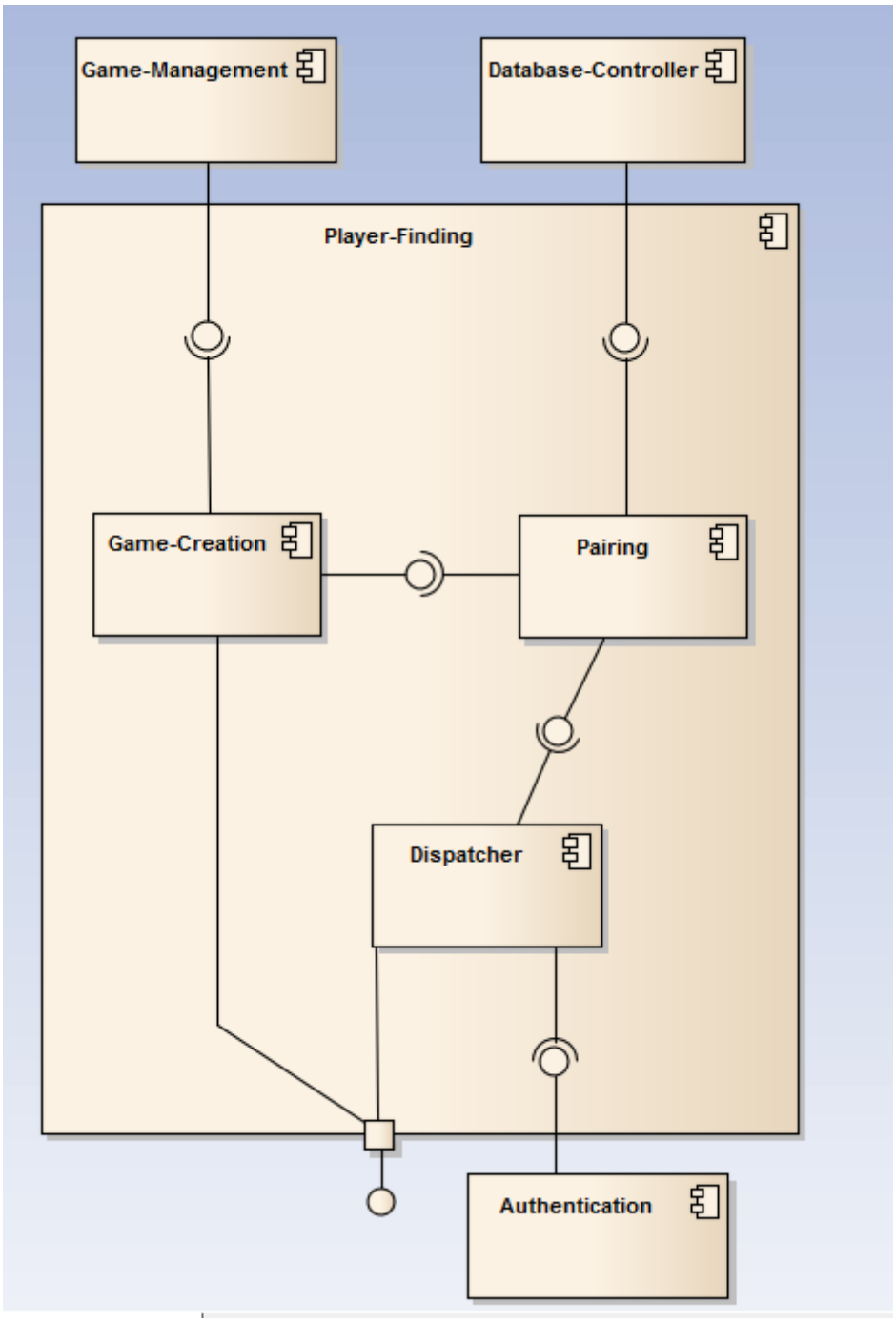


Abbildung 3.3: Aufbau der Player-Finding Komponente

werden, so beendet sich der Pairing Thread automatisch und es wird ein neuer gestartet, sobald der Client eine Aktualisierung seiner Daten an den Dispatcher versendet hat.

Wurden einer oder mehrere passende Mitspieler gefunden, so werden diese an den **Game-Initializer** weitergeleitet. Dieser stellt noch einmal eine Anfrage an jeden der gefundenen Clients, um sicher zu gehen, dass diese noch immer auf ein Spiel warten und sich nicht zwischenzeitlich beendet haben oder aus sonstigen Gründen nicht mehr zu Verfügung stehen. Sind alle Clients bereit, so sendet die Game-Initializer Komponente eine Nachricht an das Game-Management und teilt diesem mit, dass ein neues Spiel für die übermittelten Spieler erzeugt werden soll. Die Aufgabe der Player-Finding Komponente ist hiermit beendet und der Game-Initializer beendet sich selbstständig.

3.1.3 Game-Management Komponente

Die Game-Management Kompomente kümmert sich um die Verwaltung und Umsetzung von laufenden Spielen. Wie in Abbildung 3.4 erkennbar ist, besteht sie aus zwei, bzw. fünf weiteren Unterkomponenten.

Ebenso wie die Player-Finding Komponente besitzt das Game-Management einen **Dispatcher**. Dieser besitzt eine externe Schnittstelle, welche jedoch nur vom Player-Finding und nicht von den Clients angesprochen werden kann.

Der Dispatcher erzeugt bei jeder Anfrage einen eigenen Thread der **Game-Logic** Komponente. Diese besteht wiederum drei eigenen Unterkomponenten. Die Unterkomponenten können dabei sowohl auf den Database-Controller zugreifen, als auch auf die Authentication Komponente, um gegebenenfalls die Berechtigungen bestimmter Clients prüfen zu können.

Eine von ihnen ist die **Game-Creation**. Sie ist für die Vorbereitung eines Spiels verantwortlich. Je nach Art des Spiels teilt sie Spieler in Gruppen ein und weist ihnen zum Beispiel Gegenstände oder ähnliches zu. Nach der Berechnung dieser Vorverarbeitung wird ein Spiel gestartet und die notwendigen Informationen zu dem Spiel werden an die Clients weitergeleitet.

Dafür ist die **Communication-Manager** Komponente zuständig. Sie verfügt als einzige über eine externe Schnittstelle, die auch von den Clients verwendet werden kann und steuert die Client-Server Kommunikation in beide Richtungen. Es gehört ebenfalls zur Aufgabe des Managers, ein Spiel zu beenden, wenn eine Seite keine Daten mehr an den Server sendet. Erste Tests haben ergeben, dass je nach Anbieter Latenzen von 500-900 ms durchaus üblich sind, weshalb

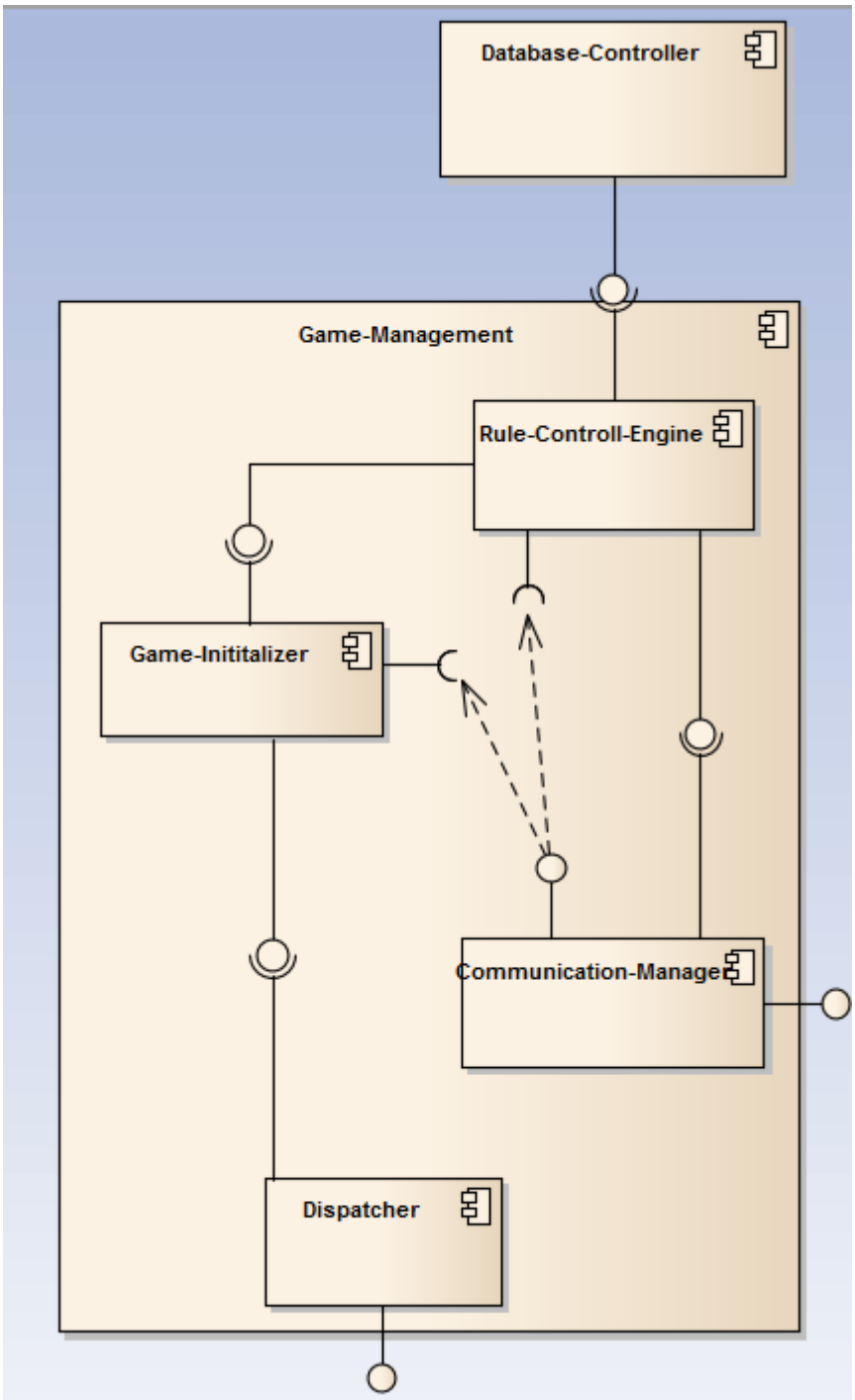


Abbildung 3.4: Aufbau der Game-Management Kompomente

jedem Client eine Toleranzzeit eingeräumt werden sollte, ehe die eingehenden Nachrichten weitergeleitet werden. Um Aufgrund der besagten Latenzschwankungen trotzdem ermitteln zu können, welche Nachricht in welcher Reihenfolge versendet wurde, ist es unabdingbar, dass die Clients einen untereinander zeitlich synchronisierten Zeitstempel mitsenden, welcher an dieser Stelle verarbeitet werden kann.

Erhält die Komponente Nachrichten von den Clients wie beispielsweise Daten, die für den Spielverlauf relevant sein könnten, so gibt sie diese an die **Rule-Controll Engine** weiter. Diese Komponente steuert alle spielrelevanten Entscheidungen. Sie verfügt über ein regelbasiertes System, mit welchem die konkreten Regeln in einer möglichst formalen Sprache definiert werden.

3.1.4 Database-Controller

Der Database Controller bietet den anderen Komponenten einen einfachen und konfortablen Zugriff auf die Datenbank. Er sorgt für die Verbindung zum Datenbankmanagementsystem und ist für ein internes objektrelationales Mapping der Datenbank zuständig, wodurch den Komponenten eine übersichtliche Schnittstelle für den Datenbankzugriff angeboten wird.

3.1.5 Authentifikation eines Spielers

Das Sequenzdiagramm 3.5 beschreibt die Authentifikation eines Spielers an der Plattform. Voraussetzung für eine erfolgreiche Authentifikation ist eine erfolgreiche Registrierung des Spielers. Die Kommunikation zwischen Spieler (Client) und Plattform (Server) erfolgt, wie bereits erwähnt, über Message-Queues.

An dem Szenario sind folgende Komponenten beteiligt: Spieler (Client), Authentifikation Komponente, Login-Queue und Player-Input-Queue. Zuerst sendet der Spieler eine asynchrone Login-Nachricht mit seinem Benutzernamen und Passwort an die Login-Queue. Währenddessen wartet die Authentifikation Komponente auf einkommende Login-Anfragen eines Clients. Sobald eine Login-Anfrage bereitsteht, liest die Authentifikation Komponente diese aus der Queue heraus und verarbeitet diese. Während der Anfrageverarbeitung prüft die Authentifikation Komponente, ob bereits ein Spieler mit dem Benutzernamen existiert. Andernfalls bricht der Authentifikationvorgang ab und eine „Authentifikation fehlgeschlagen“-Nachricht wird von der Authentifikation Komponente in die Player-Input-Queue versendet. Diese Queue

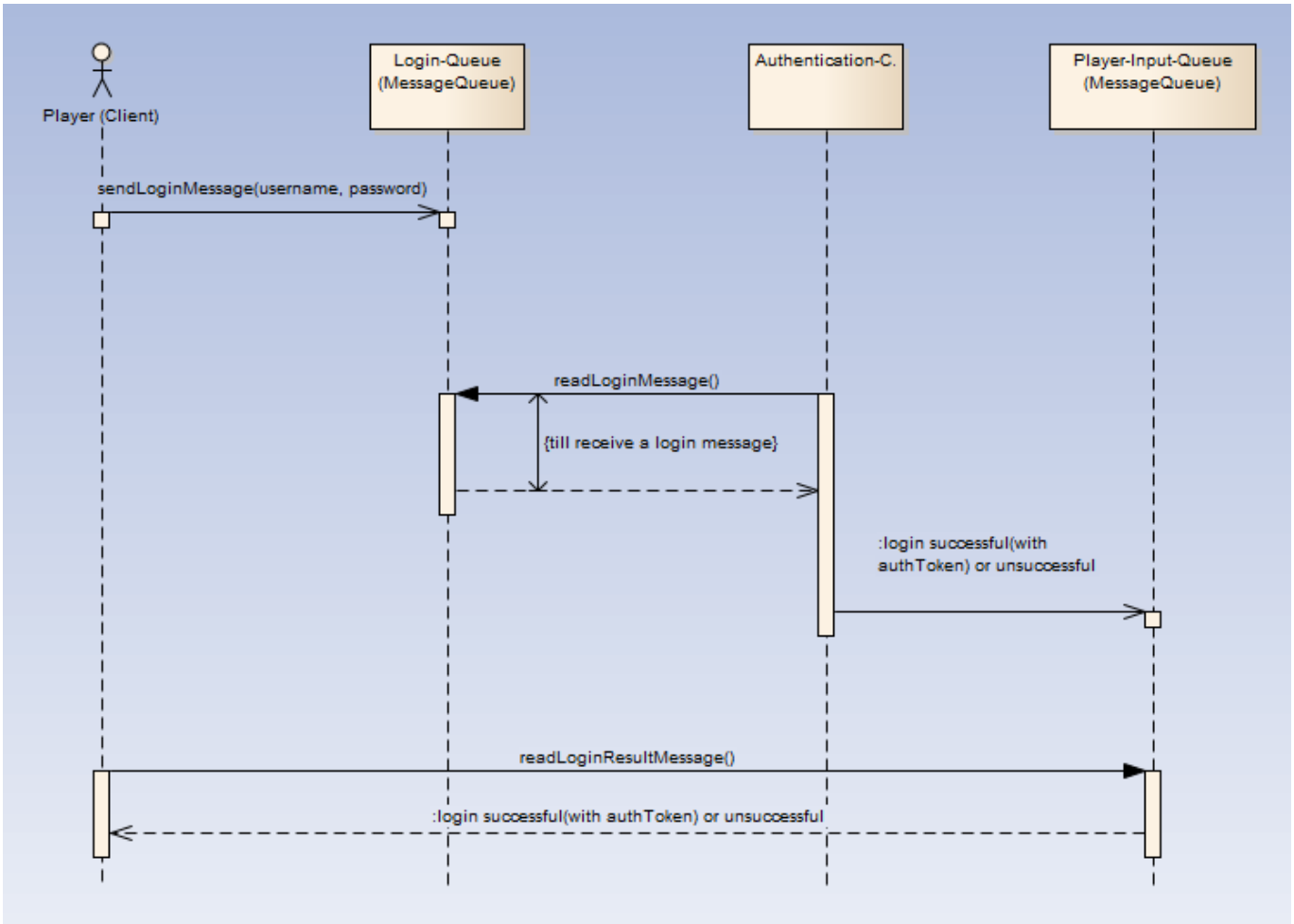


Abbildung 3.5: Authentifikationsablauf als Sequenzdiagramm

wird vom entsprechenden Client regelmäßig auf Nachrichten geprüft. Nach erfolgreicher Existenzprüfung des Benutzernamens prüft die Authentifikation Komponente das übersendete Passwort mit dem des zugehörigen Spielers auf Äquivalenz. Bei Erfolg sendet sie eine Nachricht mit Authentifikation-Token in die Player-Input-Queue, ansonsten wird eine „Authentifikation fehlgeschlagen“-Nachricht versendet.

3.1.6 Spielerfindung

Das in Abbildung 3.6 dargestellte Sequenzdiagramm beschreibt den Ablauf innerhalb der Spielerfindungskomponente. Annahme dieses Szenarios ist es, dass der Client sich bereits im

Vorfeld erfolgreich über die Authentifizierungskomponente angemeldet und authentifiziert hat.

Möchte ein Spieler ein Spiel bestreiten, so sendet er dazu eine Nachricht an die ReadyToPlay-Queue. Seine Nachricht beinhaltet dabei spielspezifische Informationen, die gegebenenfalls notwendig sind, um geeignete Mitspieler zu finden. Für das Referenzbeispiel „Räuber und Gendarm“ wären dies beispielsweise Lokationsdaten. Diese Informationen werden in eine Datenstruktur verpackt und sind im Diagramm unter der Bezeichnung „playerObject“ zu finden. Auf Seiten des Servers prüft der Dispatcher, ob sich neue Nachrichten in der ReadyToPlay Queue befinden und holt diese, falls vorhanden, heraus. Anschließend startet er einen neuen Thread der Pairing Komponente, welche das playerObject des suchenden Clients übergeben bekommt. Während der Dispatcher sich weiter um die eingehenden Nachrichten kümmert, versucht die Pairing Komponente nach geeigneten Mitspielern zu suchen. Damit der aktuell suchende Spieler auch von anderen Pairing Komponenten-Threads gefunden werden kann, wird sein playerObject zunächst in der Datenbank persistiert. Dazu sendet die Pairing Komponente eine Nachricht an den Database-Controller (storePlayerWhoIsReadyToPlay(playerObject)) und wartet auf eine positive Rückmeldung des Controllers. Nachdem die Daten persistiert wurden, holt sich die Pairing Komponente eine Liste von allen ebenfalls wartenden Mitspielern. Dies geschieht ebenfalls über den Database-Controller (getAllWaitingPlayers()). An dieser Stelle ist es durchaus sinnvoll, bereits eine Vorauswahl an wartenden Spielern zu treffen, um so nicht jedes Mal die komplette Liste aller Spieler zu erhalten, jedoch ist dies nicht Teil der vorliegenden Arbeit.

Hat die Pairing Komponente eine Liste wartender Spieler erhalten, so gilt es jetzt, den idealsten Mitspieler zu ermitteln. Dies geschieht mit einem sogenannten Feature-Vektor, mit dessen Hilfe der/die passendste/n Mitspieler anhand verschiedener Werte ermittelt werden kann/können. Die verschiedenen Werte repräsentieren bestimmte Merkmale bzw. Fähigkeiten der Spieler. Darunter sind zum Beispiel folgende Merkmale bzw. Fähigkeiten zu verstehen:

- Der Erfahrungslevel der Spieler
- Die Distanz zwischen den Spielern (basierend auf deren Geolokation)
- Gemeinsamkeiten unter den Spieler (z. B. selbe Hobbies, ähnliche Interessen)

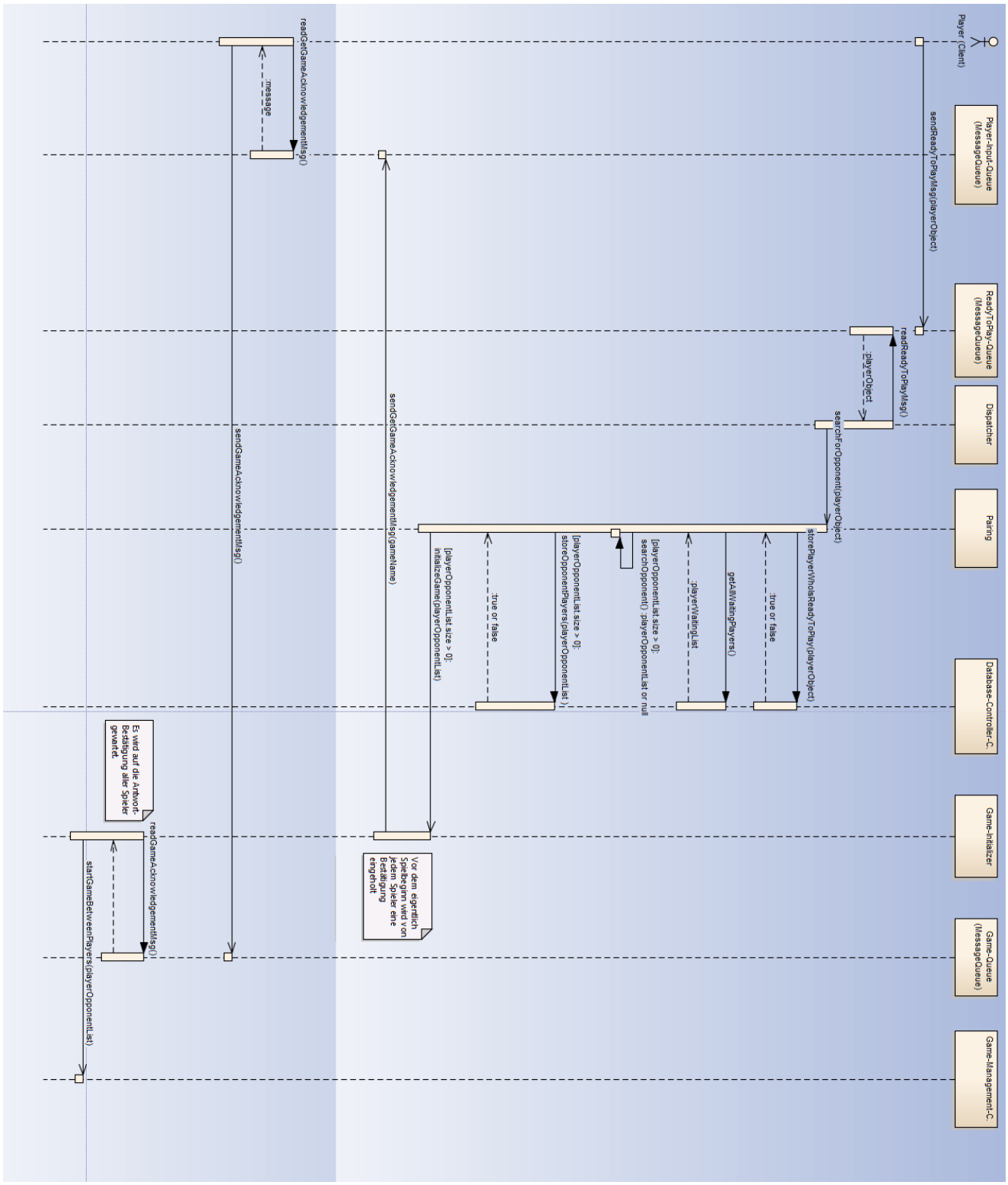


Abbildung 3.6: Spielerfindungsablauf als Sequenzdiagramm

Der folgende Term zeigt die verwendete Formel zur Berechung des Feature-Vektors.

$$\sum_{i=1}^{n} G_i * |x_i - y_i|$$

Zunächst wird die Differenz zu jedem Merkmal und jeder Fähigkeit der Spieler gebildet ($x_i - y_i$) und anschließend der Betrag aus der jeweiligen Differenz gezogen ($|x_i - y_i|$). Nachkommend wird der herausgezogene Betrag mit einer festgelegten Gewichtung multipliziert und insgesamt aufsummiert. Durch die Gewichtung ist die Wichtigkeit jedes Merkmals individuell anpassbar. Die unterschiedlichen Gewichtungen sind hierbei im $\hat{G}$-Gewichtungsvektor definiert. Der Spieler, der den höchsten aufsummierten Wert erzielen würde, biete sich als idealster Mitspieler an.

Die Suche ist im Diagramm unter der Schleifenfunktion „searchOpponent()“ zu sehen. Wurde ein oder mehrere passende/r Mitspieler gefunden, so wird diese Information zunächst in der Datenbank festgehalten (storeOpponentPlayers(playerOpponentList)). Das Persistieren dieser Information ist notwendig, da aller Wahrscheinlichkeit nach die Threads der gefunden Mitspieler ebenfalls ein identisches Ergebnis in ihrer Suche erzielt haben, allerdings nur einer der Threads mit der Weiterverarbeitung des Suchergebnisses fortfahren sollte. Sollte ein anderer Thread bereits eine Eintragung zu einem der gefundenen Spieler in der Datenbank hinterlegt haben, so gibt der Database-Controller ein „false“ zurück und die Arbeit für die Pairing Komponente ist an dieser Stelle beendet. Andernfalls sendet sie eine Nachricht an den Game-Initializer und überreicht diesem die Liste mit den Spielern, die zuvor gefunden wurden (inklusive dem ursprünglich suchenden Spieler). Zu sehen unter initializeGame(playerOpponentList). Spätestens jetzt ist die Arbeit für den Pairing Komponenten-Thread beendet. Der Game-Initializer fordert als erstes eine Bestätigung von jedem Spieler in der an ihn übergeben Liste an, um zu prüfen, ob diese noch immer gewillt sind ein Spiel zu bestreiten, da es unter Umständen passieren kann, dass Spieler noch für eine Zeit lang als suchend in der Datenbank gelistet sind, aber bereits nicht mehr zur Verfügung stehen. Dazu sendet er eine Nachricht an die Entsprechenden Queues der Spieler (sendGetGameAcknowledgementMsg(gameName)). Jeder Spieler verfügt hierbei über eine eigene MessageQueue aus der nur er lesen kann. Die Spieler versenden anschließend eine Nachricht an den Game-Initializer zurück, indem sie ihre Nachricht in eine, für den Game-Initializer bekannte „Game-Queue“-MessageQueue senden (sendGameAcknowledgementMsg()). Hat der Initializer von allen Clients eine Antwort erhalten, so sendet er eine Nachricht an die Game-Management-Komponente (startGameBetweenPlayers(playerOpponentList)), die sich um den eigentlichen Spielstart kümmert. Erhält

der Game-Initializer innerhalb eines gewissen zeitlichen Intervalls nicht von jedem Client Antwort, so gilt der Spielstart als gescheitert und es kommt kein Spiel zustande.

3.1.7 Spielablauf

Die nachkommenden Sequenzdiagramme stellen den Ablauf eines Spiels im Inneren der Game-Management-Komponente dar. Für dieses Szenario wird vorab eine erfolgreiche Spielerfindung vorausgesetzt. Der Ablauf ist auf drei Sequenzdiagramme aufgeteilt. Das erste Diagramm zeigt die Zuordnung der Spielerrollen der Spieler. Das zweite Sequenzdiagramm stellt das Aktualisieren des „playerObjects“, die Ausführung der Rule-Engine sowie die Aufforderung einer Spieleraktion dar. Das dritte Diagramm beschreibt das Ende eines Spiels.

1. Sequenzdiagramm – Spielablauf

Als erstes wird die Abbildung 3.7 vorgestellt, welche die Zuordnung der Spielerrollen als Sequenzdiagramm aufzeigt.

Nach einer erfolgreichen Spielerfindung wird die Game-Management-Komponente von der Player-Finding-Komponente zum Erstellen eines neuen Spiels angestoßen. Dafür sendet die Player-Finding-Komponente eine Nachricht mit den Spielern („startGameBetweenPlayers (playerOpponentList)“). Innerhalb der Game-Management-Komponente wird die Nachricht von der Dispatcher-Komponente verarbeitet. Diese sorgt dafür, dass ein neuer Thread gestartet wird, welcher die Nachricht („startGameBetweenPlayers(playerOpponentList)“) an die Game-Creation-Komponente weiterleitet. Währenddessen arbeitet die Dispatcher-Komponente weiterhin Spielerstellungsanfragen ab. Anschließend gibt die Game-Creation-Komponente die Spielerliste an die Rule-Control-Engine-Komponente weiter („decidePlayerRoleAssignments (playerList)“). Daraufhin „feuert“ die Rule-Control-Engine-Komponente die Regeln zur Spielerrollenzuordnung ab („fireRuleForRoleAssignment(playerList)“). Diese Regeln können individuell für jedes Spiel angepasst werden. Als Ergebnis folgt die Spielerrollenliste, welche dieselbe Reihenfolge wie die der Spielerliste aufweist. Als nächstes wird die Spielerrollenliste an die Game-Creation-Komponente zurückgegeben. Diese fordert anschließend den Communication-Manager zum Versenden der Spielerrollen an die Clients auf. Der Communication-Manager verschickt nun die jeweilige Spielerrolle an die dementsprechende „Player-Input-Queue“-MessageQueue des Spielers („sendPlayerRoleAssignmentMsg(role)“). Im nächsten Schritt

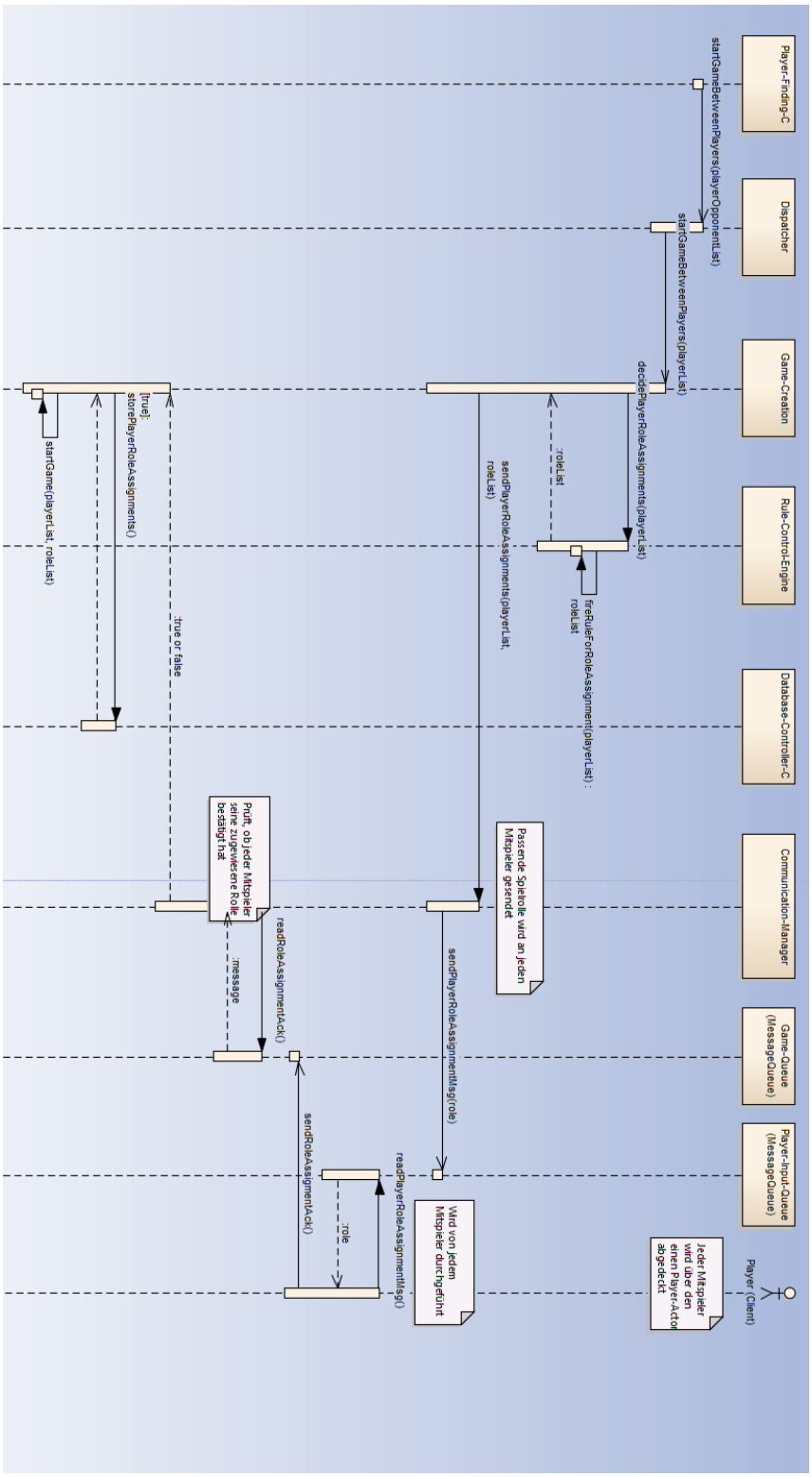


Abbildung 3.7: 1. Sequenzdiagramm – Spielablauf

liest jeder einzelne Mitspieler die ihm zugewiesene Spielerrolle aus seiner „Player-Input-Queue“-MessageQueue aus („readPlayerRoleAssignmentMsg()“) und entscheidet, ob ihm die zugewiesene Spielerrolle zusagt. Als nächstes leitet jeder einzelne Mitspieler seine Entscheidung an die „Game-Queue“-MessageQueue weiter („sendRoleAssignmentACK(decision)“). Der Communication-Manager wartet währenddessen auf die Antwort bzw. Entscheidung aller Mitspieler. Hierfür liest er die Antworten bzw. Entscheidungen der einzelnen Mitspieler aus der „Game-Queue“ aus („readRoleAssignmentACK“). Falls alle Mitspieler ihre zugewiesene Spielerrolle akzeptiert haben, wird „true“ an die Game-Creation zurückgegeben. Andernfalls wird „false“ zurückgesendet. Gesetzt dem Fall, dass alle Mitspieler ihre Spielrolle akzeptiert haben, persistiert die Game-Creation die Spielerrollenzuweisung mit Hilfe des „Database-Controllers“ („storePlayerAssigments(assigments)“) und die Game-Creation startet anschließend das Spiel mit den Spielern in den entsprechenden Spielerrollen. Sollten nicht alle Spieler ihre zugewiesenen Spielerrollen bestätigt haben, so kommt das Spiel nicht zustande.

2. Sequenzdiagramm – Spielablauf

Als nächstes wird das Sequenzdiagramm 3.8 beschrieben, welches die Aktualisierung des „playerObjects“, die Ausführung der Rule-Control-Engine sowie die Aufforderung einer Spieleraktion abbildet.

Jeder Mitspieler sendet zyklisch in regelmäßigen Abständen eine Aktualisierung seines „playerObjects“ an die „Game-Queue“-MessageQueue („sendPlayerStatusUpdateMsg“), um dadurch beispielsweise Geo-Positionsänderungen der Mitspieler zu erfahren. Währenddessen besorgt sich die Game-Creation mit Hilfe des Communication-Managers die Aktualisierung der Mitspieler („getPlayerStatusUpdate()“). Der Communication-Manager liest dafür die Änderungen der Mitspieler aus der „Game-Queue“ aus („readPlayerStatusUpdateMsg()“) und gibt die Spieleränderung an die „Game-Creation“ zurück. Anschließend übernimmt die Game-Creation die zurückgelieferte Spieländerung („updatePlayerStatus(player, status)“). Im nächsten Schritt fordert die Game-Creation die „Rule-Control-Engine“ auf, ihre definierten Spielregeln „abzufeuern“ („checkIfRuleApply()“). Sollte eine oder mehrere Regel/n greifen, die eine Aktion für einen oder für mehrere Spieler nach sich bringt, so leitet die Rule-Control-Engine den entsprechenden Spieler sowie die jeweilige Aktion an den Communication-Manager weiter („sendActionPlayer(player, action)“). An dieser Stelle sendet der Communication-Manager die übermittelte Aktion in die passende „Player-Input-Queue“-MessageQueue des Spielers („sendActionForPlayerMsg()“). Als nächstes liest der Spieler die Spieleraktion aus seiner

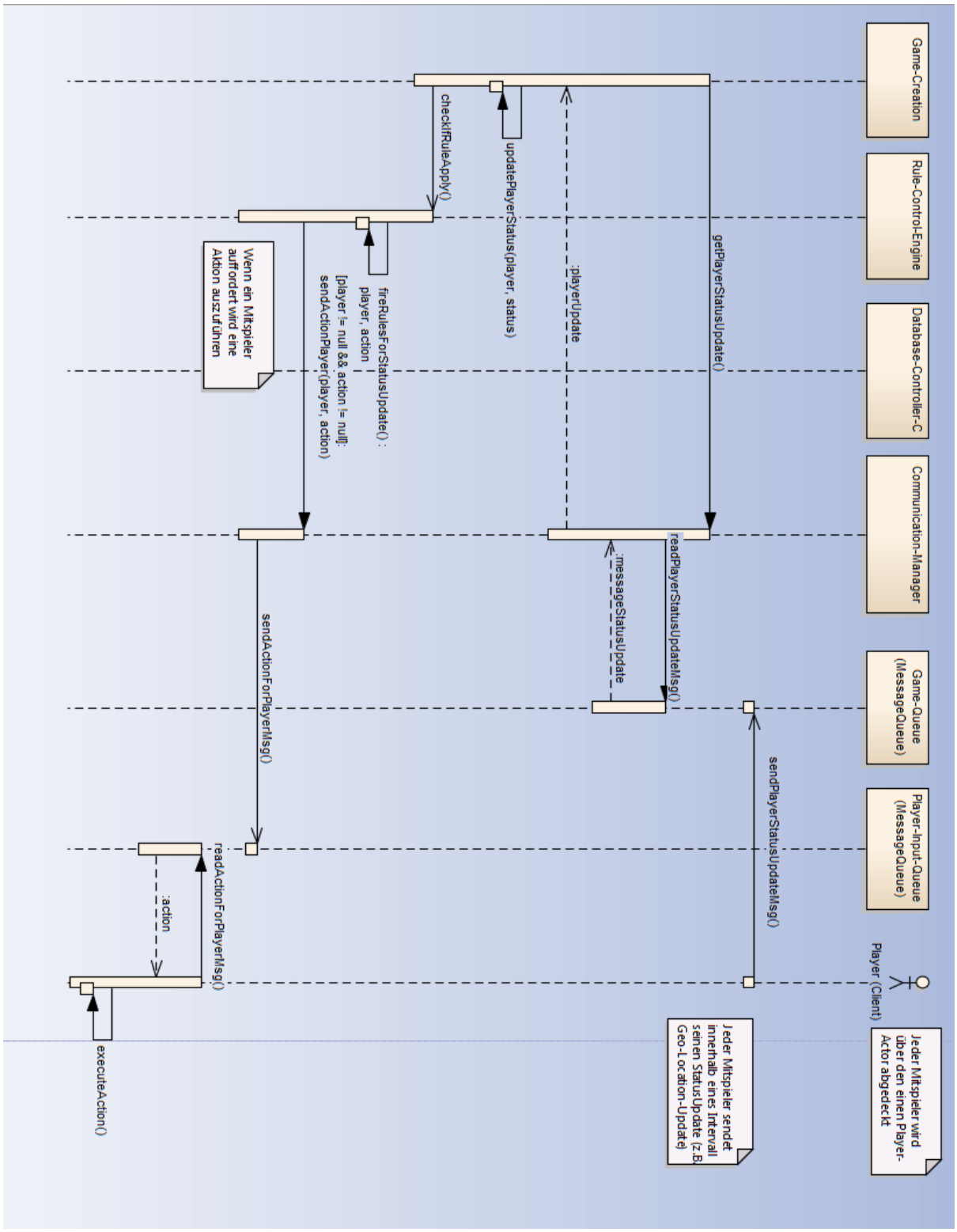


Abbildung 3.8: 2. Sequenzdiagramm – Spielablauf

„Player-Input-Queue“ aus („readActionForPlayerMsg“) und führt die entsprechende Aktion aus („executeAction()“).

3. Sequenzdiagramm – Spielablauf

Zuletzt wird der dritte Teil des Sequenzdiagramms 3.9 vorgestellt. Dieser behandelt das Beenden eines laufenden Spiels.

Dieses Sequenzdiagramm ist bis zu dem Zeitpunkt, an dem die „Rule-Control-Engine“ die Rule-Engine „abfeuert“, identisch mit dem zuletzt beschriebenen Diagramm. Daher beginnt die Beschreibung des dritten Sequenzdiagramms erst zu diesem Zeitpunkt. Sollte das Regelsystem als Ergebnis die Aktion („end_of_game“) auswerfen, so ruft die „Rule-Control-Engine“ „stopGame“ auf Seiten der Game-Creation auf. Daraufhin persistiert die Game-Creation das Spielergebnis mit Hilfe des „Database-Controllers“. Anschließend fordert die Game-Creation den Communication-Manager auf, die Kommunikation mit den Spielern zu beenden („finishCommunicationWithPlayer(gameResult)“). Dafür sendet der Communication-Manager eine Nachricht mit dem Spielergebnis an die „Player-Input-Queue“-MessageQueue aller Spieler des Spieles („sendFinishGameWithPlayer(gameResult)“). Jeder einzelne Spieler liest daraufhin das Spielergebnis aus der seinigen „Player-Input-Queue“ aus („readFinishGameMsg()“) und bestätigt die Beendigung des Spiels, indem er eine Spielbeendigung in die „Game-Queue“ sendet („sendFinishGameACK()“). Währenddessen liest der Communication-Manager die Bestätigungen der Spieler aus der „Game-Queue“-MessageQueue aus („readFinishGameACK()“). Falls alle Mitspieler die Beendigung des Spiels bestätigt haben, so wird von der Game-Creation die Beendigung des Spiels als erfolgreich markiert und mit Hilfe des „Database-Controllers“ persistiert („storeGameSuccesfulStopped()“).

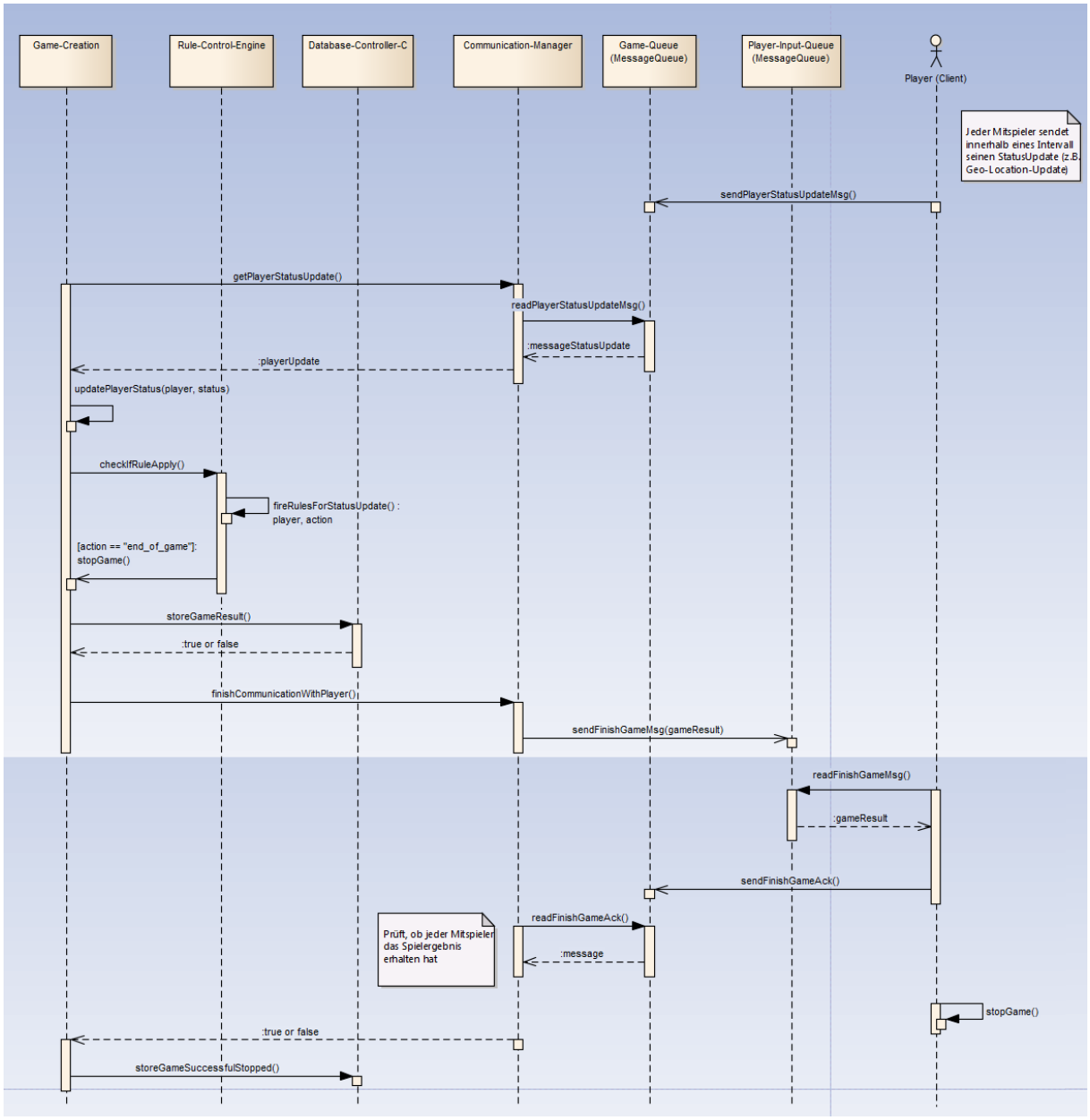


Abbildung 3.9: 3. Sequenzdiagramm – Spielablauf

3.1.8 Datenbankmodell

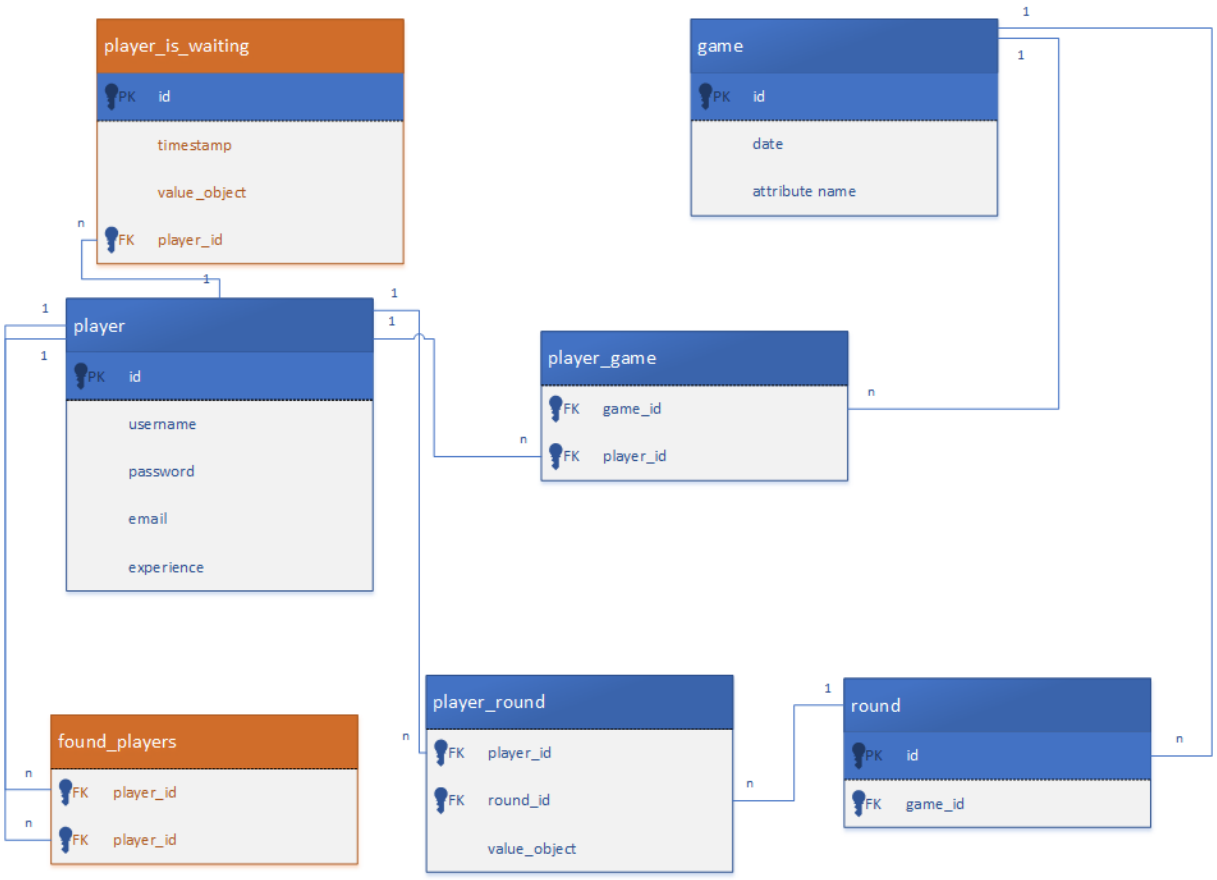


Abbildung 3.10: Datenbankmodell in ERD Notation

Abbildung 3.10 zeigt das Datenbankmodell in der bekannten ERD Notation. Die zentrale Tabelle ist hierbei die „player“-Tabelle, welche alle relevanten Informationen über einen Spieler bereithält. Auch wenn bei dem Datenbankmodel ebenfalls versucht wurde, den generischen Ansatz einzuhalten, so wird es bei gewissen Spielen notwendig sein, die „player“-Tabelle zu erweitern, wenn spezielle Attribute vonnöten sind. Die orangefarbene gekennzeichneten Tabellen („player_is_waiting“ und „found_players“) sind speziell für die Spielersuche zuständig. Sucht ein Spieler nach einem Mitspieler, so werden seine spielrelevanten Daten in dem Attribut "value_object"gespeichert, damit auch andere suchende Prozesse auf die Daten zugreifen können. Kommt ein Spiel zwischen mehreren Spielern zustande, so werden diese in der Tabelle „found_players“ temporär gespeichert, damit andere Threads wissen, dass diese Spieler bereits miteinander spielen und keine weitere Spielinstanz mit denselben Spielern erzeugt wird. Die

Tabelle „game“ dient zur Speicherung eines Spiels inklusive deren gesamten Ablauf. Ein Spiel besteht hierbei aus den teilnehmenden Spielern und verschiedenen Runden (Tabelle „round“). Jede Runde besitzt dabei die spezifischen Werte in dem Attribut „value_object“ eines jeden Spielers.

3.2 Testszenario

Das folgende Kapitel beschreibt den Aufbau eines ersten Testszenarios für die Erfassung von Testdaten. Wie bereits in Kapitel 2.3 erwähnt, dient das Spiel Räuber und Gendarm als Referenzanwendung für diese Bachelorarbeit. Aus diesem Grund fiel die Entscheidung darauf, zunächst einmal eine minimale Clientsoftware zu entwickeln, welche die notwendigen Daten für die Entwicklung der Serverarchitektur erfasst. Die Clientsoftware sollte dabei die Möglichkeit besitzen, aktuelle Geopositionen inklusive der aktuellen Zeit zu persistieren, um mit diesen Daten später einen aktuellen Spielablauf simulieren zu können. Die gesammelten Daten werden dabei inklusive der aktuellen Zeit in einer CSV (Comma-Seperated-Value) Datei gespeichert.

id;timestamp;latitude;longitude

1;1357840810656;53.59453771;10.02737638

2;1357840812812;53.59453198;10.02739947

3;1357840813717;53.59452363;10.02741713

Die Auflistung zeigt einen Auszug aus der CSV Datei. Jede Zeile erhält eine Id, einen Zeitstempel und die Geolokationen, welche durch Längen- und Breitengrade dargestellt sind. Für die erste Datenerfassung wurde der Hamburger Stadtpark als Lokation gewählt, da dieser mit seinen verzweigten Wegen eine ideale Möglichkeit bietet ein Pervasive-Game zu spielen. Außerdem bietet der Park aufgrund seiner außergewöhnlich hohen Anzahl an Grünflächen ideale Versteckmöglichkeiten, so dass der Jäger sein Ziel nicht bereits auf viele Meter sichten kann. Ein weiterer Vorteil dieser Lokation ist das Verbot von motorisierten Fahrzeugen, wodurch die Risiken eines Unfalles im Eifer eines Spiels deutlich reduziert werden.

3.2.1 Testclient

Der Testclient, welcher entwickelt wurde, um das oben genannte Szenario zu realisieren, ist zunächst für das Android OS entwickelt worden. Die Test-App verfügt über einen einzigen Screen, auf welchem die aktuelle Geoposition angezeigt wird und auf einer Karte visualisiert ist. Die Testperson startet dabei die Aufnahme der Geodaten manuell per Tastendruck und kann den Testlauf ebenso stoppen. Die von der App ermittelten Daten werden dabei lokal in einer CSV Datei auf dem Device gespeichert und müssen im Anschluss von Hand für eine weitere Auswertung übertragen werden. Abbildung 3.11 zeigt den Screen der Test-App. Der Button, welcher in der unteren linken Seite zu finden ist, dient dem Starten und Stoppen eines Testlaufes. Mit dem Button auf der unteren rechten Seite springt man zu seiner aktuellen Position auf der Karte, die sich über den gesamten Bildschirm erstreckt. Im oberen Bereich der App wird die aktuelle Position in Form von Längen- und Breitengraden angezeigt.

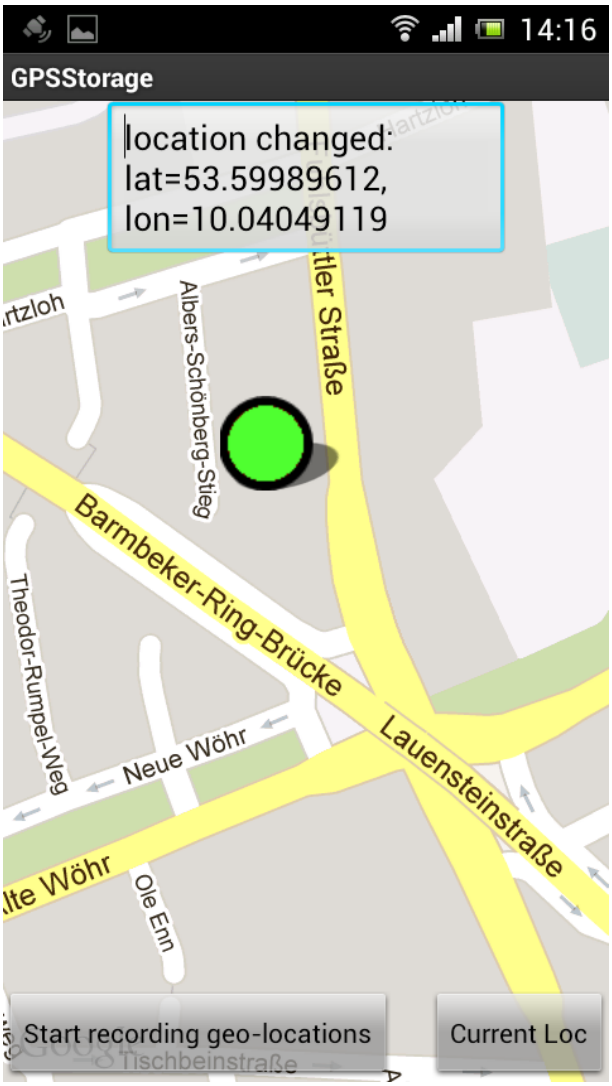


Abbildung 3.11: Test-App Design

3.2.2 Testlauf

Für den Testlauf im Hamburger Stadtpark wurden zwei GPS fähige Android Geräte mit dem Testclient bespielt. Die beiden Testpersonen hatten anschließend die Aufgabe, eine im Vorfeld besprochene Strecke mit den Geräten abzulaufen. Dabei hatten beide Personen einen unterschiedlichen Startpunkt, und sollten dabei die Rollen von Jäger und Gejagtem einnehmen, um eine möglichst realistische Situation des echten Spieles zu erzeugen. In den Abbildungen 3.12 und 3.13 lassen sich die Laufwege der beiden Spieler nachvollziehen. Die dort zu sehenden

lexikographisch sortierten Punkte wurden dabei stichprobenartig aus der von der Testapp erzeugten CSV Datei ausgewählt, um den groben Weg der Spieler darstellen zu können. Die dargestellte Tabelle unter den beiden Abbildungen zeigt hierbei den gemessenen Zeitstempel der beiden Testpersonen zu den jeweiligen Messungspunkten an.

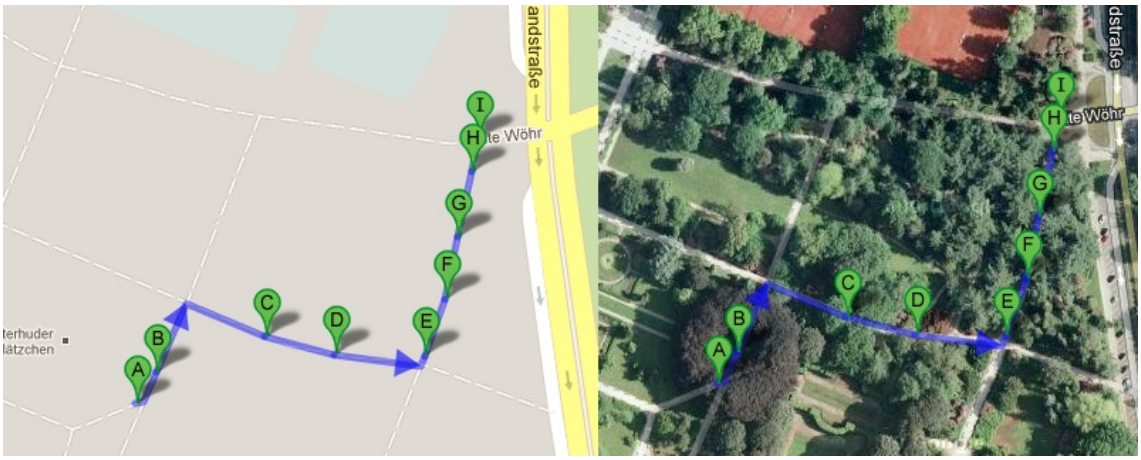


Abbildung 3.12: Testlauf des Gejagten

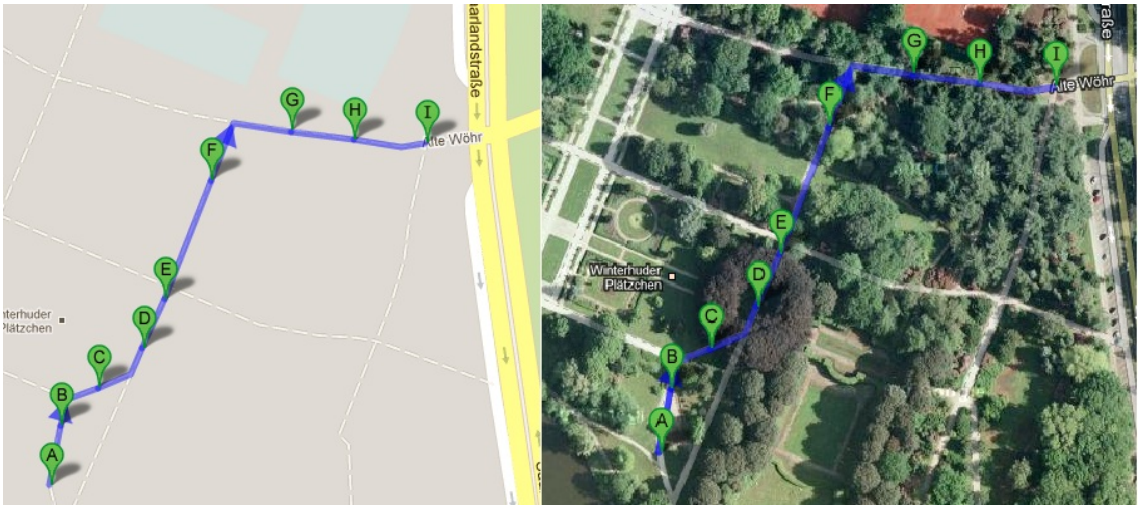


Abbildung 3.13: Testlauf des Jägers

Index	Gejagter	Jäger
A	18:00:39	18:00:29
B	18:00:59	18:00:50
C	18:01:25	18:01:12
D	18:01:50	18:01:32
E	18:02:39	18:01:52
F	18:03:04	18:02:51
G	18:03:29	18:03:22
H	18:04:05	18:04:02
I	18:04:22	18:04:27

3.3 Persistierung der Daten

Das folgende Kapitel beschäftigt sich mit der Persistierung von Daten, welche in der Softwarearchitektur verwendet werden. Wie bereits in der Analyse vorausgesetzt wurde, ist es von großer Wichtigkeit, die Daten vor Ausfällen zu schützen und auch dafür zu sorgen dass, diese immer verfügbar sind. Außerdem sollte die Möglichkeit des dauerhaften Speicherns gegeben sein. Bei einem skalierbaren System sollte zusätzlich darauf geachtet werden, dass das so genannte ACID Prinzip eingehalten wird. Aus diesem Grund wurde für die Persistierung ein Datenbankmangementsystem gewählt, da ein solches die benötigten Anforderungen erfüllt und darüber hinaus weitere Eigenschaften bietet, die den Umgang mit den gespeicherten Daten erleichtern. Als konkretes System wird „MySql“ verwendet, weil „MySql“ zur kostenlosen Verwendung verfügbar ist und die am weitesten verbreitete Open-Source-Datenbank ist [The MySQL team, 2013]. Für eine komfortable Arbeit mit der Datenbank bietet es sich an, ein Objektrelationales Mapping als Zwischenschicht zwischen dem Server und der Datenbank beizulegen.

3.3.1 Objektrelationales Mapping

Eine grundlegende Problematik bei der Speicherung von Programmdaten in eine Datenbank besteht in den unterschiedlichen Konzepten beider Systeme. In objektorientieren Programmiersprachen werden Daten in Objekte gekapselt. Relationale Datenbanksysteme kapseln ihre Daten in Tabellen auf Grundlage der relationalen Algebra. Diese beiden Konzepte sind vom Grund auf verschieden und stehen in einem konzeptionellen Widerspruch zueinander. Dieser

Widerspruch wird auch als „Object-Relational impedance missmatch“ bezeichnet [Neward, 2013]. Abhilfe für diese Problematik soll das Objektrelationale Mapping (ORM) schaffen. Ein ORM erlaubt dem Entwickler ein Mapping zwischen einem Objektmodel einer Programmiersprache und einer Tabelle eines Datenbankschemas. Das Mapping zwischen Objekten und der Datenbank wird dabei durch Metadaten beschrieben. Durch das Mapping wird die Menge von Datenbankzugriffscodeteilen maßgeblich reduziert, wodurch die Produktivität der Entwickler erhöht wird, da sich diese nicht mehr um die Datenbankzugriffe kümmern müssen, sondern einfach mit den gewohnten Objekten der Sprache weiterarbeiten können. Das ORM fungiert dabei als Middleware zwischen Anwendung und Datenbank [Richardson, 2008].
Folgende Grundbedingungen werden heutzutage an eine ORM Lösung gestellt:

- Es sollte eine API vorhanden sein, welche die Ausführung von Datenbankgrundoperationen(CREATE, READ, UPDATE und DELETE) auf die Objekte persistierter Klassen anbietet.
- Es muss die Möglichkeit geben, über eine API oder eigene Sprache Querys zu spezifizieren, damit man bestimmte Klassen oder Klassenattribute erhalten kann.
- Das Mapping an sich sollte ebenfalls spezifizierbar sein (beispielsweise durch die Verwendung von Metadaten).
- Eine weitere wichtige Grundbedingung ist die Möglichkeit von Optimierungsfunktionen, wie beispielsweise dirty checking oder lazy assiociation fetching, auf Objekte ausführen zu können, welche sich in einer momentanen Transaktion befinden.

3.3.1.1 Vorteile bei der Verwendung von ORM

Die Verwendung eines ORM Frameworks bietet dem Entwickler eine Menge Vorteile, wenn es um die Datenpersistierung geht. Die wichtigsten sind hierbei folgende:

3.3.1.2 Produktivität

Die Verwendung von ORM reduziert die Entwicklungszeit maßgeblich, da der Entwickler nach erfolgreicher Einbindung des Frameworks mit den Objekten arbeiten kann und sich nicht noch zusätzlich Gedanken um deren Persistierung machen muss.

3.3.1.3 Wartbarkeit

Die Tatsache, dass sich weniger Lines of Codes im Programm befinden, macht das System verständlicher und übersichtlicher. Hierdurch wird zudem auch noch das refactoring deutlich vereinfacht. Bei Persistierungssystemen, welche von Hand codiert werden entstehen zwangsläufig Abhängigkeiten zwischen den Objekten und den dazugehörigen Entitäten. Änderungen auf der einen Seite tragen durch diese Abhängigkeiten häufig weitere Änderungen auf der anderen Seite mit sich. Des Weiteren müssen meistens Kompromisse bei den Objekt- und Relationsmodellen eingegangen werden, damit diese zusammenpassen. Ein ORM dient hier als Puffer zwischen beiden Modellen und kapselt so Änderungen voneinander ab, so dass die besagten Abhängigkeiten minimiert werden.

3.3.1.4 Performance

Codiert man die Persistenz von Hand, so ist es zwar durchaus möglich diese ebenso performant wie bei einem ORM zu erzeugen, jedoch ist der Zeitaufwand, den ein Entwickler investieren muss, für Performanceoptimierung bei Abfragen ziemlich hoch. Ein ORM bietet diese Optimierung bereits an, wodurch die eingesparte Zeit für die Optimierung an anderer Stelle genutzt werden kann.

3.3.1.5 Hersteller Unabhänigkeit

Ein ORM abstrahiert die Anwendung von der SQL Datenbank und seinem spezifischen Dialekt. Dadurch ist die Portierbarkeit anderer Datanbanksysteme mit unterschiedlichen Dialekten problemlos möglich, ohne dass hierfür das System geändert werden muss. Bei einer händischen Lösung müssten bei einem Datenbankwechsel mindestens die Querys neu formuliert werden [Bauer und King, 2006, S. 28-29].

3.3.1.6 ORM Typen

Bevor man sich für ein ORM entscheidet, sollte zunächst beachtet werden, dass es verschiedene Arten von ORM gibt. Dabei sind zwischen vier verschiedenen Typen zu unterscheiden.[Bauer und King, 2005, S. 24-25] Der folgende Inhalt bezieht sich des Öfteren auf die Drei-Ebenen Architektur.

Pure Relational

Bei dem Pure Relational ORM wird die gesamte Applikation, inklusive der grafischen Benutzeroberfläche auf Grundlage des Relationenmodells und deren SQL Operationen entwickelt. Für kleine Systeme, bei denen eine gewisse Anzahl von mehrfach verwendetem Code tolerierbar ist und dessen Datenbanktransaktionen auf ein Minimum reduziert sind, kann eine solche Art des ORM durchaus eine Existenzberechtigung haben. Der Vorteil der direkten SQL Abfragen liegt darin, dass sie genau auf das entsprechende Programm abgestimmt werden können. Dafür ist das System jedoch schlecht portierbar und wartbar, da eine Änderung der Datenbank gleichzeitig eine Änderung der Abfragen und gegebenenfalls des Programmcodes mit sich zieht. Besonders langfristig gesehen ist dieses System nicht zu empfehlen. Des Weiteren verwenden Programme dieser Art häufig eine große Anzahl von „store proecudes“ und verschieben die Arbeit der Business-Ebene an die Persistierungs-Ebene, weswegen es für größere Systeme ziemlich ungeeignet ist.

Light Object Mapping

Light Object Mapping bildet Entitäten auf Klassen ab. Das Mapping zwischen Entität und Klasse geschieht dabei von Hand. Die codierten Datenbankzugriffe (SQL Querys) werden in der Business Logik mithilfe von Designpatterns versteckt. Dieser Typ des ORM ist in der Praxis relativ weit verbreitet und eignet sich gut bei einer geringen Anzahl von Entitäten oder auch bei Datenmodellen, welche durch Metadaten automatisch generiert wurden.

Medium Object Mapping

Bei diesem ORM Typ wird das System auf Grundlage eines Objektmodelles entwickelt. Der SQL Code wird beim Building von einem „code generation tool“ entwickelt oder auch zur Laufzeit durch Framework-Code. Objektassoziationen werden dabei durch Persistierungsmechanismen unterstützt. Die SQL Abfragen lassen sich durch eine so genannte „object orientated expression language“ definieren. Die vom System verwendeten Objekte werden von der Persistierungsschicht gecashed. Viele der sich heutzutage auf dem Markt befindenden ORM Frameworks unterstützen diese Ebene des Mappings. Diese Art es ORM eignet sich gut für mittelgroße Systeme mit komplexen Transaktionen und auch, wenn eine Portierbarkeit

zwischen verschiedenen Datenbanksystem als wichtig zu erachten ist. In der Regel werden hierbei keine „store procedures“ verwendet.

Full Object Mapping

Das Full Object Mapping ist das mächtigste der hier dargestellten Typen. Es unterstützt komplexe Objektmodellierungen wie Kompositionen, Vererbung oder auch Polymorphie. In der Persistenzschicht ist die Persistierung transparent gehalten. Die persistierten Klassen müssen dabei kein besonderes Interface oder eine Oberklasse implementieren, sondern verhalten sich wie reguläre Klassen. Fetching Strategien wie lazy oder eager fetching sind ebenso wie caching transparent implementiert. Das Full Object Mapping erfordert eine hohe Entwicklungszeit und ist deshalb meist zu zeitaufwendig, um selbst implementiert zu werden. Aus diesem Grund wird in der Regel ein bereits vorhandenes Framework eingebunden.

Nach ausgiebiger Prüfung der hier zuvor vorgestellten ORM Typen, wurde das Full Object Mapping für die Serverarchitektur gewählt, da diese den größtmöglichen Freiraum bei der Softwaremodellierung bietet und von den meisten, sich auf den Markt befindlichen, Frameworks angeboten wird. Für die Programmiersprache Java bieten sich Toplink, OpenJPA und Hibernate als verfügbare Frameworks an. Für die in dieser Arbeit behandelte Serverarchitektur wurde Hibernate verwendet, da bereits ausgereifte Vorkenntnisse und Erfahrung zu diesem Produkt vorliegen. Des Weiteren wird Hibernate in der Praxis von vielen zehntausend Entwicklern verwendet [The Hibernate Team, 2013], weshalb hier ein deutlich höherer Support im Vergleich zu den anderen Produkten gegeben ist.

3.3.2 Hibernate

Hibernate ist ein Open Source ORM Framework, welches von der Firma Jboss entwickelt wurde. Der folgende Abschnitt beschäftigt sich mit der Struktur und der Funktionsweise des Frameworks, damit dieses ordnungsgemäß in dem hier behandelten Serversystem eingebunden werden kann.

Abbildung 3.14 zeigt einen Überblick über die wichtigsten Schnittstellen von Hibernate. Hibernate besitzt ein großes API Design. Auch wenn versucht wurde, die Schnittstellen schlank zu halten, so zeigt die Praxis, dass diese doch recht mächtig sind und eine wichtige Rolle innerhalb des Frameworks einnehmen. Hibernate verfügt über mehrere APIs, um das System von

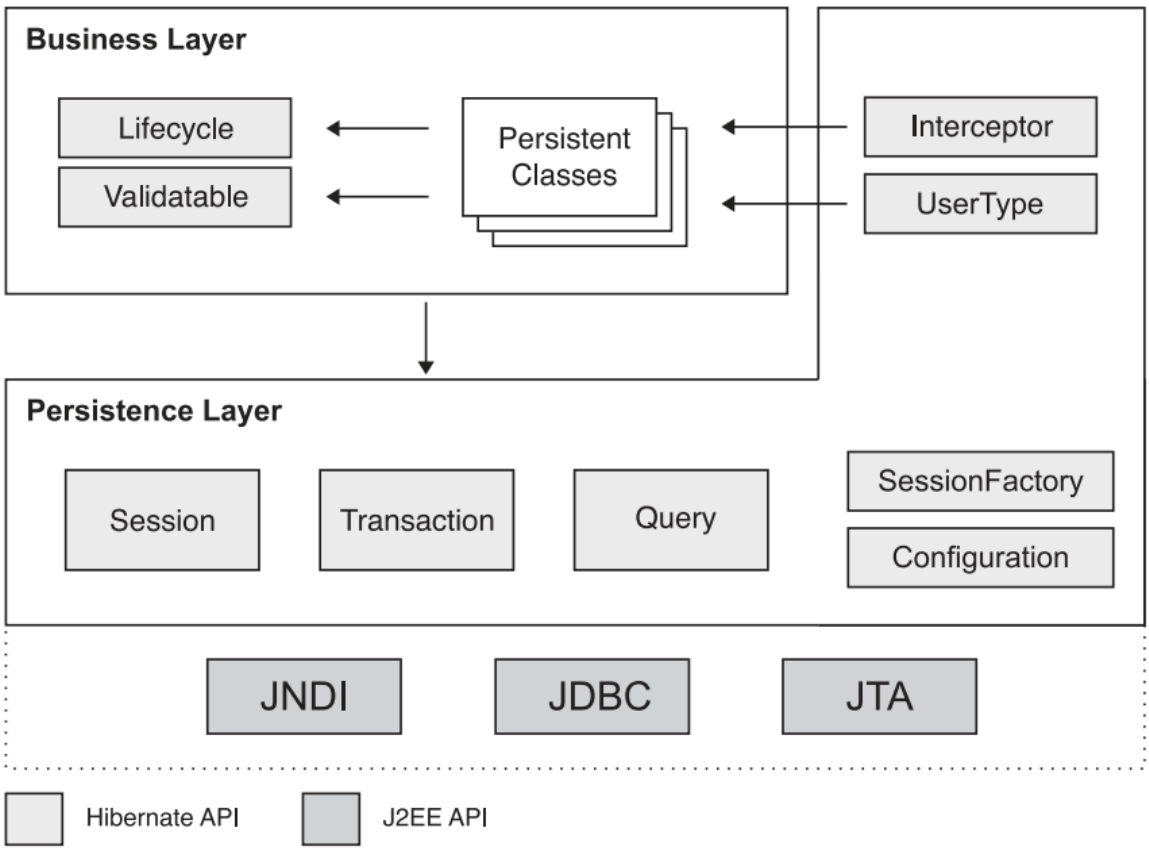


Abbildung 3.14: Hibernate-API [Bauer und King, 2005, S. 37]

der relationalen Datenbank zu kapseln. Für Standard CRUD Datenbankoperationen werden die Session, Transaction und Query Schnittstellen von der Anwendung verwendet. Sie sind gleichzeitig der wichtigste Punkt bezüglich der Abhängigkeit zwischen der Persistenz- und der Business Schicht. Die Configuration-Schnittstelle dient, wie der Name schon sagt, zur Konfiguration von Hibernate. Die Schnittstellen Interceptor, Lifecyle und Validatable sind so genannte Callback-Schnittstellen. Sie erlauben es der Anwendung, auf Events in Hibernate zu reagieren. Mit der Usertype-Schnittstelle kann das Mapping individualisiert werden. Des Weiteren verwendet Hibernate ebenso bereits existierende Java Apis wie JNDI, JDBC und JTA. Mit der JDBC kann Hibernate auf die meisten relationalen Datenbanken zugreifen, während JNDI und JTA die Integration mit J2EE Servern erlauben.

3.3.2.1 Kern-Schnittstellen

Hibernate besitzt fünf Kern-Schnittstellen, welche in nahezu jeder Hibernate Anwendung verwendet werden. [Bauer und King, 2005, S. 38-40] Die besagten Schnittstellen werden im Folgenden genauer erläutert.

Session-Schnittstelle

Die Session-Schnittstelle kann als die primäre Schnittelle des Hibernate Frameworks angesehen werden. Sie ist sehr leichtgewichtig und sowohl bei Erzeugung, als auch bei der Zerstörung ressourcensparend. Diese Eigenschaft ist von großer Wichtigkeit, da Sessions regelmäßig erzeugt und zerstört werden. In der Regel geschieht dies bei jedem neuen Request. Eine Session gilt nicht als Threadsave, weshalb jeder Thread seine eigene Session verwenden sollte. Des Weiteren dient die Session als Cache oder Collection für geladene Objekte und ist zuständig für das Storing und Retrieving von Objekten.

SessionFactory-Schnittstelle

Die SessionFactory gibt der Anwendung eine neue Instanz eines Session-Objektes. Eine SessionFactory ist nicht leichtgewichtig, weshalb für eine Anwendung auch nur eine Factory erzeugt werden sollte, zumindest wenn die Anwendung nur mit einer einzigen Datenbank interagiert. Andernfalls sind weitere SessionFactorys notwenig, da jede Datenbank ihre eigene benötigt. Die Schnittstelle sorgt des Weiteren für das Cachen von generierten SQL-Statements und der Mapping-Metadaten, welche Hibernate zur Laufzeit verwendet.

Configuration-Schnittstelle

Diese Schnittstelle wird beim Konfigurieren und Bootstrapping von Hibernate verwendet. Sie beinhaltet die Lokationen der Mapping-Dateien und enthält darüber hinaus Hibernate-spezifische Werte. Auch die Instanziierung der SessionFactory wird über die Configuration-Schnittstelle vorgenommen.

Transaction-Schnittstelle

Die Transaction-Schnittstelle ist optional und muss nicht zwingend von Hibernate verwendet werden. Sie abstrahiert den Anwendungscode von der darunterliegenden Transaktionsimplementierung, wie z.B. die JDBC Transaktion. Außerdem dient sie Hibernate bei der Hilfe von Portierbarkeit unterschiedlicher Container.

Query- und Criteria Schnittstelle

Die Query-Schnittstelle erlaubt das Stellen von Anfragen an die Datenbank. Sie kontrolliert deren Ausführung, indem sie die Anzahl an Return-Ergebnissen limitiert, Queryparameter bindet oder auch den eigentlichen Querybefehl ausführt. Die Querys in Hibernate sind in der internen Hibernate Query Language (HQL) dargestellt, native SQL Befehle sind jedoch ebenfalls möglich. Die Criteria-Schnittstelle verhält sich zu dem der Query ziemlich ähnlich. Sie erzeugt und führt objektorientierte Quers aus. Dadurch wird der Code noch einmal kürzer und kompakter. Beide Schnittstellen sind leichtgewichtig, jedoch nicht außerhalb der dazugehörigen Session-Instanz verwendbar.

3.3.2.2 Mapping mit Hibernate

Für ein reguläres Mapping mit Hibernate bietet es sich an, die Software auf der Grundlage eines Domänen-Modelles abzubilden. Das Modell sollte hierbei keine Abhängigkeiten zu anderen javaspezifischen APIs haben, wie zum Beispiel bei einem Datenbank-Aufruf via JDBC. Des Weiteren sollte sich die Modellierung nur auf der Business-Ebene befinden und keine direkten Abhängigkeiten zu der Persistierungsschicht besitzen. Die Persistierung hingeben wird von Hibernate transparent gehalten. Es existieren dabei keine Abhängigkeiten zwischen dem Persistierungsmechanismus und dem Persistenzobjekt, was bedeutet, dass keine der implementierten Persistenzklassen des Domänen-Modells eine Hibernate API, ein Interface oder eine Superklasse einbinden muss. Dadurch können die besagten Klassen auch außerhalb des Hibernate-Kontextes, wie zum Beispiel bei einem Unittest, problemlos verwendet werden. Diese Transparenz garantiert wiederum ein hohes Maß an Portierbarkeit, da die gesamte Business-Ebene komplett ohne die Persistierung in anderen Systemen wiederverwendet werden kann. [Bauer und King, 2005, S. 60]

Objekte als POJOs

Die Objekte des Domänen-Modells werden in Form von Plain Old Java Objects (POJOs) umgesetzt, da die Bedingungen, welche Hibernate an das Domänenmodell stellt, am besten durch POJOs sichergestellt werden. [Bauer und King, 2005, S. 67] Der Begriff „Plain Old Java Object“ bezeichnet ein einfaches Java-Objekt, welches sich lediglich an die Java Language Specification zu halten hat, wodurch die Vorspezifizierung von Klassen und Schnittstellen verhindert wird.

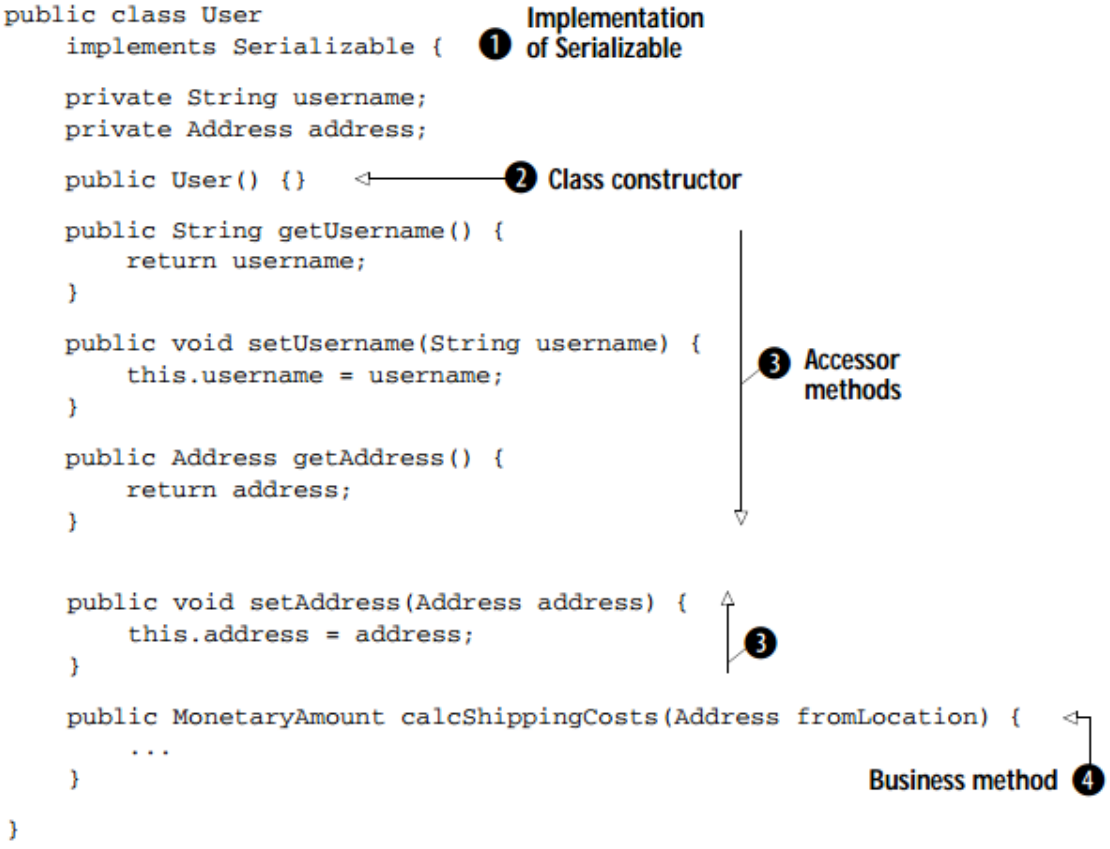


Abbildung 3.15: Beispielumsetzung POJO [Bauer und King, 2005, S. 67-68]

Abbildung 3.15 zeigt eine Beispielumsetzung eines POJOs mit den dazugehörigen Bedingungen, welche Hibernate an die Objekte stellt.

1. Für Hibernate ist es nicht zwingend erforderlich, die Schnittstelle „Serializable“ zu implementieren, bei Objektspeicherung innerhalb einer HTTPSession ist es jedoch notwendig, was bei einer Hibernate-Anwendung des Öfteren vorkommt.

2. Hibernate benötigt einen parameterlosen Konstruktor für jede Perstistierungsklasse. Es ist nicht notwendig, den Konstruktor auf „Public“ zu setzen, allerdings sollte er zumindest in dem entsprechenden Package sichtbar sein.

3. Eigenschaften werden als Instanzvariablen implementiert. Zu diesen Variablen müssen die entsprechenden Zugriffsmethoden, in Form von „Gettern“ und „Settern“ implementiert werden. Wichtig ist es hierbei, sich um die von JavaBean vorgegebene Namenskonvention zu halten, da Hibernate bei einem Zugriff auf die Eigenschaft gezielt über den Namen nach den entsprechenden Methoden sucht. Die „getter“-Methoden beginnen dabei mit dem Wort „get“, gefolgt von dem Variablennamen, der immer mit einem Großbuchstaben beginnt. Die „setter“-Methoden beginnen mit einem „set“. Eine Besonderheit bieten Boolean-Variablen, die mit einem „is“ statt einem „get“ beginnen können. Für Hibernate ist es nicht notwendig die Zugriffsmethoden mit „Public“ zu deklarieren. Auch „private“-Methoden können problemlos verwendet werden.

4. In einem POJO können ebenfalls normale Business-Methoden implementiert werden, welche keinen direkten Einfluss auf Hibernate besitzen.

Assoziationen zwischen POJO

Abbildung 3.16 zeigt zwei POJO–Klassen, die sich in einer „many-to-many“ Assoziation befinden. Des Weiteren befindet sich die hier zu sehende Klasse Category in einer „one-to-many“ Assoziation zu sich selbst. Die folgenden Code-Ausschnitte zeigen hierbei die Implementierung der POJO-Klassen, für eine mögliche Persistierung der Assoziationen durch Hibernate.

Das Codebeispiel, welches hier in Abbildung 3.17 zu sehen ist, zeigt zunächst nur die Assoziationsumsetzung der Category-Klasse zu sich selbst. Jede Kategorie kann dabei mehrere Unterkategorien(childCategories) besitzen, aber selbst nur eine Oberkategorie(parentCategory). Die „Eins“ einer „one-to-many“ Assoziation wird durch eine Variable des konkreten Objektes dargestellt. In diesem Fall durch Category parentCateory; Das „Viele“ wird durch einen Collectiontyp dargestellt. Hibernate benötigt ein Collection-Interface als Variablendeklarierung. In der Abbildung zu sehen als Set childCategories = new HashSet();. Es ist nicht zwingend notwendig

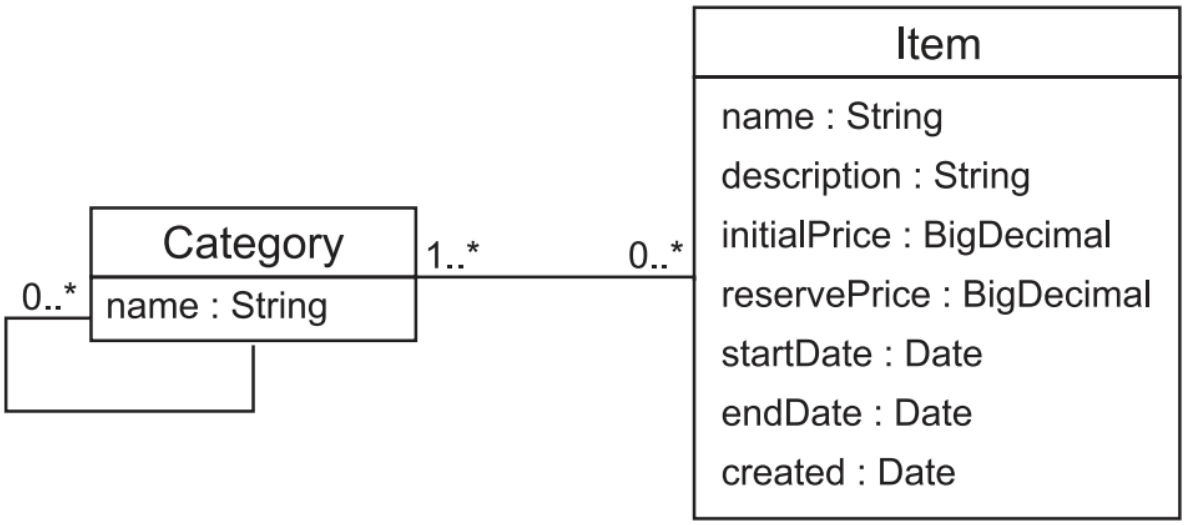


Abbildung 3.16: Assoziation von POJO-Klassen [Bauer und King, 2005, S. 71]

```
public class Category implements Serializable{

    private String name;
    private Category parentCategory;
    private Set<Category> childCategories = new HashSet<Category>();

    public Category() {};

    /*
     ...
     accesor methods
     ...
     */
```

Abbildung 3.17: Codebeispiel für eine Assoziationsumsetzung

das Interface Set zu verwenden(auch andere Schnittstellen vom Typ Collection sind möglich), jedoch sind Duplikate nicht erlaubt, weswegen sich das Set hier anbietet. Implementierung einer Assoziation zweier Category-Objekte

Anhand der Abbildung 3.18 ist zu erkennen, dass bei der Erzeugung einer Assoziation Manipulation an beiden Objekten zu machen sind. Dem Eltern-Objekt wird das Kind-Objekt in die Liste aller Kindobjekte hinzugefügt (aParent.getChildCategories().add(aChild)), während bei dem Kind-Objekt das Eltern-Objekt als explizite Oberkategorie gesetzt wird (aChild.setParentCategory(aParent)). Um den Entwickler zu entlasten und sicher zu stellen, dass bei einer Assoziationserzeugung

```
Category aParent = new Category();
Category aChild = new Category();
aChild.setParentCategory(aParent);
aParent.getChildCategories().add(aChild);
```

Abbildung 3.18: Codebeispiel zur Erzeugung einer Assoziation Manipulation zwischen zwei Objekten

beide Objekte manipuliert werden, bietet es sich an, eine sogenannte Convience Methode zu erzeugen, welche die Aufrufe zusammenfasst. Für das hier gezeigte Beispiel sieht eine solche Methode folgendermaßen aus:

```
public void addChildCategory(Category childCategory) {
    childCategory.setParentCategory(this);
    childCategories.add(childCategory);
}
```

Abbildung 3.19: Codebeispiel für eine Convience Methode

Convience Methoden sind nicht von Hibernate gefordert und helfen allein dem Entwickler bei der Sicherstellung der doppelten Objektmanipulation und Reduzierung des Programmcodes.

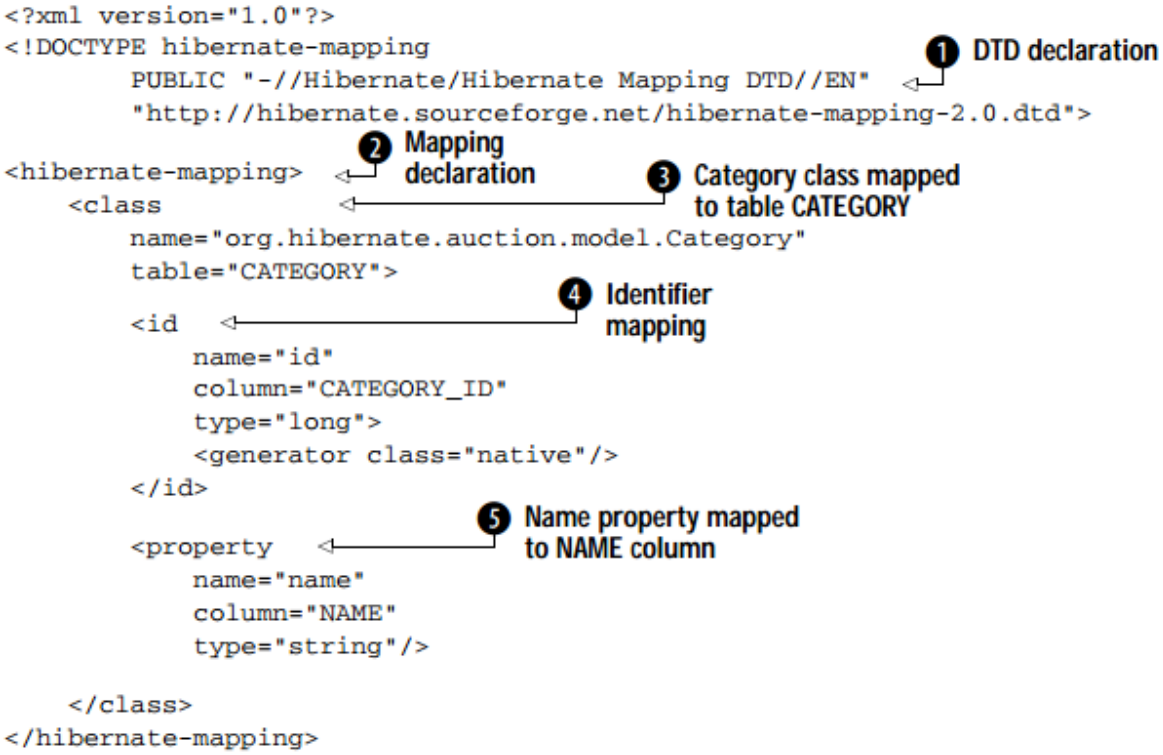
Für die Erzeugung der „many-to-many“ Assoziation zwischen den Klassen Category und Item ist es nötig, in beiden Klassen ein jeweiliges Set der anderen Klasse zu implementieren. Ausgehend davon müsste die Klasse Category um Folgendes erweitert werden:

```
private Set<Items> items = new HashSet<Items>();

public Set<Items> getItems() {
    return items;
}

public void setItems(Set<Items> items) {
    this.items = items;
}
```

Abbildung 3.20: Codebeispiel zur Erzeugung einer „many-to-many“ Assoziation zwischen zwei Klassen

Die Klasse Items erhält im Gegenzug dazu ein Set bestehend aus Categories. Da die Struktur die Selbige ist, kann auf eine explizite Darstellung des Codes in diesem Fall verzichtet werden [Bauer und King, 2005, S. 68]. Mapping Metadaten ORM Tools spezifizieren für gewöhnlich das

Mapping zwischen den Klassen, Tabellen, Attributen, Spalten, Assoziationen, Fremdschlüsseln etc. in einem Metadaten Format. Die über das Mapping gespeicherten Informationen werden auch als „opject/relational mapping metadata“ bezeichnet und definieren die Transformationen zwischen den unterschiedlichen Datentypen und Beziehungsrepräsentationen. Hibernate speichert diese Metadaten im XML Format, da dieses für den Mensch leicht lesbar und verständlich ist. Des Weiteren ist es problemlos von Hand änderbar und zudem sehr leichtgewichtig. Hibernate beinhaltet zusätzlich für viele Metadaten Defaultwerte, welche bei fehlenden Attributen auf die zu mappende Klasse angewandt werden kann.




```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
        PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class
        name="org.hibernate.auction.model.Category"
        table="CATEGORY">

        <id
            name="id"
            column="CATEGORY_ID"
            type="long">
            <generator class="native"/>
        </id>

        <property
            name="name"
            column="NAME"
            type="string"/>

    </class>
</hibernate-mapping>
```

Abbildung 3.21: Hibernate-Mapping [Bauer und King, 2005, S. 76]

Abbildung 3.21 zeigt eine Hibernate Mappingdatei im XML Format.

1. Die DTD (Document Type Definition) gibt an, dass es sich um eine Hibernate Mapping Datei handelt. Diese Definition wird von Hibernate bei der syntaktischen Validierung des Dokumentes benötigt.

2. Alle mappingspezifischen Attribute werden innerhalb eines <hibernate-mapping> Elementes definiert. Das besagte Element kann hierbei durchaus mehrfach in einer einzigen Datei vorkommen.

3. Die zuvor gezeigt Klasse Category wird hier auf die Tabelle „CATEGORY“ gemappt. Unter dem Attribut „name“ findet sich der vollständige Name einer Klasse inklusive des Package, in welchem sie sich befindet. Das Attribut „table“ gibt den Namen der Tabelle an. Jede Tabellenzeile repräsentiert hierbei eine Instanz der Klasse Category.

4. Da jede Tabelle eines relationalen Datenbankmodelles einen eindeutigen Primärschlüssel besitzt, ist es ebenfalls notwendig, einen solchen in der der Mappingdatei zu definieren. Das geschieht mit dem Tag „id“. Der Identifkator eines Objektes taucht für gewöhnlich nicht im Domänen-Modell auf, sondern nur in der entsprechenden Tabelle. Trotzdem ist es möglich, die Id eines Objektes über den Standard Setter-Definition zu erhalten, sofern dieses zuvor über die Session API von Hibernate aus der Datenbank geladen wurde [Snyder u. a., 2011, S. 88]. Die Zeile <generator class =“Native“/> sorgt dafür, dass sich Hibernate intern automatisch um die Identifikatoren kümmert.

5. Das Property Tag sorgt für das Mapping zwischen den normalen (nicht Schlüsselattributen) Attributen des Objektes und den Datenbanktabellen. Das Attribut „name“ wählt aus, welche Objektvariable gemappt werden sollte. „Column“ benamst dabei die entsprechende Spalte der Tabelle. Der „type“ legt den Datentyp fest.

3.3.2.3 Perstistierung mit Hibernate

Der folgende Abschnitt zeigt die Persistierung mittels Hibernate in der Praxis [Bauer und King, 2005, S.126-128]. Hierfür besitzt nahezu jedes „Perstistence Tool“ eine Perstistence-Manager-API, welche mindestens die Standard CRUD Operationen unterstützen, so wie auch Query Ausführungen und Transaktionskontrolle. Hibernate verfügt mit Session, Query, Criteria und Transaction gleich über mehrere Schnittstellen des Persistent-Managers. Die zentrale Schnittstelle zwischen der Applikation und Hibernate ist hierbei die Session. Sie ist der Startpunkt aller Operationen. Sowohl das Speichern, als auch das Laden von Objekten wird von der Session ausgeführt. Die Speicherung eines Objektes sieht in der Praxis folgendermaßen aus:

```
User user = new User();
user.getName().setFirstname("John");
user.getName().setLastname("Doe");

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

session.save(user);

tx.commit();
session.close();
```

Abbildung 3.22: Objektpersistierung in Hibernate [Bauer und King, 2005, S. 126-127]

Als erstes wird ein neues Objekt „user“ erzeugt. Die Klasse User ist hierbei eine normale POJO Klasse und ist wie in dem Abschnitt zuvor beschrieben implementiert. Nach der Initialisierung des Objekts wird eine neue Session erzeugt. Sessions ist hierbei eine Instanz der Sessionfactory und kümmert sich um die Erzeugung. Im nächsten Schritt wird eine Datenbank Transaktion gestartet. Um das Objekt zu persistieren, muss es mit dem Aufruf save() von der Session gespeichert werden. Nach Aufruf der save()-Methode, steht das Objekt in einer Assoziation zu der aktuellen Session, allerdings macht Hibernate noch keinen SQL Insert. Ein SQL Statement wird von der Session nur ausgeführt, wenn es absolut notwendig ist. Die Synchronisation zwischen Objektänderungen und der Datenbank geschehen erst bei dem commit der Transaction. Dabei ruft Hibernate den JDBC Treiber auf und führt einen einzigen Insert-Befehl auf die Datenbank aus. Anschließend wird die Session geschlossen und die JDBC Verbindung wird freigegeben. Alle Änderung die Innerhalb einer Transaktion ausgeführt werden (beginTransaction() und commit()), treten in einer einzigen Datenbank-Operation auf.

Objekte, die erst nach geschlossener Session eine Manipulation der Daten erhalten, bringen keine Veränderungen in der Datenbank mit sich. Solche Objekte werden auch als „detached instance“ bezeichnet. Eine Möglichkeit solch bearbeitete Objekte in der Datenbank zu persistierten ist es, im Anschluss eine neue Session zu erzeugen und die update() Methode der Session aufzurufen. Das Codebeispiel in Abbildung 3.23 zeigt ein solches Szenario.

Der Update-Befehl sorgt für ein Persistierungsupdate, durch die Verwendung es SQL Update Statements, in der Datenbank. Es macht keinen Unterschied, ob eine Objektänderung, vor oder nach dem update() angewandt wird, so lange es sich innerhalb der Transaktion befindet. Der Updatebefehl sorgt dafür, dass das Objekt dirty ist.

```
user.setPassword("secret");

Session sessionTwo = sessions.openSession();
Transaction tx = sessionTwo.beginTransaction();

sessionTwo.update(user);

user.setUsername("jonny");

tx.commit();
sessionTwo.close();
```

Abbildung 3.23: Objektaktualisierung in Hibernate [Bauer und King, 2005, S. 128]

Für das Laden eines persistierten Objekts aus der Datenbank wird ebenfalls die Session-Schnittstelle verwendet. Das folgende Codebeispiel zeigt einen einfachen Aufruf, in welchem ein Objekt anhand seiner ID aus der Datenbank geladen wird:

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));

tx.commit();
session.close();
```

Abbildung 3.24: Objektloading in Hibernate

Die get-Funktion sorgt dabei für das Laden des Objektes.

Zusammenfassend kann über Hibernate gesagt werden, dass es ein sehr komplexes Framework ist, welches den Benutzern zahlreiche Konfigurationsmöglichkeiten bietet. Auch wenn der Einstig einiges an Einarbeitungszeit erfordert und sich durchaus schlankere ORM-Frameworks auf dem Markt befinden, so ist Hibernate dennoch ideal für die hier thematisierte Plattform geeignet und wird in dieser seinen Einzug finden.

3.4 Kommunikation

Im folgenden Kapitel wird auf die Kommunikation zwischen den unterschiedlichen Parteien innerhalb der Plattform eingegangen. Die Plattform agiert als verteiltes System sowie mit mobilen Geräten. Da die Verfügbarkeit eines mobilen Geräts nicht immer gegeben ist, wird

eine lose Kopplung zwischen den unterschiedlichen Instanzen bevorzugt. Daher fiel die Wahl auf eine nachrichtenorientierte Middleware.

3.4.1 Nachrichtenorientierte Middleware (Message Oriented Middleware)

Basis der nachrichtenorientierte Middleware ist die asynchrone Kommunikation zwischen Server und Client in einem verteiltem System. Im Gegensatz dazu steht die synchrone Kommunikation, bei der Client und Server miteinander, zeitgleich und direkt in Verbindungen stehen. Durch diese Eigenschaften können sich Server und Client gegenseitig blockieren, falls Verbindungsprobleme auftreten. Aufgrund dessen wird die nachrichtenorientierte Middleware auch als lose Kopplung zwischen den Beteiligten bezeichnet. Für gewöhnlich verläuft die Organisation der Nachrichten in der Form einer Warteschlange (Queue). Dies führt dazu, dass der Empfänger die Nachrichten zu einem beliebigen Zeitpunkt aus der Queue entnehmen kann. Die unabhängige Übermittelung der Nachrichten ist der wesentliche Vorteil innerhalb der nachrichtenorientierte Middleware. Folglich ist die nachrichtenorientierte Middleware zu bevorzugen gegenüber klassischen verteilten Systemen, die mittels Remote-Procedure-Call (RPC) funktionieren. Des Weiteren bietet die nachrichtenorientierte Middleware, welche anhand von Queue arbeitet, eine Möglichkeit dem Empfänger bei der Nachrichtenabholung gleich gebündelte Nachrichten anzubieten. Demzufolge kann eine bessere Performanz durch die Bündelung von Nachricht beim Nachrichtentausch erreicht werden. Die nachrichtenorientierte Middleware stellt somit ein Konzept zum Austausch von Nachrichten losgelöst von Plattform und Programmiersprache bereit.

Die Abbildung 3.25 veranschaulicht eine fundamentale Architektur für eine nachrichtenorientierte Middleware zur Entwicklung eines verteilten Systems.

Ausgangspunkt ist ein nachrichtenorientiertes Modell, welches den asynchronen Nachrichtenaustausch zwischen unterschiedlichen Prozessen regelt. Für gewöhnlich wird dieses Konzept als Message Queue, Messaging und Message Oriented Middleware bezeichnet. Typischerweise kommt hierbei die Warteschlange-Technik zum Einsatz, welche folgendermaßen aufgebaut ist:

- Sender stellt eine Nachricht in der Empfänger-Queue bereit
- Empfänger und Sender arbeiten durchgehend losgelöst voneinander

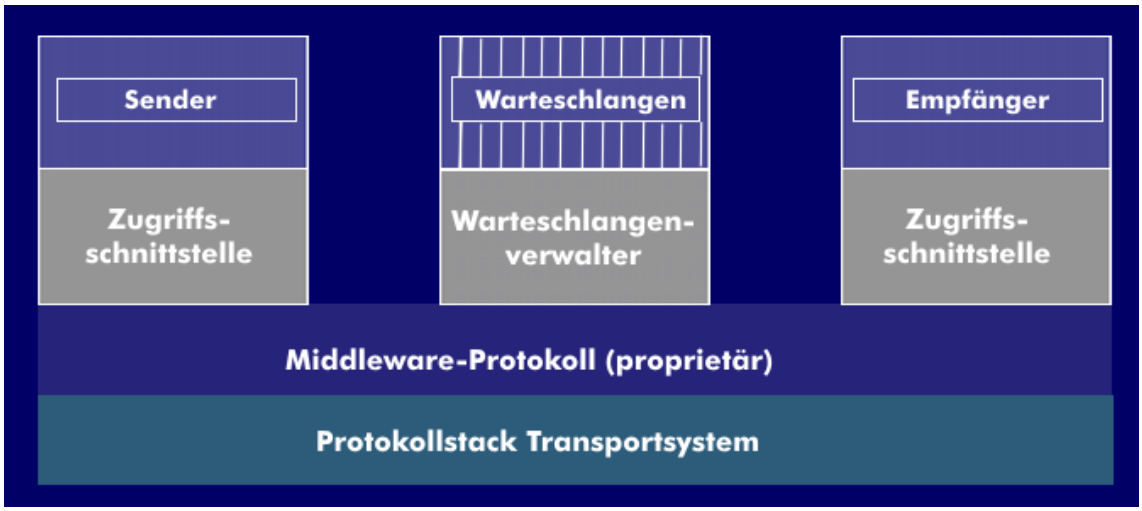


Abbildung 3.25: Architektur für eine nachrichtenorientierte Middleware [ITWissen, 2013]

- Sender geht seinen eigentlichen Aufgaben weiter nach, ohne zu Wissen, in welchem Status sich die versendete Nachricht befindet
- Empfänger entnimmt zu einem geeigneten Zeitpunkt die Nachrichten aus der Empfänger-Queue

Der Warteschlangenverwalter steht im Mittelpunkt der abgebildeten Architektur und ist wesentlicher Bestandteil der Architektur. Dieser ist auch als Provider oder Manager bekannt. Die Verwaltung der Warteschlange übernimmt folgende Zuständigkeiten:

- Einordnung der Nachrichten in die Empfänger-Queue
- Informierung des Empfänger
- Initialisierung und Kontrolle der Queues
- Bereitstellung eines Übertragungsprotokolls
- Bereitstellung von Zugriffsschnittstellen

Außerdem bieten einige nachrichtenorientierte Middlewares das Publish-Subscribe-Modell an, welches nach der „Veröffentlichen und Abonnieren“-Technik funktioniert, wie in Abbildung 3.26 zu sehen ist.

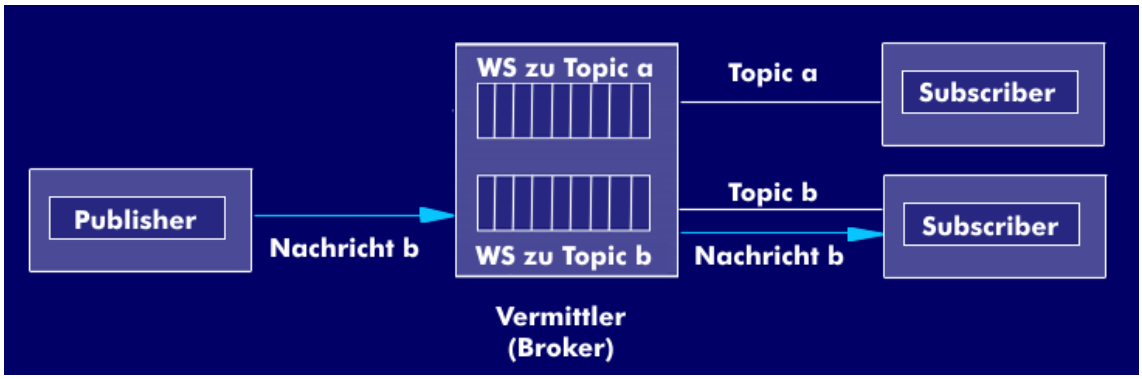


Abbildung 3.26: Publish-Subscribe-Modell [ITWissen, 2013]

Hierbei veröffentlicht ein Publisher Nachrichten zu einem Thema (Topic). Durch den Vermittler (Broker) werden diese an die Abonnenten (Subscriber) weitergeleitet. Exemplarische Beispiele zu dieser Technik sind RSS-Feeds oder Börsendienste [ITWissen, 2013].

Einen Einsatz dieser Technik ist vorzeitig für die zu entwickelnde Plattform jedoch noch nicht zu sehen.

3.4.2 Sicherheit

In diesem Kapitel wird der Aspekt der Sicherheit unter Einbeziehung der hier thematisierten Serverarchitektur analysiert. Um ein verteiltes System sicher zu machen, gilt es, eine sichere Kommunikation zwischen dem Client und dem Server herzustellen. Eine solche Kommunikation geschieht über einen sogenannten „Sicheren Kanal“ [Tanenbaum und van Steen, 2007, S. 432]. Ein sicherer Kanal sorgt bei beiden Parteien für den Schutz vor Änderungen, Fälschungen und Abfangen von Nachrichten. Gegen einen Schutz vor Störungen kann er nicht garantieren, dafür wird jedoch gewährleistet, dass keine Nachrichten von dritten belauscht werden können. Um den Kanal vor Änderungen und Fälschungen durch Eindringlinge zu schützen, werden Protokolle benötigt, welche die wechselseitige Authentifizierung und Nachrichtenintegrität gewährleisten. Da sich, wie bereits im Kapitel Architektur erwähnt, für eine Client-Server-Architektur entschieden wurde, wird insbesondere auf eine funktionsfähige Authentifizierung, sowie eine sichere Nachrichtenintegrität und Vertraulichkeit wert gelegt. Die besagten Punkte werden im folgenden genauer erläutert.

3.4.2.1 Authentifizierung

Für eine sichere Client-Server-Kommunikation ist eine Authentifizierung beider Seiten unvermeidlich. Wichtig hierbei ist allerdings zu erwähnen, dass die Authentifizierung auf die Nachrichtenintegrität angewiesen ist und umgekehrt. Schickt beispielsweise Client C eine Nachricht an den Server S und es existiert zwar eine Authentifizierung, aber keine Nachrichtenintegrität, so kann S zwar sicher sein, dass die Nachricht von C versendet wurde, jedoch nicht, dass die Nachricht auf ihrem Weg nicht von dritten manipuliert wurde. Gäbe es in einem umgekehrten Beispiel zwar eine Sicherheit auf Nachrichtenintegrität, jedoch nicht auf die Authentifizierung, so wüsste S zwar, dass die Nachricht auf ihrem Weg nicht verändert wurde, allerdings könnte er sich nicht darüber sicher sein, dass die Nachricht auch tatsächlich von C versendet wurde. Aus diesem Grund ist es wichtig, diese beiden Sicherheitsaspekte nicht von einander zu trennen, sondern, wie von den meisten Protokollen umgesetzt, gemeinsam zu behandeln. Eine Möglichkeit einer solchen Umsetzung basiert auf der Authentifizierungsgrundlage eines gemeinsamen geheimen Schlüssels.

Authentifizierungsgrundlage eines gemeinsamen geheimen Schlüssels

Bei diesem Protokoll wird davon ausgegangen, dass zwei Endgeräte, welche miteinander kommunizieren möchten, bereits über einen gemeinsamen geheimen Schlüssel verfügen. Bei einem Authentifizierungsversuch fordert eine Seite die andere zu einer Antwort auf, die nur beantwortet werden kann, wenn sie über den gemeinsamen Schlüssel verfügt. Eine solche Lösung ist auch unter dem Begriff Challenge Response Verfahren bekannt. Die Abbildung 3.27 erläutert dieses Vorgehen noch einmal grafisch.

In der Abbildung 3.27 möchte Endgerät Alice (A) einen Kommunikationskanal mit Bob (B) einrichten und sendet diesem zunächst seine Identität (Schritt 1). B sendet daraufhin eine Aufgabe (R_B) an A (Schritt 2), welche von A beantwortet werden muss. Eine Aufgabe kann z.B. aus einer einfachen Zufallszahl bestehen. A verschlüsselt anschließend die Aufgabe Mithilfe des gemeinsamen Schlüssels ($K_{A,B}$) und sendet das verschlüsselte Packet zurück zu B. B entschlüsselt im Anschluss das Packet, indem er ebenfalls den gemeinsamen Schlüssel verwendet und kann dadurch prüfen, ob es sich bei seinem Kommunikationspartner tatsächlich um A handelt. B kann sich jetzt sicher sein, um wen es sich bei A handelt, jedoch hat A noch keine Bestätigung über die Identität von B, weshalb A im nächsten Schritt eine Aufgabe an B

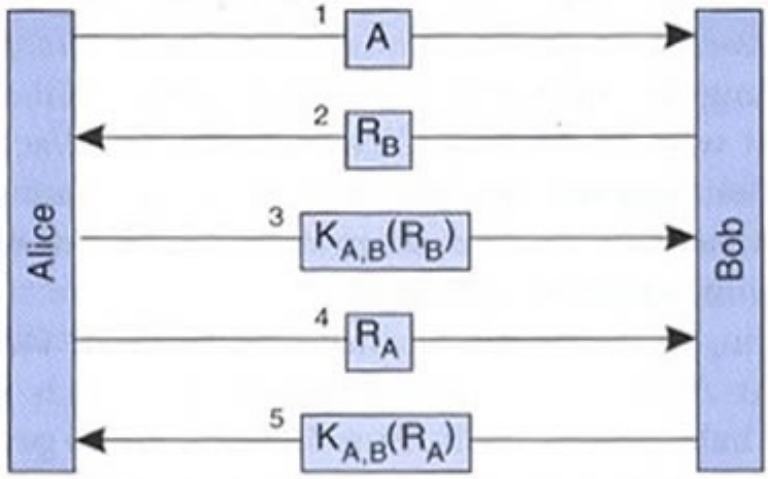


Abbildung 3.27: Authentifizierung auf der Grundlage eines gemeinsamen geheimen Schlüssels

sendet (Schritt 4), welcher dieser ebenfalls nach dem bereits erläuterten Verfahren verschlüsselt und zurück zu A sendet (Schritt 5) [Tanenbaum und van Steen, 2007, S. 433-434].

Authentifizierung mittels öffentlicher Schlüssel

Dieses Protokoll basiert auf der Grundlage von öffentlichen Schlüsseln. Jedes Endgerät besitzt dabei einen geheimen privaten Schlüssel, welchen nur er selbst kennt, und einem öffentlich Schlüssel, der im Besitz beider Seiten ist.

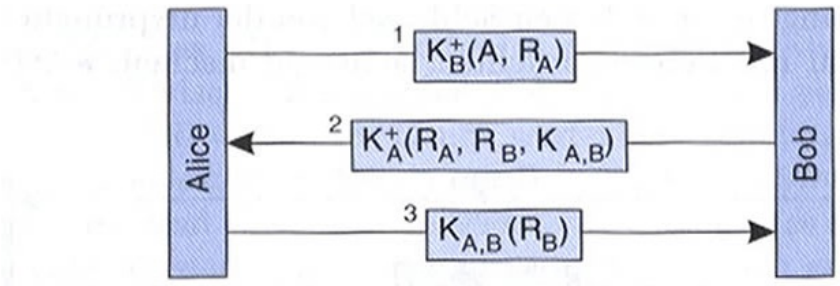


Abbildung 3.28: Wechselseitige Authentifizierung in einem Verschlüsselungssystem mit öffentlichen Schlüsseln

Anhand der hier gezeigten Abbildung 3.28, lässt sich erkennen, dass A eine Authentifizierungsanfrage an B sendet. Hierbei wird von A eine Aufgabe (R_A erzeugt und mit dem öffentlichen

Schlüssel von B (K_B^+)verschlüsselt. Diese Aufgabe kann lediglich mit dem privaten Schlüssel, welchen nur B besitzt, entschlüsselt werden. Hat B die Aufgabe entschlüsselt, so packt er diese gemeinsam mit einer eigenen Aufgabe (R_B) für A und einem Sitzungsschlüssel ($K_{A,B}$) in ein Paket. Dieses Paket wird anschließend mit dem öffentlichen Schlüssel (K_A^+) von A verschlüsselt. Der Sitzungsschlüssel dient für spätere Kommunikationen über den gesicherten Kanal. Empfängt A dieses Paket, so entschlüsselt er es mit seinem privaten Schlüssel und kann anhand der entschlüsselten Aufgabe sicher sein, dass es sich tatsächlich um B handelt. Im Anschluss verschlüsselt A die entschlüsselte Aufgabe von B mit dem Sitzungsschlüssel und sendet diese zurück zu B, damit auch B sicher sein kann wirklich mit A zu kommunizieren. Bei diesem Protokoll ist es wichtig, dass beide Seiten die Garantie haben, dass der jeweilige öffentliche Schlüssel auch wirklich zu der richtigen Person gehört und nicht zu einer, die sich fälschlicherweise als diese ausgibt [Tanenbaum und van Steen, 2007, S. 440].

Darüber hinaus existieren noch weitere Authentifizierungsprotokolle, wie z.B. ein KDC (Key Distribution Center), bei dem sich eine zentrale Verwaltungsstelle um die Authentifikation kümmert. Ein solches Verfahren ist für die hier vorgestellte Serverarchitektur allerdings zu umfangreich, weshalb hier nicht weiter darauf eingegangen wird.

3.4.2.2 Nachrichtenintegrität und Vertraulichkeit

Neben der Authentifizierung ist es von großer Wichtigkeit, dass ein sicherer Kanal ebenfalls die Nachrichtenintegrität und auch Vertraulichkeit garantieren kann. Die Nachrichtenintegrität schützt dabei vor betrügerischen Änderungen von Nachrichten. Die Vertraulichkeit stellt sicher, dass Nachrichten während des Versendens weder abgefangen, noch gelesen werden können. Eine Realisierung der Vertraulichkeit ist relativ unkompliziert zu realisieren, indem man, wie auch bei der Authentifizierung, die zu verschickende Nachricht Mithilfe eines privaten oder öffentlichen Schlüssels verschlüsselt. Für eine Nachrichtenintegrität zu sorgen, ist hingegen deutlich schwieriger. Hierfür eigenen sich folgende Verfahren [Tanenbaum und van Steen, 2007, S. 440]:

Digitale Signaturen

Neben dem Anspruch eines sicheren Kanals gehen die Anforderungen der Nachrichtenintegrität häufig über die eigentliche Ermittlung hinaus. Wird davon ausgegangen, dass A eine Nachricht an B sendet, so müssen folgende Anforderungen gewährleistet sein:

- A muss zugesichert werden, dass es B nicht möglich ist, den Wert der Nachricht zu verändern und anschließend zu behaupten, sie käme von A.
- B hingehen benötigt eine Zusicherung, dass A nicht abstreiten kann, die Nachricht gesendet zu haben.

Diese Anforderungen sind erfüllbar, wenn A seine Nachricht so signiert, dass die Signatur eindeutig mit dem Inhalt der Nachricht verknüpft ist. Sollte eine Nachricht verändert werden, so kann dies durch die Hilfe der Signatur aufgezeigt werden. Für die Umsetzung einer Signatur gibt es verschiedene Möglichkeiten. Eine verbreitete Möglichkeit ist die Verwendung eines öffentlichen Schlüssels, wie die Abbildung 3.29 zeigt.

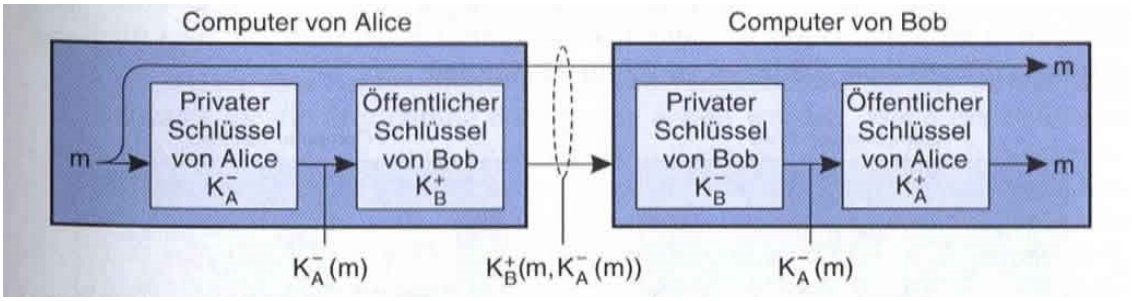


Abbildung 3.29: Das digitale Signieren einer Nachricht mittels Kryptografie mit öffentlichen Schlüsseln

Wenn A eine Nachricht an B senden möchte, so verschlüsselt A die Nachricht (m) mit seinem privaten Schlüssel (K_A^-) und sendet diese verschlüsselte Nachricht (K_A^-(m)) gemeinsam mit derselben Nachricht in unverschlüsselter Form an B. Wenn die Nachricht ebenfalls vertraulich sein soll, so werden die Nachrichten ebenfalls mit dem öffentlichen Schlüssel (K_B^+) von B verschlüsselt, damit dieser der einzige ist, der das Paket lesen kann. B kann jetzt das Paket mit seinem privaten Schlüssel (K_B^-) entschlüsseln und anschließend die verschlüsselte Nachricht Mithilfe des öffentlichen Schlüssels von A entschlüsseln. Wenn B die Garantie hat, dass der öffentliche Schlüssel wirklich der von A ist, so kann er sich jetzt sicher sein, dass die Nachricht von A kam, indem er die entschlüsselte Nachricht mit der bereits unverschlüsselten Nachricht abgleicht. A ist im Gegenzug dazu vor unerwünschten Änderungen der Nachricht durch B geschützt, da B nachweisen müsste, dass die geänderte Version auch von A signiert wurde [Tanenbaum und van Steen, 2007, S. 441].

Sitzungsschlüssel

Ein Sitzungsschlüssel wird nach einer erfolgreichen Authentifizierung, während der Benutzung eines sicheren Kanals verwendet. Er wird auf eine sichere Art verworfen, sobald ein Kanal nicht mehr in Benutzung ist. Statt einen Sitzungsschlüssel zu generieren, wäre es auch möglich, die für die Authentifizierung verwendeten Schlüsselpaare zu benutzen, jedoch besitzt ein Sitzungsschlüssel einige Vorteile, die für eine sichere Kommunikation förderlich sind. Je häufiger derselbe Schlüssel verwendet wird, desto einfacher fällt es einem Eindringling diesen aufzudecken, da er anhand der Datenmengen bestimmte Eigenschaften des Schlüssels herausfinden kann. Der Austausch eines Authentifizierungsschlüssel findet in der Regel über so genannte out-of-band Mechanismen, wie z.B. Post oder Telefon, statt, weshalb ein häufiger Wechsel dieser Schlüssel mit einem hohen Aufwand verbunden ist. Bei einem gleichbleibenden Schlüssel besteht außerdem die Möglichkeit einer Wiedereinspielung von Daten vergangener Sitzungen. Sollte es einem Eindringling gelingen den Schlüssel herauszufinden, so könnte er damit ebenfalls, alle gespeicherten Nachrichten aus vergangenen Sitzungen lesen. Bei einem Sitzungsschlüssel, der für jeden sicheren Kanal neu generiert wird, kann der Eindringling im schlimmsten Fall die Nachrichten der aktuellen Sitzung auslesen, nicht aber die aus vergangenen Kommunikationen. Ebenso ist ein Einspielen von älteren Nachrichten mit einem wechselnden Schlüssel nicht möglich [Tanenbaum und van Steen, 2007, S. 443].

Im Nachfolgenden wird nach einer Lösung gesucht, welche die hier analysierten Sicherheitsaspekte bei der Nachrichtenübertragung beinhaltet.

3.4.3 ActiveMQ

Die Wahl der nachrichtenorientierte Middleware fiel auf die ActiveMQ, da diese alle nötigen Aspekte zur Umsetzung der Plattform abdeckt und als Open-Source-Software verfügbar ist. Außerdem liegen fundierte Vorkenntnisse im Umgang mit der ActiveMQ vor. Zusätzlich gehört die ActiveMQ zu den leistungsstärksten und bekanntesten Message Queues. Des Weiteren bietet das besagte Framework ein hohes Maß an der zuvor geforderten Sicherheit an, wodurch sich Kompomenten wie die Authentifizierung realisieren lassen. Weitere Kriterien, die für die ActiveMQ sprechen, sind:

- Persistierung der Nachrichten
- Authentifikation über LDAP

- Unterstützung vieler sprachübergreifender (Cross Language) Clients
- Unterstützung diverser Protokolle
- Verschlüsselung der Kommunikation
- unkomplizierte Handhabung
- einfache Konfiguration
- sehr gute Performanz

Als nächstes werden die wichtigsten Kriterien näher erläutert und in Verbindung zum praktischen Kontext gesetzt.

3.4.3.1 Persistierung der Nachrichten

Die JMS-Spezifikation beschreibt zwei Arten der Nachrichtenauslieferung. Einmal die persistente und einmal die nicht persistente. Eine auszuliefernde Nachricht, welche die Persistent-Eigenschaft enthält, muss zwingend auf einem persistent Speicher gesichert werden. Dagegen muss eine Nachricht, die keine Persistent-Eigenschaft enthält, nicht auf einem persistenten Speicher gesichert werden. Da die ActiveMQ die JMS-Spezifikation erfüllt, bietet sie auch beide Arten der Nachrichtenauslieferung an. Außerdem unterstützt die ActiveMQ die Wiederherstellung von persistenten Daten, was bei einem Ausfall eines ActiveMQ-Vermittlers (Broker) sehr nützlich ist. Die ActiveMQ stellt mehrere Möglichkeiten zur Persistierung bereit:

- In-Memory Persistierung
- Dateibasierte Persistierung
- Persistierung in relationalen Datenbanken

Für die zu entwickelnde Plattform wird die Option der Persistierung in relationalen Datenbanken gewählt, da bereits für die „Persistierung der Daten“ eine relationale Datenbank eingesetzt wird (siehe Kapitel 3.3). Außerdem können zu einem späterem Zeitpunkt nutzvolle Queries und Statistiken über die erfassten Nutzerdaten gebildet werden. Üblicherweise werden persistente Nachrichten dann verwendet, wenn Nachrichten, die der Vermittler (Broker) bereits empfangen hat, immer für den Empfänger zur Verfügung stehen, auch wenn dieser zum

Versandzeitpunkt der Nachricht nicht im Betrieb ist, der Empfänger ausgefallen oder nicht verfügbar ist. Gerade bei mobilen Geräten mit mobilen Netz ist die Verfügbarkeit nicht immer gegeben. Im Falle, dass der Empfänger eine Nachricht konsumiert und den Empfang dieser Nachricht bestätigt hat, wird die Nachricht aus der Queue, sowie aus dem persistenten Speicher gelöscht. Nicht-persistente Nachrichten werden gewöhnlich dann genutzt, wenn eine sichergestellte Nachrichtenauslieferung nicht unbedingt nötig ist und perfomancekritische Problematiken zu lösen sind.

Die ActiveMQ nutzt die JDBC-Datenbankschnittstelle, um persistente Nachrichten in relationale Datenbanken zu sichern. Dadurch bringt die ActiveMQ mehr Flexibilität bei der Unterstützung unterschiedlicher relationaler Datenbanken ein, da für die JDBC-Datenbankschnittstelle bereits eine Vielzahl von Treiber für die unterschiedlichen relationalen Datenbanken vorliegen. Hierzu gehören u.a. folgende:

- Apache Derby
- MySQL
- PostgreSQL
- Oracle
- SQL Server
- Sybase
- Informix
- MaxDB

Es können allerdings alle Datenbanken, die einen JDBC-Treiber anbieten, verwendet werden.

Die Performanz der Nachrichtenspeicherung über die JDBC-Datenbankschnittstelle erreicht im Verhältnis zu den anderen Persistierungsmöglichkeiten der ActiveMQ nicht das beste Ergebnis [Snyder u. a., 2011, S. 107]. Falls sich die Speicherung der Nachrichten über die JDBC-Datenbankschnittstelle als Engpass der zu entwickelnde Plattform aufweist, dann wird eine performantere alternative Speicheroption der ActiveMQ zur Nachrichtensicherung gewählt.

Die Persistierung der Queue-Nachrichten setzt die ActiveMQ wie folgt um: Sie erzeugt hierbei zwei Datenbankentabellen mit den Namen „ACTIVEMQ_LOCK“ und „ACTIVEMQ_MSGS“,

welche für die Sicherung der Nachrichten benutzt werden. Die Tabelle „ACTIVEMQ_LOCK“ sorgt dafür, dass innerhalb eines bestimmten Zeitraumes nur ein ActiveMQ-Vermittler (Broker) Nachrichten in der relationalen Datenbank persistiert, um den Problemen, die bei nebenläufigen Lese -und Schreibzugriffen auf einem Shared-Memory/Storage (in diesen Fall, die relationale MySQL-Datenbank) auftreten können, zu umgehen. Die ActiveMQ setzt dies durch ein Blockierverfahren um, indem der Vermittler (Broker), der eine Nachricht sichern möchte den Zugriff auf die Tabelle blockiert und erst nach dem Persistieren der Nachricht wieder freigibt. Zu diesem Zweck enthält die Tabelle „ACTIVEMQ_LOCK“ zwei Spalten. Die eine Spalte „ID“ sorgt für die Blockierung der Tabelle. Dazu enthält sie einen eindeutigen Identifier. Die andere Spalte „Broker Name“ beinhaltet den Namen des ActiveMQ-Vermittlers (Brokers), der die Tabelle blockiert, um eine Nachricht zu persistieren. Die Tabelle „ACTIVEMQ_MSGS“ enthält die gesicherten Nachrichten mit den zusätzlich nötigen Informationen, wie dem Queue-Name, in dem sich die Nachrichten befindet, den Identifier des Nachtrichtenerzeugers, die Anzahl der Nachrichten innerhalb der Queue und die Dauer in Millisekunden bis zur Ungültigkeit der Nachricht.

Konfiguration der ActiveMQ zur Persistierung von Nachrichten mit einer relationalen MySQL-Datenbank

Folgende Schritte sind zur Konfiguration der Nachrichtenpersistierung mit einer relationalen MySQL-Datenbank unter ActiveMQ nötig:

- Ablage des MySQL-JDBC-Treibers unter dem Pfad ACTIVEMQ_HOME/libs/optional
- Erstellung einer leeren Datenbank mit einem sinngemäßen Namen. Exemplarisch wird der Name „activemq“ für die Datenbanken verwendet
- Anpassung der ActiveMQ-Konfigurationsdatei
  - Anpassung des ActiveMQ-PersistenceAdapter

```
<persistenceAdapter>
    <jdbcPersistenceAdapter dataDirectory="${activemq.data}" dataSource="#mysql-ds"/>
</persistenceAdapter>
```

– Einrichtung der MySQL-Datenquelle

```
<!-- MySql DataSource Sample Setup -->

<bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/activemq?relaxAutoCommit=true"/>
    <property name="username" value="root"/>
    <property name="password" value=""/>
    <property name="maxActive" value="200"/>
    <property name="poolPreparedStatements" value="true"/>
</bean>
```

3.4.3.2 Authentifikation

Eine häufige Problematik, die beim Einsatz der ActiveMQ gelöst werden muss, ist die Zugriffssicherheit zum ActiveMQ-Nachrichtenvermittler (Broker), sowie zu den Nachrichtenempfängern. Aus diesem Grund bietet die ActiveMQ ein flexibles und anpassbares Sicherheitsmodel anl, welches dem Sicherheitsmechanismus der umzusetzenden Plattform angeglichen werden kann.

Bevor auf die Sicherheitseinrichtung der ActiveMQ eingegangen wird, werden kurz einige grundlegende Begriffe zum Thema Sicherheit dargelegt und in Verbindung mit dem Sicherheitsmodell der ActiveMQ gebracht.

Authentifikation ist der Vorgang, bei dem die Integrität einer Entität oder eines Anwenders, welche/welcher den Zugriff auf eine gesicherte Ressource anfordert, verifiziert wird. Es existieren mehrere Möglichkeiten, über die eine Authentifikation abgewickelt werden kann. Exemplarisch gehört dazu die Authentifikation über Passwörter im Klartextformat, Einmalpasswörter, Chipkarten (Smart Cards) oder das Kerberos-Verfahren. Die ActiveMQ stellt eine simple Authentifikation, sowie eine „Java Authentication and Autorization Service“ JAAS-Authentifikation bereit. Bei dieser handelt es sich um eine Java-API für die Bereitstellung besonderer Authentifizierungs- und Zugriffsdienste in Java-Programmen. Außerdem stellt die ActiveMQ eine API zur Verfügung, um eigenständig angepasste Authentifikation-Plugins entwickeln zu können. Bei einer erfolgreichen Authentifikation ist der Zugriff auf das System gewährt. Damit ist allerdings nicht gewährleistet, dass die Ausführung aller Operation erlaubt ist. Dies könnte zum Beispiel für Operationen gelten, welche während der Ausführung auf Systemressourcen zugreifen. Solche Operationen benötigen wiederum zur Ausführung eine

bestimmte Autorisierung, die der Anwender oder die Entität erbringen muss [Snyder u. a., 2011, S. 117-120].

Autorisation legt die Zugriffsrechte einer Entität oder eines Benutzers zu einer gesicherte Ressource fest. Autorisation baut auf die Authentifikation auf, da diese den Beitritt von nicht zulässigen Benutzern auf das System verhindert. Allerdings prüft die Autorisation, ob ein Benutzer die entsprechenden Privilegien besitzt, um bestimmte Aktionen ausführen zu dürfen/können. Ein Beispiel soll dies verdeutlichen: Besitzt Benutzer X die benötigten Berechtigungen zur Ausführung des Programms Y auf dem System Z? Um diese Frage kümmert sich die Autorisation. Solche Benutzerprivilegien werden häufig als Access Control Lists (ACLs) bezeichnet und legen fest was oder wer auf eine bestimmte Ressource bei der Ausführung einer bestimmten Operation zugreifen kann.

Als nächstes werden zwei Authentifikation-Plugins der ActiveMQ vorgestellt. Zuerst wird exemplarisch das simple Authentifikation-Plugin vorgeführt und anschließend wird die Authentifikation über die JAAS-API umgesetzt. Beide Authentifikationen werden direkt in der ActiveMQ-XML-Konfigurationsdatei als Plugins eingebunden. Die ermöglicht eine einfache Konfiguration sowie die Möglichkeit der Anpassung über das <plugin>-XML-Element [Snyder u. a., 2011, S. 117-120].

Einbindung des simplen Authentifikation-Plugins

Die einfachste Art und Weise um den Zugriff auf einen ActiveMQ-Vermittler (Broker) zu sichern, ist durch direktes Einfügen der Benutzeranmeldedaten in die ActiveMQ XML Konfigurationdatei realisierbar. Diese Arbeitsweise wird durch das simple Authentifikation-Plugin ermöglicht, welches wiederum von der ActiveMQ zur Verfügung gestellt wird. Der folgende Konfigurationsausschnitt zeigt einen exemplarischen Einsatz des simplen Authentifikation-Plugins der ActiveMQ [Snyder u. a., 2011, S. 118-120].

Während des Einsatzes der Konfiguration aus Abbildung 3.30 ist nur den fünf Benutzern (admin, dispatcher, game_intializer, game_creation, player_philip, player_vassili) der Zugriff auf den ActiveMQ-Vermittler (Broker) gestattet. Bei der Authentifikation auf den ActiveMQ-Vermittler (Broker) benötigt jeder Benutzer einen Benutzernamen und ein Passwort. Darüber hinaus bietet das simple Authentifikation-Plugin die Möglichkeit, Benutzer in Gruppen einzuteilen. Diese Funktionalität stellt das Plugin über das XML-Attribute „groups“ bereit. Dabei kann jeder Benutzer in mehreren Gruppen eingeteilt werden. Dafür müssen die Gruppennamen

```
<plugins>
    <!-- Configure authentication; Username, passwords and groups -->
    <simpleAuthenticationPlugin>
        <users>
            <authenticationUser username="admin" password="password"
                groups="dispatchers,admins"/>

            <authenticationUser username="dispatcher" password="password"
                groups="dispatchers"/>

            <authenticationUser username="game_initializer" password="password"
                groups="dispatchers"/>

            <authenticationUser username="game_creation" password="password"
                groups="dispatchers"/>

            <authenticationUser username="player_philip" password="password"
                groups="players"/>

            <authenticationUser username="player_vassili" password="password"
                groups="players"/>
        </users>
    </simpleAuthenticationPlugin>
</plugins>
```

Abbildung 3.30: Einsatz des simplen Authentifikation-Plugins der ActiveMQ

der Gruppen, in denen jener Benutzer hinzugefügt werden soll, per Komma (,) voneinander getrennt angegeben werden, wie es exemplarisch beim Benutzer „admin“ zu sehen ist. Diese Gruppeninformationen können bei der Autorisation innerhalb der ActiveMQ verwendet werden. Diese Funktion wird zu einem späteren Zeitpunkt detaillierter beschrieben.

Bei dem Einsatz des simplen Authentifikation-Plugins werden die Passwörter allerdings im Klartextformat gespeichert und übertragen. Dies hat Auswirkungen auf die Sicherheit des ActiveMQ-Vermittlers (Brokers). Nichtsdestotrotz verhindern auch Klartext-Passwörter den Zugriff von unberechtigten Clients auf die ActiveMQ, was für bestimmte Szenarien völlig ausreichend ist. Außerdem bietet die ActiveMQ die Möglichkeit, das simple Authentifikation-Plugin mit einem SSL-Transport zu kombinieren, um so die Übertragung von Klartext-Passwörtern über das Netz zu umgehen. Allerdings ist dies nicht für die zuentwickelnde Plattform ausreichend, da diese mit sensiblen Daten des Anwenders, wie zum Beispiel die Geolokation des Anwenders, hantiert. Aus diesem Grund wird als nächstes auf das „Java Authentication and

Autorization Service“ JAAS-Plugin der ActiveMQ eingegangen, welches Optionen zur nötigen Sicherheit bereitstellt.

Einbindung des „Java Authentication and Autorization Service“-Plugins

Eine ausführliche Beschreibung des „Java Authentication and Autorization Service“ (JAAS) würde den angesetzten Umfang dieser Arbeit überschreiten, weshalb nur die elementaren JAAS-Konzepte und die Entwicklung eines „PropertiesLoginModule“, welches denselen Funktionsumfang des simplen Authentifikation-Plugin abdeckt, nähergebracht werden. JAAS stellt eine pluggable (ansteckbare) Authentifikation bereit. Dies wiederum bedeutet, dass die ActiveMQ dieselbe Authentifikation-API einsetzt [Snyder u. a., 2011, S. 121]. Dabei ist es unerheblich, welche Technologie zur Verifikation der Anmeldedaten der Benutzer Anwendung findet. Diese Unabhängigkeit bietet der zu entwickelnden Plattform ein hohes Maß an Flexibilität, da auch zu einem späteren Zeitpunkt ein Wechsel der Authentifikationstechnologie möglich ist. Zur Einbindung des JAAS-Plugins sind folgende zwei Aufgaben zu erledigen. Einmal die Entwicklung einer Implementation der „javax.security.auth.spi.LoginModule“-Schnittstelle, sowie die Modifizierung der Konfiguration des zum Einsatz kommenden Authentifikations-Plugins der ActiveMQ. Die ActiveMQ stellt für einige Module Implementationen bereit. Zu diesen gehören diejenigen, welche die Authentifikation über Properties, LDAP oder SSL-Zertifikate erlauben. Ein Vorteil, welcher das Login-Modul bietet, ist eine vergleichsweise unkomplizierte Konfiguration, die durch die Einhaltung einer Spezifikation vonseiten der Login-Module erzielt werden konnte.

Im folgenden Abschnitt wird die Authentifikation über das JAAS Login-Modul, welches die Properties (Eigenschaften)-Datei-Variante verwendet, dargelegt. Zur Einbindung des „PropertiesLoginModule“ wird eine Datei mit dem Namen „login.config“ benötigt. Inhalt der Datei ist ein standardisiertes Format zur Konfiguration von JAAS Benutzern und Gruppen. Die Abbildung 3.31 stellt eines dieser dar.

Die Abbildung 3.31 (login.config) zeigt einige unterschiedliche Elemente, die zur Konfiguration eines JAAS Moduls Anwendung finden. Das „active-domain“-Element enthält die gesamte Konfiguration des Login-Moduls. Die erste Zeile beginnt mit dem vollständigen Namen des „PropertiesLoginModule“,welche eine Implementation eines JAAS Login-Modules ist und von der ActiveMQ zur Verfügung gestellt wird. Beendet wird die Zeile mit der Notation „required“, die dafür sorgt, dass eine Authentifikation ohne Login-Modul nicht

```
activemq-domain {
    org.apache.activemq.jaas.PropertiesLoginModule required
        debug=true
        org.apache.activemq.jaas.properties.user="users.properties"
        org.apache.activemq.jaas.properties.group="groups.properties";
};
```

Abbildung 3.31: Konfiguration eines JAAS Moduls

weiter fortgesetzt wird. Die zweite Zeile bewirkt die Aktivierung des Debug-Loggens für das Login-Modul. Diese Eigenschaft ist optional zu definieren. Die dritte Zeile zeigt die „org.apache.activemq.jaas.properties.user“-Eigenschaft, die den Verweis zur „user.properties“-Datei angibt. Die vierte Zeile stellt die „org.apache.activemq.jaas.properties.group“-Eigenschaft, welche den Verweis zur „group.properties“-Datei beschreibt. Des Weiteren sind die zwei Properties (Eigenschaften)-Dateien (user.properties , group.properties) zu erstellen.

Die Festlegung der Anmeldedaten von Benutzern in der Properties-(Eigenschaft)-Datei gestaltet sich als unkompliziert. Die Erstellung der „user.properties“-Datei wird dafür ebenso wie die Festlegung der Benutzeranmeldedaten benötigt. Hierfür wird für jeden Benutzer eine Zeile mit Benutzername und Passwort definiert. Die einzelnen Benutzer werden durch den Zeilenumbruch voneinander getrennt. Die Abbildung 3.32 veranschaulicht eine exemplarische „user.properties“-Datei.

```
admin=password
dispatcher=password
game_initializer=password
game_creation=password
player_philip=password
player_vassili=password
```

Abbildung 3.32: Beispiel zur Festlegung einer „user.properties“-Datei

Die „group.properties“-Datei wird benötigt, um die Festlegung der Gruppen zu bestimmen. Dazu wird die „group.properties“-Datei erzeugt und die Gruppen mit den entsprechenden

Benutzern gelistet. Die Trennung der einzelnen Gruppen erfolgt dabei gleichermaßen per Zeilenumbruch. Zuerst wird der Gruppenname definiert und darauffolgend die Benutzer per Komma voneinander getrennt angegeben, die dieser Gruppe zugehörig sind. Die nachkommende Abbildung stellt ein Beispiel für die „group.properties“-Datei dar.

```
admins=admin
dispatchers=admin,game_initializer,game_creation
players=player_philip,player_vassili
```

Abbildung 3.33: Beispiel zur Festlegung einer „group.properties“-Datei

Zuletzt fehlt noch die Einbindung des JAAS-Plugins in die ActiveMQ-XML-Konfiguration. Dies beschreibt die Abbildung 3.34.

```
<plugins>
    <jaasAuthenticationPlugin configuration="activemq-domain"/>
</plugins>
```

Abbildung 3.34: Einbindung des JAAS-Plugins in die ActiveMQ-XML-Konfiguration

In diesem Fall wird die Konfiguration „active-domain“ verwendet, die zu einem vorherigen Zeitpunkt in der „login.config“-Datei festgelegt wurde. Das JAAS-Plugin benötigt hierfür nur den Namen der JAAS-Domäne, die innerhalb der „login.config“-Datei definiert wurde. Die ActiveMQ findet die „login.config“-Datei anhand des Klassenpfades. Des Weiteren gibt es die Möglichkeit, die Lokation der „login.config“-Datei über die „java.security.auth.login.config“ Systemeigenschaft anzugeben, die in der Abbildung 3.37 Anwendung findet.

```
<ACTIVE_HOME>/bin/activemq start -Djava.security.auth.login.config=<PATH>/login.config
```

Abbildung 3.35: Angabe der Lokation einer „login.config“-Datei über die „java.security.auth.login.config“ Systemeigenschaft

An dieser Stelle bietet das JAAS-Plugin exakt dieselbe Funktionalität, die auch mit dem simplen Authentifikation-Plugin der ActiveMQ abgedeckt wurde. Allerdings mit dem Unterschied, dass

der standardisierte Java-Mechanismus zur Authentifikation verwendet wird. Dieser Unterschied hat zum Vorteil, dass nun auch beliebige existierende Sicherheitsrichtlinie realisiert werden können.

3.4.3.3 Autorisation

Dieses Kapitel beschreibt die Einrichtung sowie Konfiguration einer Autorisation innerhalb der ActiveMQ. Die Autorisation baut auf der Authentifikation auf, welche im vorherigen Kapitel vorgestellt wurde. Da die Authentifikation exklusiv für die Zugriffskontrolle einer Entität oder eines Anwenders auf eine gesicherte Ressource (System) zuständig ist, bietet diese die Möglichkeit, eine feingranulare Zugriffsüberprüfung für die jeweiligen Clients einzuführen, um so nur „auserwählten“ Benutzern den Zugriff für bestimmte Aufgaben zu gewähren. Aktienhandel-Anwendungen, bieten ein gutes Beispiel für ein solches Szenario. Bei derartigen Anwendungen reicht eine „bloße“ Authentifikation nicht aus, da es nur bestimmten Applikationen erlaubt sein sollte, neue Aktienpreise zu veröffentlichen. Andernfalls könnte „jede“ dieser Anwendungen einen neuen Aktienpreis veröffentlichen. Aus diesem Grund sollten nur authentifizierten und autorisierten Applikationen diese Option freistehen [Snyder u. a., 2011, S. 123-124].

Infolgedessen stellt die ActiveMQ zwei Ebenen zur Autorisation bereit:

- Autorisation auf Operationsebene
- Autorisation auf Nachrichtenebene

Diese beiden Arten der Autorisation ermöglichen eine viel detaillierte Zugriffskontrolle im Verhältnis zur einfachen Authentifikation.

Als nächstes wird die Autorisation auf Operationsebene beschrieben, um dann durch einige Beispiele genauer erläutert zu werden.

Autorisation auf Operationsebene

Um mit einer ActiveMQ-Destination (Queues und Topics) zu interagieren, gibt es drei verschiedene Arten von Operationen auf Benutzerebene. Zu diesen Operationen gehören:

- Read (Lesen) – Die Fähigkeit zum Empfangen von Nachtrichten seitens einer ActiveMQ-Destination
- Write (Schreiben) – Die Fähigkeit zum Senden von Nachrichten zu einer ActiveMQ-Destination
- Admin (Administrieren) – Die Fähigkeit zur Administration einer ActiveMQ-Destination Zum Beispiel fällt darunter die Fähigkeit zur Erstellung sowie Entfernung einer ActiveMQ-Destination

Anhand dieser drei Operationen kann festgelegt werden, welche Entität, Benutzer oder Gruppe die Berechtigung bekommen soll, um die jeweilige Operation ausführen zu können. Zur Einbindung dieser Autorisationsmöglichkeit wie auch zur Festlegung der Autorisation gebraucht es einer Anpassung der ActiveMQ-XML-Konfigurationsdatei. Die Abbildung 3.36 beschreibt die Einbindung von Definitionen exemplarischer operationsspezifischer Autorisationen zu einigen ActiveMQ-Destinationen.

Die Abbildung 3.36 zeigt einen Ausschnitt aus der ActiveMQ-XML-Konfigurationsdatei, welche eine Beispieldefinition für das umzusetzende Autorisation-Plugin der ActiveMQ auf Operationsebene beschreibt. Zuerst wird das JAAS Authentifikation-Plugin eingebunden, ebenso wie im vorherigen Abschnitt „Einbindung des JAAS-Plugins“ beschrieben. Anschließend wird das JAAS Autorisation Plugin definiert, welches auf die „activemq-domain“ zeigt und aufbaut. Die „activemq-domain“ Konfiguration wird innerhalb der „login.config“-Datei festgelegt. Das JAAS Autorisation-Plugin der ActiveMQ stellt eine Map mit Autorisationseinträgen („authorizationEntries“) bereit. Bei der Konfiguration eines Autorisationseintrags („authorizationEntry“) innerhalb der Map wird zunächst die ActiveMQ-Destination angegeben, welche gesichert werden soll. Hierfür wird passend zur ActiveMQ-Destination das Attribut „Queue“ oder „Topic“ gesetzt. Als Wert des entsprechenden Attributes wird der Name der ActiveMQ-Destination definiert. Anschließend muss festgelegt werden, welche Benutzer und/oder welche Gruppen die Berechtigung für jene Operationen („read“, „write“, „admin“) auf die ActiveMQ-Destination erhalten.

Bei der Definition des ActiveMQ-Destination-Namens gibt es die nützliche Möglichkeit Platzhalter (Wildcards) einzusetzen. Der „.“ Platzhalter erlaubt eine hierarchische Aufteilung des ActiveMQ-Destination-Namens wie in der Zeile „... <authorizationEntry queue=“Player.Input.*“ .../>...“ zu sehen ist. Ein weiteres Beispiel für einen Platzhalter ist beim Autorisationseintrag („authorizationEntry“) „... <authorizationEntry topic=“ActiveMQ.Advisory.>“ .../>...“ sichtbar.

```
<plugins>
    <jaasAuthenticationPlugin configuration="activemq-domain"/>

    <authorizationPlugin>
        <map>
            <authorizationMap>
                <authorizationEntries>

                    <!-- queues authorization -->
                    <authorizationEntry queue=">"
                        read="admins" write="admins" admin="admins" />

                    <authorizationEntry queue="Login"
                        read="dispatchers" write="players" admin="admins" />

                    <authorizationEntry queue="ReadyToPlay"
                        read="dispatchers" write="players" admin="admins" />

                    <authorizationEntry queue="Player.Input.*"
                        read="players" write="dispatchers" admin="admins,dispatchers" />

                    <authorizationEntry queue="Game.*"
                        read="dispatchers" write="players" admin="admins,dispatchers" />

                    <!-- topics authorization -->
                    <authorizationEntry topic=">"
                        read="admins" write="admins" admin="admins" />

                    <authorizationEntry topic="ActiveMQ.Advisory.>"
                        read="admins,dispatchers,players"
                        write="admins,dispatchers,players"
                        admin="admins,dispatchers,players" />

                </authorizationEntries>
            </authorizationMap>
        </map>
    </authorizationPlugin>
</plugins>
```

Abbildung 3.36: Beispiel zur Einbindung sowie Definitionen exemplarischer Operationsspezifischer Autorisationen zu einigen ActiveMQ-Destinationen

Der „>“ Platzhalter sorgt dafür, dass die Berechtigung rekursiv an alle ActiveMQ-Destinationen weitergeben wird, die sich unterhalb der „ ActiveMQ.Advisory“-Hierarchie befinden. Des Weiteren bietet der „*“ Platzhalter die Option, für eine beliebige Anzahl von ActiveMQ-Destinationen die Operationsberechtigungen festzulegen. Der Autorisationseintrag („authorizationEntry“) „...

<authorizationEntry queue="Player.Input.*" .../>..." zeigt ein Beispiel für diesen Platzhalter. Alle ActiveMQ-Destination bei denen der Name in das Schema „Player.Input.*" passt und die ActiveMQ-Destination vom Typ „Queue" ist, erhalten die selbe Operationsberechtigungen. [The ActiveMQ team, 2013c]

Bei der Angabe der Entitäten/Gruppen für die jeweilige Operation steht die Option zur Verfügung, diese per Komma voneinander zu trennen oder eine einzelnd anzugeben. Exemplarisch ist dies am letzten Autorisationseintrag („authorizationEntry") zu sehen („... read="admins,dispatchers,players" ...").

Für das abgebildete Autorisationsbeispiel bedeutet dies:

- Nur Benutzer der Gruppe „admins" haben einen Vollzugriffs auf alle Queues. Dies bedeutet, dass alle Benutzer dieser Gruppe Nachrichten von allen Queues empfangen, Nachrichten an alle Queues versenden sowie diese Queues erzeugen und löschen können.
- Benutzer der Gruppe „dispatchers" ist das Konsumieren von Nachtrichten aus der Queue „Login" gestattet. Nur Benutzer der Gruppe „players" dürfen Nachrichten an die „Login"-Queue senden. Allerdings ist die Erstellung sowie Entfernung der „Login"-Queue nur Benutzern der Gruppe „admins" erlaubt.
  Selbiges Verhalten gilt auch für die „ReadyToPlay"-Queue.
- Nur Benutzern der Gruppe „players" sind zum Empfangen von Nachrichten der „Player.Input.*"-Queue autorisiert. Benutzern der Gruppe „dispatchers" ist das Versenden von Nachrichten an die „Player.Input.*"-Queue erlaubt. Das Erstellen und Entfernen der „Player.Input.*"-Queue ist nur Benutzern der Gruppen „admins" und „dispatchers" gestattet.
- Alle Benutzer der Gruppe „dispatchers" sind für das Empfangen von Nachrichten aus der „Game*"-Queue autorisiert. Das Produzieren von Nachrichten in die „Game.*"-Queue ist nur Benutzern der Gruppe „players" erlaubt. Die Erzeugung sowie Löschung der „Game.*"-Queue ist nur Benutzern der Gruppe „dispatchers" und „admins" erlaubt.
- Nur Benutzer der Gruppe „admins" haben einen Vollzugriff auf alle Topics.
- Die Berechtigungen für die Topics „ActiveMQ.Advisory.>" sorgen für die „Java Management Extensions"(JMS)-Unterstützung der ActiveMQ. Der JMX-Support bietet die Möglichkeit zur Überwachung und Steuerung des Verhaltens eines ActiveMQ-Vermittlers (Broker) [The ActiveMQ team, 2013a].

Bei der Definition der Autorisationseinträge („authorizationEntries“) wurde wie folgt vorgegangen: Zuerst wurde der Zugriff auf jede Operation („read“, „write“, „admin“) aller Queues nur der „admins“- Gruppe gewährt. Erst anschließend wurden die Berechtigungen für die semantisch korrekten Gruppen passenden zu den relevanten Queues freigeschaltet. Mit dem ActiveMQ-Destinationstyp „Topic“ wurde äquivalent verfahren. Dies bietet den Vorteil, dass genau die ActiveMQ-Destinationen (Queues und Topics) freigegeben werden, die für die Umsetzung der Plattform notwendig sind.

Da die „active-domain“-Authentifikation-Konfiguration aus dem vorherigen Kapitel verwendet wurde, ist der ActiveMQ-Vermittler (Broker) auf die selbe Art und Weise zu starten.

```
<ACTIVE_HOME>/bin/activemq start -Djava.security.auth.login.config=<PATH>/login.config
```

Abbildung 3.37: Angabe der Lokation einer „login.config“-Datei über die „java.security.auth.login.config“ Systemeigenschaft

An dieser Stelle ist der Zugriff auf jede Operation („read“, „write“, „admin“) aller ActiveMQ-Destinationen nur Benutzern erlaubt, welche für den Zugriff auf den ActiveMQ-Vermittler (Broker) authentifiziert und für die jeweilige Operation auf der ausgewählten ActiveMQ-Destination autorisiert sind.

3.4.3.4 Verschlüsselung der Kommunikation

In diesem Abschnitt wird vorgestellt, wie eine verschlüsselte Kommunikation zwischen Nachrichtensendern und ActiveMQ-Vermittlern (Broker) sowie zwischen ActiveMQ-Vermittlern und Nachrichtenempfängern, exemplarisch für die umzusetzende Plattform, realisiert wurde. Da die Kommunikation zwischen den Benutzern und dem ActiveMQ-Vermittler über ein ungesichertes Netz verläuft und sensible Daten der Benutzer, wie zum Beispiel die Geo-Koordinaten vermittelt werden, ist eine Verschlüsselung der Kommunikation zwischen den Parteien nicht zu vernachlässigen. Für diese Problematik stellt die ActiveMQ die Nutzung des „Secure Socket Layer“ (SSL)-Protokoll zur Verfügung. Aufbauend auf dem SSL-Protokoll ist eine gesicherte Datenübertragung/Kommunikation zwischen den Benutzern und dem ActiveMQ-Vermittler (Broker) umsetzbar. Das SSL-Protokoll wurde entworfen, um verschlüsselte Daten über das TCP-Netzwerkprotokoll übertragen zu können. Das SSL-Protokoll verwendet ein Schlüsselpaar, welches aus einem öffentlichen Schlüssel und einem privaten Schlüssel besteht. Ein Schlüssel-

paar wird für einen gesicherten Kommunikationskanal benötigt. Der SSL-Transport-Connector wird genau für diese Angelegenheit von der ActiveMQ bereitgestellt. Der SSL-Transport-Connector setzt eine SSL-Schicht über den TCP-Kommunikationskanal [Snyder u. a., 2011, S. 70]. Die SSL-Schicht sorgt wiederum für die verschlüsselte Kommunikation zwischen ActiveMQ-Vermittler und Benutzern (Clients).

Als nächstes wird die Konfiguration zur verschlüsselten Kommunikation vorgestellt. Um den ActiveMQ-Vermittler (Broker) auf eine SSL-Übertragung umzustellen, sind einige Änderungen in der ActiveMQ-XML-Konfigurationsdatei nötig. Zunächst muss die URI des „<transport-Connector . . . />“s angepasst werden. Die URI-Syntax des SSL-Protokolls sieht wie folgt aus:

ssl://<hostname>:<port>?<key>=<value>

[Snyder u. a., 2011, S. 70] [Kilburn, 2013; The ActiveMQ team, 2013b].

Nach der Anpassung des „<transportConnector . . . />“s auf Basis der SSL-URI-Syntax schaut die ActiveMQ-XML-Konfigurationsdatei folgendermaßen aus:

```
<broker>
    ...
    <transportConnectors>
        <transportConnector name="ssl" uri="ssl://0.0.0.0:61617?trace=true"/>
    </transportConnectors>
    ...
</broker>
```

Abbildung 3.38: Anpassung des „<transportConnector . . . />“s auf Basis der SSL-URI-Syntax

Für eine funktionsfähige SSL-Übertragung gebraucht es einige weitere Komponenten. Diese Komponenten beinhalten SSL-Zertifikate, die für eine erfolgreiche SSL-Kommunikation erforderlich sind. Die ActiveMQ nutzt zur Umsetzung der SSL-Funktionalität die „Java Secure Socket Extension“ (JSSE)-Java-API. „Java Secure Socket Extension“ (JSSE)-Java-API gibt zwei unterschiedliche Typen von Dateien vor, die zur Speicherung von Zertifikaten und Schlüsseln benötigt werden. Der eine Dateityp enthält die privaten Zertifikaten mit ihren entsprechenden privaten Schlüsseln und werden als „Keystores“ bezeichnet. Der andere Dateityp trägt die Bezeichnung „Truststores“, welche vertrauenswürdige Zertifikate (öffentliche Schlüssel) von anderen Entitäten (Anwendungen) beinhalten [Oracle, 2013a] [Snyder u. a., 2011, S. 70/71].

Nun wird die Erstellung der fehlenden Datei vorgeführt. Zuerst wird eine „Keystore“-Datei und das entsprechende Zertifikat für den ActiveMQ-Vermittler (Broker) erzeugt. Hierfür kommt das Kommandozeilentool „keytool“ zum Einsatz. „keytool“ wird zur Verwaltung von „Keystores“ und „Truststores“, die per Java verteilt, beziehungsweise genutzt werden, verwendet. Das „keytool“ wird von dem Java Development Kit bereitgestellt [Oracle, 2013b]. Zur Erstellung eines „Keystores“ (Schlüsselpaares) kommt das „keytool“ wie in Abbildung 3.39 zum Einsatz:

```
keytool -genkey -alias broker -keyalg RSA -keystore mybroker.ks
```

Abbildung 3.39: Befehl zur Erstellung eines „Keystores“ mit Hilfe des „keytools“

Das „keytool“ fordert zur Eingabe eines Passworts sowie zur Eingabe von Zertifikatsdaten auf. Beispiele hierfür sind:l „Wie lautet Ihr Vor- und Nachname?“, „Wie lautet der Name Ihrer organisatorischen Einheit?“, „Wie lautet der Name Ihrer Organisation?“, „Wie lautet der Landescode (zwei Buchstaben) für diese Einheit?“ Anschließend sichert das „keytool“ den „Keystore“ mit dem angegeben Passwort und den angegeben Zertifikatsinformationen in der „mybroker.ks“-Datei.

Als nächstes wird das Zertifikat aus dem „mybroker.ks“-“Keystore“ exportiert, damit dieses Zertifikat an die Clients des ActiveMQ-Vermittlers (Broker) verteilt werden kann. Die Abbildung 3.40 zeigt den zuständigen Befehl hierfür:

```
keytool -export -alias broker -keystore mybroker.ks -file mybroker_cert
```

Abbildung 3.40: Befehl zur Exportierung des Zertifikats eines „Keystores“

Bei der Ausführung des Befehls wurde das ActiveMQ-Vermittler-Zertifikat aus der „Keystore“-Datei („mybroker.ks“) in die „mybroker_cert“-Datei exportiert. Diese enthält nun das Zertifikat des oben erstellten „Keystores“, also die des ActiveMQ-Vermittlers in diesem konkreten Fall.

Jetzt wird der „Truststore“ für den Client benötigt. Der „Truststore“ muss erstellt werden und das Zertifikat („mybroker_cert“) des ActiveMQ-Vermittlers enthalten. Dafür muss das ActiveMQ-Vermittler-Zertifikat („mybroker_cert“) in den „Truststore“ importiert werden. Die Abbildung 3.41 zeigt den benötigen Befehl für das Vorgehen:

Das importierte „mybroker_cert“-Zertifikat ist nun in der „myclient.ts“-„Truststore“-Datei zu finden.

```
keytool -import -alias broker -keystore myclient.ts -file mybroker_cert
```

Abbildung 3.41: Befehl zur Importierung eines Zertifikats in einen „Truststore“

An dieser Stelle sind alle fehlenden Komponenten vorhanden. Als nächstes bedarf es einer weiteren Anpassung der ActiveMQ-XML-Konfigurationsdatei. Dem ActiveMQ-Vermittler (Broker) muss ein „<sslContext .../>“ hinzugefügt werden. Damit wird die zu nutzenden „Keystore“-Datei angegeben. Die Abbildung 3.42 zeigt die nötigen Änderungen innerhalb der ActiveMQ-XML-Konfigurationsdatei.

```
<broker>
    ...
    <sslContext>
        <sslContext keyStore="file:<keystore path>" keyStorePassword="<keystore password>"/>
    </sslContext>

    <transportConnectors>
        <transportConnector name="ssl" uri="ssl://0.0.0.0:61617?trace=true"/>
    </transportConnectors>
    ...
</broker>
```

Abbildung 3.42: Anpassungen zur Einbindung eines „<sslContext .../>“ innerhalb einer ActiveMQ-XML-Konfigurationsdatei

Die Konfiguration des ActiveMQ-Vermittlers (Broker) ist somit abgeschlossen.

Zuletzt muss dem Client die „Truststore“-Datei („myclient.ts“) mitgegeben werden. Hierfür muss der Java Virtual Machine bei der Ausführung des Clients folgendes Argument wie in Abbildung 3.43 überreicht werden.

```
-Djavax.net.ssl.trustStore=<truststore path>
```

Abbildung 3.43: „Truststore“-Pfadmitgabe als JVM-Argument

Ab diesem Zeitpunkt verläuft die Kommunikation zwischen Client und ActiveMQ-Vermittler verschlüsselt.

Des Weiteren besteht die Möglichkeit einer Verifizierung des Clients auf Seiten des ActiveMQ-Vermittlers. Dieses Vorgehen wird allerdings nicht weiter in dieser Arbeit beschrieben, da es nicht Schwerpunkt der Arbeit ist.

3.4.3.5 REST-Unterstützung

In diesem Abschnitt werden die nötigen Schritte zur Konfiguration der REST-Unterstützung für die ActiveMQ vorgeführt. Anschließend wird exemplarisch die Kommunikation über die REST-API beschrieben.

Die Kommunikation über die REST-API soll dazu dienen, dass von den unterschiedlichsten mobilen Plattformen sowie Webseiten aus mit der ActiveMQ zusammengearbeitet werden kann. Dies wird durch die Einbindung der REST-API erreicht, da diese „nur“ mit Standard HTTP-Verbindungen arbeitet. Diese Verbindungen werden heutzutage wiederum von den meisten mobilen Plattformen unterstützt.

Die ActiveMQ stellt die nötige Umgebung zur Kommunikation über REST bereit. Folgende Komponenten werden von der ActiveMQ mitgebracht:

- ein Webserver (Jetty)
- eine erforderliche Web-Infrastruktur
  - REST-API

Zur Einbindung des Jetty-Webservers „muss“ die Default-Jetty-Konfigurationsdatei „jetty.xml“ in die ActiveMQ-XML-Konfigurationsdatei importiert werden. Die Abbildung 3.44 zeigt die benötigten Änderungen in der ActiveMQ-Konfigurationsdatei.

```
<beans>
    ...
    <import resource="jetty.xml"/>
    ...
</beans>
```

Abbildung 3.44: Anpassungen der ActiveMQ-Konfigurationsdatei zur Einbindung des Jetty-Webservers

Anschließend wird eine WebApp innerhalb der Jetty-Konfigurationsdatei „jetty.xml“ definiert. Dieser wird der „HandlerCollection“ in der Jetty-Konfigurationsdatei hinzugefügt. Die folgende Grafik zeigt, welche Anpassungen in der Jetty-Konfigurationsdatei vorgenommen werden müssen.

```
...
<bean id="securityHandler" class="org.eclipse.jetty.security.ConstraintSecurityHandler">
    ...
    <property name="handler">
        <bean id="sec" class="org.eclipse.jetty.server.handler.HandlerCollection">
            <property name="handlers">
                <list>
                    ...
                    <bean class="org.eclipse.jetty.webapp.WebAppContext">
                        <property name="contextPath" value="/rest" />
                        <property name="resourceBase" value="${activemq.home}/webapps/rest" />
                        <property name="logUrlOnStart" value="true" />
                    </bean>
                    ...
                </list>
            </property>
        </bean>
    </property>
</bean>
...
```

Abbildung 3.45: Anpassungen der Jetty-Konfigurationsdatei zur Einbindung einer WebApp

Nach Angabe der WebApp gebraucht es einer weiteren Konfigurationsdatei für die WebApp selbst. Diese wird unter dem angegeben Pfad („resourceBase“), der im vorherigem Schritt in der Jetty-Konfigurationsdatei („jetty.xml“) definiert wurde („... ${activemq.home}/webapps/rest />“), plus „WEB-INF/web.xml“ hinterlegt. Der WebApp-Konfigurationsdatei fehlt nun die Einbindung der REST-API. Die REST-API wird von dem „org.apache.activemq.web.MessageServlet“ Servlet implementiert [Snyder u. a., 2011, S. 249]. Aus diesem Grund muss dieses Servlet in der WebApp-Konfigurationsdatei („web.xml“) definiert werden und auf ein URL-Pattern abgebildet. Die Abbildung 3.46 zeigt die nötigen Anpassungen.

An dieser Stelle ist die Einbindung der REST-API innerhalb der ActiveMQ soweit abgeschlossen. Die ActiveMQ-Destinationen sind nun unter der angegeben URL („/message/*“) erreichbar. Zum Beispiel ist die Queue „Test.Spieler“ unter der URL „`http://localhost:8161/rest/message/Test/Spieler?type=queue`“ zu erreichen. Die hierarchische Trennung der Destination über das „.“-Symbol wird durch das typische URL-Trennsymbol „/“ ersetzt. Über GET-und POST-Anfragen können Nachrichten von den ActiveMQ-Destinationen

```
<!-- the subscription REST servlet -->
<servlet>
    <servlet-name>MessageServlet</servlet-name>
    <servlet-class>org.apache.activemq.web.MessageServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<!-- servlet mappings -->
<servlet-mapping>
    <servlet-name>MessageServlet</servlet-name>
    <url-pattern>/message/*</url-pattern>
</servlet-mapping>
```

Abbildung 3.46: Anpassungen der WebApp-Konfigurationsdatei

empfangen werden sowie an die ActiveMQ-Destinationen versendet werden. Die Abbildung 3.47 zeigt wie mit dem „curl“-Tool über eine GET-Anfrage eine Nachricht aus der „Test.Spieler“-Queue ausgelesen werden kann.

```
curl "http://localhost:8161/rest/message/Test/Spieler?type=queue&clientId=consumerA"
```

Abbildung 3.47: Beispiel zum Empfangen einer Nachricht per GET-Anfrage mit Hilfe des „curl“-Tools

Die Abbildung 3.48 beschreibt, wie eine Nachricht mit dem „curl“-Tool über eine POST-Anfrage an die „Test.Spieler“-Queue versendet wird.

```
curl -d 'body="Hello World"' "http://localhost:8161/rest/message/Test/Spieler?type=queue&clientId=producerA"
```

Abbildung 3.48: Beispiel zum Versenden einer Nachricht per POST-Anfrage mit Hilfe des „curl“-Tools

3.4.4 Datenformat

Das Datenformat für die Übermittlung der Daten sollte ein hohes Maß an Kompaktheit besitzen und dazu von Computern leicht zu verarbeiten sein. Da in der Regel keine großen Datenmengen übertragen werden, sondern lediglich einzelne Werte, ist es sinnvol,l das Format

und dessen Struktur so einfach wie möglich zu halten. Für die hier thematisierte Lösung bietet sich deshalb die Wahl des JSON (Javascript Object Notation) als Datenstruktur an, da es sich hierbei um ein schlankes Format handelt, welches sowohl von Menschen einfach zu lesen und schreiben ist, als auch von Maschinen einfach zu parsen und zu generieren ist [The JSON Group, 2013]. Des Weiteren lassen sich Nachrichten im Json-Format ohne weiteren Aufwand problemlos über die ActiveMQ versenden.

```
{
   "player":{
      "id":"2",
      "operation":"update",
      "value_object":{
         "geo_data":{
            "id":"87",
            "timestamp":"1357840839530",
            "latitude":53.597077,
            "longitude":10.042936
         }
      }
   }
}
```

Abbildung 3.49: Beispielnachricht als JSON-Format

Abbildung 3.49 zeigt eine exemplarische Nachricht im Json-Format. In diesem Fall sendet ein Client eine Aktualisierung seiner Geodaten an den Server. Der Server kann dies an dem Keyword „operation“erkennen für welches in diesem Fall der Wert „update“hinterlegt ist. Um den generischen Ansatz auch hier so weit wie möglich beizubehalten, wurde sich für eine spielubanhängige Struktur entschieden, welche immer den gleichen Aufbau hat. Ein Spieler übergibt bei jeder Nachricht seine Id, die zu tätigende Operation und ein Objekt mit den spezifischen Werten eines Spieles ("value_object"), in welchem konkrete Information zu der jeweiligen Operation stehen. Ziel ist es, das value_object erst in der „Rule-Controll-Engine“zu behandeln.

3.5 Spielelogik

Die Spielelogik fällt die Entscheidung innerhalb eines laufenden Spieles. Sie legt fest, wann beispielsweise ein Spiel beendet ist, welche Spieler gewonnen oder verloren haben und welche

Aktionen zu welchen Zeitpunkt ausführt werden müssen. Damit die Spielelogik Entscheidungen treffen kann, benötigt sie hierfür bestimmte Regeln, weshalb sie im Besitz einer eigenen Rule Engine ist und sich um diese Regeln kümmert.

3.5.1 Rule Engine

Eine Rule Engine ist ein Tool für die Formulierungen von Regeln in einer eigenen Regelsprache. Die Art der Regelsprache hängt dabei von der Umsetzung der jeweiligen Engine ab, kann jedoch soweit abstrahiert werden, dass es möglich ist, die Regeln in einer natürlichen Sprache zu formulieren. Die Regeln werden dabei auf konkrete Entitäten und Methoden angewandt. Der Vorteil in der Verwendung einer Rule Engine liegt darin, dass für die Erzeugung keine expliziten Programmierkenntnisse notwendig sind, so dass diese sowohl von Informatik-Experten, als auch von Programmierlaien erzeugt werden können. Des Weiteren ist ein starres Regelkonzept in einer Software mit einem generischen Ansatz, wie in dieser Arbeit behandelt, nur bedingt einsetzbar und hätte bei Anpassungen massive Änderungen zur Folge. Rule Engines bieten darüber hinaus folgende Eigenschaften an, welche jedoch je nach Tool variieren können [Wunderlich, 2006, S. 21]:

- Eine Organisierung für die einzelnen Regeln, d.h. Anordnungen der Regeln durch Algorithmen, Sortierungen, Kategorisierungen oder Priorisierungen.
- Die Möglichkeit von Berechtigungsvergaben, Authentifizierung oder Autorisierung für verschiedene Regelersteller und Benutzer.
- Der einfache Austausch von Regeln, während der Laufzeit der dazugehörigen Applikation.
- Eine Import- und Exportmöglichkeit von fremden Regeldefinitionen.
- Eine Konvertierung von Entscheidungsbäumen oder Entscheidungstabellen in Regeln.
- Die Visualisierung von Regeln durch Darstellungen in Form von UML Diagrammen, Programmablaufplänen etc.

Aufbau einer typischen Rule Engine

Die Regeln innerhalb einer Engine setzen sich aus der Aussagelogik zusammen. Dabei setzen sich mehrere Bedingungen mittels einer Konjunktion oder einer Disjunktion zu einem Verbund zusammen und werden daraufhin durch eine oder mehrere Implikationen oder Aktionen (Konsequenzen) zu Regeln zusammengefasst [Wunderlich, 2006, S. 45].

Wissensdatenbank (Knowledge base)

Die Wissensdatenbank einer Rule Engine beschreibt die Menge aller Regeln, welche in dieser vorhanden sind. Sie wird dabei für einen spezifischen Kontext und ein konkretes Wissensgebiet zusammengestellt. Sie fungiert als spezielle Datenbank für das Wissensmanagement und ist die Grundlage für die Sammlung von Informationen. Für gewöhnlich besteht die Wissensdatenbank dabei aus expliziten Informationen einer Organisation und aus Problemlösungen, Artikeln, White Paper, Benutzerhandbüchern und ähnlichen Daten. Die Datenbank benötigt dabei eine Klassifizierung ihres Inhaltes sowie dessen Formatierung und einer Suchfunktion für den Benutzer. Rule Engines existieren auf der Grundlage von solchen Wissenssammlungen. Die Sammlung von derartigen Informationen nennt man „Wissensaquisition“ (knowledge acquisition oder auch knowledge engineering) und bezeichnet die Transformation von Expertenwissen in eine vom Computer interpretierbare Form der Darstellung. Die Sammlung und Erzeugung einer solchen Wissensbasis ist ein zeitaufwendiger Prozess, weswegen durchaus versucht wird, eine solche Datenbank mit Hilfe von eigenständigem Lernen (z. B. neuronale Netze) zu erzeugen. Diese Thematik ist jedoch nicht Inhalt der hiesigen Arbeit, da sie in der hier vorgestellten Architektur nicht verwendet wird.

Regelverarbeitung

Eine Regel besteht für gewöhnlich aus einer oder mehreren Prämissen (Aussagen), welche miteinander verknüpft sind. Diese Verknüpfungen führen anschließend durch Konklusionen zu einer oder mehreren Aktionen. Die Rule Engine verarbeitet dabei eine Menge von Regeln, welche aus der Wissensdatenbank kommen. Vor der Anwendung muss entschieden werden, welche Regel, für den entsprechenden Einsatz überhaupt relevant sind. Die Menge von aktiv anwendbaren Regeln sind dabei der Regelsatz (Rule set). Bei der Regelverarbeitung einer Regel wird die Prämisse (linker Teil einer Regel) geprüft. Bei erfolgreicher Prüfung wird

die Konsequenz (rechter Teil einer Regel) ausgeführt. Diese Prüfung geschieht durch die Kombination von Fakten, welche einen anschließenden Wahrheitswert wiedergeben. Die Fakten stammen dabei aus einer sogenannten „Fact Base“. Das nachfolgende Beispiel erläutert den Zusammenhang zwischen Fakt, Prämisse und Konsequenz:

Fakt: Die Herdplatte hat eine Temperatur an der Oberfläche von 52 Grad Celsius. *Prämisse:* Die Herdplatte ist heißer als 50 Grad. (Prämisse ist erfüllt) *Konsequenz:* Die Temperatur herunterdrehen. (Konsequenz ist auszuführen)

Frames

Als Frames bezeichnet man bei einer Rule Engine die Sammlung von Attributen, die kein Verhalten aufweisen. Ein Frame besitzt dabei Datenzustände, sogenannte Slots, zu definierten Zeitpunkten. Wird ein Datenzustand eines Frames geändert, so kann diese Änderung zu Ausführung einer Aktion vor, während oder danach führen.

Case-based reasoning

Geht man davon aus, dass Frames eine Darstellung einer Datenstruktur vor der Verarbeitung einer Regel repräsentieren, dann ist Case-based reasoning die Anwendung von Konsequenzen auf diese Frames. Ein Frame gilt also als fallbasierte Schlussfolgerung, sozusagen als Lösung für eine spezifische Aufgabenstellung. Bei einem Case-based resoning System erhält die Rule Engine den Ausgangsdatenzustand (problem input). Ein Retriever bearbeitet diese Daten und findet den case, welcher am besten auf den Frame passt. Das Ergebnis (match) wird anschließend von dem Retriever zurückgegeben. In der Praxis wird ein Ergebnis allerdings nur ausgeführt, wenn zu 100 Prozent matched, d.h. wenn ein exakter Wahrheitswert ermittelt wurde [Wunderlich, 2006, S. 46].

Inference Engine

Die Inference Engine kann als Motor der Rule Engine angesehen werden. Sie zieht Schlussfolgerungen auf Grundlage der definierten Regeln, also die Ausführung einer Konsequenz der damit verbundenen Regel.

Working Memory

In dem Working Memory werden die Fakten der Regelanwendungen gespeichert. Die Rule Engine nimmt sich dort die zur verarbeitenden Elemente heraus, um zu prüfen, ob eine Regel anwendbar ist oder nicht. Der explizite Aufruf einer Regelausführung wird als Rules Firering (Feuern) bezeichnet. Die Art Der Rule Engine bestimmt hierbei die Reihenfolge der Regeln und dessen Feuerung [Wunderlich, 2006, S. 47].

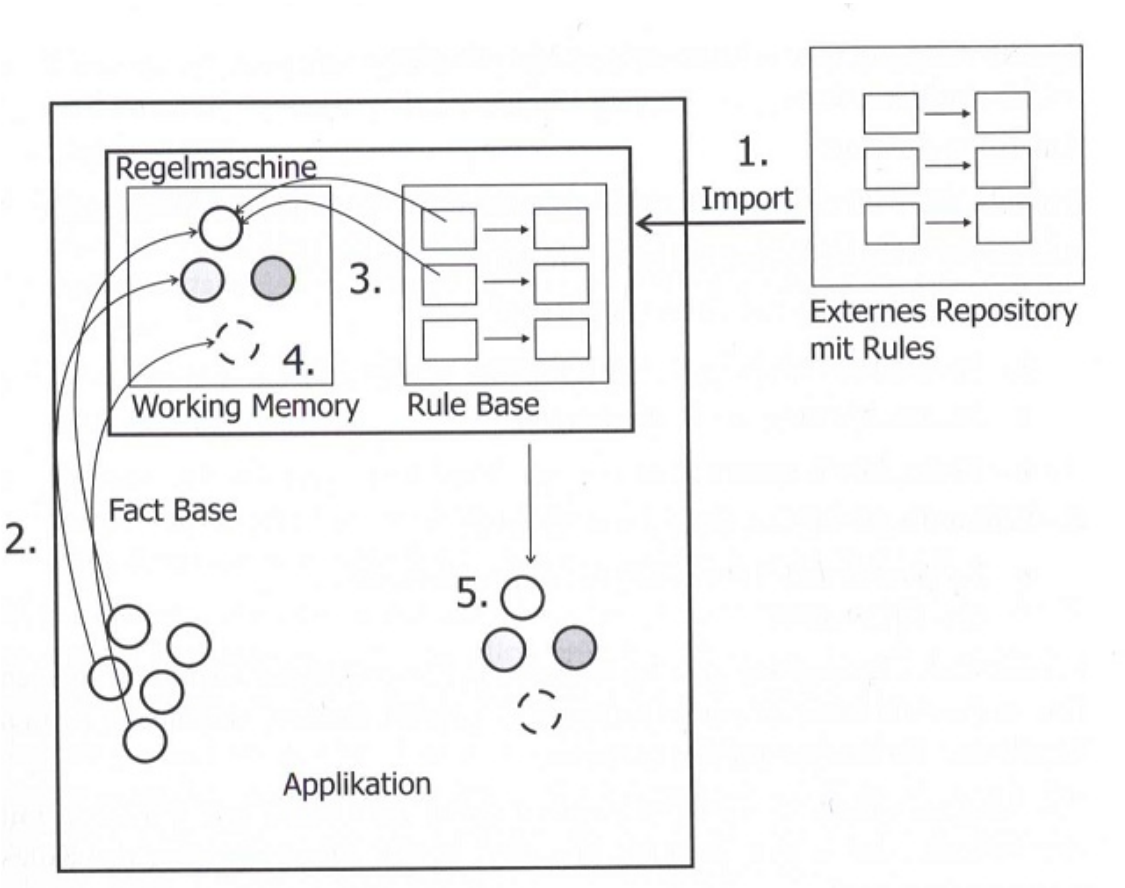


Abbildung 3.50: Zusammenspiel einer Rule Engine mit den Regeln und Fakten [Wunderlich, 2006, S. 52]

Abbildung 3.50 zeigt das Zusammenspiel einer Rule Engine mit den Regeln und Fakten [Wunderlich, 2006, S. 52].

1. Bevor die Rule Engine starten kann, muss sie zunächst die Regeldaten aus einem externen Repository importieren. Ein Repository kann hierbei eine Datei, eine Datenbank oder eine sonstige Quelle sein, welche Daten bereitstellen kann.

2. Anschließend findet die Erzeugung des Working Memorys statt. Ausgewählte Anwendungsdaten werden hierbei dem Working Memory von der Fact Base zur Verfügung gestellt.

3. Einige Rule Engines entscheiden bereits bei der Einfügung in das Working Memory, welche Bedingungsteile der Regel erfüllt sind. Eine solche Zuordnung von Regeln und ausführenden Aktionen wird in der Agenda vorgemerkt.

4. Die Rule Engine feuert die Regeln ab, wodurch es gegebenenfalls zu Modifizierung, Löschung oder auch Neuerzeugung von Objekten in dem Working Memory kommen kann. Auch Daten, welche außerhalb liegen, können manipuliert werden, jedoch ist hiervon abzuraten.

5. Als letztes wird der Zustand des Working Memorys, bzw. die dort vorhandenen Objekte an die Anwendung zurückgegeben, damit diese mit den Veränderungen weiterarbeiten kann.

3.5.2 Drools

Drools ist eine Open Source Rule Engine und wurde von der Firma Jboss Inc. entwickelt. Drools verwendet Forward Chaining. Es unterstützt also damit die typische Schlussfolgerungsverkettung, d.h. Aktionen werden aufgrund von Bedingungen ausgeführt. Für die Abarbeitung der Regeln verwendet Drools den RETE Algorithmus, welcher sicherstellt, dass nur Regeln durchlaufen werden, welche durch Objekte im Working Memory effektiv anschlagen und genutzt werden können. Bei der Auswahl der Regeln existiert keine feste Reihenfolge. Die Organisation der Regelausführung geschieht durch eine Regelstruktur. Der RETE Algorithmus ordnet Regeln in einer Baumstruktur an und durchläuft nur die Zweige, die sinnvoll hintereinander folgen. Für die interne Abbildung von Regeln verwendet Drools Java-Objektklassen. Die aktuelle Drools-Version (Drools 5.0) ist in fünf Module unterteilt [The JBoss Drools team, 2013a]:

- Expert : Die traditionelle Rule Engine

- Guvnor: Ein webbasiertes Steuerungssystem, das Regeln als Business-Rule-Management-System (BRMS) referenziert.
- Fusion: Eigenständiges Modul, welches für event processing verantwortlich ist und diese steuert.
- Flow: Liefert den workflow oder business Prozessmöglichkeiten an die Drools Plattform
- Planner: Optimiert die Planung und reduziert die Ressourcen.

Für die hier thematisierte Plattform kommt besonders das Drools Expert in Betracht, weswegen die anderen Versionen hier nicht weiter ausgeführt werden. Drools Expert bietet die benötigtige Rule Engine sowie die Erstellung von Regeln an.

Drools Expert

Expert ist als spezifische Rule Engine von Drools entwickelt worden und basiert auf dem RETE Algoritmus.

```
package rules
import com.data.Player
import com.data.RuleController

//declare any global variables here
global com.data.BasicLocationOperations basicOperations


rule "Hunter caught victim"
    when
    $p1 : Player(role == "hunter")
    $p2 : Player(role =="victim")
    $ruleController: RuleController()

    eval(basicOperations.distanceBetweenToGeoLocationsInKilometers
        ($p1.getLatitude(),$p1.getLongitude(),$p2.getLatitude(),$p2.getLongitude())< 0.005)

    then
        System.out.println( "Player: "+$p2.getName()+" was caught by: "+$p1.getName());
        $ruleController.gameOver($p1, $p2);
end
```

Abbildung 3.51: Aufbau einer Beispielregel in Drools Expert

Abbildung 3.51 zeigt den Aufbau einer Regel in Drools Expert. Die hier zu sehende Regel wurde aus dem Referenzbeispiel „Räuber und Gendarm„“ entnommen und überprüft, ob ein

Jäger sein Ziel bekommen hat. Da Drools mit Objekten arbeitet, ist es notwendig, die Klassen zu importieren, welche von der/den Regel/n verwendet werden. In diesem Fall ist es die Klasse „Player“, welche alle spielrelevanten Eigenschaften eines Spielers in Form einer einfachen POJO Klasse bereitstellt. Des Weiteren beinhaltet die Regeldatei eine Referenz des RuleControllers, um diesen über bestimmte Ergebnisse einer Regelausführung zu informieren. Als letztes wird eine Klasse benötigt, um spezielle regelspezifische Berechnungen durchführen zu können. Es ist zwar möglich eine solche Berechnung mithilfe von Java-Funktionen direkt in dem Rule-File zu implementieren, jedoch ist es das Ziel eines generischen Ansatzes, so wenig Code wie möglich in den Files zu haben, weswegen die Berechnung in die Klasse BasicLocationOperations ausgelagert wurde.
Das Schlüsselwort „rule“ deklariert den Anfang der Regeldefinition samt dem dazugehörigen Regelnamen. Innerhalb des „when“ Blockes können die Bedingungen für die entsprechende Regel eingetragen werden. In diesem konkreten Beispiel geht es darum, dass ein Player-Objekt existieren muss, welche für das Attribut „role“ den Wert „hunter“ besitzt. Um in weiteren Bedingungen auf dieses Objekt zugreifen zu können, wird es an die lokale Variable $p1 gebunden. Zusätzlich dazu muss ein zweites Player-Objekt vorhanden sein, bei welchem „role“ den Wert „victim“ erhalten hat. Dadurch wird sichergestellt, dass es sich um zwei unterschiedliche Rollen handelt. In diesem einfachen Beispiel gibt es eigentlich nur eine konkrete Bedingung, anhand der festgestellt werden kann, ob ein Jäger sein Ziel gefangen hat, nämlich die Entfernung zu diesem. Hierfür ist eine Berechnung der Entfernung beider Spieler notwendig. Eine solche Berechnung geschieht in der zuvor erwähnten BasicLocationOperations. Abbildung 3.52 zeigt eine solche Berechnung im Java-Code. Dabei wurde auf die bewährte Spherical Law of Cosines zurückgegriffen, die für die Entfernungsberechnung zwischen zwei Punkten, in Form von Längen- und Breitengraden zuständig ist [Weisstein, 2013]. Ein Jäger hat also demnach ein Spiel gewonnen, wenn er sich weniger als fünfzig Meter von seinem Ziel entfernt befindet. Das Drools-Keyword „eval“ prüft, ob der Wert der Berechnung den Wahrheitswert „true“ zurückliefert. Sind alle Bedingungen des „when“ Blocks erfüllt, so wird als Konsequenz der „then“ Teil ausgeführt. Hier wird der ruleController über die Beendigung des Spieles informiert, indem die Drools-Engine eine seiner Methoden aufruft und dieser die Gewinner und Verlierer (der erste Übergabeparameter ist der Gewinner) übermittelt.

```
public static double distanceBetweenToGeoLocationsInKilometers(
                    double lat_1, double lon_1, double lat_2, double lon_2) {

    double lat1 = lat_1 * Math.PI / 180.0; // convert from degrees to radians
    double lon1 = lon_1 * Math.PI / 180.0; // convert from degrees to radians;
    double lat2 = lat_2 * Math.PI / 180.0; // convert from degrees to radians;
    double lon2 = lon_2 * Math.PI / 180.0; // convert from degrees to radians;

    double R = 6371; // km
    double d = Math.acos(Math.sin(lat1)*Math.sin(lat2) +
                    Math.cos(lat1)*Math.cos(lat2) *
                    Math.cos(lon2-lon1)) * R;
    return d;
}
```

Abbildung 3.52: Berechnung der Entfernung zweier Spieler anhand deren Längen- und Breitengraden

Um die in Abbildung 3.51 gezeigte Regel im Programmcode einbinden zu können, sind noch einige Drools-spezifische Anpassungen notwendig. Abbildungen 3.53 und 3.54 zeigen einen kleinen Ausschnitt dieser Anpassungen. Hierbei wird davon ausgegangen, dass sowohl die KnowledgeBase bereits das zuvor erzeugte Rule-File (geschieht in der Methode readKnowLedgeBase()) beinhaltet, als auch die Initialisierung der hier verwendeten Objekte stattgefunden hat. Für den Regeldurchlauf wird als nächstes eine Knowledge-Session erzeugt und dieser die notwendigen Objekte hinzugefügt. Anschließend wird mit der Methode fireAllRules() dafür gesorgt, dass alle Regeln durchlaufen werden und gegebenenfalls matchen.

```
private static KnowledgeBase readKnowledgeBase() throws Exception {
    KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
    kbuilder.add(ResourceFactory.newClassPathResource("GameRules.drl"), ResourceType.DRL);
    KnowledgeBuilderErrors errors = kbuilder.getErrors();
    if (errors.size() > 0) {
        for (KnowledgeBuilderError error: errors) {
            System.err.println(error);
        }
        throw new IllegalArgumentException("Could not parse knowledge.");
    }
    KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
    return kbase;
}
```

Abbildung 3.53: Beispielprogrammcode zur Einbindung einer Rule-Datei in eine Drools-KnowledgeBase

```
// load up the knowledge base
KnowledgeBase kbase = readKnowledgeBase();
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
// go !
ksession.insert(player1);
ksession.insert(player2);
ksession.insert(ruleControler);

ksession.fireAllRules();
```

Abbildung 3.54: Hinzufügung der zu verwendeten Objekte innerhalb einer Knowledge-Session

Nach einer ersten Analyse dieses Verfahrens bleibt festzustellen, dass die Regeldefinition trotz Auslagerung der spezifischen Berechnungen noch immer sehr codelastig ist und die Regeln von einem Nicht-Programmierer so nur schwer umzusetzen sind. Aus diesem Grund wird im nächsten Abschnitt auf die Erstellung und Einbindung einer Domain Specific Language in Drools beschrieben, um die Regelerstellung auch für Nicht-Programmierer umsetzbar zu gestalten.

3.5.3 Domain Specific Language (DSL)

Dieser Abschnitt beschreibt Erstellung sowie Einbindung einer DSL innerhalb von Drools Expert. Da im vorherigen Absatz festgestellt wurde, dass die Definition von Regeln in Drools sehr codelastig und für Nicht-Programmierern nur schwer umzusetzen ist, wurde der Entschluss getroffen, eine eigenständige Domain Specific Language (DSL) zu entwickeln, welche anschließend zur Regeldefinition verwendet wird, um so den Nicht-Programmierern bei der Festlegung von Regeln zu unterstützen. Genau für solche Fälle bietet Drools die Option zur Einbindung einer solchen Domain Specific Language an [The JBoss Drools team, 2013b, Kap. 4.10.1, S. 215].

3.5.4 Domain Specific Language (DSL) in Drools

Domain Specific Languages bieten die Möglichkeit zur Erstellung einer „eigenen“ Regelsprache, welche genau für die entsprechende Problemdomäne zuständig bzw. geeignet ist. DSL-Definitionen bestehen aus DSL-„Sätzen“, welche in die typischen Regelkonstrukte (.drl Dateikonstrukte) von Drools Expert umgewandelt werden. Dies bringt den Vorteil, dass die komplette zugrundeliegende Regelsprache und die Engine Features genutzt werden können. Im

Falle einer bestehenden DSL werden die zu entwickelnden Regeln in DSL-Regeldateien (.dslr Dateien) definiert, welche wiederum in die typischen Regelkonstrukte (.drl Dateikonstrukte) übersetzt werden. .dsl und .dslr Dateien sind Klartextdateien. Domain Specific Languages haben keine Auswirkungen auf die Laufzeit der Rule-Engine, da die Übersetzung der DSL in das .drl-Format zur Compilezeit stattfindet [The JBoss Drools team, 2013b, Kap. 4.10.1, S. 215].

Konditional-Ausdrücke („condition“/„when“ -Statements) und Konsequenz-Aktionen („consequence“/„then“ -Statements) sind durch den Drools DSL-Mechanismus individuell anpassbar. Die Abbildung 3.55 zeigt ein Beispiel für einen Konditional-Ausdruck.

```
[condition]Set player with role {role} to {var}={var}: Player(role=={role})
```

Abbildung 3.55: Beispiel für ein DSL-Mapping

Das in Abbildung 3.55 genutzte Schlüsselwort „[condition]“ gibt den Gültigkeitsbereich des Ausdrucks an. Es sagt also aus, ob ein Ausdruck für den Konditional-Bereich (die linke Seite der Regel) oder für den Konsequenz-Teil (die rechte Seite der Regel) gültig ist. Das Schlüsselwort „[condition]“ oder „[when]“ wird für konditional Ausdrücke verwendet, während das Schlüsselwort „[consequence]“ oder „[then]“ für Konsequenz-Aktionen genutzt wird. Der Bereich hinter dem eckig eingeklammerten Slüsselwort („[condition]“) und dem Gleichheitszeichen („=“) entspricht dem Ausdruck, welcher bei der Regelerstellung verwendet wird. Innerhalb des Ausdrucks wird üblicherweise eine natürliche Sprache passend zur einzusetzenden Domäne angewendet. Bei der zu entwickelnden DSL wurde sich darauf geeignet, die natürliche englische Sprache zu gebrauchen und passende Begriffe aus der Spieldomäne mit einzubringen. Der Bereich nach dem Gleichheitszeichen („=“) entspricht dem Mapping des Ausdrucks in die Drools-Regelsprache. Die Struktur des Strings ist abhängig von dem Gültigkeitsbereich des Ausdrucks (Konditional- oder Konsequenz-Bereich). Falls der Ausdruck für den konditional Bereich vorgesehen ist, so sollte die Bedingung der regulären Konditional-Syntax entsprechen, wie in Abbildung 3.55 zu sehen ist. Falls der Ausdruck für den Konsequenz-Teil einer Regel vorgesehen ist, so entspräche der Term einem Java-Statement.

Immer wenn der DSL-Parser eine Zeile aus der DSL-Regeldatei (Dateiendung .dslr) einem Ausdruck aus der DSL-Definition (Dateiendung .dsl) zuordnen kann, führt der DSL-Parser für die entsprechende Zeile drei Phasen von String-Manipulationen durch. Im ersten Schritt werden vom DSL-Parser die Stringwerte der Ausdrücke, welche Variablennamen in geschweiften Klammern enthalten (siehe Abbildung 3.55: role) extrahiert. Anschließend werden die

extrahierten Stringwerte in die gleichbenannten „Platzhalter“ mit geschweiften Klammern auf der rechten Seite des DSL-Mapping eingefügt. Zuletzt ersetzt der eingefügte Stringwert die passenden „Platzhalter“ in der Zeile der DSL-Regeldateien.

Die Abbildung 3.56 zeigt eine Erweiterung bei der Erstellung einer DSL-Definition in Drools.

```
[condition]Set player with role "{role:\w+}" to {var}={var}: Player(role=="{role!lc}")

[condition]Set player with id {id:\d+} to {var}={var}: Player(id=={id})
```

Abbildung 3.56: Erweitertes Beispiel für ein DSL-Mapping

Des Weiteren bietet Drools die Möglichkeit, reguläre Ausdrücke auf der linken Seite (vor dem Gleichheitszeichen („=“)) des DSL-Mappings einzusetzen, wie in der Abbildung 3.56 zu sehen {role:\w+}/{id:\d+}. Zur Definition von regulären Ausrücken werden die Standard-Java-Pattern (Zeichenklauseln) verwendet. Aus diesem Grund sorgt das Pattern „\w+“ dafür, dass nur Wortzeichen ([a-zA-Z_0-9]) und mindestens ein Wortzeichen für den Platzhalter „{role:\w+}“ eingesetzt werden können/kann. Ebenfalls sorgt das Pattern „\d+“ dafür, dass nur Ziffern und mindestens eine Ziffer für den Platzhalter „{id:\d+}“ eingefügt werden können.

Zusätzlich besteht die Option, String-Manipulationen auf der rechten Seite der DSL-Definition anzuwenden, so wie in dem obigen Beispiel {role!lc} zu sehen. Die String-Manipulationen „!lc“ dient dazu, dass alle Buchstaben in Kleinbuchstaben umkonvertiert werden. Solche String-Manipulationen werden häufig dann verwendet, wenn von einer natürlichen Sprache abstrahiert werden soll, um dadurch eindeutige Namen erhalten zu können.

Außerdem bietet Drools bei der DSL-Definition den Einsatz von Anführungszeichen, wie in der vorherigen Abbildung „”{role:\w+}”“ zu sehen. Diese sind besonders nützlich bei der Angabe von textuellen Daten, da deren Grenzen an den Anführungszeichen zu erkennen sind [The JBoss Drools team, 2013b, Kap. 4.10.1, S. 215].

Die Abbildung 3.57 zeigt die für die Architektur entwickelte Drools DSL. Diese DSL definiert einige exemplarische kontextsensitive Beispiele. Eine vollständige DSL würde den Rahmen dieser Arbeit überschreiten.

Die ersten beiden Konditional-Ausdrücke („Set player with role/id … to {var}“) sorgen dafür, dass Spieler anhand ihrer Spielerrolle/Spieler-Id identifiziert und in einer Variablen zwischengespeichert werden können. Die folgenden zwölf Konditional-Ausdrücke („Is distance

Language Expression	Scope
Set player with role "{role:\w+}" to {var}	[condition]
Set player with id {id:\d+} to {var}	[condition]
Is distance between player {playerX} and player {playerY} less than or equal to {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player {playerX} and player {playerY} less than {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player {playerX} and player {playerY} greater than or equal to {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player {playerX} and player {playerY} greater than {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player {playerX} and player {playerY} equal to {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player_role "{playerRoleX:\w+}" and player_role "{playerRoleY:\w+}" less than or equal to {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player_role "{playerRoleX:\w+}" and player_role "{playerRoleY:\w+}" less than {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player_role "{playerRoleX:\w+}" and player_role "{playerRoleY:\w+}" greater than or equal to {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player_role "{playerRoleX:\w+}" and player_role "{playerRoleY:\w+}" greater than {inKilometers:\d+\.\d+} kilometers	[condition]
Is distance between player_role "{playerRoleX:\w+}" and player_role "{playerRoleY:\w+}" equal to {inKilometers:\d+\.\d+} kilometers	[condition]
LogVar : {var}	[consequence]
LogText : "{text}"	[consequence]
Game over winner is player {winner} and loser is player {loser}	[consequence]
Prompt player {player} to go to geolocation latitude {latitude:\d+\.\d+} longitude {longitude:\d+\.\d+} with minimum distance {inKilometers:\d+\.\d+} kilometers	[consequence]
Show player {playerX} the geolocation position from player {playerY} with hint text "{hintText}"	[consequence]
Show distance between player {playerX} and player {playerY} with hint text "{hintText}"	[consequence]

Abbildung 3.57: Entwickelte Drools DSL der Architektur

between player/player_role ... and player ... less than or equal to/less than/greater than or equal to/greater than/equal ... kilometers“) eignen sich dafür, Regeln zu definieren, welche bei einer angegeben Distanz (in Kilometern) zwischen zwei Spielern und einer passenden ausgewählten Ordnungsrelation/Äquivalenzrelation „abgefeuert“ werden. Die als nächstes kommenden Konsequenz-Aktionen („LogVar“ und „LogText“) finden Verwendung bei der Ausgabe von Logs. „LogVar ...“ wird beim Logging von zwischengespeicherten Variablen eingesetzt, welche zu einem früheren Zeipunkt mit den Konditional-Ausdrücken „Set player with ...“ gesetzt wurden. „LogText ...“ findet Verwendung beim Logging von Text. Die nachfolgende Konsequenz-Aktion („Game over winner is player ... and loser is player ...“) dient dazu, ein laufendes Spiel zu beenden und den entsprechenden Spielern das Spielergebnis mitzuteilen. Die nächste Konsequenz-Aktion („Prompt player ... to go to geolocation latitude ... longitude .. with minimum distance ... kilometers“) wird in den Fällen eingesetzt, wenn ein Spieler aufgefordert werden soll, zu einer bestimmten Geolokation zu gehen. Die anschließende Konsequenz-Aktion („Show player ... the geolocation from player ... with hint text ...“) findet Anwendung, wenn einem Spieler die Geo-Position eines anderen Mitspielers übergeben werden soll. Die letzte Konsequenz-Aktion („Show distance between player ... and player ... with hint text ...“) dient zur Übermittlung der Distanz zweier Spieler.

Die Abbildung 3.58 zeigt das „vereinfachte“ Regelbeispiel („Hunter caught victim“) aus dem vorherigem Abschnitt unter Zuhilfenahme der entwickelten DSL. Zur Einbindung der DSL in die Regeldatei (.dslr) wird die Zeile „expander <DSL-Dateiname>.dsl“ benötigt.

```
rule "Hunter caught victim"
    when
        Set player with role "victim" to $victim
        Set player with role "hunter" to $hunter
        Is distance between player $victim and player $hunter less than or equal to 0.05 kilometers
    then
        Game over winner is player $hunter and loser is player $victim
end
```

Abbildung 3.58: Das „vereinfachte“ Regelbeispiel unter Zuhilfenahme der entwickelten DSL

Im „when“-Teil der Regel werden die Spieler anhand ihrer Spielerrollen in die Variablen $victim und $hunter zwischengespeichert. Sollte die Distanz zwischen $hunter und $victim kleiner gleich 50 m (0.05 km) sein, so ist das Spiel beendet und der $hunter ist Gewinner des Spiels und $victim ist Verlierer des Spiels.
Durch den Einsatz der DSL sind zur Erstellung der Regeln keine Programmierkenntnisse mehr nötig, da die vordefinierten DSL-Ausdrücke der natürliche Sprache sehr nahe kommen und von der ursprünglichen Codelastigkeit weniger zu verspüren ist.

Zur Nutzung der DSL-Definition (.dsl Dateiendung) und der DSL-Regeldefinition (.dslr Dateiendung) bedarf es einige spezifischen Drools-Anpassungen im Java-Programmcode, wie in Abbildung 3.59 zu sehen ist.

```
private static KnowledgeBase readKnowledgeBase() throws Exception {
    KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

    kbuilder.add(ResourceFactory.newClassPathResource("Game.dsl"), ResourceType.DSL);
    kbuilder.add(ResourceFactory.newClassPathResource("Game.dslr"), ResourceType.DSLR);

    KnowledgeBuilderErrors errors = kbuilder.getErrors();
    if (errors.size() > 0) {
        for (KnowledgeBuilderError error: errors) {
            System.err.println(error);
        }
        throw new IllegalArgumentException("Could not parse knowledge.");
    }
    KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
    return kbase;
}
```

Abbildung 3.59: Einbindung einer DSL-Definition und einer DSL-Regeldefinition innerhalb Drools Expert

Der KnowledgeBase muss die DSL-Definition (.dsl Dateiendung) und die DSL-Regeldefinition (.dslr Dateiendung) als Ressourcen hinzugefügt werden, anstatt der „typischen“ Regeldefinition (.drl Dateiendung), wie im vorherigem Abschnitt zu sehen ist.

Ein weiteres Regelbeispiel folgt wiederum aufbauend auf der entwickelten DSL (Abbildung 3.60).

```
rule "Show geolocation from victim when victim is to much to far away for hunter"
    when
        Set player with role "victim" to $victim
        Set player with role "hunter" to $hunter
        Is distance between player $victim and player $hunter greater than or equal to 0.75 kilometers
    then
        Show player $hunter the geolocation position from player $victim with hint text "last geolocation from victim"
end
```

Abbildung 3.60: Weiteres Regelbeispiel unter Zuhilfenahme der entwickelten DSL

Im „when“-Teil wurde ähnlich wie um vorherigem Regelbeispiel vorgegangen. Der Spieler mit der Spielerrolle „victim“ wurde in der Variable $victim und der Spieler mit der Spielerrolle „hunter“ in der Variable $hunter zwischengespeichert. Sollte die Distanz zwischen $victim und $hunter größer gleich 750 m (0.75 km) sein, so wird dem $hunter die Geo-Position (Breiten- und Längengrad) des $victims übermittelt werden, um so dem Spieler $hunter eine Chance zu geben, den Spieler $victim im weiterem Verlauf des Spiels zu fangen.

Die Abbildung 3.61 zeigt ein weiteres Regelbeispiel aufbauend auf der entwickelten DSL.

```
rule "mission for victim when victim is too far away for hunter"
    when
        Set player with role "victim" to $victim
        Set player with role "hunter" to $hunter
        Is distance between player $victim and player $hunter greater than or equal to 0.5 kilometers
    then
        Prompt player $victim to go to geolocation latitude 53.59601775 longitude 10.03088396 with minimum distance 0.05 kilometers
end
```

Abbildung 3.61: Weiteres Regelbeispiel unter Zuhilfenahme der entwickelten DSL

Der „when“-Teil der Regel beginnt identisch wie bei den vorherigen vorgestellten Regelbeispielen. Der Spieler mit der Spielerrolle „hunter“ wird in der Variablen $hunter und der Spieler mit der Spielerrolle „victim“ in der Variablen $victim zwischengesichert. Sollte die Distanz zwischen $hunter und $victim größer gleich 500 m (0.5 km) betragen, so würde der Spieler $victim aufgefordert werden, eine bestimmte Geo-Position (Breiten- und Längengrad) anzulaufen, um so die Distanz zwischen den Spielern zu verringern.

An dieser Stelle ist die Erstellung sowie Einbindung der DSL in Drools soweit abgeschlossen. Eine Verbesserung der DSL könnte durch Adaptionen von etablierten DSLs aus dem Gaming-

Bereich ermöglicht werden. Dieses Vorhaben würde allerdings den Rahmen dieser Arbeit überschreiten.

3.6 Client

Das folgende Kapitel befasst sich mit der Umsetzung eines Referenzclients auf Grundlage des in Kapitel 2.3 dargestellten Fallbeispiels. Um eine maximale Plattformunabhängigkeit zu bieten, wurde sich für die Implementation mittels HTML und Javascript in Form einer Web-App entschieden, allerdings wird diese für die ersten Testversuche auf mobile Android-Geräte aufgespielt werden, da Google eine komplett kostenlose Entwicklungsumgebung anbietet, während bei anderen Anbietern wie z.B. Microsoft und Apple hierfür weitere Kosten entstehen würden. Des Weiteren verfügt das Android OS mit bis zu 75 % im vergangen Jahr, über den höchsten Marktanteil aller Smartphones [Financial Times Deutschland, 2012] und ist deswegen für den Einstieg in den Markt mit Abstand am geeignetesten.

Um eine möglichst vollständige Interaktion mit allen Bereichen der Plattform gewährleisten zu können, hat die Clientsoftware folgende Punkte zu erfüllen:

- Login
- Spielerfindung
- Eigentliches Spiel

Login

Der Login/Authentifizierung wurde auf der Clientseite relativ schlank gehalten und setzt einen bestehenden Account auf der Serverseite voraus. Nach Eingabe des korrekten Benutzernamens und Passworts gelangt man im Anschluss zu der Spielerfindung.

Abbildung 3.62: Client-Screenshot vom Login/Authentifizierung-Layout

Spielerfindung

Die Spielerfindung läuft für den Client vollständig transparent ab. Der Benutzer soll lediglich seine Bereitschaft, zu spielen, signalisieren und darüber informiert werden, wenn ein passender Mitspieler gefunden wurde. Der Client sendet dabei im Hintergrund seine aktuelle Position in regelmäßigen Abständen an den Server.

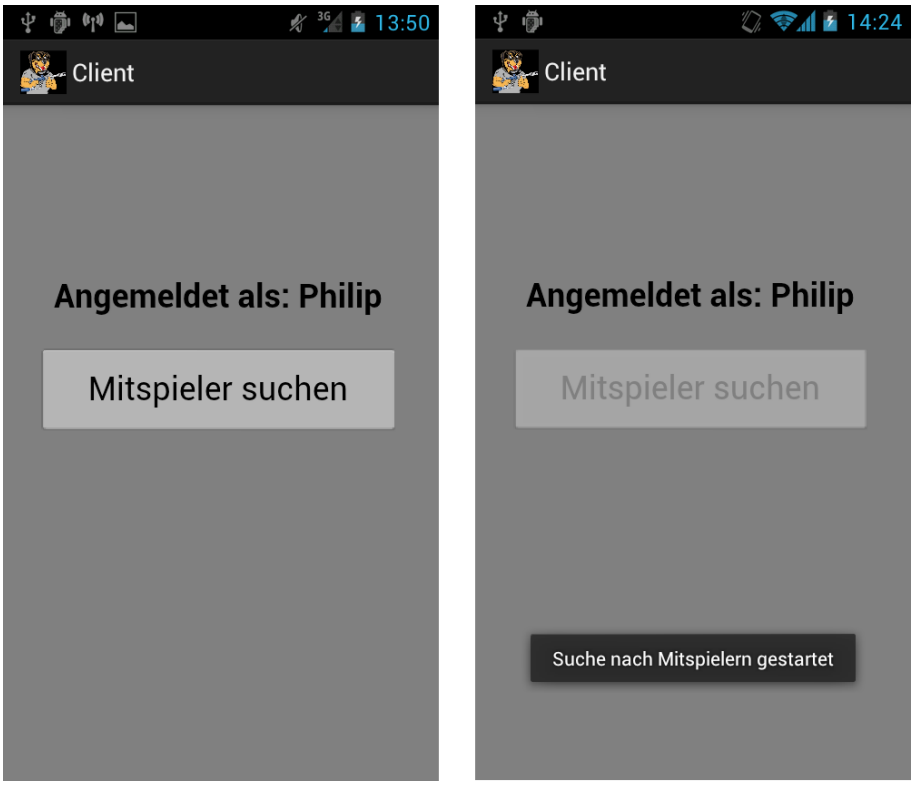


Abbildung 3.63: Client-Screenshots vom Spielerfindung-Layout

Eigentliches Spiel

Wurde ein passender Mitspieler gefunden, so beginnt das eigentliche Spiel. Wie in Abbildung 3.64 zu sehen, ist der Bildschirm zweigeteilt. Der mit der „Eins“ deklarierte Teil stellt das Infofenster dar. Hier wird der Spieler am Anfang eines Spieles über seine vom Server zugeteilte Rollte informiert und kann zudem weitere spielrelevante Informationen erhalten. Der zweite Teil besteht aus der Google-Maps Karte, auf welcher die aktuelle Position des Spielers dargestellt wird (Punkt 2). Je nachdem, ob er sich seinem Kontrahenten nähert oder sich von diesem entfernt, verändert sich das Positionssymbol zu einem – oder einem +. In besonderen Fällen kann auch kurzzeitig die Position des Kontrahenten angezeigt werden, jedoch wird ein solches Szenario ausschließlich von Seiten des Servers erzeugt.

Abbildung 3.64: Client-Screenshot vom Spiel-Layout

3.7 Fazit

In diesem Kapitel wurde die Architektur für die zu entwickelnde Plattform dargestellt und mittels Sequenzdiagrammen, welche auf Grundlage des Referenzbeispieles aus Kapitel 2.3 erschaffen wurden, detailliert dargestellt. Außerdem wurde auf die verwendeten Technologien zur Datenpersistierung, Nachrichtenübertragung und Regelwerk-Steuerung eingegangen. Die erfolgreiche Umsetzung wurde durch die Entwicklung und Verwendung eines kompakten Testclients demonstriert. Zusammenfassend lässt sich sagen, dass die Realisierung der hier thematisierten Plattform erfolgreich umgesetzt werden konnte, jedoch konnte der generische Ansatz nicht, wie anfangs geordert, vollständig übernommen werden, so dass bei der Erzeugung eines neuen Spieles auch serverseitig einige Codeanpassungen notwendig sind. Gründe hierfür liegen u.a. darin, dass die Drools-Rules-Engine mit konkreten Java-Objekten arbeitet, welche sich von Spiel zu Spiel verändern können. Ebenso wurde aus Gründen des Umfanges darauf verzichtet, den Feature-Vektor innerhalb der Pairing Komponente zu dynamisieren,

wodurch dieser ebenfalls mit konkretem Java-Code umgesetzt wurde. Abgesehen davon sind die konkreten Spieleregeln durch die Verwendung einer DSL soweit abstrahiert worden, dass es möglich ist, diese unter bestimmten Rahmenbedingungen auch ohne die Verwendung von Programmcode einzubinden und zu verändern. Die DSL orientierte sich dabei an der natürlichen Sprache und zeigt damit, dass die Grundidee der Arbeit erfüllt wurde. Da die Plattform aufgrund der zeitlichen Begrenzung durch die hiesige Thesis noch nicht im vollen Umfang ihrer Möglichkeiten ausgeschöpft wurde, bieten sich folgende Themen als eine hervorragende Erweiterung an:

- Das Kapitel 3.5.4 hat bereits gezeigt, wie der generische Ansatz innerhalb der Regelbasierung mittels einer eigens geschriebenen DSL auszusehen hat. Diese ist in ihrem Umfang jedoch beschränkt, da sie exemplarisch ihm Rahmen dieser Thesis geschrieben wurde. Eine Erweiterungsmöglichkeit bestünde darin, eine bereits, speziell für Games, entwickelte DSL zu verwenden. Eine detaillierte Befassung mit diesem Thema liefert die Masterarbeit von Andre Goldflam, welche eine Vertiefung in diesen Abschnitt liefert [Goldflam, 2012].

- Ein weiteres, bereits im Fazit des Kapitels Design & Realisierung, angesprochenes Problem ist die Verwendung von konkreten Java-Objekten in der Drools-Rule-Engine. Dieses Verhalten steht etwas im Widerspruch zum gewünschten generischen Ansatz. Glücklicherweise bietet Drools jedoch die Möglichkeit an, Objekte mittels type declaration zu erzeugen, welche wiederum durch eine DSL definiert werden könnten. Diese Art und Weise der Objekterzeung ist jedoch ziemlich zeitaufwendig und würde den Rahmen dieser Arbeit bei Weitem überschreiten.

- Da die Authentifizierung und die Datensicherheit nicht als Schwerpunkt dieser Arbeit galt, wurden diese Bereiche zwar im theoretischen Teil angesprochen, jedoch nicht mit umgesetzt. Eine für einen möglichen Release der Plattform notwenige Erweiterung wäre deshalb die Implementierung eines Protokolls, welches für Sicherheit notwendig ist. Empfehlenswert wäre in diesem Fall die Verwendung von LDAP, da dieses problemlos mit der ActiveMQ eingebunden werden kann.

- Ein weiterer Punkt ist die Skalierbarkeit der Plattform. Zwar wurde die Server-Architektur so designt, dass die Software auch parallel auf verschiedenen Maschinen laufen kann, jedoch wurde dieses Verhalten noch nicht ausreichend getestet und gilt deshalb noch nicht als sicher. Des Weiteren ist die Datenbank bisher an nur einer zentralen Stel-

le vorhanden, wodurch sich dort ein potentieller Engpass (bottleneck) befindet. Die Replizierung des Datenbanksystems wäre deshalb eine sinnvolle Erweiterung.

- Die Umsetzung hat außerdem gezeigt, dass es bei der Entwicklung und insbesondere beim Testing von großem Nutzen wäre, wenn es eine eigene Testumgebung für die Plattform gäbe, sofern man nicht für jede Art der Datensammlung das Haus verlassen möchte. Kapitel 3.2 zeigt bereits die Umsetzung einer TestApp für die Beschaffung exemplarischer Daten, welche noch erweitert werden könnte, um ein weiteres Spektrum abzudecken. Auch die Erschaffung einer Art Entwicklungsumgebung wäre denkbar, mit welcher die späteren Spieledesigner auf komfortable Art und Weise Regeln definieren und verändern könnten.

4 Ausblick

Eine Limitierung bei der Erzeugung neuer Spiele existiert neben den Einschränkungen der Plattform durch die aktuelle Hardware von mobilen Geräten. So ist es momentan nur schwer möglich, genaue Positionsdaten zu erhalten, wenn man sich beispielsweise in Gebäuden befindet. Auch das mobile Internet ist in der heutigen Zeit noch nicht als zuverlässig einzustufen, weshalb temporäre Verbindungsabbrüche mit einzukalkulieren sind. Diese Einschränkungen behindern die Entwicklung von Pervasive Games maßgeblich. Da sich beide Technologien erst seit wenigen Jahren für die Allgemeinheit durchgesetzt haben, sind die dadurch resultierenden Probleme nachvollziehbar. Anhand des mobilen Internets lässt sich bereits bei Betrachtung ein deutlicher Fortschritt in letzten Jahren erkennen. So haben zum Beispiel viele Provider den Wirkungsbereich ihrer Netze erweitert, so dass diese zumindest deutschlandweit in allen urbanen Gegenden verfügbar sind. Die Geschwindigkeit des Netzes hat sich außerdem durch LTE (Long Term Evolution) deutlich verbessert. Geht man also von der vergangenen Entwicklung aus, so kann eine Beseitigung der momentanen hardwarebedingten Einschränkungen in naher Zukunft durchaus als realistisch bezeichnet werden. Idealerweise muss sich der zukünftige Entwickler von Pervasive Games keine Gedanken über die Verfügbarkeit von Lokationsdaten und mobilem Netzwerk machen. Auch Latenzprobleme sollten der Vergangenheit angehören. Der Markt für diese Art von Spielen vergrößert sich von Jahr zu Jahr, da für die Digital Natives das Smartphone immer zum Alltagsgegenstand wird. Bereits im vergangenen Jahr besaß fast jeder zweite Jugendliche ein Smartphone [Informationszentrum Mobilfunk e.V. (IZMF), 2012].

Die hier vorgestellte Plattform könnte besonders für Betriebe im Einzelhandel von Interesse sein, da so Kunden mithilfe von Pervasive Games spielerisch geworben werden können. Auch die Serviceleistung der besagten Unternehmen kann durch diese Weise verbessert werden. So können die Kunden beispielsweise durch das Spiel direkt vor das Geschäft geführt werden, um dort eine Aufgabe zu erfüllen. Ebenso wären Pervasive Games in der Tourismusbranche denkbar, bei denen die Benutzer spielerisch die urbane Gegend mit ihren Points of Interest kennenlernen.

Ob das steigende Interesse an Pervasive Games auch in naher Zukunft bestand haben wird, muss sich erst noch zeigen. Durch die Entwicklung neuer Technologien können ebenso neue Interessensfelder gebildet werden, welche die Möglichkeiten dieser Art von Spielen erweitern oder sogar ihre Entwicklung in völlig neue Bahnen lenken könnten. Abschließend bleibt zu sagen, dass diesem interessanten Themenbereich jede Menge Potenzial offensteht, welches darauf wartet, umgesetzt zu werden. Auch die Zukunft wird den Pervasive Games zahlreiche ungeahnte Möglichkeiten bieten, von denen wir heutzutage noch nicht die geringste Vorstellung haben.

Abbildungsverzeichnis

Literaturverzeichnis

[AndroidPit - Swetlana Soschnikow 2013] AndroidPit - Swetlana Soschnikow: *800.000 Apps: Play Store überholt Apple.* 2013. – URL http://www.androidpit.de/anzahl-der-apps-google-ueberholt-apple. – Zugriffsdatum: 2013-06-28

[Balzert 2009] Balzert, H.: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering.* Spektrum Akademischer Verlag GmbH, 2009. – ISBN 9783827422460

[Bauer und King 2005] Bauer, C. ; King, G.: *Hibernate in action.* Manning Publications Company, 2005 (In Action Series). – URL http://books.google.de/books?id=WCmSQgAACAAJ. – ISBN 9781932394153

[Bauer und King 2006] Bauer, C. ; King, G.: *Java Persistance With Hibernate.* Dreamtech Press, 2006. – URL http://books.google.de/books?id=bTwfHzMxtQcC. – ISBN 9788177227192

[BITKOM 2013] BITKOM: *Smartphones etablieren sich als Spieleplattform.* 2013. – URL http://www.bitkom.org/de/presse/8477_75517.aspx. – Zugriffsdatum: 2013-06-28

[Exit Games 2013a] Exit Games: *Photon Server Realtime Backend Architecture.* 2013. – URL http://www.exitgames.com/Photon/Architecture. – Zugriffsdatum: 2013-05-22

[Exit Games 2013b] Exit Games: *Realtime. Multiplayer. Cross Platform.* 2013. – URL http://www.exitgames.com/Photon/. – Zugriffsdatum: 2013-05-22

[Financial Times Deutschland 2012] Financial Times Deutschland: *Google-Betriebssystem: Android erreicht Marktanteil von drei Vierteln.* 2012. – URL http://www.ftd.de/it-medien/it-telekommunikation/:google-betriebssystem-android-erreicht-marktanteil-von-drei-vierteln/70113730.html. – Zugriffsdatum: 2013-06-12

[Goldflam 2012] Goldflam, Andre: *Eine domänenspezifische Sprache zur Implementierung komplexer Abläufe in Gesellschaftsspielen*, Hochschule für Angewandte Wissenschaften Hamburg, Masterarbeit, 2012. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/goldflam.pdf

[Groundspeak, Inc. 2013] Groundspeak, Inc.: *Geocaching*. 2013. – URL http://www.geocaching.com/. – Zugriffsdatum: 2013-07-14

[HealthOn 2013] HealthOn: *30 Millionen Smartphone-Nutzer in Deutschland*. 2013. – URL http://www.healthon.de/2013/02/06/30-millionen-smartphone-nutzer-in-deutschland/. – Zugriffsdatum: 2013-07-03

[Informationszentrum Mobilfunk e.V. (IZMF) 2012] Informationszentrum Mobilfunk e.V. (IZMF): *JIM-Studie 2012: Fast jeder zweite Jugendliche besitzt ein Smartphone*. 2012. – URL http://www.izmf.de/de/content/jim-studie-2012-fast-jeder-zweite-jugendliche-besitzt-ein-smartphone. – Zugriffsdatum: 2013-07-20

[ITWissen 2013] ITWissen: *Message oriented middleware*. 2013. – URL http://www.itwissen.info/definition/lexikon/message-oriented-middleware-MOM.html. – Zugriffsdatum: 2013-05-24

[Jonsson und Waern 2008] Jonsson, Staffan ; Waern, Annika: The art of game-mastering pervasive games. In: *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*. New York, NY, USA : ACM, 2008 (ACE '08), S. 224–231. – URL http://doi.acm.org/10.1145/1501750.1501803. – ISBN 978-1-60558-393-8

[Kilburn 2013] Kilburn, Colin: *Setting up the Key and Trust Stores*. 2013. – URL http://activemq.apache.org/how-do-i-use-ssl.html. – Zugriffsdatum: 2013-06-27

[LDAP 2013] LDAP: *Lightweight Directory Access Protocol*. 2013. – URL http://de.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#St.C3.A4rken_und_Anwendungsschwerpunkte_von_LDAP. – Zugriffsdatum: 2013-06-12

[Neward 2013] Neward, Ted: *The Vietnam of Computer Science*. 2013. – URL http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx. – Zugriffsdatum: 2013-06-03

[Oracle 2013a] Oracle: *JavaTM Secure Socket Extension (JSSE) Reference Guide*. 2013. – URL http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html. – Zugriffsdatum: 2013-06-27

[Oracle 2013b] Oracle: *JavaTM Secure Socket Extension (JSSE) Reference Guide*. 2013. – URL http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html. – Zugriffsdatum: 2013-06-27

[Richardson 2008] Richardson, Chris: ORM in Dynamic Languages. In: *Queue* 6 (2008), Mai, Nr. 3, S. 28–37. – URL http://doi.acm.org/10.1145/1394127.1394140. – ISSN 1542-7730

[Snyder u. a. 2011] Snyder, B. ; Bosnanac, D. ; Davies, R.: *ActiveMQ in Action*. Manning Publications Company, 2011 (In Action Series). – URL http://books.google.de/books?id=_jjCPwAACAAJ. – ISBN 9781933988948

[Tanenbaum und van Steen 2007] Tanenbaum, Andrew S. ; Steen, Maarten van: *Verteilte Systeme*. 2., Aufl. PEARSON STUDIUM, 2007. – ISBN 3827372933

[The ActiveMQ team 2013a] The ActiveMQ team: *JMX*. 2013. – URL http://activemq.apache.org/jmx.html. – Zugriffsdatum: 2013-06-23

[The ActiveMQ team 2013b] The ActiveMQ team: *The SSL Transport*. 2013. – URL http://activemq.apache.org/ssl-transport-reference.html. – Zugriffsdatum: 2013-06-26

[The ActiveMQ team 2013c] The ActiveMQ team: *Wildcards*. 2013. – URL http://activemq.apache.org/wildcards.html. – Zugriffsdatum: 2013-06-22

[The Foursquare team 2013] The Foursquare team: *Foursquare*. 2013. – URL https://foursquare.com/. – Zugriffsdatum: 2013-07-03

[The Hibernate Team 2013] The Hibernate Team: *Hibernate FAQ - Product Evaluation FAQ*. 2013. – URL https://community.jboss.org/wiki/HibernateFAQ-ProductEvaluationFAQ. – Zugriffsdatum: 2013-06-04

[The JBoss Drools team 2013a] The JBoss Drools team: *Drools - JBoss Community*. 2013. – URL https://www.jboss.org/drools/. – Zugriffsdatum: 2013-07-03

[The JBoss Drools team 2013b] The JBoss Drools team: *Drools Expert User Guide*. 2013. – URL http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/html_single/index.html. – Zugriffsdatum: 2013-06-20

[The JSON Group 2013] The JSON Group: *Einführung in JSON*. 2013. – URL http://json.org/json-de.html. – Zugriffsdatum: 2013-06-29

[The MySQL team 2013] The MySQL team: *Marktanteil*. 2013. – URL http://www.mysql.de/why-mysql/marketshare/. – Zugriffsdatum: 2013-07-05

[Weisstein 2013] Weisstein, Eric W.: *Spherical Trigonometry*. MathWorld. 2013. – URL http://mathworld.wolfram.com/SphericalTrigonometry.html. – Zugriffsdatum: 2013-06-22

[Wunderlich 2006] Wunderlich, L.: *Java Rules Engines: Entwicklung von regelbasierten Systemen*. Entwickler.Press, 2006. – URL http://books.google.de/books?id=oxOvAAAACAAJ. – ISBN 9783935042758

Abgrenzung

Im Folgenden ist definiert, welcher Autor für welches Kapitel maßgeblich verantwortlich war.

Philip Rose verfasste Kapitel 1, 4 und die Abschnitte 2.1, 2.2, 2.6, 2.7, 2.8, 3.1, 3.2, 3.3, 3.5.1, 3.5.2, 3.6, 3.7.

Vassilios Stavrou verfasste Kapitel 4 und die Abschnitte 2.3, 2.4, 2.5, 3.1, 3.4, 3.5.3, 3.5.4, 3.7.

Versicherung über Selbstständigkeit

Hiermit versichern wir, dass wir die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt haben.

Hamburg, 29. August 2013 Philip Rose

Hamburg, 29. August 2013 Vassilios Stavrou