

RouteMe - Dokumentation des Refactoring

Julian Dehne Hendrik Geßner

22. Oktober 2013

Student:

Julian Dehne, Matrikelnummer: 752711

B.A. European Studies

Fachsemester: 06

Hochschulsemester: 08

Adresse:

Stahnsdorfer Straße 156b

14482 Potsdam

Julian.Dehne@googlemail.com

Student:

Hendrik Geßner, Matrikelnummer: 751352

Fachsemester: 07

Hochschulsemester: 07

Adresse:

Stahnsdorfer Straße 148a

14482 Potsdam

HENGEO1@googlemail.com

Institution:

Universität Potsdam

Institut für Informatik

Inhaltsverzeichnis

Inhaltsverzeichnis	2
1 Einleitung	3
2 Prozessbeschreibung	3
2.1 Ausgangslage	3
2.2 Refaktorisierung des PHP Quellcodes	4
2.3 Entscheidung gegen PHP und für Java	6
2.4 Entscheidung für das Hibernate Datenbank Framework	7
2.5 Entscheidung für das Google Web Toolkit	8
2.6 Entscheidung für eine zusätzliche REST Schnittstelle	10
2.7 Entscheidung für eine Refaktorisierung der Persistenzschicht und der Schnittstellenschicht	12
2.8 Lokalisierung und GPS als programmatische Herausforderung	13
2.9 Entscheidung für manuelle Suche nach einer geeigneten Örtlichkeit	14
2.10 Zusammenfassung des Prozesses und der Phasenübergänge	16
3 Technische Beschreibung der Software	17
3.1 Spielinhalt aus technischer Sicht	17
3.2 Tests	17
3.3 Architektur der RouteMe Implementierung	17
3.3.1 Datenbank	18
3.3.2 Dependency Injection	22
3.3.3 Timer	23
3.3.4 Klient	23
3.3.5 Schnittstellen	32
3.4 Architektur der Tools	37
3.4.1 GPS-Scouting	38
3.4.2 Dummy-Player	43
3.4.3 Performance Parser	45
3.4.4 Settings Manipulator	45
3.5 Installationsanleitung	45
3.5.1 Entwicklungsumgebung	45
3.6 Deployment	46
4 Notwendige Verbesserungen und Erweiterungen	47
5 Ausblick	47
6 Literaturverzeichnis	49

7 Abbildungsverzeichnis	50
--------------------------------	-----------

Zusammenfassung

Pervasives und spielbasiertes Lernen gehören zu einer neuen Gattung innovativer Lernmethoden. In diesem Kontext wurde an der Universität ein innovatives Spiel RouteMe entwickelt. Die Implementation auf PHP Basis offenbarte bei dem Versuch Fehler zu beheben und Struktur in den Quellcode zu bringen strukturelle wie auch konzeptionelle Schwächen. Daher haben wir uns entschieden, das Spiel auf Java Basis weitestgehend neu zu schreiben. Die vorgenommenen Änderungen bieten nachfolgenden Arbeiten die Möglichkeit auf einer stabilen und stärker gekapselten Architektur aufzubauen.

Keywords: *Pervasives Lernen, Spielbasiertes Lernen, Evaluation, Refaktorisierung*

1 Einleitung

Da nachfolgende Dokument enthält eine technische Beschreibung des Prozesses der Refaktorisierung des Spiels RouteMe.

Diese Arbeit ist ein Teil eines größeren Forschungsprojektes des Lehrstuhls für komplexe multimediale Anwendungsarchitekturen, welches sich in folgenden Teilschritten bewegt:

1. Das Spielkonzept basiert auf einer Diplomarbeit und lässt sich in einschlägigen Veröffentlichungen nachlesen. [MLZ11]
2. Eine lerntheoretisch fundierte Evaluation des Spielkonzeptes wurde in einer Bachelorarbeit erstellt. [Jul12]
3. Das Refaktorisierungsprojekt ist Gegenstand dieser Arbeit
4. Entwicklung weiterer Features stehen in Aussicht (Bachelorarbeit von Hendrik Geßner)

2 Prozessbeschreibung

2.1 Ausgangslage

Im Herbst 2011 hat Julian Dehne die erste Implementierung des Spiels RouteMe von Tobias Moebert übernommen, um sie auszurollen und weiterzuentwickeln. Dies konnte aus folgenden Gründen nicht durchgeführt werden:

Das erste Hindernis war, dass die neueren PHP Versionen (5.4+) bezüglich der JSON Klasse nicht rückwärtskompatibel sind (*breaking changes*). Der Institusadmin war aus Sicherheitsgründen nicht bereit, eine ältere PHP Version einschließlich ihrer Sicherheitsprobleme zu installieren. Auf dem (einzigsten) verfügbaren Produktionsserver liefen zu diesem Zeitpunkt noch andere Projekte, die durch eine Umkonfiguration der PHP-Engine gefährdet gewesen wären.

Das zweite Hindernis bestand darin, dass Julian Dehne drei unterschiedliche Versionen des Quellcodes zu Verfügung standen, die sich zum Teil stark voneinander unterschieden. Eine Version wurde als Diplomarbeit eingereicht. Eine weitere Version mit Optimierungen wurde von Tobias Moebert zugeschickt und eine dritte Version stammt von einem Deployment auf einem externen Server von Raphael Zender.

Ein drittes Hindernis bestand darin, dass sich die MySQL-Version seit der ersten Implementierung weiterentwickelt hatte (5.1 auf 5.3). Auch hier gab es institutionelle Probleme, eine ältere MySQL Version zu installieren

Nachdem die oben stehenden Probleme umgangen oder behoben wurden, stellte sich heraus, dass die Software noch Programmierfehler enthielt. Dies wurde von Tobias Moebert im Emailverkehr bestätigt.

Da in jedem Fall die Programmierfehler behoben werden mussten und ein Upgrade der Implementation auf aktuelle Versionen der Programmierumgebung notwendig wurde, kam es zur Entscheidung, die Software zu refaktorieren.

2.2 Refaktorisierung des PHP Quellcodes

Im folgenden beschreiben wir unsere Vorgehensweise und unsere Lösungsstrategien, um mit auftretenden Problemen umzugehen.

In dem Quellcode haben wir eine Reihe von konzeptionellen Schwächen entdeckt: Die Objektorientierung, die in PHP5 bereits angelegt ist, wurde nur sehr begrenzt verwendet. Alleine die Datenbanktabellen werden auf Klassen gemappt, wobei es hier zu einer Überladung mit Funktionen einzelner Klassen kommt. Zum Beispiel ist ein Großteil des Algorithmus in einer einzigen Klasse implementiert, die gleichzeitig auch sämtliche Felder eines Datenobjektes trägt.

Eine Trennung von Logik, Anzeige und Persistenz wurde nicht durchgeführt. In Abbildung 1 wird durch die farbliche Markierung deutlich, wie zusätzlich zu dem PHP und HTML Code, Javascript in das Dokument integriert wird. Des Weiteren wird sogar in der dritten Zeile des ersten Javascript Blocks Javascript von PHP aus dynamisch erstellt. Dies belegt die fehlende Trennung von serverseitiger und klientenseitiger Implementierung. Konsequenter wäre es gewesen, die fehlenden Daten auch hier per Webservice oder mit einem blockierenden AJAX Aufruf aufzurufen.

Unserer Meinung nach ist es bei PHP im Gegensatz zu Java oder C# üblicher, Teillogik in den HTML Quellcode zu schreiben, allerdings setzten sich auch hier Template-Frameworks wie Smarty [New13] durch. Außerdem ist es üblicher geworden, den Javascript Code zu injizieren, anstatt ihn *inline* zu schreiben. Der Javascript-Code hat meist die Funktion Events abzufangen, wohingegen der PHP-Code die Logik oder das ViewModell übernimmt. Durch die Verwendung von AJAX, welches hier komplette HTML Seiten lädt, die mit PHP serverseitig erstellt wurden, werden diese Prinzipien in einer Weise verletzt, dass es unmöglich machte, etwas an der Darstellung zu ändern, ohne auch die Logik-Komponenten anzufassen.

Ein weiteres Indiz für die Validität dieser Programmierprinzipien lässt sich an den Wiederholungen von identischem Code festmachen, der entweder durch *copy-paste* oder andere problematische Techniken auftreten.

Das Konzept von Namespaces, welches in PHP5 eingeführt wurde, wurde nicht benutzt, was dazu führte, dass einzelne Codeteile nicht sauber gekapselt sind. Eine Variable, die an einer beliebigen Stelle im Code deklariert wird, kann über PHP-*includes* an einer ganz anderen Stelle verwendet werden. Dies ist nicht nur problematisch, weil es zu schwer erklärbaren Überschattungen zur Laufzeit kommen kann, sondern auch, weil dadurch die Wartbarkeit leidet.

```

<?php if (!empty($gameSettings)) { ?>
    <?php $centerLatLong = $gameSettings->getPlayingFieldCenterLatLong(); ?>
    <script type="text/javascript" charset="utf-8">
        $(function() {
            getGameStats();
            initializeAdminMap(<?php echo $centerLatLong["latitude"] ?>, <?php echo $centerLatLong["longitude"] ?>);
        });
    </script>
    <div id="gameStats"></div>
    <br/>
    <?php if ($gameSettings->isRunning): ?>
        <script type="text/javascript" charset="utf-8">
            $(function() {
                getPlayerStats();
                updateRemainingPlayingTime();
                placeNewItems();
                updateBonusGoals();
                updateNodeBatteries();
            });

            <?php
                if ($gameSettings->protocol == "aodv") {
                    echo "aodvProcessRoutingMessages();";
                    echo "aodvProcessDataPackets();";
                }
            ?>
        );
    </script>
    <button class="ui-state-highlight" type="button" onclick="stopGame()">Spiel unterbrechen</button>
<?php else: ?>

```

Abbildung 1: Durchmischung der Programmierebenen

Gleiches gilt für Funktionen und IDE Unterstützung: Der PHP-Code wurde mit Netbeans [Ora13] entwickelt, welches PHP-Entwicklern ermöglicht, die schwache Typisierung der Sprache durch PHP-Doc [Wen00] auszugleichen, wodurch es möglich wird, die Klasse des Rückgabewerts von Funktionen zu kennen, ohne die Funktion semantisch lesen zu müssen. Des Weiteren besteht dadurch die Möglichkeit, die Aufruf-Hierarchie von Funktionen im Code nachzuvollziehen.

Ein rein wahrnehmungstechnisches Problem ergibt sich aus dem folgenden Umstand: Wenn in einer Datei PHPCODE, HTML und Javascript durchmischt werden, leidet darunter die Lesbarkeit.

Die rote Linie zeigt, wo Netbeans dem Entwickler die Seitengrenze bzw. einen Zeilenumbruch vorschlägt, damit der Quellcode auf einer A4-Seite dargestellt werden kann, ohne extrem weit nach rechts scrollen zu müssen.

Die Unterstützung durch die IDE ist in PHP schlechter entwickelt als in Java. Es gibt unter anderem keine Möglichkeit, einzelne Programmteile halbautomatisch zu Methoden oder Klassen zusammenzufügen.

Hinzu kommt, dass es zu dem PHP-Code keinerlei technische Dokumentation gab außer wenigen unstrukturierten Kommentaren im Quellcode.

Die Refaktorisierung des PHP-Codes gestaltete sich aus den oben benannten Gründen als sehr problematisch und für uns annähernd unmöglich. An manchen Stellen konnten wir Fehler in dem Quellcode erkennen (nicht benutzte Variablen, fehlende Zweige, Fehler etc.), aber der Gesamtzusammenhang war kaum zu verstehen, da zu viele mögliche Referenzen existierten. Für jemanden, der den Code nicht selber geschrieben hat, ist es so im Nachhinein beinahe unmöglich kleinteilige Änderungen vorzunehmen. Nach einiger Zeit entstand so eine Mischung aus ursprünglichem fehlerbehafteter und neu dazu gekommener fehleranfälliger Software, die nur pessimistische Aussichten bot.

2.3 Entscheidung gegen PHP und für Java

Wie in dem vorhergegangenen Kapitel beschrieben weist der ursprüngliche Quellcode nicht bewältigbare Mängel auf, so dass wir uns für eine weitestgehende Neuimplementierung entschieden haben.

Die Entscheidung für Java lässt sich mit den folgenden Argumenten rechtfertigen:

1. Freie IDE und mächtige Open Source Community
2. Verbreitung im akademischen Bereich
3. Erfahrungen der Programmierer
4. Bekannte und getestete Frameworks
5. Betriebssystemunabhängigkeit
6. Die Integration von Zeit in kompilierenden Sprachen

Die Probleme mit dem Ausrollen zeigen einen der wichtigsten Vorteile von Java gegenüber PHP, bei dem der Interpreter massiv von dem Betriebssystem abhängig ist. Ein Beweis dafür sind die Pfadangaben: Da in PHP das Namespace-Konzept nicht durchgängig integriert ist, entstehen die meisten Verweise durch relative Pfadangaben. Diese Pfadangaben sind jedoch von einem Betriebssystem zum nächsten unterschiedlich (Linux achtet auf Großschreibung im Gegensatz zu Windows).

Ein weiteres Argument für Java gegenüber PHP ist die Programmierung von zeitabhängigen Ereignissen. Um diesen Punkt zu erklären müssen wir einen größeren Bogen spannen:

Das Spiel RouteMe funktioniert so, dass ein Routingprotokoll simuliert wird. Diese Simulation selber ist jedoch als Konzept nur wage definiert worden, was wir hier nachholen: Um eine computergesteuerte Simulation des AODV-Protokolls zu implementieren, gibt es nach unseren Überlegungen nur drei logische Varianten. Die erste wäre die, dass das Protokoll in Realzeit abläuft. Dies hätte aber das didaktische Problem zur Folge, dass die Spieler das Ergebnis ihrer Entscheidungen nicht angemessen würdigen können, da ihnen der Prozess, den sie ausgelöst haben, in seinen einzelnen Schritten verborgen bleibt. Ein entscheidendes Spielkonzept ist jenes, dass die Spieler bei einer überlasteten Route nach einiger Zeit die Entscheidung treffen können, eine neue zu wählen. Wäre die Route nur für einige Millisekunden überlastet, könnten die Lernenden diesen Vorgang nicht erfassen. Die zweite Möglichkeit ist die, eine Verzögerung einzubauen, aber dennoch den Algorithmus trotzdem dezentral ablaufen zu lassen. Dies hat zwei Nachteile: Zum einen ist es schwieriger Übersichtstabellen zu erstellen, da es keine Instanz gibt, die einen Zustand zu einem definierten Zeitpunkt zeigt. Zum anderen verbaut dies den Weg, andere Algorithmen zu integrieren, die auf einem anderen Prinzip basieren. Die dritte Möglichkeit ist die einer vollständigen Simulation. Dies ist diejenige, die in der Originalversion verfolgt wurde.

Wenn ein Algorithmus zeitverzögert mit einer zentralen Instanz simuliert werden soll, dann muss es einen Taktgeber geben, der die Zeitverzögerung orchestriert. Es ist nicht ohne Weiteres möglich, die Verzögerung dezentral einzubauen, ohne auch die Logik dezentral zu lassen, da es hier eine wechselseitige Beeinflussung gibt.

Der Taktgeber wurde in der PHP-Version dadurch realisiert, dass der Browser, in dem das Admin-interface für das Spiel läuft, in regelmäßigen Abständen ein Step-Signal an den Server sendet. Bei den Testläufen stellte sich diese Architektur als äußerst instabil heraus, da das Spiel sofort in einen undefinierten Zustand gerät, sobald dieser Browser geschlossen wird.

Das Problem bei PHP besteht darin, dass die Sprache interpretiert wird, wenn eine HTTP-Anforderung den unterliegenden Server erreicht. PHP ist nicht darauf ausgelegt eigenständige Dienste zu fahren, die zeitgesteuerte Events auslösen. Hier ist Java wesentlich besser geeignet, da in Java ein logischer Thread aufgemacht werden kann, der von dem Betriebssystem regelmäßig beachtet wird und dann aus sich heraus agieren kann. Durch die Entscheidung für Java konnten wir neben unseren anderen bereits geschilderten Bedenken auch ein klares architektonisches Argument für den Umstieg finden.

2.4 Entscheidung für das Hibernate Datenbank Framework

Die Entscheidung für das Hibernate Framework [Red11] beruht auf der Erkenntnis, die in dem PHP-Projekt bereits erfolgt ist, dass Objektorientierung nur dann konsequent umgesetzt ist, wenn die Datenbankschicht auch aus Objekten, sogenannten *Data Transfer Objects (DTO)*, besteht. In dem PHP-Projekt wurde Doctrine [WBE⁺13] verwendet. Auf Java Ebene ist eines der wichtigsten OR-Frameworks Hibernate von JBoss. Im folgenden nennen wir einige Argumente, die die Entscheidung für Hibernate beeinflusst haben:

1. Abstraktion: Es können sehr viele unterschiedliche Datenbanken angebunden werden
2. Automatische Generierung von Klassen
3. Programmatischer Auf- und Abbau der Datenbank
4. Integrierter Cache
5. Lazy Loading
6. Annotationen anstatt XML/YML
7. Kenntnisse im Team
8. Query API
9. Named Queries
10. Kenntnisse im Team (von dem FreshUP Projekt)

Im Laufe der Entwicklung hat sich herausgestellt, dass gerade der Algorithmus sehr viele Datenbank-hits verursacht. Der Algorithmus wird durch Hibernate in mehrfacher Weise beschleunigt: Es werden nur die benötigten Daten *lazy* nachgeladen, häufige Anfragen werden im Cache vorrätig gehalten.

Trotz den großen Herausforderungen, die die Einbindung eines komplexen Frameworkes mit sich bringt, hat Hibernate bezüglich der Performanz und Wartbarkeit enorme Vorteile gebracht. Dadurch, dass im Team bereits Erfahrungen mit dem Framework vorherrschen, entsteht mit jedem Projekt eine größerer Produktionsgewinn.

Im Gegensatz zu SQL hilft die Objektorientierung, die Persistenzschicht zu strukturieren und allgemein im Projekt verwendbare Datenobjekte zu schaffen.

2.5 Entscheidung für das Google Web Toolkit

Nach der Entscheidung gegen PHP auf Serverseite stellte sich die Frage, ob wir die Klientsicht auch verändern wollten. Tatsächlich ginge es technisch, eine auf REST basierende Schnittstelle zu schaffen und die dynamischen Elemente der Websites mit PHP zu scripten.

Aus pragmatischen Gründen wurde dieser Ansatz verworfen. Zum einen müsste man so einen Java-Container und einen HTTP-Server gleichzeitig konfigurieren und bedienen, zum anderen fielen dann die Vorteile von Java (Betriebssystemunabhängigkeit etc.) weg.

Ein weiteres Problem stellte die Tatsache da, dass der klientseitige Quellcode nicht sauber von der Logik entkoppelt wurde, so dass es nicht unerheblichen Aufwand bereiten würde, den PHP-Code bei verändertem Backend beizubehalten. Insbesondere die Technik, die HTML-Seiten komplett per AJAX zu verschicken anstatt den veränderten Zustand, läge hier quer.

Daher haben wir uns dafür entschieden auch auf Klientseite den PHP-Code zu ersetzen. Die Herausforderung war daraufhin die, dass die GUI in ihrem Layout und prinzipiellen Design beibehalten werden sollte. Daher mussten wir eine Technologie verwenden, die es erlaubt, das HTML, CSS und Javascript zu ummanteln, ohne dass semantische Veränderungen entstehen.

Wir haben uns hier für das Google Web Toolkit (GWT) (<https://developers.google.com/web-toolkit/>) entschieden. Hierfür gibt es folgende Argumente:

1. Es wird auf Klient und auf Server Seite in Java programmiert
2. Die Kommunikation (XML-RPC) zwischen Klient und Server wird durch den Kompiler geprüft
3. Der Debugger überspringt die Grenze zwischen Klient und Server
4. Reifes und weit verbreitetes Framework
5. Kompiliert nach Javascript und kann sowohl natives Javascript aufrufen als auch umgekehrt
6. Erfahrungen im Team
7. Entkopplung

Durch die Verwendung von GWT konnten wir einen großen Teil von der ursprünglichen Implementierung der GUI beibehalten und konsequent neu strukturieren. Hierbei haben wir das Konzept verfolgt, dass einzelne GUI-Komponenten, die logisch eigenständig sind, auch als eigenständige Dateien und Klassen existieren, so dass sie genau wie andere Komponenten dynamisch komponiert werden können. Dazu haben wir das HTML nach Funktionen untergliedert und zu jeder Funktion einen HTML-Binder (GWT-kontrolliertes HTML) erstellt. Da GWT nach Javascript kompiliert, konnten wir die meisten Javascript Funktionen übernehmen. So konnten wir Schrittweise den Clientcode transferieren. In einigen wenigen Fällen mussten wir von GWT aus Javascript-Funktionen mittels des *Javascript Native Interface* (JSNI) aufrufen, wie in Abbildung 2 zu sehen ist.

```

package de.unipotsdam.nexplorer.client.admin.viewcontroller;

import de.unipotsdam.nexplorer.shared.GameStats;

public class JSNI {

    /**
     * Diese Methode stellt sicher, dass auch bei einem Reload der Website
     * das HTML des Spiel-Pausieren-Button
     * den Zustand des System widerspiegelt
     * @param gamePaused
     */
    public static native void setEnded()/*-{  

        $wnd.setEnded();  

}-*/;

    public static native void setNotEnded()/*-{  

        $wnd.setNotEnded();  

}-*/;

    public static void updateGameState(GameStats result) {  

        switch (result.getGameStatus()) {  

            case NOTSTARTED:  

                JSNI.setNotEnded();  

                break;  

            case HASENDED:  

                JSNI.setEnded();  

                break;  

        }  

    }  

}

```

Abbildung 2: Beispiel für Aufruf von Javascript Methoden aus GWT

GWT hat einen weiteren Vorteil gebracht: AJAX-Aufrufe, die aufgrund der nun fehlenden PHP-Komponente keine Daten bekommen haben, wurden von GWT als HTTP-Fehler in der IDE gelistet. So konnten wir für die bestehenden Javascript Schnittstellen zum Backend systematisch ein neues Interface aufbauen, so dass die Funktionalität des Javascript-Codes bewahrt und gleichzeitig getestet wurde. Mit den Motiven für den Aufbau einer zusätzlichen REST-Schnittstelle beschäftigt sich das nächste Kapitel.

2.6 Entscheidung für eine zusätzliche REST Schnittstelle

Wie in dem letzten Kapitel beschrieben, haben wir uns dafür entschieden den Javascript-Code beizubehalten. Hier stellte sich jedoch die Frage, woher die Daten kommen sollen, die von Javascript mittels AJAX geladen und gesendet werden.

Hier gab es zwei Möglichkeiten: Die erste wäre die direkte Nutzung von JSNI als Übergang von Javascript in Richtung Java. Allerdings ist der Weg von JSNI zu Java schwieriger, da Javascript im Gegensatz zu Java nicht stark typisiert ist. Das hieße, dass bei den Aufrufen die Java Namespaces kodiert werden müssten. Die zweite Möglichkeit besteht darin, neben der GWT eigenen XML-RPC Schnittstelle eine weitere REST-basierte Schnittstelle zu entwickeln. Die so entstandene Struktur der Schnittstellen sieht man in der folgenden Abbildung 3.

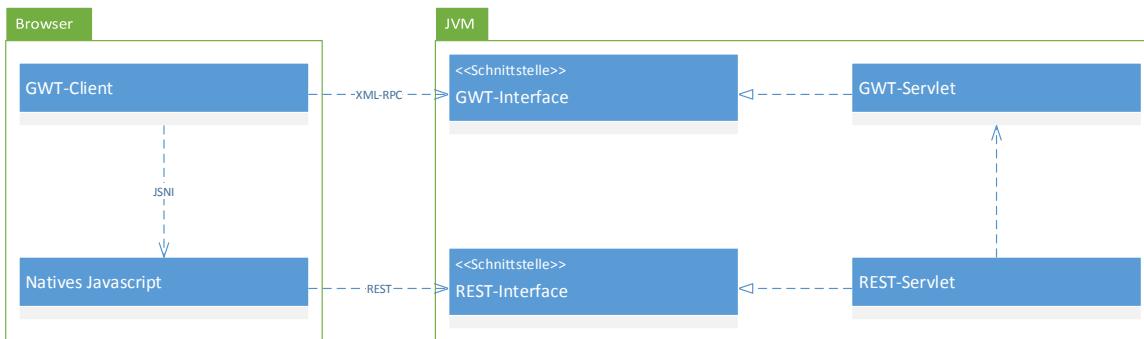


Abbildung 3: REST und XML-RPC Schnittstellen

Diese hat drei Vorteile: Zunächst arbeitet REST auf JSON, was Tobias Moebert korrekterweise als kompakteres Format bezeichnet und motiviert hat. Gerade die häufigen Positionsupdates lassen sich so schneller kommunizieren (wobei wir hier nie Zeitprobleme erkennen konnten). Zusätzlich haben wir so die Möglichkeit den Javascript-Code für die Rest-Aufrufe zu übernehmen. Schließlich bietet dies dem Programmierer die Möglichkeit komfortabel je nach Aufgabe entweder Javascript oder Java zu verwenden. Der Übergang von Java zu Javascript geschieht über JSNI. Der Übergang von Javascript zu Java geschieht über eine REST-Schnittstelle, die auf Java Seite mit Jersey [Tea13b] umgesetzt ist. Jersey erlaubt es, Java-Klassen mit Annotationen automatisch zu JSON-serialisierten Objekten umzuwandeln und diese ebenfalls mit Annotationen an einen Pfad zu koppeln. Abbildung 4 entspricht dem Code für die Möglichkeit per `[serverurl]/game_events/get_players` eine Objekt zu bekommen, welches Spielern und Koordinaten aufeinander abbildet.

```

package de.unipotsdam.nexplorer.server.rest;

/**
 * implements game events, since many events in RouteMe are now timer based
 * the specific methods are now deprecated
 * @author Julian
 */
@Path("game_events")
public class GameEvents {

    private Admin admin;

    public GameEvents() {
        this.admin = new Admin();
    }

    @POST
    @Path("get_players")
    @Produces("application/json")
    public NodeMap getPlayers() {
        try {
            List<Players> nodes = admin.getPlayerStats().getNodes();
            return new NodeMap(nodes);
        } catch (PlayerNotFoundException e) {
            throw new GameHasNotStartedYetException("admin versuchte NodeMap zu laden");
        }
    }
}

```

Abbildung 4: Beispiel für REST Schnittstelle

Dieses Beispiel zeigt auch, dass im ursprünglichen Projekt das REST-Konzept nur unzureichend reflektiert wurde. Diese Abfrage sollte eigentlich mittels eines HTTP-GET anstatt POST durchgeführt werden. Die Schnittstellen sind nun gut dokumentiert und verständlich, aber bezüglich Standardeinhaltung noch ausbaufähig.

2.7 Entscheidung für eine Refaktorisierung der Persistenzschicht und der Schnittstellenschicht

Die interne Aufteilung im Team bestand darin, dass Julian Dehne den Klient nach GWT portiert und die Schittstellen auf Java Seite implementiert hat, während Hendrik Geßner die serverseitige Logik und die Datenbankschicht portiert hat. Diese Aufteilung führte zu disjunkten Klassenstrukturen, da es zunächst nicht erkennbar war, dass es einen logischen Zusammenhang zwischen der Persistenzschicht und der serverseitigen Schnittstellenschicht gibt.

Sowohl die Schnittstellen als auch die Datenbank können als eine Art Persistenz- oder zustandsverändernde Schicht angesehen werden. Wenn der Zustand verändert wird, dann wird dieser Zustand sowohl in der Datenbank als auch gegenüber den Klienten kommuniziert. Für beide Vorgänge, Zustandsveränderung und Persistenz, lassen sich die gleichen Datentransferobjekte verwenden. Dies ist nicht zu verwechseln mit der Vermischung von View und Logik in dem ursprünglichen Projekt. Denn es gibt eine klare Trennung der Funktionen. Nur das Objekt der Funktionen ist gleichartig und kann wiederverwendet werden.

Um dieses Konzept umzusetzen, mussten wir uns intensiver mit dem Datenbankframework, der GWT gesteuerten XML-RPC Schnittstelle und dem REST-Framework auseinandersetzen. Mittels Annotations konnten wir für die verschiedenen Schnittstellen eine Untergruppe der Klassen bereitstellen, indem wir die nicht verwendeten Properties ausgeblendet haben. Auch konnten wir so Properties bei der Serialisierung mittels Jersey nach JSON umbenennen, damit keine Veränderung des Javascript-Codes notwendig wird. Auch gegenüber Hibernate mussten wir per Annotationen Änderungen vornehmen, damit einige Methoden nicht zu unnötigen Datenbankeinträgen führen.

Diese Maßnahmen führten für den Programmierer zu einer erhöhten Komplexität, da er damit rechnen muss, dass die Klasse, an der er arbeitet, von drei schwergewichtigen Frameworks (unterschiedlich) interpretiert wird. Der Vorteil liegt darin, dass Redundanz bei den Klassen verringert wird. Der Code wird so verständlicher, da für neue Felder oder Assoziationen nur eine Klasse verändert werden muss. Mittels Annotationen kann danach feingesteuert werden, ob und durch welche Schnittstelle die hinzugekommenen Felder zu den Klienten kommuniziert werden.

2.8 Lokalisierung und GPS als programmatische Herausforderung

Nach der Refaktorisierung und bei funktionierenden Unit-Tests sahen wir uns unter realen Bedingungen einer Gemengelage von Störfaktoren gegenüber, die eine verwendbare Aktualisierung von GPS Daten unmöglich machte. Zu den Problembereichen gehörten Umweltbedingungen, Hardware und Software.

Um den Einfluss von Umweltbedingungen zu erforschen stellten wir den Kontakt zu zwei Geowissenschaftlern¹ her, die in ihren Messungen häufig auf GPS-Ortungen zurückgreifen. Gemeinsam konnten wir folgende Faktoren isolieren:

1. Hauswände reduzieren die Fläche, auf der die Satelliten gefunden werden. Trotzdem wird ein Signal gefunden, aber es dauert 5-10 Minuten bis das geschieht.
2. Bäume: Baumkronen und organische Prozesse wie Photosynthese behindern GPS- Signale.
3. Wolkendecke und Nieselregen blockieren Signale
4. Zu viel Sonne kann zu Lichtreflexionen und damit beeinträchtigen Signalen führen
5. Folgende grundsätzliche Zahlen gelten für die Genauigkeit: 4-5 Satelliten bei guter Hardware bis zu 3 Meter, sonst 9-12 Meter. Bessere Ergebnisse gibt es nur mit AGPS.

¹Rafael Schäffer, Doktorand an der Technischen Universität Darmstadt, Udo Kracke, Vermessungsgenieuer, und Solveig Pospiech, Doktorandin an der Universität Göttingen

Hardware als einflussnehmende Variable wurde von uns nicht systematisch erforscht. Es ist anzunehmen, dass es zwischen unterschiedlichen Endgeräten Varianz innerhalb der Erfassungsgenauigkeit und Frequenz gibt. Auch die Zeit, bis ein erstes Signal gefunden wurde, spielt eine Rolle.

Als einflussreichste Störgröße stellte sich die Umsetzung auf Softwareseite innerhalb des Browsers heraus. Lange Zeit gingen wir von der Annahme aus, dass es keinen Unterschied machen würde, ob der GPS Empfänger des Endgerätes durch den Browser oder durch eine native Anwendung angesprochen wird.

Es stellte sich heraus, dass es zwischen den Browsern extrem große Unterschiede gibt, die aber noch von den Unterschieden zwischen einer nativen gegenüber einer webbasierten Anwendung übertroffen werden.

Der Unterschied zwischen den Browsern lässt sich auf zwei Ebenen erklären. Zum einen gibt es Browser, die für mobile Endgeräte optimiert wurden, so dass aus Gründen der Energieökonomie unterschiedliches Verhalten erfolgt. Zum anderen wird Javascript von unterschiedlichen Browsern anders interpretiert. Dies ist insbesondere bei dem übernommenen Javascript der Fall (ein verwendetes Framework funktioniert nur auf Chrome und Safari), wohingegen GWT durchaus Javascript-Code erzeugt, der Browserweichen für alle gängigen Browser enthält.

Der Unterschied zwischen einer nativen Applikation und der Browser basierten Variante lässt sich dadurch charakterisieren, dass die Applikation zumindest auf Android Basis mehr Ressourcen anfordern kann und häufigere GPS Daten erhält.

Wir konnten dieses Problem umgehen, indem wir im Hintergrund eine Applikation laufen lassen, die kontinuierlich GPS Daten abfragt. Dadurch, dass diese Daten vom Betriebssystem vorrätig gehalten werden, kommt es zu dem Effekt, dass auch der Browser aktuellere und damit genauere Daten erhält.

2.9 Entscheidung für manuelle Suche nach einer geeigneten Örtlichkeit

Bei der Suche nach Fehlern, die für die schlechte Genauigkeit verantwortlich sind, gingen wir der Frage nach, ob die Lokalität das Spielfeldes einen Einfluss auf die Empfangsgenauigkeit haben könnte. Wie in dem vorhergegangenen Kapitel beschrieben, hatten wir uns diesbezüglich fachmännischen Rat eingeholt.

In der Tat konnten wir je nach Ort stark variierende Messgenauigkeiten verzeichnen. Eine entscheidende Erkenntnis war die, dass bei fehlendem GPS-Signal das Handy in einen Modus verfällt, bei dem es über geo-IP den ungefähren Standort anhand des WLAN Netzes ermittelt. Hierdurch entstehen Lokalisationsunschärfen von 50 Metern bis mehreren Kilometern. Aber auch bei existierendem GPS-Signal konnte der Standort nicht immer unter 10-20 Meter Genauigkeit bestimmt werden. Für ein Spiel, welches auf Interaktion der Spieler im Bereich weniger Meter konzipiert wurde, ist dies unzulänglich.

Bei naiver Betrachtung käme man zu dem Schluss, dass es nun ausreichen würde, einmal mit einer mobilen Applikation denkbare Standorte abzulaufen. Leider liegt hier eine methodische Fehlvorstellung zu Grunde. Denn eine Koordinate, die an einem Tag schlechte Werte liefert, kann an einem anderen Tag wesentlich bessere Werte erbringen. Dies liegt an ungünstiger Wetterbedingungen (Wolkenbildung etc.), ungünstiger Satellitenkonstellation und anderen Faktoren.

Eine Karte mit GPS-Genauigkeit, die Jahresmittelwerte liefern könnte, gibt es unseres Wissens nicht. Daher haben wir uns dafür entschieden, eine Applikation zu programmieren, die GPS-Werte sammeln und zu einer Karte zusammenfügen kann, die Auskunft über das beste Spielfeld bietet. Hierfür Diese Applikation wurde eine Android-Applikation publiziert, auf mehreren Handys von Teammitgliedern verteilt und systematisch ausgewertet.

Die so gesammelten Daten wurden in eine Datenbank geloggt, um von dort aus mittels eines C# Projektes zu einer Karte zusammengefügt zu werden. Da hier die Datenbank eine Programmiersprachenuabhängigkeit ermöglichte, wurde C# gewählt, da hier eine sehr komfortable Karten-API zur Verfügung steht.

Eine so entstandene Karte sieht man in der Abbildung 5:

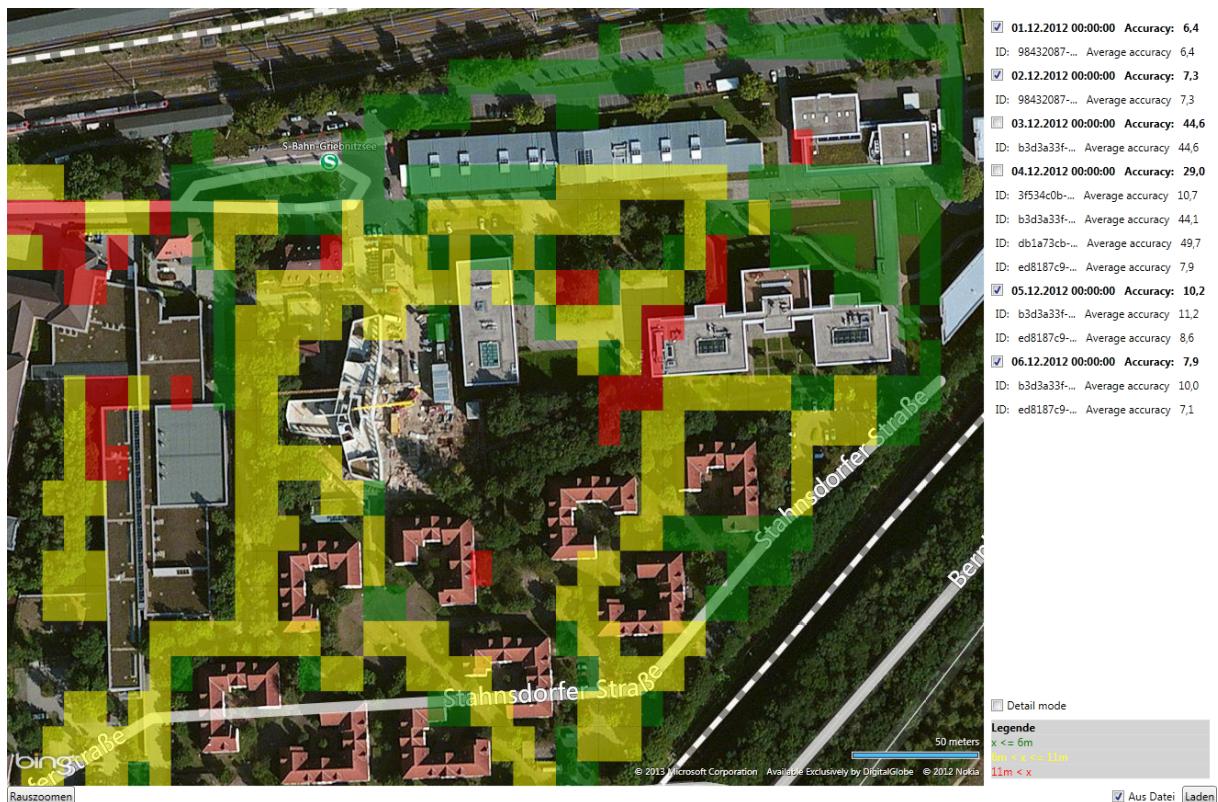


Abbildung 5: Beispielkarte aus manuell gesammelten Daten

Die Farben auf der Karte zeigen die Empfangsgenauigkeit wie in der Legende dokumentiert. Weitere Features des Tool:

1. Auswahl von Messpunkten mittels Checkboxes
2. Zoom
3. Wahl zwischen fest kodierter Datei oder URL als Datenquelle

2.10 Zusammenfassung des Prozesses und der Phasenübergänge

Unsere Arbeit lässt sich in mehrere Etappen einteilen.

Die erste Etappe beginnt bei der ersten Konfrontation mit dem PHP-Code und dem gescheiterten Versuch, diesen zum Laufen zu bringen. Dies dauerte c.a. drei Monate von Dezember 2011 bis Februar 2012.

Die zweite Etappe dauerte von Mai 2012 bis Anfang September 2012. In dieser Zeit haben wir prototypisch den Server neu strukturiert und die GUI mittels eines Template-Frameworks entkoppelt. Außerdem haben wir die Schnittstellen in REST-konformen Javascript-Code konvertiert. Bei ersten Testläufen blieben einige semantische Fragen ungeklärt, so dass unsere bereits bestehende Skepsis gegenüber der Architektur im Allgemeinen zu der bereits beschriebenen Entscheidung führte, das Projekt in Java zu übertragen und dort zu refaktorieren.

In der zweiten Septemberhälfte haben wir die Portierung nach Java durchgeführt.

Oktober 2012 korrigierten wir anfängliche Trennung der Datenbankschicht gegenüber der Schnittstellenschicht und beseitigten so auch Programmierfehler, die Redundanz g entstanden waren.

Bereits im Oktober offenbarten sich nach und nach die Probleme bei der webbasierten Lösung. Diese manifestierten sich im November und Dezember 2012, als wir die fertige Applikation zum ersten Mal unter realen Bedingungen getestet haben. Wir hatten dieses Projekt mit der Annahme begonnen, die Webbasiertheit sei bezüglich der GPS-Lokalisierung und des Javascripts gründlich durchdacht und gut konzeptualisiert. Gerade in diesem Punkt mussten wir leider aufgrund der realen Umstände einen schwierigen und aufreibenden Lernprozess durchmachen.

Seit Januar 2013 arbeitete maßgeblich Hendrik Geßner an dem Projekt, da Julian Dehne aus Zeitgründen das auf ein halbes Jahr angelegte Projekt nicht mehr gut in sein Studium integrieren konnte. Dies war auch sachlogisch, da die bestehenden Probleme sich meist auf die Timer (Race Conditions), CPU- oder Datenbank-Last bezogen.

Hendrik Geßner arbeitet seit Anfang Dezember bis Mitte Dezember an dem bereits erwähnten Karten-Tool, welches noch bis Ende Dezember optimiert wurde.

Im Januar 2013 wurden Bugfixes durchgeführt, die Fehler behoben, die durch die genaueren und schnelleren GPS Daten offenkundig wurden. Ende Januar wurde das Spiel mit bis zu 12 Spielern gespielt. Hier wurden bereits bei vier mobilen Knoten Performanzprobleme ersichtlich.

Im Februar 2013 wurde die Performanz durch eine In-Memory-Datenbank und viele kleine Veränderungen drastisch verbessert. In dieser Zeit entstand auch die Künstliche Intelligenz für mobile Klienten

Im März 2013 wurde eine abgabefertiges Packet der Software erstellt und die Dokumentation finalisiert.

3 Technische Beschreibung der Software

3.1 Spielinhalt aus technischer Sicht

Das Spiel ist auf das Lernen und Erleben von Routing-Algorithmen ausgelegt. Derzeit wird eine vereinfachte Version des AODV-Algorithmus [PBRD03] angeboten, in der Knoten nur Batterien und Booster einsammeln können. Die Implementation der AODV-Simulation stammt aus der Vorgängerarbeit [MLZ11]. Sie weicht an einigen Stellen vom RFC ab.

3.2 Tests

Das System wird mit verschiedenen Tests ausgeliefert. Die Tests befindet sich im Source-Ordner `test`. Da einige Tests auf der von Hibernate generierten Datenbank arbeiten muss jede Testklasse einzeln für sich gestartet werden. Es funktioniert nicht, alle Testklassen gleichzeitig zu starten, da Hibernate in diesem Fall die Datenbank vor einem Test nicht neu aufbaut und somit Artefakte aus den Tests zurückbleiben können.

Zudem wurde testspezifischer Code geschrieben, der sich im Source-Ordner `src` im Package `de.unipotsdam.nexplorer.testing` befindet.

3.3 Architektur der RouteMe Implementierung

Das für das Spiel relevante System unterteilt sich in verschiedenen logische Blöcke. Abbildung 6 zeigt die logischen Blöcke sowie deren Kommunikationspfade. Eine Linie zwischen Blöcken bedeutet hierbei, dass der den Blöcken zugehörige Java-Code miteinander kommuniziert oder sich gegenseitig einbindet und nutzt. Im Folgenden wird die Zugehörigkeit von Packages und Blöcken angegeben. Um die Übersicht zu erhöhen, wurde auf das Präfix `de.unipotsdam.nexplorer` verzichtet. Zudem bedeutet die Angabe eines Oberpackage, dass die dazugehörigen Unterpackages ebenfalls gemeint sind.

- DTOs: `server.shared`, `server.dto`, `server.persistence.hibernate.dto`
- Display-Logik: `client` sowie der Javascript-Code
- Sensoren: Javascript-Code
- Gateway: Package `server` ohne Unterpackages, `server.rest`
- Spiel-Logik: `server.data`, `server.di`, `server.persistence` ohne Unterpackages
- Timer: `server.time`
- Routing-Logik: `server.aodv`
- Datenbank: `server.persistence`

Abgesehen von einer Komponente (Timer, Abschnitt 3.3.3) ist der Server javaseitig zustandslos aufgebaut. Der Zustand wird nur in der Datenbank gehalten.

Im Folgenden werden die einzelnen logischen Schwerpunkte näher beschrieben.

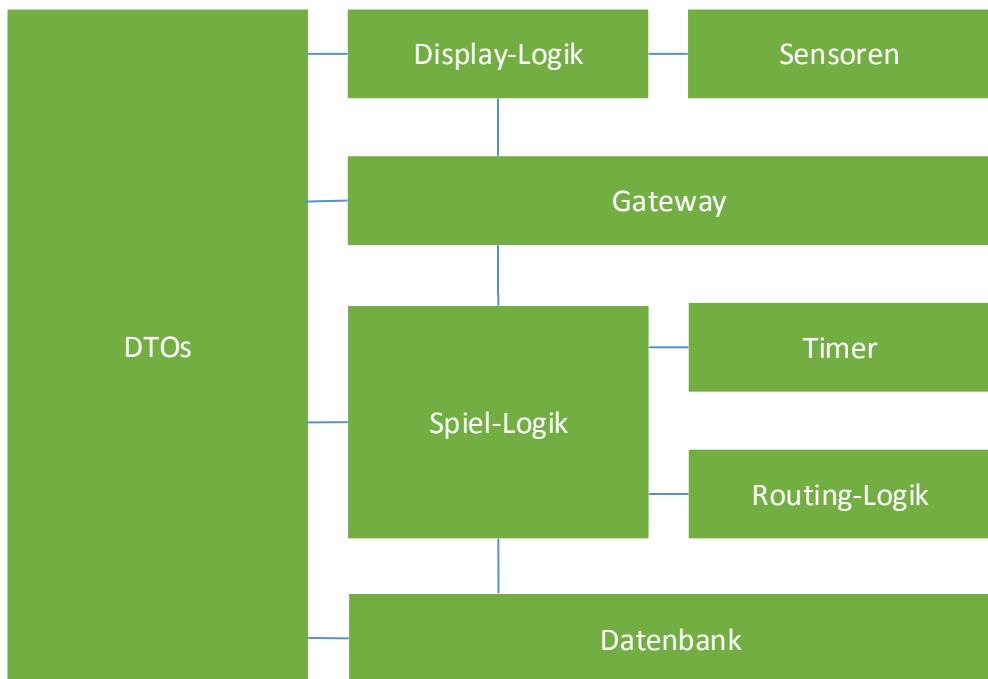


Abbildung 6: Logische Architektur der RouteMe-Implementierung

3.3.1 Datenbank

Die Datenbank ist größtenteils aus der vorhergehenden Diplomarbeit übernommen. In Abbildung 7 ist die aktuelle Version dargestellt.

Besonders auffällig sind die fehlenden Fremdschlüsselreferenzen. Obwohl an verschiedenen Stellen auf IDs aus der Players-Tabelle verweisen wird, sind diese nur selten als Fremdschlüssel eingetragen. Dies ist auf die Diplomarbeit zurückzuführen. Da die Datenbank für unsere Arbeiten genügt hat, haben wir diesen Aufbau nicht verändert.

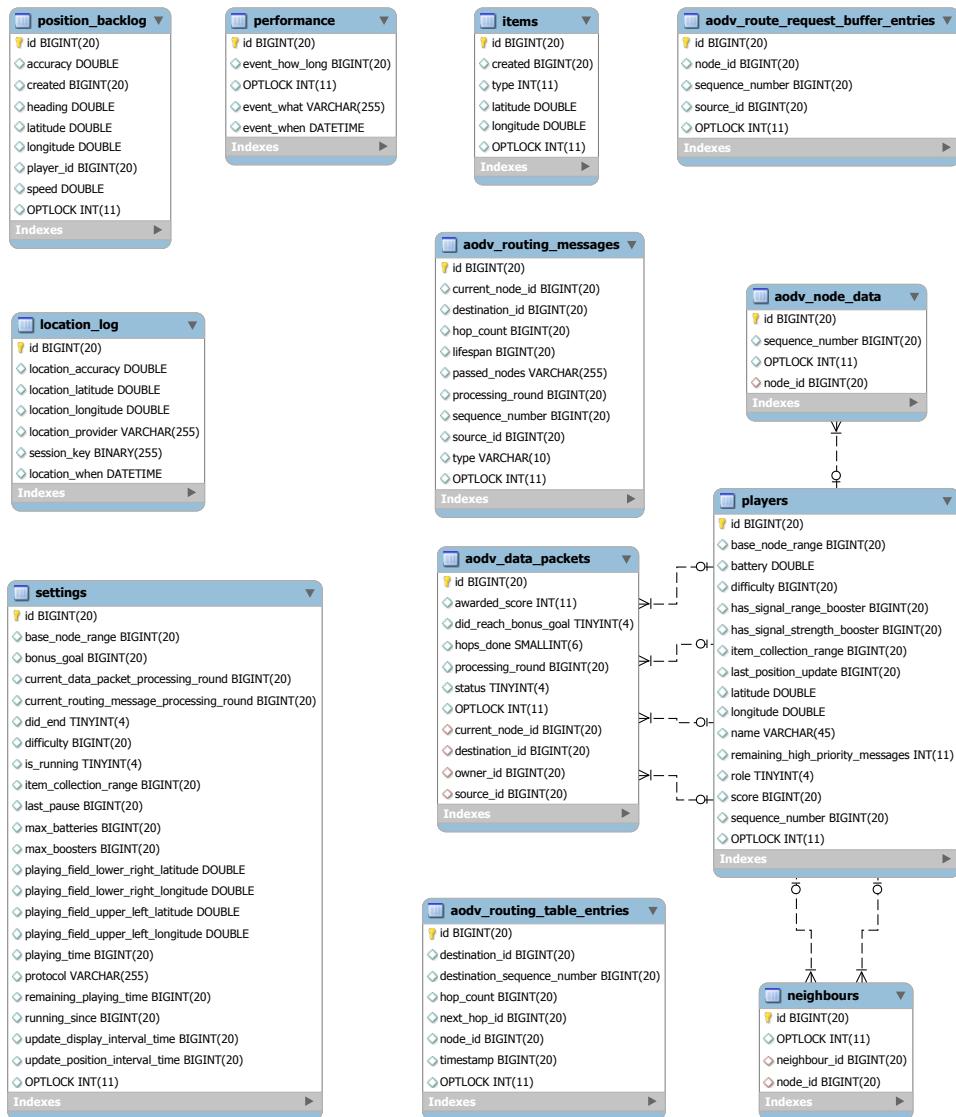


Abbildung 7: Datenbank-Schema

Denormalisierung Die Datenbank ist an einigen Stellen denormalisiert. Es handelt sich hierbei um Spalten aus der Settings-Tabelle, die in andere Tabellen kopiert wurden. Diese Entscheidung wurde während der Performance-Analysen getroffen, als sich herausstellte, dass der doppelte Zugriff auf die Datenbank (einmal für den eigentlichen Datensatz und einmal für die Settings-Spalte) zu langsam war. Die kopierten Spalten sind in der Tabellen Players zu finden, es handelt sich um base_node_range und item_collection_range. Die betroffenen Spalteninhalte werden beim Login eines Spielers, also beim Erstellen eines neuen Players-Eintrages, gesetzt. Daraus ergibt sich, dass alle Veränderungen an der base_node_range und item_collection_range, die nach dem Login eines Spielers geschehen, sowohl in der Settings- als auch in der Players-Tabelle korrigiert werden müssen.

Logischer Aufbau Die Datenbank besteht aus mehreren Untergruppen: Logging, AODV und Spiel. Diese Gruppierungen sind nicht in Schemas aufgeteilt, obwohl eine Einteilung für langfristiges Verständnis und für Wartungsarbeiten sicherlich sinnvoll wären.

- Logging: Hier werden Diagnose-Daten abgelegt, die während der Entwicklung für verschiedene Auswertungen verwendet wurden. Dazu gehören position_backlog, performance und location_log.
- AODV: Für das Routing-Protokoll AODV spezifische Informationen werden durch das Tabellen Präfix aodv_ gekennzeichnet. Fremdschlüssel-Relationen sind unidirektional. Sie zeigen auf Spiel-Tabellen. Zu den AODV-Tabellen gehören aodv_routing_messages, aodv_node_data, aodv_data_packets und aodv_routing_table_entries.
- Spiel: Spiel-bezogene Daten wie Spieler, Punkte und Spieleinstellungen werden hier gespeichert. Zu den Tabellen gehören players, neighbours, items und settings.

In der Entwicklungsphase wird die Datenbank beim Serverstart von Hibernate erstellt. Hibernate verwendet dazu die in `config/hibernate.cfg.xml` angegebenen DTOs. Diese Einstellung ist für Produktivsysteme nicht geeignet, da Hibernate für die Erstellung der Datenbank-Tabellen Administrator-Rechte benötigt, von denen im Produktivbetrieb aus Sicherheitsgründen abgeraten wird. Um ein Produktiv-Setup zu erreichen sollte daher die Einstellung `hibernate.hbm2ddl.auto` unter `config/hibernate.cfg.xml` umgestellt werden und von einer von Hibernate auf einem Entwicklungs-Datenbankserver erstellten Datenbank ein SQL-Export erstellt werden. Dieser SQL-Export kann dann auf der Produktiv-Datenbank manuell als Administrator eingespielt werden, sodass sich Hibernate auf eine korrekte Datenbank verbindet.

Da von der Diplomarbeits-Datenbank ausgegangen wurde, sind viele Altlasten übernommen worden: So existieren ungenutzte Spalten (`Players.has_signal_strength_booster`), passen Spaltenname und verwendeter Datentyp nicht zusammen (`Players.has_signal_range_booster` als Integer) und fehlen Fremdschlüsselrelationen (`Settings` und `Players`, `Aodv_routing_table_entries` und `Players`).

Die DTOs wurden über Reverse Engineering aus der Diplomarbeit-Datenbank erstellt. Änderungen an der Datenbank wurden danach in den DTOs vorgenommen, sodass Hibernate beim Start immer eine aktuelle Datenbank erstellt.

Performance Wie bereits erwähnt war die Austauschbarkeit der Datenbank eines der Entscheidungskriterien für Hibernate. Um die für den Spielbetrieb geeignete Datenbank zu finden wurden die folgenden Datenbanksysteme installiert und evaluiert:

- MySQL 5.5.28 für Windows
- H2 1.3.170 für Windows
- PostgreSQL 9.2.3.1 für Windows

PostgreSQL wurde nur kurz betrachtet, da die meisten Webserver auf MySQL setzen oder eine In-Memory-Datenbank wie H2 unterstützen. In den aufgestellten Testszenarien hat PostgreSQL die gleichen Performance-Eigenschaften wie MySQL gezeigt. Um Performance-Daten über den Server zu erlangen wurde die Knoten-KI um eine Performance-Komponente erweitert, die die Zeit zwischen Absenden der Anfrage und Erhalt der Antwort misst. Dabei hat sich gezeigt, dass die Anzahl an Datenbankzugriffen pro Abfrage maßgeblich entscheidend für die Antwortzeit ist. Tabelle 8 gibt einige Mittelwerte gegliedert nach SVN-Revision und Datenbanksystem an. In einigen Zeilen fehlen Werte, was durch einen Strich symbolisiert ist. Das Fehlen der Werte bedeutet, dass der RouteMe-Server die an ihn gestellte Last nicht bearbeiten konnte. Dies äußerte sich durch eine CPU-Auslastung von 100% und dem Verwerfen von Anfragen in Form von Timeouts. Um den Performance-Engpässen entgegenzuwirken, wurden Ausführungszeiten mit Hilfe der Sampling- und Profiling-Tools von VisualVM [SH12] ermittelt. Die größten Zeiteinsparungen ergaben sich durch die Denormalisierung der Datenbank, da die Settings-Tabelle nicht mehr abgefragt werden musste und so Mehrfachanfragen an die Datenbank verhindert wurden.

Revision	DBMS	Knoten	readGameStatus	sendLocation	collectItem
7940	mysql	2	18ms	36ms	36ms
		4	14ms	40ms	33ms
		6	15ms	45ms	28ms
		8	-	-	-
7943	mysql	6	20ms	44ms	41ms
		8	-	-	-
7945	mysql	6	16ms	47ms	45ms
		8	19ms	56ms	50ms
7946	mysql	6	19ms	13ms	25ms
		8	12ms	14ms	25ms
7948	mysql	8	12ms	17ms	27ms
		10	-	-	-
7969	mysql	8	14ms	49ms	50ms
		10	27ms	30ms	42ms
		12	24ms	24ms	44ms
	h2	10	9ms	24ms	13ms
		12	13ms	11ms	20ms

Abbildung 8: Performance-Messungen (schneller ist besser)

Ein weiterer Performance-Gewinn wurde durch die Veränderung der Locking-Strategie erzielt. *Locking* bezeichnet die Blockade einer Tabelle oder eines Datensatzes, um Race-Conditions durch gleichzeitiges Lesen und Schreiben zu verhindern. Dabei sind multiple Lesezugriffe unproblematisch, multiple Schreibzugriffe und gleichzeitige Lese- und Schreibzugriffe müssen jedoch verhindert werden, um inkonsistente Daten zu vermeiden. Hibernate verwendet standardmäßig pessimistisches Locking, was in den meisten Datenbanksystem als Table-Locking umgesetzt wird. Dabei wird eine Tabelle bei einem Schreibzugriff gesperrt, sodass nur der Schreibvorgang Datensätze ändern kann, während alle weiteren Schreib- und Lesezugriffe warten müssen. Bei häufigem und hochfrequenten Wechseln von Lese- und Schreibzugriffen reduziert diese Strategie die Datenbankperformance erheblich, weshalb in diesen Fällen *optimistisches Locking* empfohlen wird. Dabei werden - sofern es das Datenbanksystem unterstützt - keine Datensätze gesperrt, sodass alle Schreib- und Lesezugriffe ohne Verzögerung parallel weiterarbeiten. Stattdessen findet eine Prüfung auf Veränderung beim Commit statt. Falls ein zu schreibender Datensatz gegenüber seinem zuletzt bekannten Zustand von einem anderen Prozess verändert wurde, wird der Commit zurückgewiesen.

Um optimistisches Locking in Hibernate zu aktivieren ist das Hinzufügen einer besonderen Spalte - der Versions-Spalte - erforderlich [Red11]. Anhand der `@Version`-Annotation erkennt und aktiviert Hibernate das Row-Locking. Fast alle DTOs im Projekt besitzen diese Versions-Spalte. Wird ein Datensatz geladen und verändert, so überprüft Hibernate beim Commit, dass der Wert der Versions-Spalte in der Datenbank identisch mit dem des Datensatzes beim Laden aus der Datenbank ist. Bei einem erfolgreichen Commit erhöht Hibernate den Wert in der Versions-Spalte. Diese Spalte darf nicht im DTO verändert werden.

H2 unterstützt kein optimistisches Locking [Tea13a], ist aber mit der Table-Locking-Strategie schneller als MySQL. Bei MySQL brachte die Einführung des optimistischen Locking einen erheblichen Performance-Gewinn, sodass annähernd die Geschwindigkeit des H2-Datenbanksystems erreicht wurde.

Für den Testbetrieb hat sich die Nutzung von H2 als vorteilhaft erwiesen, da kein externes Datenbanksystem installiert werden muss und die Datenbank im Vergleich zu anderen Datenbanksystemen sehr schnell ist. Im Produktiveinsatz werden alle Daten in der Datenbank gelöscht, sobald die JVM beendet wird. Allerdings bietet eine dateibasierte Datenbank wie MySQL hier keinen Vorteil, da das ursprüngliche Konzept [MLZ11] keine Versionierung oder Archivierung von Spieldaten vorsieht.

3.3.2 Dependency Injection

Der Begriff *Dependency Injection* wurde 2004 von Martin Fowler geprägt [Fow04] und zielt auf die Entkopplung von Komponenten. Klassen arbeiten mit Interfaces, deren Implementierung ihnen von außen von einem *Dependency Container* übergeben werden. Auf eine Einführung wird verzichtet.

Der Server verwendet den Dependency Container *Guice* [LBWG11]. Guice findet Implementationen über *Module*. In diesen Modulen werden die Regeln angegeben, nach denen Guice anschließend Instanzen erstellt und mit den zugehörigen Abhängigkeiten befüllt. Generell sollte auf das direkte Ansprechen des Guice-Containers verzichtet werden, da der *Unit*-Konstruktor bereits alle notwendigen Aufgaben übernimmt. Lediglich Test-Klassen müssen den Container mit eventuell zu mockenden Abhängigkeiten einrichten.

Alle Klassen, die Abhängigkeiten vom Container zugewiesen bekommen, müssen ihren Konstruktor mit der `@Inject`-Annotation versehen. Diese weiß Guice an, den annotierten Konstruktor für Dependency Injection zu verwenden. Um einen eingerichteten Logger zu erhalten muss eine private Klassenvariable vom Typ `org.apache.logging.log4j.Logger` angelegt werden, die mit einer `@InjectLogger`-Annotation versehen ist. Diese Annotation weist Guice an, einen Logger für die entsprechende Klasse zu erstellen, zu initialisieren und anschließend zu setzen.

3.3.3 Timer

Von der Zustandslosigkeit ausgenommen sind die Timer in `server.time`: Um die Abhängigkeit vom Admin-Interface zu lösen mussten regelmäßige Intervalle innerhalb des Servers verarbeitet werden. Regelmäßig durchgeführte Aufgaben sind:

- Neue Items auf dem Spielfeld verteilen
- AODV-Pakete weiterleiten
- Knoten-Batterien aktualisieren
- Nachbarn aktualisieren
- Spieleinstellungen verändern

Die Timer sind größtenteils darauf ausgelegt, dass sie Logik in ihren Implementierungen vermeiden. Stattdessen rufen sie Gateway-Methoden auf, die die Delegierung der Aufgaben übernehmen.

Die Verwendung eines `TimerCallback` ist im Nachhinein nicht notwendig. Die einzige Klasse, die Gebrauch von diesem Callback macht ist `SettingsUpdateStarter`, deren Code im `Timer`-Package fehplatziert ist. Der Callback-nutzende Code sollte eigentlich im Admin-Servlet stehen. Um keine Instabilität in den Server zu bringen wurde auf kurzfristige Änderungen verzichtet. Langfristig sollte jedoch eine Migration des Codes eingeplant werden.

3.3.4 Klient

GWT kompiliert das fertige Projekt in Javascript, welches mit dem HTML zusammen übertragen wird. In dem HTML stehen nur noch Metatags und die übliche Einteilung in HEAD und BODY. Das bedeutet, dass HTML Code, der in GWT geschrieben wird, durch Javascript an die von dem Programmierer gewünschte Stelle eingefügt wird. Es gibt daher drei unterschiedliche logische Ebenen auf denen programmiert wird: HTML, GWT (Java) und Javascript. Dabei geschieht die Verknüpfung von GWT und HTML mittels IDs im HTML. Javascript wird wie bereits erwähnt von GWT per JSNI aufgerufen.

Der klientseitige Code ist sehr schlank, da die meisten Berechnungen serverseitig erfolgen. Seine Komplexität entsteht jedoch durch die verschiedenen Programmierebenen, die zum Teil Altlasten sind und es für spätere Entwickler erschweren, den Code zu verstehen. Um diesem Problem zu begegnen dokumentieren wir den Klient pro *package*.

Zur Orientierung ist es hilfreich zu wissen, dass Javascript und CSS wie in anderen Webprojekten per relativem Pfad in der HTML Datei inkludiert werden. Diese Dateien liegen gemeinsam mit den von GWT kompilierten Skripten im **war**-Verzeichnis. Die Javascript Dateien liegen in diesem Projekt im den Ordner **war/js**. Die CSS Dateien liegen in dem Ordner **war/css**. Die Datei **war/settings.xml** enthält die Konfiguration für das Spiel

Es gibt drei Klienten. Eine Adminsicht, eine Sicht für die Nachrichtenspieler und eine Sicht für die mobilen Knoten.

Toplevel Abbildung 9 zeigt das *package de.unipotsdam.nexplorer.client*. AdminService und IndoorService stellen die XML-RPC Schnittstellen zu dem Server dar. Darunter sind die Einstiegspunkte gelistet.

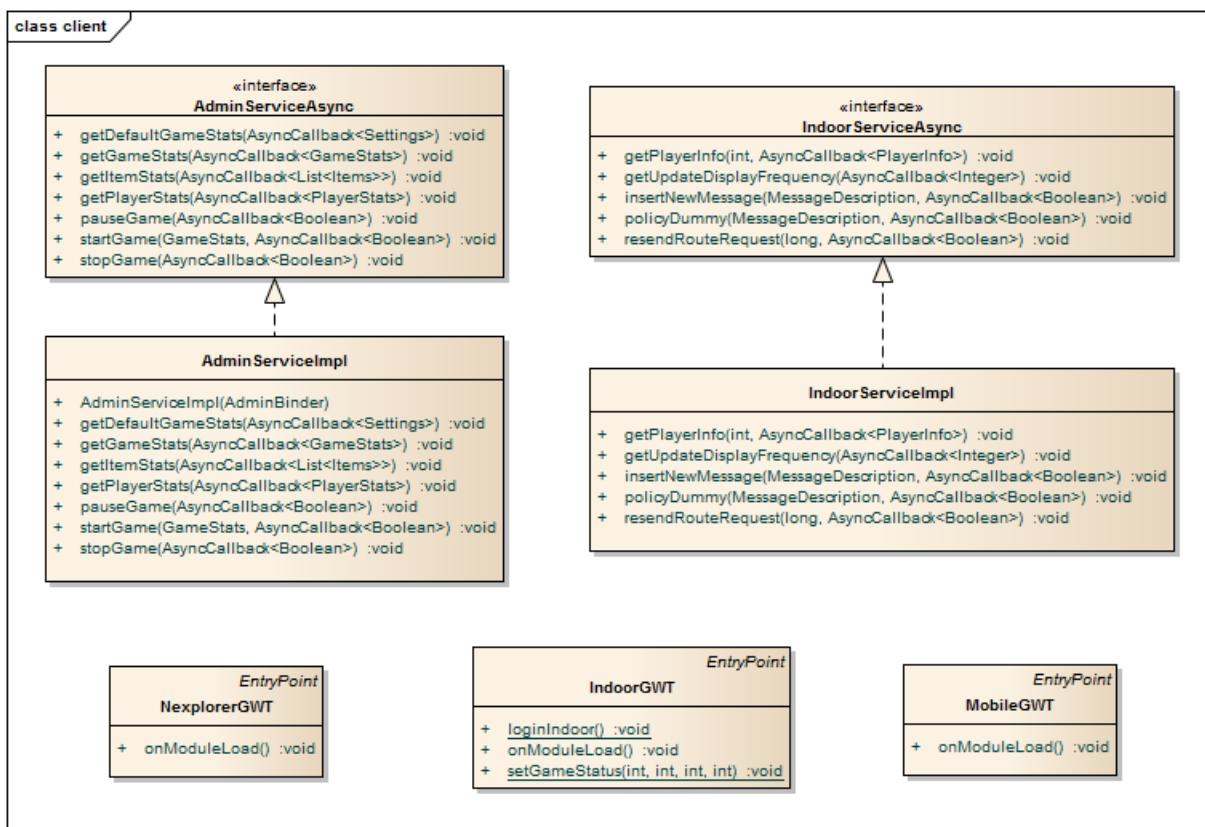


Abbildung 9: Toplevel-Klassendiagramm für den Klienten

Adminklient Das package de.unipotsdam.nexplorer.client.admin enthält die Struktur der View für den Adminklient. Diese ist in Abbildung 10 zu sehen.

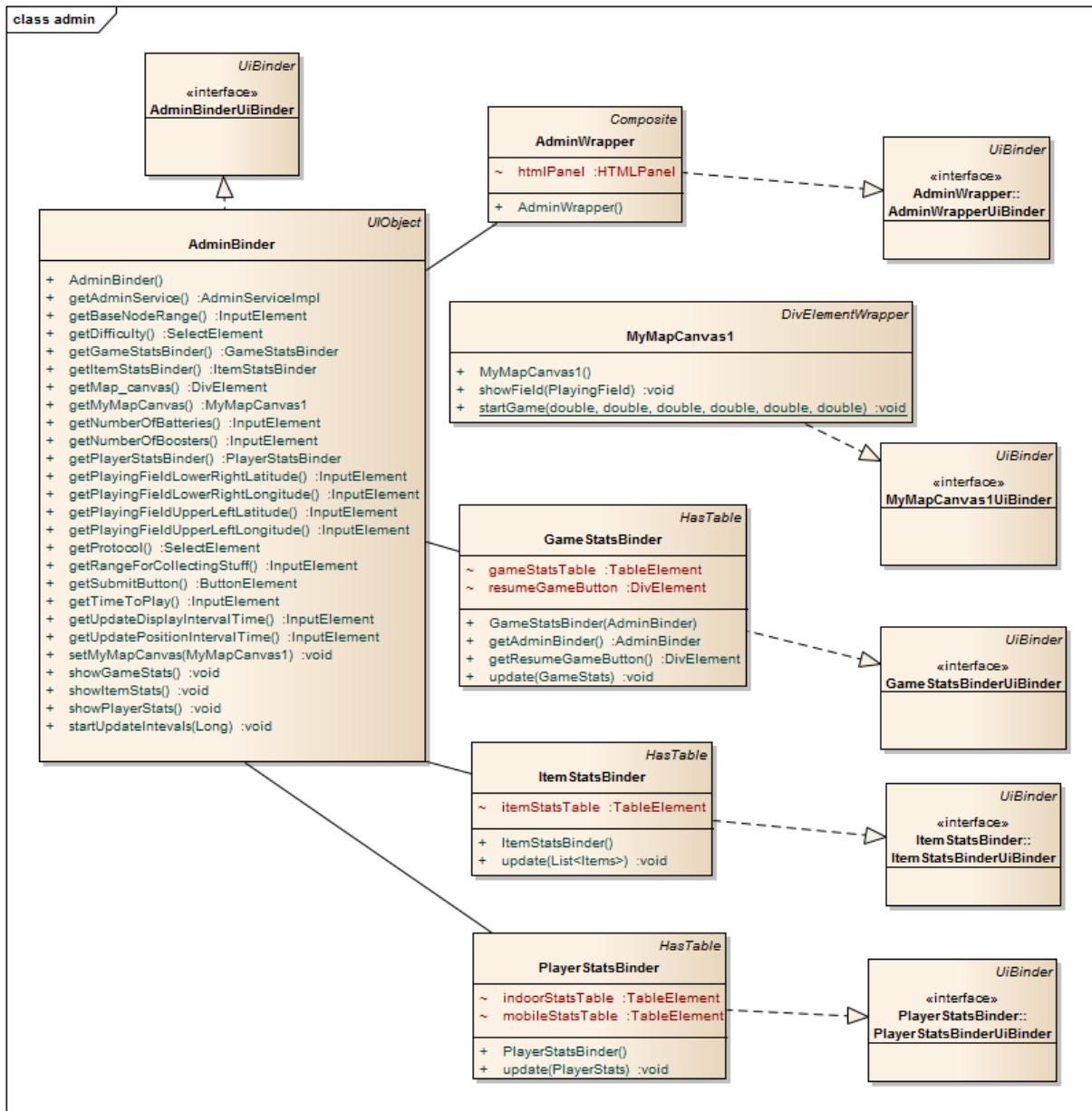


Abbildung 10: Struktur der Adminviewkomponenten

Ein dünner Wrapper enthält ein HTML-Panel, in dem das originale HTML enthalten ist. Es ist jedoch funktional untergliedert. MyMapCanvas1 enthält das HTML, welches für die geographische Übersichtskarte mit den Spielerpositionen zuständig ist. GameStatsBinder enthält das HTML für die Anzeige des Spielzustandes. ItemStatsBinder enthält das HTML für die Anzeige der Items (Batterien, Booster etc.) und PlayerStatsBinder enthält das HTML für die Anzeige der Spielerzustände.

Das package `de.unipotsdam.nexplorer.client.admin.viewcontroller` enthält den ViewControllerer für den Adminklienten. Die Aufgaben des ViewControllerer bestehen darin, die GWT-GUI bestehend aus den Bindern (letzter Abschnitt) mit den anderen Programmelementen zu verbinden.

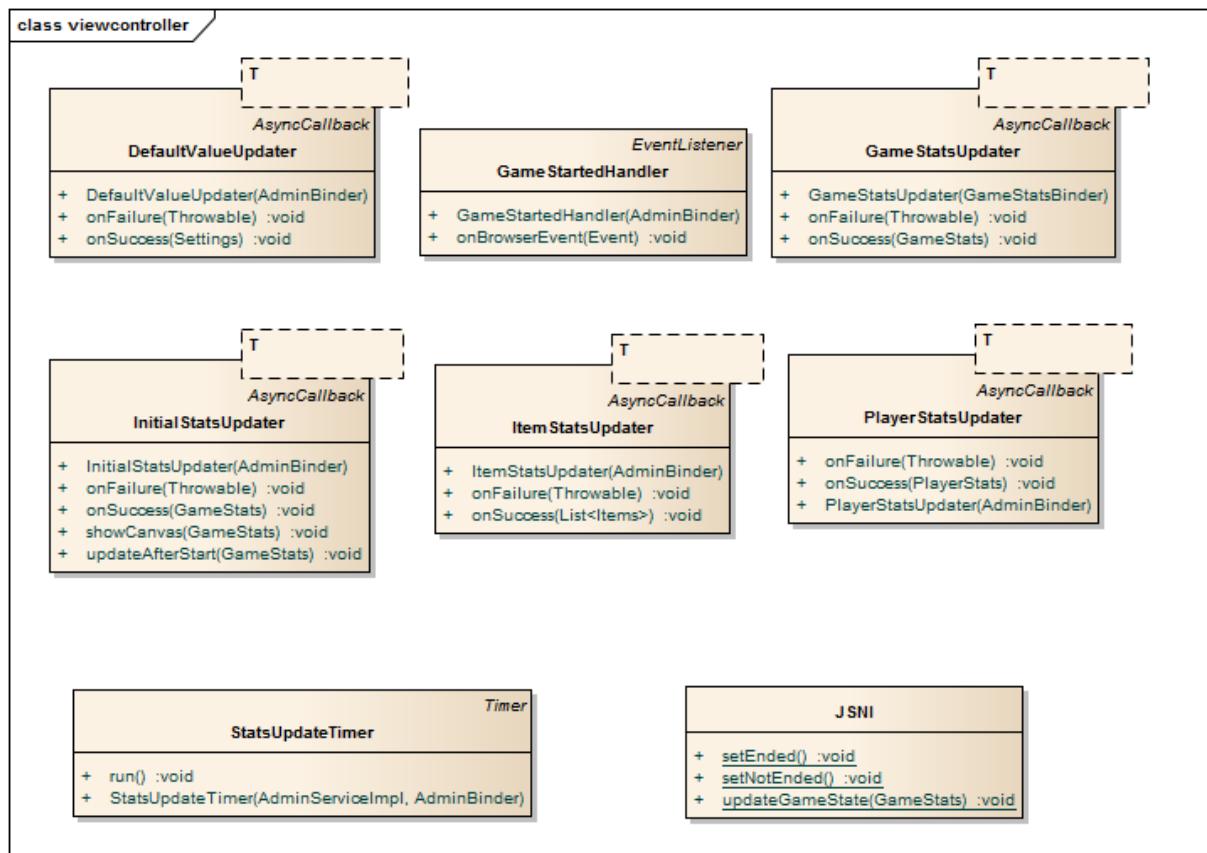


Abbildung 11: Der Controller für den Adminklienten

Die Updater, die in Abbildung 11 aufgeführt sind, stellen die Verbindung zu der Serverschnittstelle her:

1. DefaultValueUpdater sendet die gewählten Spielwerte an den Server
2. InitialStatsUpdater lädt die im XML gespeicherten DefaultSpielwerte
3. GameStartedHandler verarbeitet die Serverdaten nach dem Spielstart

4. GameStatsUpdater verarbeitet den neuen Spielzustand
5. ItemStatsUpdater verarbeitet die neuen Itemstatistiken
6. PlayerStatsUpdater verarbeitet die neuen Spielerstatistiken

Eine Sonderrolle spielen der StatsUpdateTimer und die JSNI-Klasse. Der StatsUpdateTimer reguliert die Frequenz in der die Adminsicht aufgefrischt wird. Hier kann auch speziell adjustiert werden, wenn die Spielparameter zum Spielen nicht geeignet sind. Die JSNI Klasse wurde bereits erwähnt. Sie überträgt den allgemeinen Systemzustand auf das native Javascript.

Der schematische Ablauf der Aktivitäten lässt sich in der Abbildung 12 nachvollziehen. Zunächst wird die HTML-Seite angefragt, woraufhin GWT innerhalb des Javascripts aufgerufen wird. Hier wird zunächst der *game configuration screen* gezeigt. Sollte das Spiel schon begonnen haben, wird dieser aus dem HTML gelöscht und durch die Adminsicht während dem Spiel ersetzt. An dieser Stelle wird parallel eine Schleife gestartet, die den aktuellen Zustand vom Server abfragt, wie auch die Adminübersichtskarte auf Javascript aktualisiert.

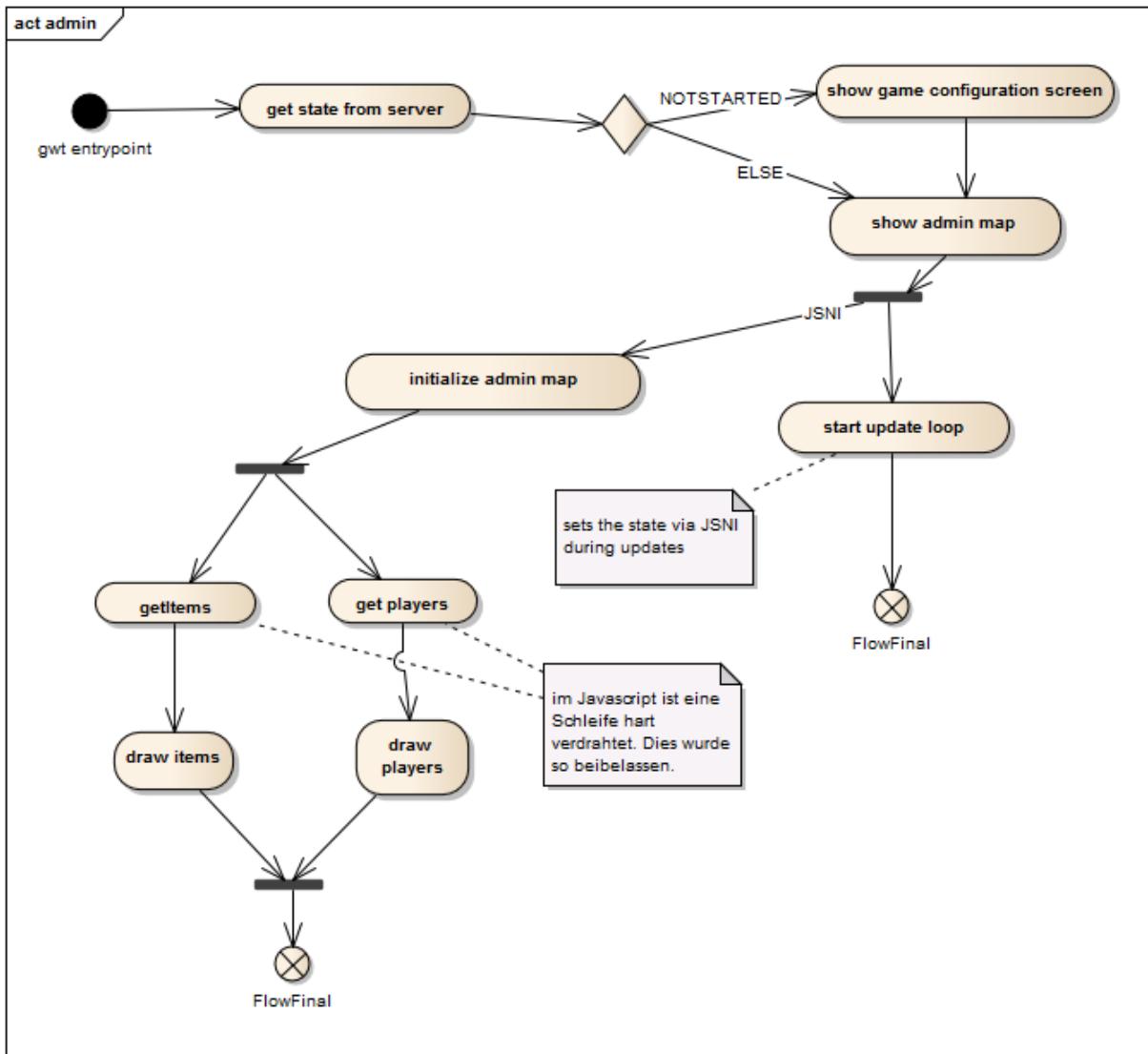


Abbildung 12: Aktivitätsdiagramm zum Adminklienten

Indoorklient Das package de.unipotsdam.nexplorer.client.indoor enthält den Klienten für die Nachrichtenspieler. Abbildung 13 zeigt eine UML-Sicht auf die Klassenstruktur.

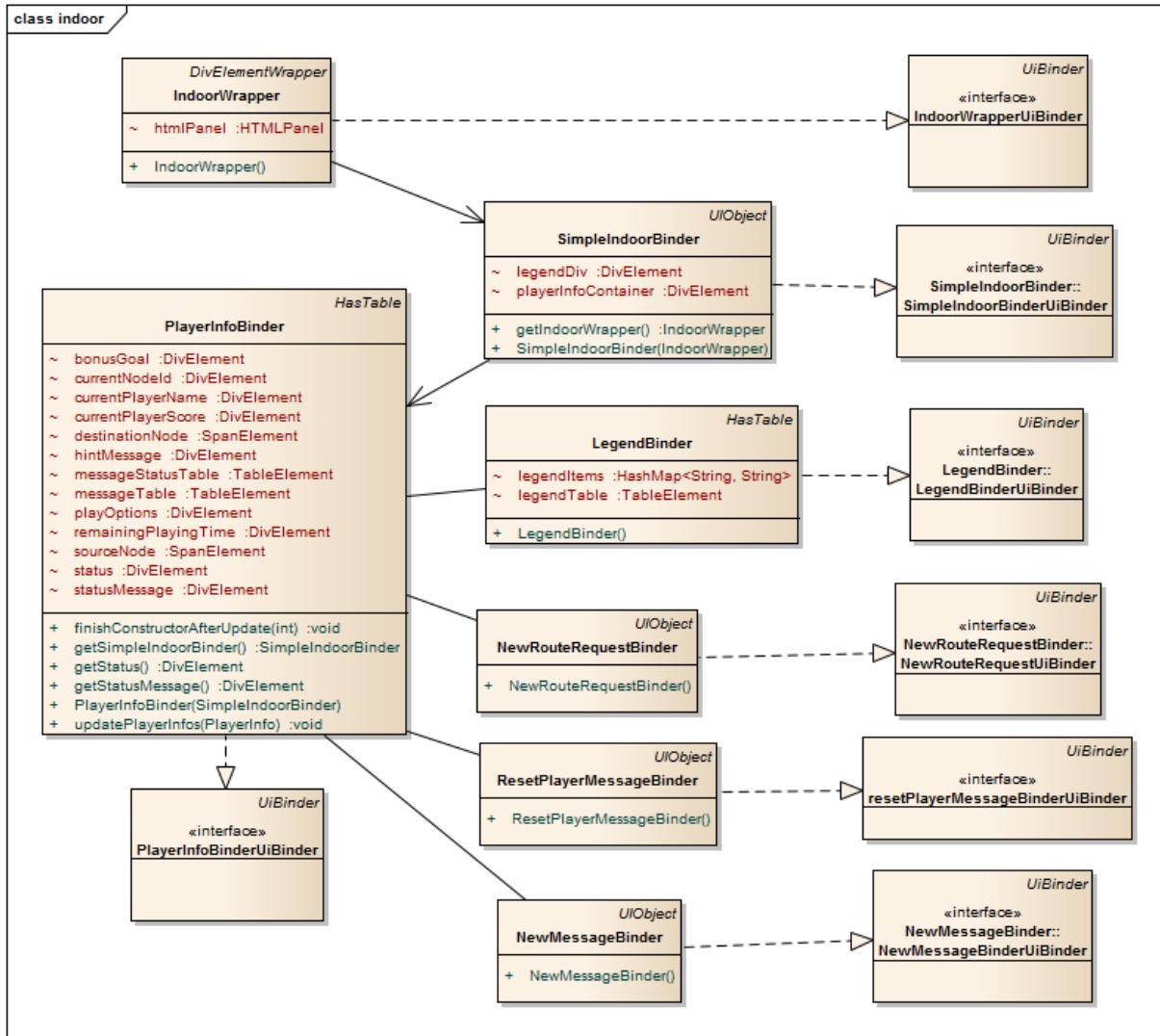


Abbildung 13: Klassendiagramm für den Indoorklienten

Wie bei der Adminsicht auch wird die Übersichtskarte von nativem Javascript übernommen. Hier wurde auch ein maßgeblicher Teil des Codes beibehalten und nur marginal angepasst oder korrigiert. Auf GWT- Seite wird die Aktualisierung der Buttons und der dahinterliegenden Funktion durch GWT übernommen. NewRouteRequestBinder steht für den Button, der mit „Nachricht neu versenden“ betitelt ist. ResetPlayerMessageBinder steht für den Button, der mit „Vorgang abbrechen“ betitelt ist. NewMessageBinder steht für den Button, mit dem man neue Nachrichten versenden kann. LegendBinder enthält die Komponente, die tabellarisch den aktuellen Zustand anzeigt.

Die einzelnen Binder enthalten intern natives Javascript *inline* im HTML mit Funktionsaufrufen, die die Buttonclicks verarbeiten. Dies ist eine Altlast, die besonders ungünstig ist, da die Funktionsaufrufe hier nicht erwartet würden. Dies war der Preis dafür, 80 Prozent des Originalcodes zu übernehmen. In Retroperspektive sollte dies refaktorisiert werden, indem die Komponente komplett auf GWT umgestellt wird.

Der ViewController besteht nur aus einem einzigen Timer, der analog zu dem StatsUpdater beim Admininterface funktioniert.

Mobileklient Der Mobile Client wurde nicht mit GWT ummantelt, da hier bereits von Tobias Moebert eine klare Schnittstelle eingeführt wurde.

Die einzige Aufgabe von GWT besteht hier darin, das HTML und native Javascript auszuliefern.

Der Ablauf im Javascript lässt sich in seinen groben Zügen in dem folgenden Aktivitätsdiagramm auf Abbildung 14 nachvollziehen:

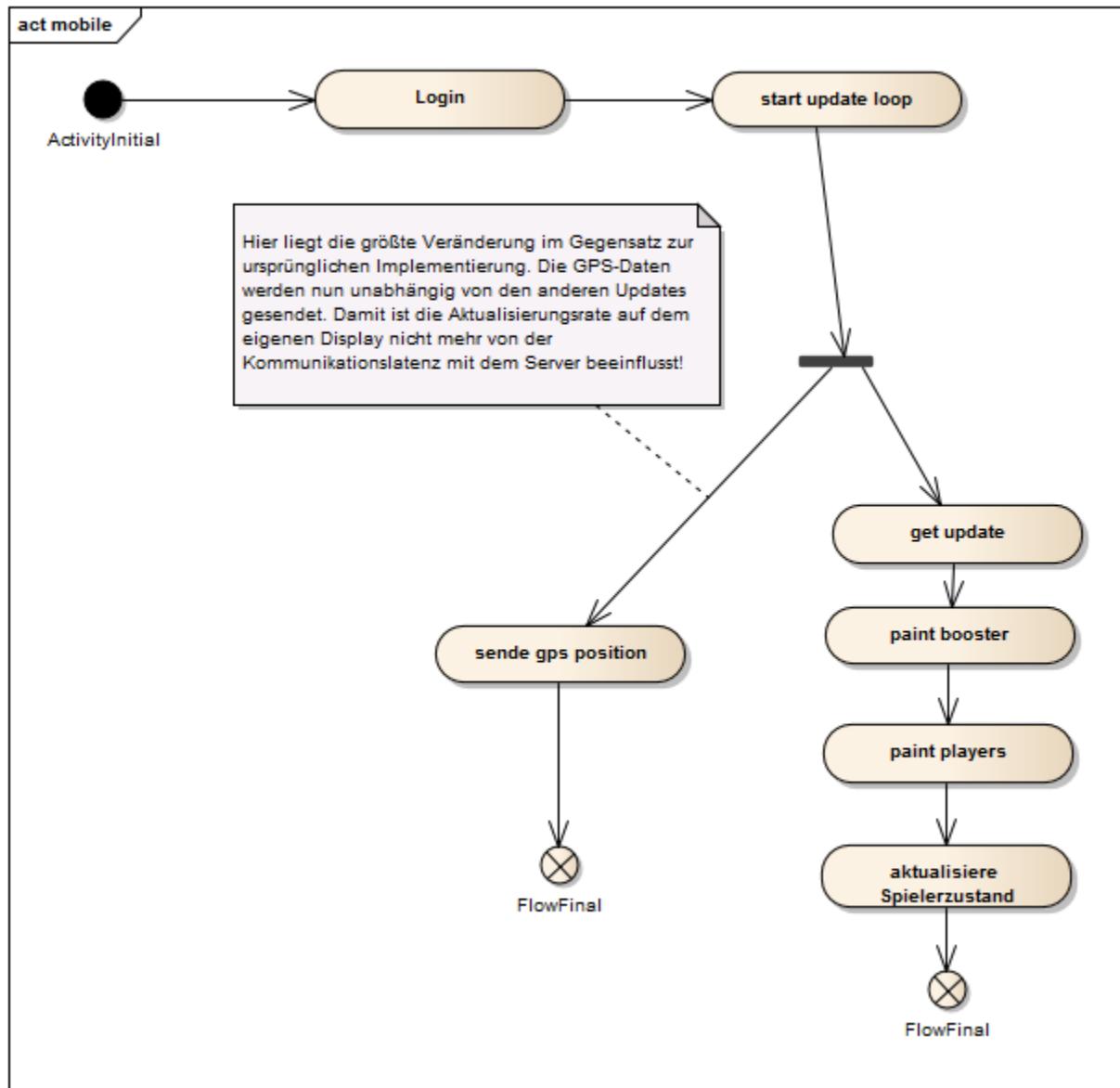


Abbildung 14: Aktivitätsdiagramm für Mobilklienten

3.3.5 Schnittstellen

Im folgenden beschreiben wir die grundsätzliche Struktur XML-RPC der Schnittstellen im Detail, um weiterführenden Arbeiten die Möglichkeit zu geben, ohne tieferes Studiums des Servers neue Klients zu bauen. Zunächst stellen wir die Schnittstellen dar, um danach die übertragenen Objekte zu erklären.

Schnittstellen-Abhängigkeiten In Abbildung 15 sieht man ein Schnittstellendiagramm, welches die logischen Packete REST, Java (GWT verwendbar) und den Anknüpfungspunkten auf Clientseite unterteilt ist. Wir betrachten hierbei das REST-*package* `de.unipotsdam.nexplorer.server.rest` und die XML-RPC-Schnittstelle `de.unipotsdam.nexplorer.server`. Das REST-*package* liegt eine Hierarchiestufe unter den XML-RPC relevanten Klassen, da es eine spezielle Ausprägung der durch die Java-Interfaces definierten Schnittstelle darstellt.

NexplorerGWT steht für die Adminsicht, die auf diese URL (`[SERVER-URL]/NexplorerGWT.html`) festgelegt ist. Analog dazu steht IndoorGWT für die Nachrichtensicht und MobileGWT für die Klients der mobilen Knoten.

GameEvents repräsentiert die ursprünglichen Methoden, um von außen den Takt oder andere Spieldereignisse anzugeben. Einen Großteil dieser Methode haben wir übernommen und implementiert, aber als *deprecated* markiert, da sie nach unserer Sicht zu einem instabilen System führen.

GameImpl ist der Einstiegspunkt, um von außen grundlegende Spielfunktionen wie Spielstart, oder Spielende zu verwenden. Diese Klasse wird sowohl von der Adminsicht als auch von der Indoorsicht genutzt.

Beide Klassen, GameEvents und GameImpl rufen innerhalb von Java die Admin-Klasse auf, die die eigentliche Implementierung zur Verfügung stellt. Die Adminklasse wird jedoch auch direkt von GWT innerhalb des Klients genutzt.

Daneben verwendet der Indoorklient noch die Indoor-Klasse im REST-package, welche wiederum die Implementation der Indoor-Funktionen im *server*-package braucht. Diese werden auch über GWT direkt verwendet.

Die Login-Klasse wird sowohl von den Nachrichtenspielern als auch von den mobilen Knotenspielern verwendet, und realisiert den Login.

Die Klasse Mobil im REST-*package* arbeitet analog zu der Klasse Indoor.

Datentransferobjekte Im JSON-Format sehen die Datentransferobjekte wie gewöhnliche unbenannte Javascript Objekte aus. Die automatische Konversion durch Jersey geschieht wie folgt:

1. Das Toplevel Objekt ist unbenannt
2. Geschachtelte Objekte tragen den Klassennamen der Java-Klasse
3. Properties in Form von Primitive Typen oder Strings werden auf einfache JSON-Properties abgebildet
4. HashMaps werden auf assoziative Arrays abgebildet



Abbildung 15: Strukturelle Übersicht über die Schnittstellen

Die Konversion ist in dem Jersey-Framework mit dem Adaptor-Pattern umgesetzt. Da wir keine Performanzprobleme durch falsche Datentypen erwarten, wie im Kapitel zu den theoretischen Vorüberlegungen beschrieben, haben wir einen abstrakten Adaptor eingeführt, der das Serialisieren und Deserialisieren generisch macht. Alle Klassen, die auf diese Art und Weise übertragen, müssen von der JSONable-Klasse erben. Die Datentypen werden damit nicht optimiert. Sollten hier irgendwann Probleme auftauchen, kann der Adaptor für den Einzelfall implementiert werden.

Die Abbildung 16 zeigt die erste von drei Graphiken, die die Struktur der serialisierten Datentransferobjekte (JSON) in Java-Notation abbildet.

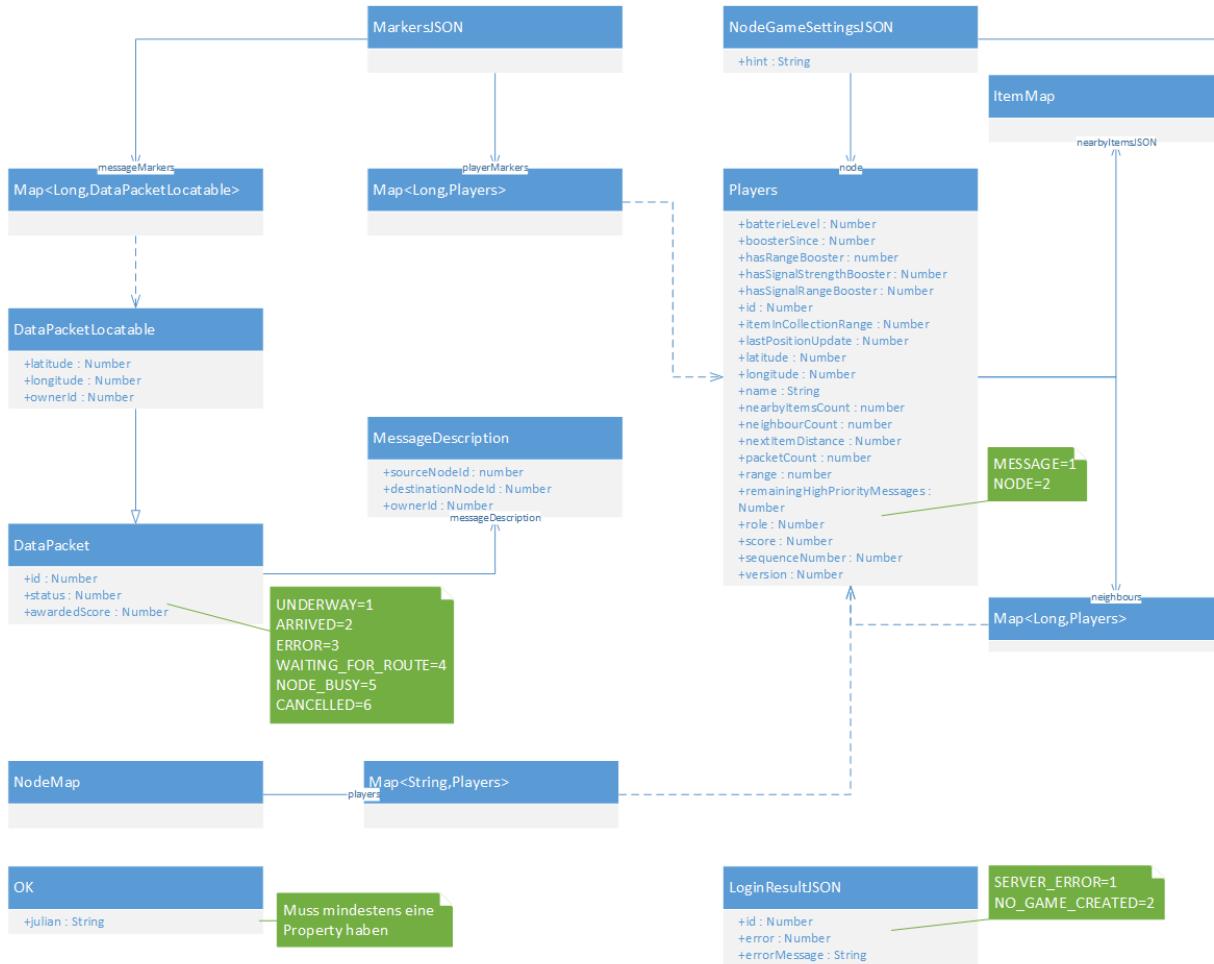


Abbildung 16: Erste Graphik zu den Datentransferobjekten

MarkersJSON liefert die notwendigen Daten für die Indoorspieler. Zum einen benötigen diese die Position der mobilen Spieler abhängig von deren ID, zum anderen eine Liste von Datenpaketen und die „locatable“, also auf einer Karte verortbar, sind. Eine weitere Anforderung besteht darin, dass die Datenpakete den ursprünglichen Auftrag (von-nach) und ihren Auftraggeber kennen, da dies für die Anzeige der Flaggen notwendig ist.

NodeMap ist eine Klasse, welche die Spieler ihren IDs zuordnet.

Die Abbildung 17 zeigt die zweite von drei Graphiken. Sie knüpft rechts an Abbildung 16 an.

Als zentrale Objekte werden hier die Player Klasse und die Itemklasse dargestellt. Die Playerklasse enthält die Spielerdaten. Die Itemklasse enthält die Spielitems, namentlich Booster und Batterien. Auch die Items werden mit IDs als assoziative Arrays nach JSON serialisiert.

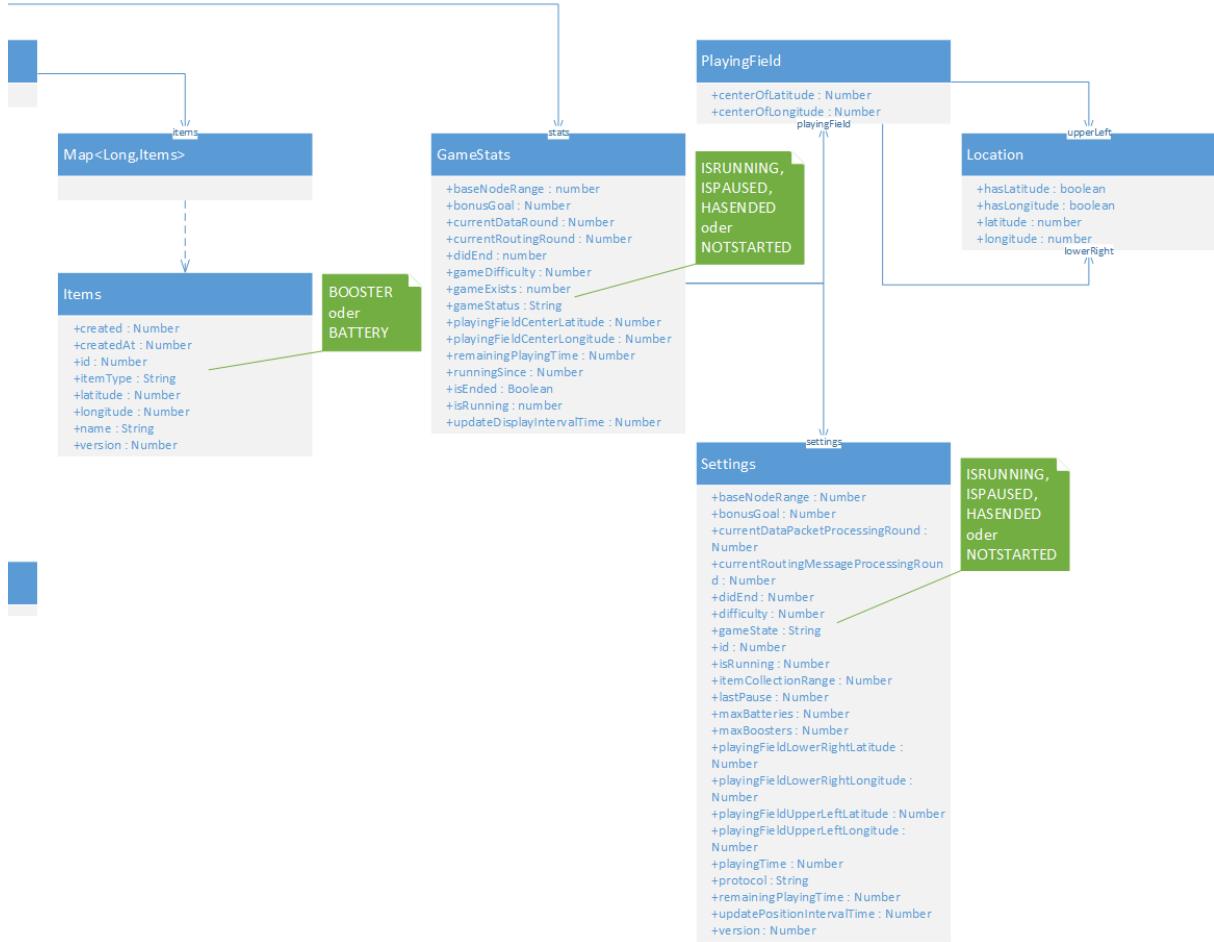


Abbildung 17: Zweite Graphik zu den Datentransferobjekten

Sie zeigt den Zusammenhang zwischen der GameStats und der Settings Klasse. Beide halten die wichtigsten Informationen zu dem Spielzustand und den gewählten Einstellungen. Spielzustand und Einstellungen sollten eigentlich unterschiedliche Objekte sein, werden aber als Altlast strukturell von uns übernommen.

In der Tat wurden in dem Originalprojekt sehr viele ungünstige Datentypen verwendet, welche sich in der Datenbank zeigen, (Long für semantische Booleans, int für semantische Booleans, long für Datetimestamps etc.). Diese Fehlgriffe führen zu überflüssigem Code bei der Auswertung im Klienten. Damit diese Probleme sich nicht innerhalb des GWT-Code fortpflanzen, haben wir die GameStats Klasse als Mantel für die Settingsklasse geschaffen, die die gleichen Properties, aber als höherwertige Datentypen bereitstellt. Damit es hier keine Coderedundanz gibt, enthält diese GameStats-Klasse die Settings-Klasse als Datenhalter.

3.4 Architektur der Tools

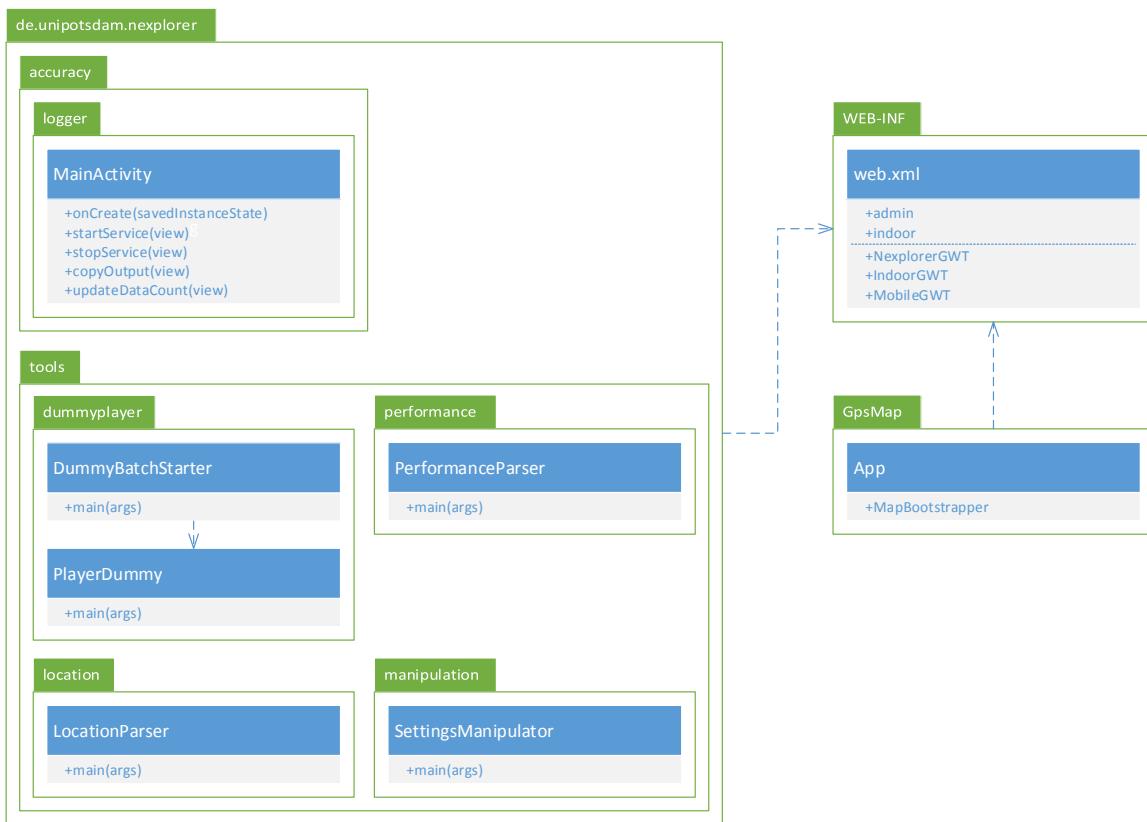


Abbildung 18: Übersicht der ausführbaren Bestandteile

Im Rahmen des Projektes sind neben dem Routing-Spiel verschiedene Werkzeuge entstanden. Eine Übersicht ist in Abbildung 18 gegeben. Die meisten Bestandteile sind Java-Programme, die über eine `main`-Methode in einem *Unterpackage* von `de.unipotsdam.nexplorer` gestartet werden.

Eine Ausnahme bildet `accuracy.logger`. Es handelt sich hierbei um eine Android-App, die zusammen mit `tools.location` und `GpsMap` zum GPS-Scouting verwendet wurde. Diese Werkzeuge werden in Abschnitt 3.4.1 näher beleuchtet.

Die mobile KI unter `tools.dummyplayer` kann Mobile-Spieler simulieren. Sie wird in Abschnitt 3.4.2 vorgestellt.

Um Performance-Daten zu analysieren, loggt der Server alle benötigten Zeiten innerhalb der Servlets in ein Performance-Log. Dieses Log wird von `tools.performance` gelesen und die darin enthaltenen Informationen in eine Datenbank geschrieben, um später weiterverarbeitet zu werden. Die Funktionsweise des Lesevorganges ist in Abschnitt 3.4.3 dokumentiert.

Werte wie die Sendereichweite der mobilen Knoten werden zu Spielbeginn festgelegt und sind nicht mehr über die Adminschnittstelle änderbar. Um eine optimale Abstimmung zu finden lassen sich ausgewählte Einstellungen mit Hilfe von `tools.manipulation`, welches in Abschnitt 3.4.4 beschrieben ist, auch im laufenden Spiel ändern.

Alle Werkzeuge benötigen einen laufenden Server, um Daten abzurufen oder zu senden. Darüber hinaus greifen einige Werkzeuge direkt auf die lokal in der `hibernate.cf.xml` eingestellte Datenbank zu, auch wenn der laufende Server eine andere Konfiguration hat.

3.4.1 GPS-Scouting

Wie in Abschnitt 2.9 dargelegt wurden mit Hilfe der Android-Applikation GPS-Daten gesammelt

Accuracy App Die Android-App ist im Play Store [Goo13] unter dem Namen UP Accuracy Logger zu finden. Sie kann nur von Geräten gefunden werden, die mindestens API-Level 15 (Android 4.0.3) haben. Die App verlangt folgende Rechte:

- Den GPS-Standort, um Daten zu sammeln
- Netzwerkzugriff, um die gesammelten Daten an den Server zu senden
- Standby-Modus deaktivieren, um auch dann noch GPS-Daten zu sammeln, wenn das Smartphone schon in den Standby-Betrieb wechselt würde

Im Folgenden ist die Arbeitsweise mit der App beschrieben. Ein Screenshot ist in Abbildung 19 zu sehen.

Der Vorgang des Datensammelns wird über *Start service* begonnen. Mit *Stop service* wird der Vorgang beendet. Nach dem Start des Vorgangs kann einige Zeit vergehen, bevor die ersten Daten empfangen werden. Die Anzahl der gespeicherten Datensätze kann jederzeit über *Update data count* abgerufen werden. Um die Daten an den Server zu senden sind die Adresse des Servers sowie sein Port anzugeben (zum Beispiel *141.89.53.181:8080*). Die App informiert über den Abschluss des Sendens, zusätzlich dazu wird auch auf dem Server eine Log-Nachricht ausgegeben.

Die App sammelt Daten von zwei Provider: GPS und NETWORK. Die gesammelten Daten werden nach dem Senden nicht vom Smartphone gelöscht. Bei wiederholten Sendevorgängen werden bereits gesendete Datensätze erneut übermittelt. Der Server hat keine Möglichkeit, doppelte Datensätze zu identifizieren. Um gesammelte Daten zu löschen muss in den Android-Einstellungen die App-Übersicht des Accuracy Logger aufgerufen werden, welche in Abbildung 19 zu sehen ist. Die Option *Daten löschen* entfernt die gesammelten Datensätze. Dieser aufwändige Weg wurde bewusst gewählt, um zu verhindern, dass beauftragte Datensammler (siehe Abschnitt 2.9), die mit dem Umgang der Smartphones nicht versiert sind, versehentlich Einträge löschen.

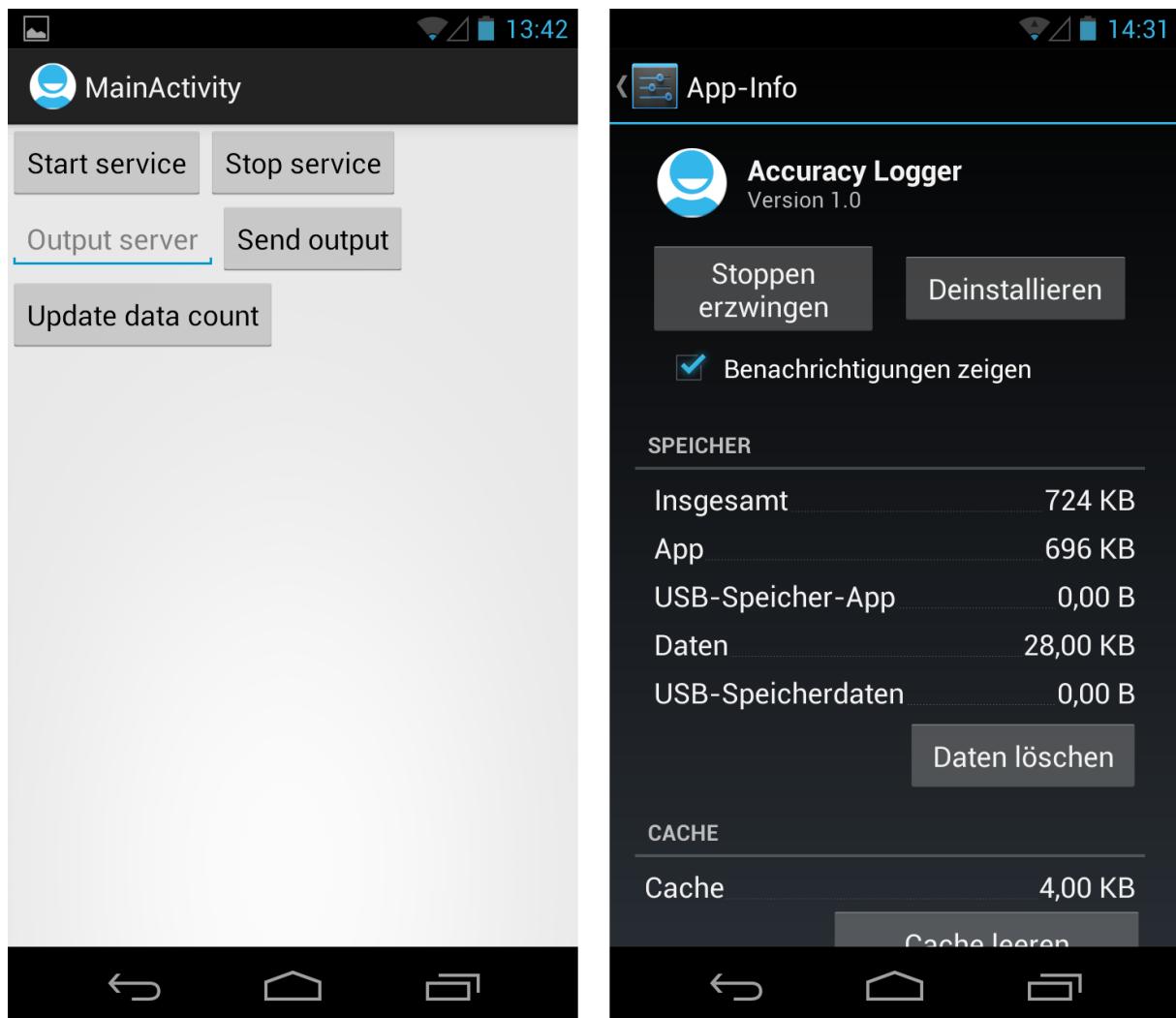


Abbildung 19: Screenshot der Accuracy App und der App-Einstellungen

Wie im Klassendiagramm in Abbildung 20 zu sehen ist besteht die App aus nur einer Activity und einem Service. Der Service koordiniert den Empfang von Positionsdaten, die sowohl aus Netzwerk- als auch aus GPS-Daten stammen. Die Daten werden von der Klasse GpsReceiver empfangen und an den GpsDatabaseConnector weitergegeben, wo sie in eine interne Datenbank geschrieben werden. Die Activity MainActivity stellt die graphische Benutzeroberfläche bereit, über der sich der GpsService starten und stoppen lässt.

Wird über die Benutzeroberfläche der Befehl zum Übertragen der Daten an den Server ausgelöst, so liest der GpsService mit Hilfe des GpsDatabaseConnector alle Datensätze aus der Datenbank, serialisiert sie nach JSON und sendet sie an den angegebenen Server, der über `new URL("http://"+ host + "/rest/accuracy/saveObject")` addressiert wird. Die Serialisierung der Daten erfolgt über flexjson [Hub10].

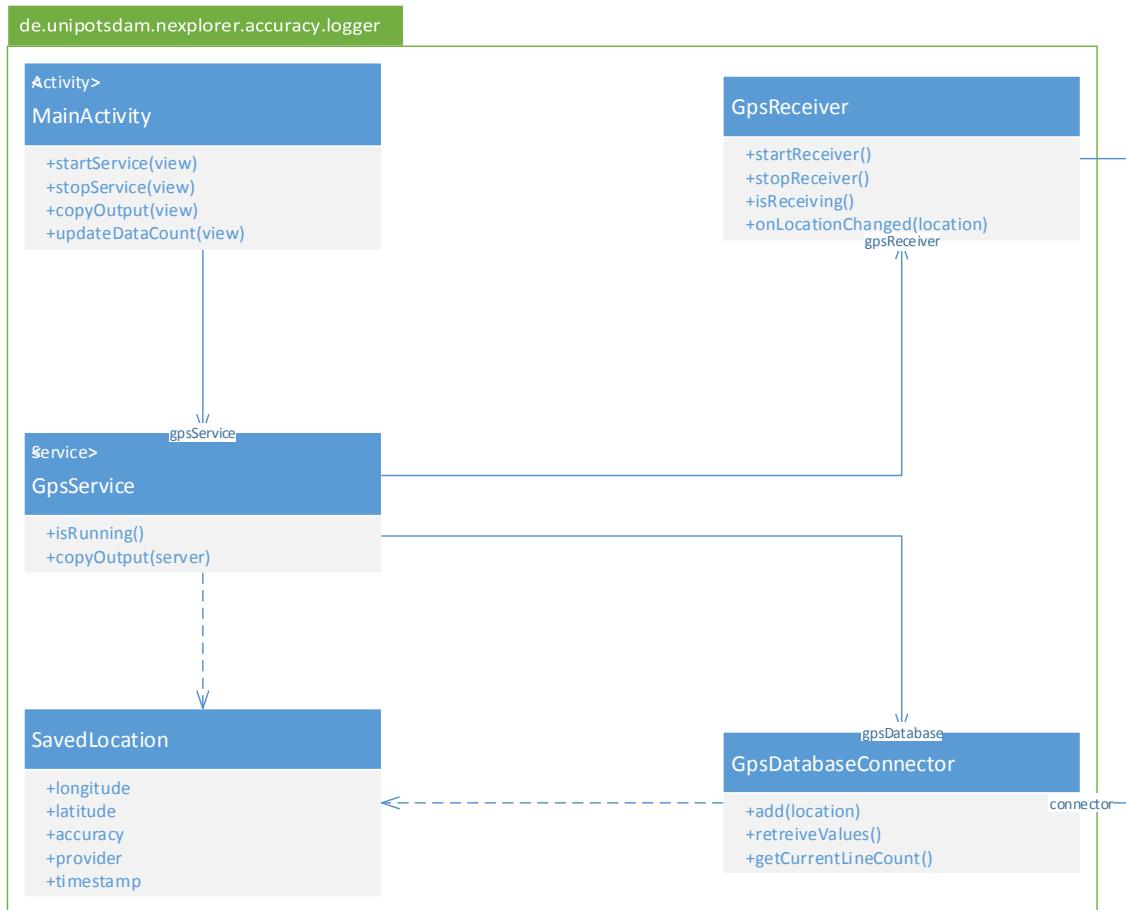


Abbildung 20: Klassendiagramm der Accuracy App

Accuracy Server Auf dem Accuracy Server werden die gesammelten GPS-Daten gesichert und können von der GPS-Map abgerufen werden.

Der Accuracy Server ist Teil des RouteMe-Servers und schreibt seine Daten in die gleiche Datenbank. Falls Hibernate so eingestellt ist, dass die Datenbank beim ersten Zugriff gelöscht wird, so werden damit auch alle in der Datenbank vorhandenen GPS-Daten gelöscht. Als Backup-Mechanismus dient das *Location log*, welches über den *Location Parser* ausgelesen wird.

Der Accuracy Server bietet nur wenige Methoden über REST an. Eine grafische Repräsentation ist in Abbildung 21 gegeben.

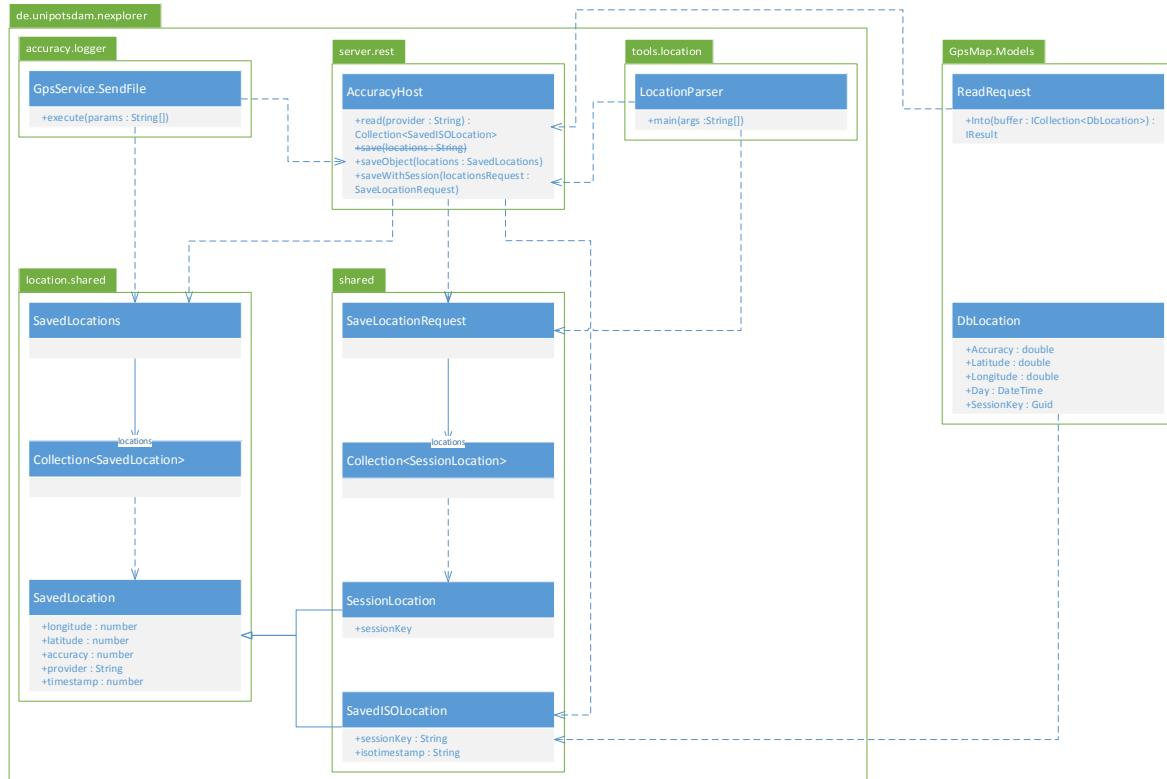


Abbildung 21: Übersicht der Schnittstellen des Accuracy Servers

1. *save*, das von der ersten App-Version genutzt wird. Diese Methode ist fehleranfällig. Außerdem werden klientseitig falsch codierte Daten als Serverfehler interpretiert. Sie ist als **deprecated** markiert. Ihr Nachfolger ist *saveObject*. Sie ruft intern *saveWithSession* auf.
2. *saveObject*, das als Ersetzung für *save* gedacht ist. Eingehende Daten werden mit einer zufälligen Session-UUID versehen und gespeichert. Sie ruft intern *saveWithSession* auf und ist für neue App-Versionen gedacht.
3. *saveWithSession*, das die gegebene Session-UUID übernimmt und die eingehenden Daten speichert. Diese Methode wird vom Location Parser genutzt.
4. *read*, das alle gespeicherten Daten eines gegebenen Providers (*gps* oder *network*) liest und zurückgibt.

Alle in `saveWithSession` eingehenden Daten (also auch über `save` und `saveObject` eingehende Daten) werden unter `logs/accuracy.log` im CSV-Format gesichert. CSV ist besonders leicht zu lesen und zu schreiben, bietet aber im Gegensatz zu XML nur flache Strukturen. Ein Auszug aus einem Log ist in Abbildung 22 zu sehen. Zu Beginn jedes Datensatzes steht eine Kopfzeile. Da alle Datensätze in die gleiche Datei geschrieben werden, existieren Header zwischen den Daten.

```
Longitude;Latitude;Accuracy;Provider;1354802702175;Session
13.1303934;52.3932995;37.0;network;1354546582426;b3d3a33f-ccdf-429f-9e7e-85d429f188cf
13.130043195560575;52.39347130525857;30.0;gps;1354551144000;b3d3a33f-ccdf-429f-9e7e-85d429f188cf
13.12915463000536;52.39335018675774;50.0;gps;1354551145000;b3d3a33f-ccdf-429f-9e7e-85d429f188cf
13.129849825054407;52.393332961946726;50.0;gps;1354551146000;b3d3a33f-ccdf-429f-9e7e-85d429f188cf
```

Abbildung 22: Auszug aus einer accuracy.log-Datei

Es hat sich als sinnvoll erwiesen, die Log-Datei in ein separates Verzeichnis (`location-logs`) zu verschieben, um alle Daten zu einem späteren Zeitpunkt mit Hilfe des Location Parser erneut einlesen zu können.

Location Parser Um die in der Log-Datei gesicherten Daten möglichst komfortabel in den Server zurückzuspielen existiert der Location Parser. Nach dem Start werden die zu lesenden Dateien ausgewählt (Mehrzahlselektion ist möglich), die anschließend an eine lokale Instanz des Accuracy Servers (`http://127.0.0.1:8080/rest/accuracy/saveWithSession`) gesendet werden. Dabei werden die vorgegebenen Session-UUIDs beachtet und Header-Zeilen gegebenenfalls übersprungen.

GPS-Map Um die gesammelten Daten möglichst intuitiv darzustellen zeigt GPS-Map die Analyseergebnisse auf einer übersichtlichen Karte. Ein Screenshot der Anwendung findet sich in Abbildung 5.

Die Anwendung ist in C# als WPF-Projekt mit Bing-Maps geschrieben. Die Entscheidung für C# fiel auf Grund der leicht einzubindenden Karte. Die Daten werden entweder von einem lokalen Accuracy Server über REST oder aus einer Datei gelesen. Die dem Projekt beigelegte Datei enthält einen Zwischenstand der gesammelten Daten vom 26.12.2012. Das Format der Datei ergibt sich aus dem Format der Serverausgabe, da die JSON-serialisierten Daten des Servers aus einer Datei gelesen werden.

Die gelesenen Dateien werden analysiert und auf der Karte dargestellt. Im Detail-Modus werden die gesammelten Datenpunkte ihrer Genauigkeiten entsprechend mit grün, gelb oder rot gezeichnet. Der Analyse-Vorgang im Übersichts-Modus ist in Abbildung 23 abgebildet. Anfangs werden die verfügbaren Datenpunkte gefiltert, da der Nutzer Daten nach Datum gruppiert abwählen kann. Anschließend werden die benötigten Kacheln ermittelt. Dabei hat sich eine Kachelgröße 0.000125 Grad als praktisch erwiesen. In Kombination mit der Definition von Längen- und Breitengraden entstehen so rechteckige Flächen. Diese Struktur bildete einen guten Kompromiss zwischen Darstellung und Programmierung.

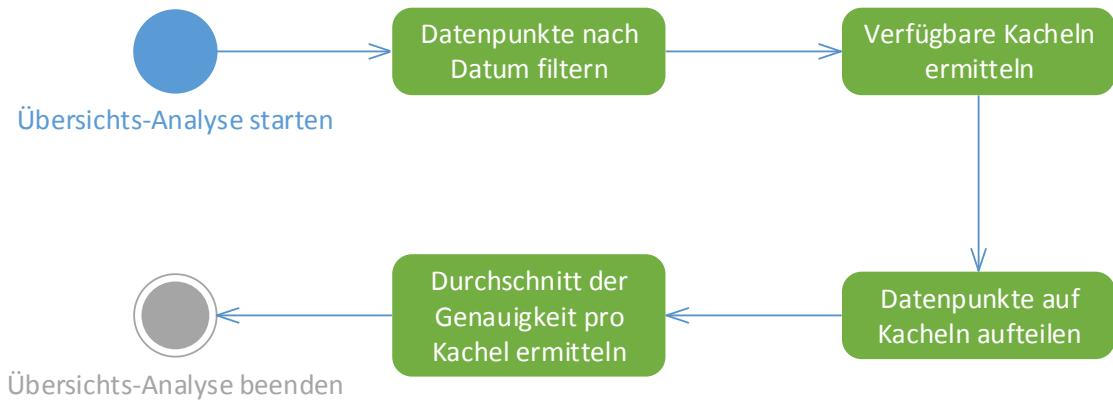


Abbildung 23: Analyse-Prozess der GPS-Map

Das Projekt ist in drei Namespaces aufgeteilt: Models, Views und ViewModels. Diese Dreiteilung heißt MVVM [Smi09] und ist typisch für WPF-basierte Anwendungen. Die Bindung zwischen Views und ViewModels übernimmt Caliburn Micro [Eis12]. Caliburn Micro setzt auf das Paradigma *Convention over Configuration*, was bedeutet, dass sich das Framework an den Klassen- und Variablennamen orientiert, um Zugehörigkeiten zu erkennen.

3.4.2 Dummy-Player

Da sich das Testen mit realen Spielerknoten als zu aufwendig erwiesen hat, ist während des Projektes ein oberflächenloser Client entstanden, der mit einer rudimentären KI ausgestattet wurde. Außerdem werden Performance-Daten auf der Konsole ausgegeben, um eine Übersicht über die Server-Performance zu erlangen. Gestartet wird sie über die Klasse `PlayerDummy` oder `DummyBatchStarter`. `DummyBatchStarter` startet dabei mehrere `PlayerDummy`-Instanzen und nimmt zu diesem Zweck auch Kommandozeilen-Parameter für Serveradresse und Anzahl an Instanzen entgegen. Ein Aufruf ohne Parameter entspricht dem Aufruf `DummyBatchStarter 127.0.0.1:8080 5`.

Lebenszyklus einer Instanz Die KI meldet sich am Server an. Aus den empfangenen Spielfelddaten werden eine zufällige Startposition und Startausrichtung ermittelt und ausgehend von dieser zufällige Bewegungen generiert, die an den Server übertragen werden. Falls sich ein Gegenstand in der Nähe befindet wird die serverseitige Einsammeln-Methode `collectItem` aufgerufen. Mit Beenden des Spieles beendet sich die Instanz.

Bewegungsmuster Um ein möglichst realistisches Bewegungsmuster durch die KI zu erreichen sind Beobachtungen aus Testspielen mit realen Spielern in den Entwurf des Bewegungsalgorithmus' eingeflossen. So war zu beobachten, dass Spieler relativ gleichmäßige Bewegungen im Schritttempo bevorzugen und ihre Laufrichtung beibehalten, wobei sie Kurskorrekturen von etwa 45 Grad nach rechts oder links vornahmen. Bei erreichen des Spielfeldrandes blieben sie kurz stehen, richteten sich neu zum Spielfeld aus und setzten ihr Bewegungsverhalten fort.

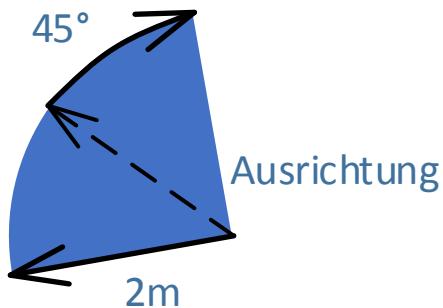


Abbildung 24: Visualisierung der Bewegungsstrategie

In Abbildung 24 ist die Bewegungsstrategie visualisiert: Ausgehen von einem aktuellen Punkt und einer aktuellen Ausrichtung wird der Folgepunkt einem einem Schwenkradius von 45 Grad nach recht oder links und einer Distanz zwischen null und zwei Metern auf Zufallsbasis ermittelt. Das bedeutet, dass der Folgepunkt innerhalb des blauen Viertelkreises liegt. Falls der Folgepunkt außerhalb des Spielfeldes liegt wird ein neuer Punkt ermittelt, für den zwar die Distanz zwischen null und zwei Metern gilt, der Schwenkradius allerdings auf 360 Grad erweitert ist.

Zwei Meter pro Sekunde sind umgerechnet etwa sieben Kilometer pro Stunde. Eine solche Geschwindigkeit entspricht einem Gehen, ohne zu rennen, was in Bezug auf die beobachteten Spielerbewegungen die Höchstgeschwindigkeit darstellt.

Einsammel-Verhalten Auch das Einsammel-Verhalten ist auf Basis intensiver Spielerbeobachtung entstanden. Dabei war auffällig, dass Spieler bei Sichtung eines Gegenstandes die *Gegenstand-einsammeln*-Schaltfläche in kurzem Abstand oft nacheinander gedrückt haben (*button spamming*). Dieses Verhalten wird von der KI simuliert, indem bei Sichtung eines Gegenstandes die Einsammeln-Methode `collectItem` fünfmal pro Sekunde aufgerufen wird.

3.4.3 Performance Parser

Der Performance-Parser liest die in `logs/performance.log` gespeicherten Daten zur benötigten Ausführungszeit von Methoden aus und schreibt sie über eine REST-Schnittstelle in die Datenbank. Dort können sie abgefragt und ausgewertet werden. Das Tool wurde in der Endphase des Projektes durch die Performance-Messungen (siehe Abschnitt 3.3.1) der Knoten-KI (siehe Abschnitt 3.4.2) abgelöst.

3.4.4 Settings Manipulator

Um optimale Parameter für das Spielen mit Studierenden zu finden, sollte es möglich sein, Parameter wie die Sendereichweite eines Knotens in einem laufenden Spiel anzupassen. Zu diesem Zweck wurde der Settings Manipulator geschrieben. Er spricht eine REST-Schnittstelle im Server an, in der übergebene Werte in der Settings-Datenbank verändert werden. Der Settings Manipulator wurde nie produktiv eingesetzt und auch nicht an die Denormalisierung der Datenbank angepasst.

3.5 Installationsanleitung

3.5.1 Entwicklungsumgebung

Im Folgenden wird beschrieben, welche Schritte zum Erhalt einer Entwicklungsumgebung erforderlich sind. Diese Anleitung ist bei der Einrichtung eines neuen Rechners entstanden und verwendet den damaligen Versionsstand. Es kann sein, dass zu einem späteren Zeitpunkt einige Versionen nicht mehr verfügbar sind. Neuere Versionen erfordern möglicherweise ein anderes Vorgehen.

Es ist wichtig, die hier aufgeführte Reihenfolge einzuhalten und Installationen sequentiell durchzuführen, da es andernfalls zu Problemen kommen kann.

Tortoise SVN Version 1.7.11 ist erhältlich unter <http://tortoisessvn.net/>. Während der Installation werden die Standardvorgaben des Wizards verwendet.

Eclipse IDE Version 4.2 (Juno) in der Edition Eclipse IDE for Java EE Developers ist erhältlich unter <http://www.eclipse.org/>. Die IDE wird nicht installiert, sondern an einen geeigneten Ort entzippt.

Android SDK Version 21.1 ist erhältlich unter <http://developer.android.com/sdk/index.html>. Hierbei wird nicht das ADT Bundle, sondern das SDK (verfügbar unter Use an existing IDE) verwendet. Während der Installation werden die Standardvorgaben des Wizards verwendet.

Über den SDK-Manager, der am Ende der Installation gestartet wird, sind folgende Optionen auszuwählen und zu installieren:

- Android SDK Platform-tools (Version 16.0.1)
- Android 4.2 (API 17)
- Android 4.0.3 (API 15)

- Android Support Library (unter Extras)
- Google USB Driver (unter Extras)

Android SDK unter Eclipse einrichten Die Anleitung ist angelehnt an <http://developer.android.com/sdk/installing/index.html>.

Eclipse starten. Unter Help -> Install New Software -> Add das Repository <https://dl-ssl.google.com/android/eclipse/> mit dem Namen ADT Plugin hinzufügen. In der geladenen Liste Developer Tools auswählen und installieren. Anschließend Eclipse neustarten.

Falls das SDK nicht automatisch gefunden wird, ist folgendermaßen vorzugehen: Android SDK Manager starten, SDK Path notieren und Android SDK Manager schließen. Eclipse starten, unter Windows -> Preferences -> Android den notierten SDK Path einfügen. Falls der Pfad korrekt eingefügt wurde, erscheinen die zuvor installierten APIs (siehe 3.5.1) in einer Tabelle.

GWT unter Eclipse einrichten Version 2.5.0 ist erhältlich unter <https://developers.google.com/web-toolkit/download>.

Eclipse starten. Unter Help -> Install New Software -> Add das Repository <http://dl.google.com/eclipse/plugin/4.2> mit dem Namen GWT Plugin hinzufügen. In der geladenen Liste folgende Optionen auswählen und installieren:

- Google Plugin for Eclipse
- GWT Designer for GPE
- Google Web Woolkit SDK 2.5.0

Anschließend Eclipse neustarten.

Visual Studio Visual Studio wird nur benötigt, wenn Änderungen an der GPS-Karte notwendig sind.

Visual Studio 2010 Ultimate ist nicht frei verfügbar. Im Folgenden wird eine installierte Version mit C#-Voreinstellungen vorausgesetzt. Studenten und Mitarbeiter der Informatik haben im Allgemeinen Zugang zu einem MANIAC-Server, über den Visual Studio 2010 Ultimate bezogen werden kann.

Visual Studio starten. Unter Tools -> Extension Manager -> Online Gallery den NuGet Package Manager suchen und installieren. Visual Studio neustarten. Unter Tools -> Options -> Package Manager -> Package Restore die Option *Allow NuGet to download missing packages during build* auswählen und mit *OK* bestätigen.

3.6 Deployment

Der System wird als WAR-Datei ausgeliefert, die auf ein geeignetes ContainerSystem installiert werden kann. In der Testphase wurde der von GWT mitgelieferte Jetty-Server genutzt.

4 Notwendige Verbesserungen und Erweiterungen

Im Laufe des Projektes haben sich viele Veränderungen gezeigt, die notwendig wären, aus Zeitgründen jedoch nicht durchgeführt werden konnten. Zu Dokumentationszwecken werden diese im Folgenden notiert.

So ist eine vollständige Portierung der Admin- und Indoor-Oberfläche sinnvoll, um die vollen Kapazitäten von GWT auszunutzen und Fehler durch Wechsel der Programmierebene zu vermeiden.

Außerdem ist die Portierung der Mobile-Oberfläche auf eine native App empfehlenswert, um von der verbesserten Unterstützung nativer Applikationen zu profitieren und auf lokale Ressourcen zugreifen zu können.

Viele Einstellungen, die derzeit im Code stehen, sollten flexibilisiert und über die Admin-Oberfläche eingestellbar werden. Dazu gehören Punktevergabekriterien, Timer-Raten und Einstellungen bezüglich der GPS-Genauigkeit.

Performance-Analysen wurden bisher nur auf Basis der Mobilknoten vorgenommen, da für diese eine KI existiert. Es ist jedoch sinnvoll, eine Indoor-KI zu entwickeln und zu implementieren, um die Performance-Eigenschaften der Indoor-Oberfläche zu ermitteln und gegebenenfalls zu korrigieren.

Die Rest-Schnittstelle ist aus dem Vorgängerprojekt übernommen und wird an vielen Stellen als JSON-RPC eingesetzt. Hier sollten die Standards von Rest-Schnittstellen herausgearbeitet und eingehalten werden, um Rest-spezifische Vorteile zu nutzen.

5 Ausblick

Im universitären Kontext haben wir uns der Herausforderung gestellt, ein Softwareprojekt von einem anderen Studenten zu übernehmen, welches nicht für längerfristige Entwicklung, sondern für einen *Proof-Of-Konzept* ausgelegt war.

Dieses Projekt hat sich zur Aufgabe gestellt, den bestehenden Code für eine längerfristige Entwicklung reif zu machen. Aber auch hier besteht noch ein großer Arbeitsbedarf. Um den Logik-Teil nach außen zu modellieren, käme eine Prozessengine wie *Activity* [Rad13] in Frage. Diese könnte die Hürde der vollständigen Dokumentation des Quellcodes senken. Insbesondere die Dokumentation ist im universitären Kontext problematisch, da studentische Projekte für die Studenten gedanklich mit dem Erwerb der Credits enden. Die Prozessengine würde die wichtigsten Teile der Serverlogik transparent machen. Alternativ wäre es plausibel, die Serverlogik auf eine funktionale Sprache wie Scala umzustellen, was gleich zwei Vorteile hätte: Der Quellcode wäre kürzer und besser lesbar und bei vielen Spielern könnte die Serverlast verteilt werden, da Scala zustandslose Programmierung fördert und damit Clustering vereinfacht.

Eine andere sinnvolle Erweiterung bestände darin, die Kommunikation der mobilen Knoten untereinander von dem Server zu entkoppeln. Hier könnten auf Javascript basierende Pushframeworks wie „Push-Fork“ helfen, die Serverlast zu drosseln und die Programmierung von Spielementen zwischen einzelnen Knoten zu verbessern.

Es fehlen nach wie vor Spielkonzepte. Bei unseren Testläufen haben die Testspieler angemerkt, dass die derzeitige GUI den Ablauf des Algorithmus, der gelernt werden soll, nicht verständlich darstellt. Hier sollten Schnittstellen entwickelt werden, die es erlauben, Spielemente und Lerninhalte hinzuzufügen, ohne den Kern der Architektur zu berühren.

Die Bachelorarbeit von Hendrik Geßner wird sich mit der Portierung der Knotenschnittstelle auf nativen Androidquellcode befassen, wodurch es einfacher wird, eine interaktive und ansprechende GUI zu entwickeln. Desweiteren werden die Spielemente ausgebaut.

6 Literaturverzeichnis

- [Eis12] Rob Eisenberg. Caliburn.Micro, 2012. <http://caliburnmicro.codeplex.com/>.
- [Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. <http://martinfowler.com/articles/injection.html>.
- [Goo13] Google. Google play, 2013. <https://play.google.com/store>.
- [Hub10] Charlie Hubbard. Flexjson, 2010. <http://flexjson.sourceforge.net/>.
- [Jul12] Julian Dehne. Evaluation pervasiver Lernspiele, 2012. <http://apache.cs.uni-potsdam.de/de/profs/ifi/mm/studentische-arbeiten/Dehne2012.pdf>.
- [LBWG11] Bob Lee, Kevin B, Jesse Wilson und Christian Gruber. google-guice, 2011. <https://code.google.com/p/google-guice/>.
- [MLZ11] Tobias Moebert, Ulrike Lucke und Raphael Zender. A Pervasive Educational Game on Pervasive Computer Network. In *Int. Conf. on E-Learning; Honolulu*, 2011.
- [New13] New Digital Group, Inc. smarty Template Engine, 2013. <http://www.smarty.net/>.
- [Ora13] Oracle Corporation. NetBeans IDE, 2013. <http://netbeans.org/>.
- [PBRD03] C. Perkins, E. Belding-Royer und S. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing*. Network Working Group, 2003. <http://www.ietf.org/rfc/rfc3561.txt>.
- [Rad13] Tijs Rademakers. Activiti BPM Platform, 2013. <http://www.activiti.org/>.
- [Red11] Red Hat, Inc. *Hibernate Community Documentation*, 2011. <http://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html>.
- [SH12] Jiri Sedlacek und Tomas Hurka. VisualVM 1.3.5, 2012. <https://visualvm.java.net/>.
- [Smi09] Josh Smith. WPF-Anwendungen mit dem Model-View-ViewModel-Entwurfsmuster. *MSDN Magazin*, Februar 2009. <http://msdn.microsoft.com/de-de/magazine/dd419663.aspx>.
- [Tea13a] H2 Team. *H2 Advanced*, 2013. http://www.h2database.com/html/advanced.html#transaction_isolation.
- [Tea13b] Jersey Team. Project Kenai, 2013. <https://jersey.java.net/>.
- [WBE⁺13] Jonathan H. Wage, Guilherme Blanco, Benjamin Eberlei, Bulat Shakirzyanov und Juozas Kaziukenas. doctrine, 2013. <http://www.doctrine-project.org/>.
- [Wen00] Ulf Wendel. PHPDoc, 2000. <http://www.phpdoc.de/>.

7 Abbildungsverzeichnis

1	Durchmischung der Programmierebenen	5
2	Beispiel für Aufruf von Javascript Methoden aus GWT	10
3	REST und XML-RPC Schnittstellen	11
4	Beispiel für REST Schnittstelle	12
5	Beispielkarte aus manuell gesammelten Daten	15
6	Logische Architektur der RouteMe-Implementierung	18
7	Datenbank-Schema	19
8	Performance-Messungen (schneller ist besser)	21
9	Toplevel-Klassendiagramm für den Klienten	24
10	Struktur der Adminviewkomponenten	25
11	Der Controller für den Adminklienten	26
12	Aktivitätsdiagramm zum Adminklienten	28
13	Klassendiagramm für den Indoorklienten	29
14	Aktivitätsdiagramm für Mobilklienten	31
15	Strukturelle Übersicht über die Schnittstellen	33
16	Erste Graphik zu den Datentransferobjekten	35
17	Zweite Graphik zu den Datentransferobjekten	36
18	Übersicht der ausführbaren Bestandteile	37
19	Screenshot der Accuracy App und der App-Einstellungen	39
20	Klassendiagramm der Accuracy App	40
21	Übersicht der Schnittstellen des Accuracy Servers	41
22	Auszug aus einer accuracy.log-Datei	42
23	Analyse-Prozess der GPS-Map	43
24	Visualierung der Bewegungsstrategie	44

