HOME
HOCHSCHULE
MERSEBURG
University of
Applied Sciences

Masterarbeit

Thema:

Konzeption und Implementation einer Softwarearchitektur anhand eines VR-Spiels für Mobilgeräte

Fachbereich: Ingenieur- und Naturwissenschaften

Abgeben von: Stefan Walesch

Matr. 21970

Abgeben am: 08.06.2019

Betreuer: Prof. Dr. Uwe Schröter

Fabian Bull

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als angegeben verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Ort:

Datum: Unterschrift

Inhaltsverzeichnis

1 Einleitung

Ein jedes Softwareprojekt startet zunächst klein, kann jedoch mit genügend Entwicklungszeit zu einem *Monstrum* an Quelldateien, Klassen, Abhängigkeiten, Skripten und Ressourcen anwachsen. Damit sich derart große Projekte sinnvoll erweitern und pflegen lassen, braucht es grundlegende Konzepte, die sich mit den Herausforderungen einer solchen Anwendung beschäftigen. Die Aufgabe der Softwarearchitektur ist es, solche Konzepte aufzuzeigen, mit dem Ziel, allgemein übliche Verfahren zu nutzen und Fallstricke zu umgehen.

Im Rahmen dieser Arbeit werden einige dieser Architekturen aufgegriffen und anhand des selbst gewählten Projekts YARPG analysiert. Da Software kein generisches Produkt ist, sondern stark durch Projektziele, Zielsysteme und den Entwickler geprägt wird, müssen diese Themen unbedingt Teil der Analyse sein. Das Projekt YARPG wird in diesem Zusammenhang so behandelt, als wäre eine Veröffentlichung geplant. Ziel ist es, das Projekt durch wohl überlegte Entscheidungen zu formen und im Rahmen einer gewählten Architektur schematisch umzusetzen. Das Ergebnis kann dabei aufgrund des enormen Umfangs keine vollständige Anwendung sein. Präsentiert wird jedoch ein Auszug der wichtigsten Softwarekomponenten, die im Anschluss an diese Masterarbeit um weitere Features ergänzt werden könnten.

Um dies zu erreichen, soll zunächst der Begriff Softwarearchitektur genauer untersucht werden, damit dessen Umfang, Ziele und Beschränkungen klar sind. Anschließend werden das Projekt und dessen Projektziele für eine theoretische Veröffentlichung vorgestellt. Ausgehend von der Spielidee wird gezeigt wie die grundlegenden Softwareentscheidungen getroffen werden. Die Diskussionsgrundlage dafür bildet das Projekt selbst sowie eine Auswahl bekannter und verbreiteter Architekturen. Letztere werden daraufhin untersucht, inwieweit sie den Anforderungen gerecht werden. Da es nicht die eine richtige Lösung für ein solch umfangreiches Projekt wie YARPG gibt, können lediglich die zu erreichenden Ziele festgelegt werden. Den Abschluss dieser Arbeit bildet eine detaillierte Auswertung. Darin wird geprüft, inwieweit diese Ziele erreicht wurden. Es wird ein Fazit gezogen und sogar ein kleiner Ausblick auf eine mögliche Weiterentwicklung gewährt.

2 Grundlagen

2.1 Softwarearchitektur

Eine allgemeingültige Definition für Softwarearchitektur gibt es seit der ersten Erwähnung bis heute noch nicht. Es lässt sich jedoch zunehmend ein Konsens erkennen, der durch folgende Bestandteile geprägt wird:

- Festlegung der Softwarekomponenten, die durch die Architektur beschrieben werden
- Abstraktionsebene und Detailgrad
- Beantwortung, was eine gute Softwarearchitektur ist

Vgl. (Posch, Gerdom, & Birken, 2012):S.15 ff.

2.1.1 Festlegung der Softwarekomponenten

Im Zuge der Softwarearchitektur wird ein Softwareprojekt in Komponenten zerlegt, welche die verschiedenen Teilbereiche der Anwendung darstellen. Der Architekt entscheidet beim Design in wie viele und welche. Beispielsweise kann es sich bei dem Projekt um das Einlesen eines Textdokuments handeln, dessen Verarbeitung und Ausgabe mit neuem Format oder neuer Formatierung. Einlesen, Verarbeitung und Ausgabe könnten drei voneinander getrennte Bereiche sein, welche über eine definierte Schnittstelle miteinander arbeiten. Die Architektur beschreibt dabei die Schnittstelle und die Aufgabe dieser Komponenten, nicht jedoch deren Innenleben. Das Aufteilen in solche Komponenten hilft, Softwareprojekte thematisch besser zu strukturieren und somit auch besser zu warten. Insbesondere bei dynamischen Entwicklungen ermöglicht dies, Teilbereiche später zu überarbeiten, zu ergänzen oder sogar gänzlich zu ersetzen. Das Ziel der Trennung nicht zusammenhängender Strukturen ist es, den Quellcode testbarer und lesbarer zu gestalten, um in Summe die Anzahl der Fehler während der Entwicklung zu reduzieren. Jede Schnittstelle stellt dabei jedoch einen zusätzlichen Entwicklungsaufwand dar. Somit ist die Festlegung der Komponentenanzahl eine Gratwanderung zwischen Arbeitsaufwand und Sinnhaftigkeit.

In der Softwarearchitektur werden solche Komponenten und Schnittstellen vom Architekten definiert und in verschiedenen Abstraktionsebenen mit entsprechendem Detailgrad dokumentiert. Ein Softwareentwickler muss mithilfe dieser Dokumentation in der Lage sein, die Software entwickeln zu können und somit auch die Anforderungen an das Projekt zu erfüllen. Vgl. (Posch, Gerdom, & Birken, 2012):S.16 ff.

2.1.2 Abstraktionsebene und Detailgrad

Im Zuge der Planungsphase entstehen diverse Dokumente, welche den Ablauf und die Architektur für alle Beteiligten möglichst verständlich beschreiben. Inhaltlich werden alle Softwareanforderungen vom Stakeholder erfasst und beschrieben.

Allgemein müssen jede Komponente und jede Schnittstelle so genau wie möglich, jedoch auch nur so umfangreich wie nötig beschrieben werden.

Für Komponenten genügt es in der Regel zu beschreiben, *was* in ihr erledigt werden soll und welche Information dafür zur Verfügung steht. Das *Wie* ist dabei dem Entwickler überlassen. Eine ausreichende Erklärung kann somit ohne Klassenbeschreibung oder Algorithmen auskommen.

Über Schnittstellen werden Datenobjekte ausgetauscht, mit deren Hilfe die Teilbereiche der Software kommunizieren und verarbeitet werden. Für diese Schnittstelle ist ein Klassendiagramm meist unumgänglich, um eindeutig zu bleiben sowie um Vorgaben für Eingabe und Ausgabe der Komponenten zu schaffen.

Handelt es sich um eine sicherheitsrelevante, schwer verständliche oder anderweitig kritische Komponente, muss der Detailgrad soweit angehoben werden, bis sichergestellt ist, dass ein ausreichendes Maß an Gewissheit vorhanden ist, um alle Softwareanforderungen zu erfüllen. In einem solchen Fall kann es sogar vorkommen, dass die Architektur Vorgaben bis hin zur Speicherebene macht.

Als Beispiel sei auf der einen Seite eine Webseite für Hotelreservierung und auf der anderen ein Notfallbremssystem eines Autos zu Projektieren. Typische Schnittstellen zwischen Komponenten der Webseite wären Kunden- und Raumdaten sowie Reservierungsvereinbarung. Diese werden durch die einzelnen Komponenten entweder bearbeitet, geprüft oder schlicht

angezeigt. Die Datengrundlage ist weitestgehend durch das Hotel und dessen Betriebsablauf vorgegeben. Die zugehörigen Komponenten entsprechen dabei hauptsächlich den Handlungen, welche ein Kunde oder Angestellter auf dieser Datengrundlage ausführen möchte, z. B. Reservierung vornehmen, stornieren oder die Rechnung der Minibar aufrufen. Bei jeder dieser Komponenten genügt es, das *Was* zu beschreiben. Das *Wie* kann dabei vernachlässigt werden, da keine zeitkritischen oder sicherheitsrelevanten Abläufe einzuhalten sind. Lediglich die Schnittstellen sind im Detail zu beschreiben.

Die Anforderung an die Reaktionszeit eines Notfallbremssystems gibt hingegen einen klar definierten Zeitrahmen vor. Das bedeutet, Laufzeit und Ablauf unterliegen festen Vorgaben, wodurch sichergestellt sein soll, dass das System rechtzeitig reagieren kann und im Zweifel Leben rettet. Es wäre daher zu erwarten genaue Details darüber in der Softwarearchitektur zu finden, also nicht nur darüber, welche Anforderung (was) eingehalten werden muss, sondern auch wie diese eingehalten werden soll.

Vgl. (Posch, Gerdom, & Birken, 2012):S.18 ff.

2.1.3 Gute Softwarearchitektur

Die Motivation für die Erarbeitung einer guten Architektur ist es,

- Risiken des Scheiterns zu minimieren,
- Software zeit- und kosteneffizient zu entwickeln sowie instand zu halten,
- eine Kommunikationsgrundlage mit den Stakeholdern aufzubauen und
- eine wiederverwendbare Basis zu schaffen.

Vgl. (Poort & Vliet, 2012)

Die Absicht einer Softwarearchitektur besteht immer darin, das Projekt umzusetzen, das heißt, die vom Stakeholder vorgegebenen Ziele inhaltlich, zeitlich und kostentechnisch zu erfüllen. Um den Erfolg des Projekts wahrscheinlich zu machen, sind Softwareeigenschaften nötig, welche die oben genannten Motivationen positiv bedienen. Eine gute Architektur besteht somit aus Entscheidungen über Aufbau und Struktur des Projektes, welche genau solche positiven Eigenschaften mit sich bringen.

Eigenschaften guter Software:

- **Funktional:** Sie ist dazu in der Lage, ihre Aufgabe zu erfüllen.
- **Robust**: Die Anwendung ist resistent gegenüber Abstürzen oder fehlerhaften Laufzeitzuständen, erkennt und behandelt Fehlerfälle entsprechend der Möglichkeiten.
- **Messbar:** Die Software kann daraufhin überprüft werden, ob sie die Aufgaben entsprechend der Bedürfnisse der Anwender erfüllt.
- ***Debuggable*:** Der Quellcode kann bezüglich Fehlerhaften Verhaltens sinnvoll untersucht werden.
- **Wartbar:** Die Software kann leicht instand gehalten werden durch konstantes Design, gute Modularisierung, gute Testabdeckung oder anderen Prinzipien, welche dabei helfen, vorhandene Funktionen nicht unwissentlich zu verändern.
- **Wiederverwendbar:** Die Software enthält für ein wiederkehrendes Problem generalisierte Lösungen und reduziert dadurch möglich Fehlerquellen.
- **Erweiterbar:** Neue oder geänderte Anforderungen können im vorhandenen Quellcode mit überschaubarem Aufwand eingearbeitet werden.
- **Sicher**: Die Anwendung ist nicht nur durch unbefugte Zugriffe von außen geschützt, sondern beinhaltet auch Mechaniken, um konsistente Datenobjekte zu gewährleisten beziehungsweise nicht jeder Programmebene Zugriff auf alles zu geben. Jede Komponente braucht nur Zugriff auf so viel Information wie für deren Aufgabe erforderlich ist.

Vgl. (McCall & Boehm, 1991) und (Stephen, 2016)

Diese Eigenschaften lassen sich zum Teil nur schwer quantisieren beziehungsweise nur am fertigen Produkt messen. Vor allem für die Eigenschaften *robust*, *wiederverwendbar* und *wartbar* kann die Architektur verschiedenste Designvorschläge unterbreiten, um letztlich eine gute Software zu gewährleisten.

2.2 Pattern (Entwurfsmuster)

Ein Pattern ist eine generalisierte Architekturlösung. Es ist kein fertiges Softwaredesign, welches direkt in Quellcode verwandelt werden kann, vielmehr beschreibt es eine bewährte Methode oder Idee, um wiederkehrende Probleme zu lösen. Der Architekt kann sich solcher

Entwurfsmuster bedienen, um das Projekt in entsprechende Teilbereiche zu zerlegen oder für einzelne Problemstellungen Lösungsansätze vorzugeben. Zum Teil lassen sich solche Entwurfsmuster in Softwarebibliotheken wiederfinden, welche in das eigene Projekt übernommen werden können. Eine solche Bibliothek kann dann Beispielsweise den Aufbau von Datenobjekten vorgeben oder bestimmte Schnittstellen erwarten, um genutzt zu werden. Es könnte sich dabei um eine Bibliothek handeln, welche sich mit der Datenbankanbindung beschäftigt und wie die Datenbank konsistent in wohldefinierter Form gelesen und beschrieben wird. Durch den Einsatz einer solchen Bibliothek lässt sich nicht nur ein funktionales Problem lösen, sondern auch ein bestimmtes Muster übernehmen, um die Funktion überhaupt nutzen zu können. Sofern diese Bibliotheken richtig eingesetzt werden, können sie die Entwicklungszeit verkürzen, indem typische Stolperfallen bereits gelöst und somit von Anfang an ausgeschlossen sind. Es kann außerdem dazu beitragen, das Verständnis der einzelnen Softwarekomponenten zu steigern, da die Entwickler mit deren typischen Aufbau vertraut sind. Zudem lässt sich die Kommunikation zwischen den beteiligten Personen verbessern, wenn auf weit verbreitetes oder bekanntes Vokabular zurückgegriffen werden kann. Häufig verwendete Lösungen als Teil einer Softwarebibliothek können zudem von einer breiteren Masse und über einen längeren Zeitraum immer weiter verbessert werden. Das macht sie letztlich auch stabiler gegenüber Einzellösungen. Vgl. (Shvets, 2014):S.6

Als Beispiel für ein Entwurfsmuster kann das *Model-View-Controller Modell (MVC)* dienen. In diesem wird die Anwendung grob in drei Teilbereiche aufgeteilt (Datenhaltung, Präsentation und Programmlogik), mit dem Ziel nicht zusammenhängende Strukturen voneinander zu trennen. Diese Aufteilung gibt dem Softwareprojekt einen grundlegenden Aufbau und damit einen groben Programmablauf. Die Aufteilung beschreibt darüber hinaus jedoch nicht wie die einzelnen Komponenten arbeiten. MVC ist somit eine Art Grobkonzept beziehungsweise eine Idee, welche sich durch das Projekt zieht. Wird die Anwendung nun in Komponenten und Schnittstellen zerlegt, sollte es sich im Rahmen dieses Konzepts bewegen. Für die einzelnen Komponenten und Schnittstellen können darüber hinaus weitere Entwurfsmuster genutzt werden, um beispielsweise die Datenobjekte und deren Verarbeitung zu beschreiben.

Ein Pattern kann unterschiedlich großen Einfluss auf die Anwendung haben. So muss es nicht wie am Beispiel des MVC Modells den gesamten Aufbau der Anwendung definieren. Es gibt Entwurfsmuster zu jedem Thema, wie etwa über die Struktur von Klassen und Methoden, den Lebenszyklus von Datenobjekten oder andere Details. Im Buch „Design Patterns: Elements of Reusable Object-Oriented Software“ (Gamma, Helm, Johnson, & Vlissides, 1994) werden

viele Ansätze und Ideen bezüglich dieses Themas diskutiert und mögliche Lösungen aufgezeigt, welche jeweils unterschiedliche Auswirkungen, Einsatzmöglichkeiten oder Begründungen haben. Viele dieser Entwurfsmuster kommen mit Vor- und Nachteilen, schließen sich gegenseitig aus oder sind perfekt kombinierbar. Manche, wie das MVC Modell, haben einen großen Einfluss auf die gesamt Softwareumsetzung, andere beschreiben lediglich den Objektlebenszyklus oder abstrakte Datentypen. Ein Softwareprodukt kann also aus vielen verschiedenen Pattern bestehen, welche passend zusammengefügt wurden oder nur das grobe Konzept beziehungsweise die Grundidee dessen aufgreifen. Auch hier gilt wieder: Es gibt nicht die eine richtige Lösung oder die perfekte Umsetzung eines Patterns, lediglich das Erreichen von Zielen.

2.3 Softwaredesign

Während Softwarearchitektur den Rahmen des Projekts beschreibt, es in Komponenten aufteilt und somit auch das Pattern vorgibt, beschreibt das Design einzelne Komponenten und deren Aufbau. Es ist somit eine tiefere Detailstufe und dafür gedacht, sich Gedanken über das „Wie“ zu machen. Dabei spielen Datentypen, Klassen und Funktionen eine zentrale Rolle. Wichtig sind hierbei auch die Vorgaben der Architektur. Diese müssen eingehalten werden, um das Konzept zu wahren und nicht gegen dieses zu arbeiten. Es bildet schlussendlich die Grundlage für die eigentliche Implementierung und beschreibt die Spezifikationen für jede Komponente. In der realen Softwareentwicklung ist der Übergang zwischen Architektur und Design fließend. Für sehr kleine Projekte ist der Umfang an Personal und Arbeitszeit oft gering, wodurch es sich nicht immer sinnvoll auf einzelne Arbeitspakete aufteilen lässt. So werden Architektur und Design gelegentlich als ein Dokument behandelt. Bei sehr großen Projekten ist es hingegen zumeist unmöglich, in nur einer Iteration von der Architektur zum Design zu gelangen. Zudem sind Anforderungen oft noch im Entscheidungsprozess und nehmen rückwirkend wieder Einfluss. So kann auch das Design die Architektur im Projektverlauf beeinflussen, wenngleich dies natürlich dem eigentlichen Sinn von Architektur und Design zuwiderläuft.

Vgl. (Koffmann & Wolfgang, 2013)

2.4 Auswahl von Architektur und Design

Um nachvollziehbare und systematisch zutreffende Entscheidung zur Architektur vornehmen zu können, ist es sinnvoll, sich Schritt für Schritt bewusst zu werden, welche Ziele erfüllt werden müssen. Dabei können Verfahren wie das *Attribute-driven design (ADD)* helfen, um Entscheidungsprozesse gezielt zu unterstützen. Bei diesem werden die gewünschten Attribute genutzt, welche einen Einfluss auf den Aufbau der Software oder das Geschäftsmodell haben könnten, um auf deren Basis die Aufgabenstellung komponentenweise zu lösen. Diese Attribute, nachfolgend als *Architecturally significant requirements (ARS)* bezeichnet, beinhalten funktionale Anforderungen, Qualitätsmerkmale beziehungsweise nichtfunktionale Anforderungen sowie alle Arten von Randbedingungen. Das Ergebnis eines ADD sind mehrere Skizzen, welche genutzt werden können, um eine vollständige Architektur zu beschreiben. Sofern dies parallel zum Entwicklungsprozess verläuft, kann das ADD auch als Dokumentation genutzt werden, die erläutert, warum welche Entscheidung getroffen wurde. Es ist somit eine Hilfestellung oder auch ein Leitfaden, den Fokus auf die wichtigen Aspekte der Software legen zu können.

ADD ist in vier sich solange wiederholende Schritte unterteilt bis das Softwareprojekt als solches vollständig erfasst beziehungsweise bis alle Anforderungen erfüllt wurden. Dabei wird in Schritt eins jeweils eine noch nicht betrachtete Komponente gewählt, welche als nächstes analysiert werden soll oder, sofern es der erste Durchlauf ist, das System beziehungsweise der Projektrahmen als Ganzes. In Schritt zwei werden alle relevanten ARS für dieses Element identifiziert und priorisiert. Dabei wird zwischen technischer- und geschäftlicher Ebene unterschieden und deren Einfluss von niedrig über mittel bis hoch bewertet. Schritt drei stellt das Herzstück des ADD-Prozesses dar. Architektur und Design werden darin so ausgesucht, dass die wichtigsten ARS für die gewählte Komponente erfüllt werden. Da oft nicht alle Anforderungen vollständig erfüllt werden können, muss eine Entscheidung zugunsten der wichtigsten getroffen werden. Handelt es sich bei den zu betrachtenden Architekturen um altbekannte Modelle, kommen diese meist mit Vor- und Nachteilen einher, anhand derer Entscheidungen getroffen werden können. Wird ein völlig neues Konzept gewählt, ist es wichtig, dieses vorab im Rahmen der Möglichkeiten zu testen und zu bewerten. Wie in der Einführung beschrieben, sollte möglichst auf eigene Architekturen verzichtet werden, da sie mit einem enormen Mehraufwand verbunden sind und Schwächen unter Umständen erst sehr spät in der Entwicklung oder sogar erst in der Produktion auffallen. Schlussendlich wird die Komponente anhand der gewählten Struktur initialisiert, das heißt, die Vorgaben der Architektur werden anhand des Projekts umgesetzt. Anschließend wird getestet, ob die Ziele erfüllt sind.

Im vierten und damit letzten Schritt wird ein Teil der übrig gebliebenen Anforderungen beziehungsweise Komponenten ausgewählt und in Schritt eins erneut betrachtet. Sind alle Anforderungen erfüllt, endet hier der ADD-Prozess. Vgl. (Professor Gielen, 2018)

Andere Varianten der Entwicklung sind beispielsweise *Material-driven design (MDD)*, *Behavior-driven development (BDD)*, *Test-driven development (TDD)* und *Process-driven development (PDD)*. Diese Ansätze legen den Entscheidungsprozess jeweils auf einen anderen Schwerpunkt und haben dementsprechend einen ähnlichen Aufbau. MDD fokussiert sich auf einen Werkstoff und deren Eigenschaften. PDD auf den Geschäftsprozess selbst und dessen Verlauf. Beide Varianten sind im Zusammenhang mit dieser Arbeit nicht von Interesse, da weder physische Komponenten geplant sind noch ein Prozess als solcher erfüllt werden muss.

BDD und TDD stellen das Verhalten und Testen einer Anwendung in den Vordergrund und werden meist kombiniert eingesetzt. So werden die gewünschten Eigenschaften mithilfe von Tests abgebildet und vor der Umsetzung der eigentlichen Softwarelösung geschrieben. Anhand dieser Tests kann dann entwickelt werden und zu jedem Zeitpunkt überprüft werden welche Eigenschaften erfüllt wurden.

Auch der ADD-Prozess beinhaltet das Testen einer Anwendung, jedoch werden die gewünschten Eigenschaften nach der Implementierung anhand der Architekturentscheidung überprüft und nicht umgekehrt. Wie so oft kann es jedoch auch hier eine Vermischung geben. BDD und TDD gliedern sich in eine existierende Softwareumgebung gut ein, sofern diese um einzelne Komponenten erweitert werden soll. Sie eignen sich jedoch weniger gut für das grundlegende Entwerfen einer neuen Anwendung, da sich kaum ein sinnvoller Test schreiben lässt, wenn noch nichts vorhanden ist. Somit kann es sinnvoll sein, beispielsweise mit ADD zu beginnen, um die Software grundlegend zu designen und im Verlauf eines dynamisch entwickelten Projekts den Prozess entsprechend zu ändern. Diese Arbeit konzentriert sich auf den ADD-Prozess, um mittels der ASR das Thema Softwarearchitektur zu diskutieren.

3 AR-Spiel für Mobilgeräte (YARPG)

3.1 Augmented Reality (AR)

Um die theoretischen Ansätze zu verdeutlichen und zu testen, soll das in der Einleitung bereits erwähnte Beispielprojekt YARPG genutzt werden. Die Idee zu diesem Projekt entstand ursprünglich im Rahmen einer Veranstaltung der Hochschule Merseburg. YARPG steht dabei für *Yet Another Rollplay Game* und verrät augenzwinkernd, dass die zugrundeliegende Spielidee nicht wirklich neu ist. In ausgereifter Form wäre für YARPG jedoch durchaus ein potenzieller Markt vorhanden.

Nachfolgend wird die Spielidee beschrieben und daraufhin eine Anforderungsanalyse durchgeführt. Dabei wird nur so weit ins Detail gegangen, wie es für eine Architekturanalyse notwendig ist. Ziel ist es nicht, am Ende der Arbeit ein fertiges oder marktfähiges Produkt erstellt zu haben, sondern prototypisch zu verdeutlichen wie das Projekt umgesetzt werden kann. Darüber hinaus sollen mögliche Vor- und Nachteile bestimmter Softwareentscheidungen aufgezeigt werden.

Der Clou von YARPG liegt darin, die virtuelle mit der realen Welt zu verbinden. Möglich machen das die GPS-Koordinaten, die das Mobilgerät erfasst. Diese werden genutzt, um die Position der Spielfigur zu definieren. Bewegt sich der Spieler beispielsweise um 100 Meter nach Norden, bewegt sich gleichsam seine Figur in der virtuellen Welt nach Norden. Damit das Erlebnis möglichst realistisch ist, müssen Dimensionen und Verkehrswege der virtuellen Welt identisch zur realen Welt sein. Für eine spannende Handlung muss die virtuelle Karte jedoch auch digitale Elemente enthalten, die es in der realen Welt nicht gibt.

Diese virtuelle Erweiterung der Realität nennt sich *Augmented Reality (AR)*. Der Begriff wurde erstmals 1990 vom Boing-Forscher Thomas Caudell verwendet. Er beschrieb damit eine Visualisierungstechnik der Elektriker, die sich mit komplizierten Verkabelungen beschäftigten. Vgl. (Rouse, 2016) Moderne Anwendungen sind beispielsweise ein Head-up-Display im Auto, das die aktuell geltenden Verkehrsregeln einblendet oder die Google Glass, welche Information zu Gebäuden anzeigt, die gerade betrachtet werden. Damit relevante Information angezeigt werden kann, ist die relative Position des Nutzers entscheidend. Diese kann beispielsweise mittels GPS oder Kameras ermittelt werden.

AR-Spiele wie YARPG geben dem Nutzer das Gefühl, die Spielfläche sei reale die Welt, auch wenn es lediglich eine Abbildung dieser ist. Bekannteste Vertreter solcher Spiele sind *Ingress* und *PokemonGo* des US-amerikanischen Entwicklerstudios *Niantic, Inc*. Beide erfreuen sich bis heute größter Beliebtheit. Für die Entwicklung von *PokemonGo* benötigte *Niantic, Inc.* übrigens mehrere Jahre. Allein von *Nintendo* erhielt das Entwicklerstudio 30 Millionen Dollar Förderung und das, obwohl *PokemonGo* eine nahezu identische Spielmechanik wie sein inoffizieller Vorgänger *Ingress* aufweist. Vgl. (Takahashi, 2018) Daran lässt sich ermessen wie aufwändig ein solches Projekt werden kann. Im Rahmen des praktischen Teils dieser Arbeit müssen daher zwangsläufig Abstriche gemacht werden, selbst für einen Prototyp.

3.2 Aufbau und Grundidee

Im AR-Spiel YARPG übernimmt der Spieler, beziehungsweise die Spielfigur, die Kontrolle über eine Abenteurergruppe. Die Spielfigur zieht dabei durch eine *High-Fantasy-Welt* und erledigt Aufgaben. Die virtuelle Welt ist durch ein idealisiertes europäisches Mittelalter geprägt, das zusätzlich um Fabelwesen, Magie sowie eine alternative Geschichte und Religion erweitert wird. Solche Inhalte sind jedoch nur dann für die Architektur relevant, wenn sie einen Einfluss auf softwaretechnische Entscheidungen haben. Die Details der Spielwelt sind daher lediglich im Anhang A thematisiert. Wichtiger sind dagegen die visuelle Präsentation und die Benutzerinteraktion, welche nachfolgend beschrieben werden.

Nach der Anmeldung mit Benutzername und Passwort erscheint im Hintergrund bildschirmfüllend eine Karte. Bewohnte Gebiete und Straßen sind auf dieser in unterschiedlichen Farben abgebildet, ähnlich wie in einem Navigationssystem. Die Ansicht ist auf einen farbigen Punkt zentriert, der die Spielfigur sowie die aktuelle GPS-Position des Spielers beschreibt. Die angezeigten farbigen Flächen entsprechen einer zweidimensionalen Abbildung der an dieser Stelle real existierenden Verkehrswege und Gebäude in einer für das Spiel ausreichenden Genauigkeit. Teilweise verdeckt wird diese Karte von verschiedenen Grafikelementen, welche spielrelevante Information bezüglich der Spielfigur oder andere Inhalte anzeigt. Einige der angezeigten Elemente können durch ein Fingertippen Bildschirmfüllend expandiert werden, um entweder noch umfangreichere Spieldaten oder ein Optionsmenü anzeigen zu können.

Auf der Karte selbst werden, neben Straßen und Gebäuden, Spielelemente mittels verschiedener Symbole angezeigt. Die Symbole sind georeferenzierte Objekte, an die Handlungsanweisungen geknüpft sind, welche unter bestimmten Bedingungen ausgelöst werden. Die

Symbole repräsentieren somit die verschiedenen virtuellen Aspekte des Spiels. Sobald sich der Spieler in dessen Nähe aufhält, kann er mit diesem Teil der virtuellen Welt interagieren. Die Spielfigur umgibt zu diesem Zweck ein Kreis. Alles innerhalb dieses Kreises befindet sich in Interaktionsreichweite, alles außerhalb nicht.

Die Position und somit auch die Bewegung der Spielfigur ist an die GPS-Koordinaten des Mobilgeräts gekoppelt, auf dem das Spiel ausgeführt wird. Der Spieler muss das Mobilgerät daher physisch an eine andere Stelle bewegen, um in gleichem Maße die Figur zu bewegen. Auf diese Art kann sich der Spieler durch die virtuelle Welt bewegen und die beschriebenen Symbole ansteuern.

Befindet sich ein Symbol in Interaktionsreichweite, kann es entweder durch Antippen aktiviert werden oder aktiviert sich selbstständig. Entsprechend der Spielmechanik öffnet sich ein Fenster, das den Spieler zum Handeln auffordert. Abhängig vom Ergebnis der Handlung führt dies zu einer Veränderung von gespeicherter Information über das Symbol und den beteiligten Spieler. Eine solche Handlung kann unter anderem ein Kampf, die Erfüllung einer gestellten Aufgabe, Handel mit anderen Spielern oder der Ausbau der Siedlung sein. Weitere Details zu diesen Spielmechaniken sind in Anlage A erläutert.

Ziel des Spiels ist es, Erfahrungspunkte zu sammeln, Aufgaben zu erfüllen sowie Güter herzustellen und diese zu handeln. Es handelt sich somit um ein fortlaufendes Spielprinzip mit Interaktionsmöglichkeiten zwischen verschiedenen Spielern. Je nach Spielmechanik beeinflusst der Spieler durch seine Handlungen manche Symbole in seiner Umgebung. So kann das Erfüllen einer Aufgabe dazu führen, dass ein solches Symbol nicht mehr angezeigt wird oder sich zu einem anderen Symbol verändert. Dabei werden jedem Spieler nur diejenigen Symbole angezeigt, welche er gemäß Handlung und Spielmechanik sehen darf. Per Zufall können aber auch unsichtbare Symbole aktiviert werden, also dann, wenn sich der Spieler zufällig am richtigen Ort befindet.

3.3 Auswahl der Spielelemente für die Softwareanalyse

Wie zuvor bereits erwähnt dient das Spiel YARPG lediglich als Grundlage, die Softwarearchitektur zu untersuchen und soll im Zuge der Arbeit nicht vollständig umgesetzt werden. Damit das Spiel jedoch in Grundzügen erkennbar ist, wurden einige Kernelemente des Spiels ausgewählt und umgesetzt, darunter das Tracking und Anzeigen der Spielfigur sowie die Interaktion mit den Symbolen auf der Karte. Für diese Elemente braucht es zumindest einen Datensatz mit Symbolen und natürlich ein Mobilgerät mit GPS-Empfänger. Ebenfalls von Interesse ist die Spielmechanik des Erzählers, der zufällig Symbole mitsamt Inhalt und Aufgaben auf der Karte erzeugt. Der Erzähler ist die treibende Kraft, um die virtuelle Welt möglichst dynamisch und spannend zu gestalten. Softwaretechnisch verbirgt sich dahinter ein Service, der zeitlich oder eventgesteuert Objekte erzeugt, verändert oder löscht.

Wie beschrieben handelt es sich bei YARPG um ein AR-Spiel, das die reale Welt als Abbildung nutzt. Die reale Welt bestimmt somit auch die Ausmaße des Spielfelds. Im Ergebnis bedeutet das viel Platz für viele Symbole und potentiell viele Spieler, die gleichzeitig mit ihrer virtuellen Umgebung interagieren wollen. Softwaretechnisch bedeutet dies sehr viele Datensätze und potenziell viele Datenbankzugriffe, welche möglichst verteilt abgearbeitet werden müssen.

3.3.1 Interaktive Karte

Wie in Kapitel *3.2 Aufbau und Grundidee* beschrieben, bestimmt die Karte grundlegend die Optik des Spiels und beinhaltet den Spieler sowie die Symbole mit den Interaktionsmöglichkeiten. Ein solches Symbol kann beispielsweise die Darstellung einer Siedlung sein, ein Fähnchen oder ein Pin. Verantwortlich für die Symbole und deren Darstellungen ist ein Softwareobjekt, das an verschiedene Spielelemente gekoppelt sein kann. Weitere Details dazu sind im Anhang beschrieben. Grundlegend wird dabei in Sichtbarkeit und Aktivierungsmechanik unterschieden. Die Sichtbarkeit beschreibt, ob der Spieler die Symbole auf der Karte sehen kann. Dies kann durch verschiedene Spielmechaniken beeinflusst werden, so zum Beispiel durch die Fähigkeiten oder das Level der Spielfigur. Die Aktivierungsmechanik ist entweder aktiv oder passiv. Passive Symbole warten auf den Spieler, bis dieser die Interaktion beginnt. Ein typisches Beispiel dafür sind die Siedlungen, deren Interaktionsmenü sich

erst dann öffnet, wenn der Spieler die Siedlung antippt. Aktive Symbole initiieren hingegen bereits eine Interaktion, sobald der Spieler in Reichweite ist. Ein unsichtbares aktives Symbol kann beispielsweise eine *Monstergruppe* sein, die im Falle einer Begegnung einen Kampf auslöst. Unsichtbare passive Symbole sind im Unterschied dazu nicht zu erreichen, weil die Spielfigur die dafür notwendigen Voraussetzungen noch nicht erfüllt. Jedes Symbol ist dabei georeferenziert. Durch die Übermittlung der GPS-Daten vom Client zum Server werden entsprechende Symbole generiert.

3.3.2 Erzähler

Der Erzähler steuert den Ablauf des Spiels. Er informiert den Spieler, wenn ein neues Ereignis eintritt oder ein Symbol verschwindet. Dabei wird zwischen zufälligen und festen Ereignissen unterschieden. Für zufällige Ereignisse werden beispielsweise Monster oder Höhlen erzeugt, die der Spieler finden kann, indem er sich auf der Karte bewegt. Solche zufälligen Ereignisse sind lediglich für eine beschränkte Zeit aktiv und werden danach wieder gelöscht. Somit erfahren alle Spieler, die sich zur selben Zeit am selben Ort befinden dasselbe Zufallsereignis. Die Ereignisse lassen sich jedoch auch an das bisher erreichte Level des Spielers knüpfen und im Sinne eines facettenreicheren Spielerlebnisses variieren. Ein festes Ereignis ist hingegen eine vom Spielfluss vorgegebene Situation, welche entweder aus einer Handlung folgt oder ab einem bestimmten Zeitpunkt eintritt. Feste Ereignisse sind zum Beispiel Aufgaben, die von einem *Nicht-Spieler-Charakter (NSC)* vergeben wurden, Tutorials oder globale Ereignisse, die von den Administratoren des Spiels gestartet wurden. Diese festen Ereignisse sind nicht zwingend an einen bestimmten Ort oder eine Zeit gebunden. Für deren Eintreten müssen stattdessen bestimmte Bedingungen beziehungsweise Aufgaben erfüllt sein. Somit können mehrere Spieler unabhängig von ihrer Position oder des Zeitpunkts das gleiche Ereignis erhalten.

3.4 Projekt spezifische Anforderungen und Ziele

Neben den Spielinhalten sind auch die Projektziele entscheidend für die Auswahl einer geeigneten Softwarearchitektur. So ist beispielsweise das Zielsystem eine vom Entwickler getroffene Entscheidung, die sich auf zu verwendende Technologien oder deren Beschränkung auswirken kann. Entscheidungskriterien sind in diesem Zusammenhang häufig die Ziel-

gruppe, der Ressourceneinsatz und eine mögliche Monetarisierung. Damit auch solche Kriterien in die Betrachtung einfließen können, werden nachfolgend die wichtigsten Projektziele festgelegt.

Ziele des Entwicklers

- Schaffung eines marktfähigen AR-Spiels, das über entsprechende App-Stores an Mobilegeräte vertrieben werden kann
- Mögliche Monetarisierung durch Werbung und/oder Premium-Abonnements, die Unterhaltungskosten decken beziehungsweise eine Gewinnabschöpfung ermöglichen
- Anbieten einer offenen Schnittstelle für die Spielergemeinschaft, um Eigenentwicklungen oder Erweiterungen zu unterstützen und um langfristig eine treue Spielergemeinschaft zu erschaffen
- Muss aufgrund der Einzelarbeit und der fehlenden Finanzierung im Rahmen eines Hobbyprojekts umsetzbar sein, das heißt, es soll ohne große Eigenentwicklung und möglichst ohne Zukauf von teuren Software- oder Hardwarelösungen umgesetzt werden

Zielsystem Mobilgeräte

- Das Spiel soll GPS-fähige Mobilgeräte mit den Betriebssystemen *Android*, *Windows Phone* und *iOS* sowie alle gängigen Browser unterstützen, um eine möglichst große Nutzergruppe zu erreichen
- Möglichst geringe Datenübertragung
- Anpassung an unterschiedliche Bildschirmgrößen
- Gutes Rückmeldeverhalten der Anwendung

4 Planen einzelner Komponenten mittels ADD

Wie in den Grundlagen beschrieben sollen mittels *Attribute-driven design (ADD)* einzelne Komponenten des Projekts ausgewählt und anhand ihrer *Architecturally significant requirements (ARS)* Diskussionen um mögliche Softwarelösungen geführt werden. Die erste Komponente ist dabei immer das System selbst.

4.1 System

4.1.1 ARS des Systems

Die wichtigsten ARS für das System lauten wie folgt:

- Testbarkeit
- Robustheit
- Erweiterbarkeit bezüglich aktueller und zukünftiger Spielelemente
- Sicherheit bezüglich der Spielerfahrung (Betrug erschweren)
- Anbieten einer Schnittstelle für alternative Oberflächen
- Skalierbarkeit bezüglich der Nutzerzahl
- Mobilgeräte mit den Betriebssystemen *Android*, *iOS* und *Windows Phone* als Hauptplattform für das Spiel
- Browser auf dem PC als alternative Plattform zur Unterstützung der Spielerfahrung (für Spielelemente die kein GPS benötigen)
- Möglichst geringer Entwicklungsaufwand
- Möglichst geringe Entwicklungskosten
- Möglichst gute Lesbarkeit des Quellcodes durch Einsatz bekannter Architekturen

4.1.2 Diskussion möglicher Architekturen

Das System umfasst den Rahmen des Projekts, welches die Darstellung des Spiels mit all seinen Elementen sowie die Infrastruktur für die Datenerfassung, Kommunikation und Verarbeitung beinhaltet. Als erstes muss entschieden werden, wie die Anwendung generell aufgebaut sein soll. Laut Beschreibung handelt es sich um ein Spiel mit Mehrspielermodus, welches eine Kommunikation zwischen den einzelnen Benutzern voraussetzt. Außerdem soll ein zentralisiertes Ereignissystem (Erzähler) die Spielwelt mit Aufgaben versorgen. Laut Projektziel soll zudem eine Schnittstelle geschaffen werden, mit der die Gemeinschaft individualisierte Darstellungen entwickeln kann. Damit ist vorgegeben, dass mindestens das Ereignissystem für alle Benutzer erreichbar gemacht wird und die Spiellogik in irgendeiner Form für die Entwicklung bereitsteht. Damit Spieler miteinander kommunizieren können, ist es außerdem erforderlich, Kontaktinformation auszutauschen. Es muss also eine geteilte Schnittstelle geben, beispielsweise als Server. Eine offensichtliche Lösung für diese Problemstellung wäre es, Darstellung und Verarbeitung voneinander zu trennen und so die Spiellogik über einen Server für alle erreichbar zu machen. So kann die mobile Anwendung über die gegebene Schnittstelle Information abrufen und austauschen. Dies ist jedoch nicht die einzig mögliche Lösung.

Es könnte auch ein Softwarepaket entwickelt werden, welches die Daten auf den Zielsystemen selbst verarbeitet und speichert, also direkt auf den Mobilgeräten. Lediglich für den Informationsaustausch zwischen den Geräten und für den Erzähler bräuchte es für die erste Kontaktaufnahme eine Lösung, Direktverbindungen aufzubauen. Beides könnte relativ einfach mittels RSS-Feed, Mailingliste, Twitter-Account oder ähnliches erreicht werden. Alle Benutzer müssten sich dort initial einmal melden und wären dann für alle Geräte bekannt. Darüber hinaus kann diese Form der Kontaktaufnahme genutzt werden, um ein Ereignis des Erzählers für alle erreichbar zu machen. Vorteil einer solchen Lösung wäre es, keine großen Server unterhalten zu müssen, da die Anwendung verteilt auf den Mobilgeräten der Benutzer läuft. Lediglich für den Erzähler müsste eine für alle erreichbare Lösung angeboten werden, um eine zentral gesteuerte, fortlaufende Geschichte aufbauen zu können. Da eine solcher Erzähler hauptsächlich Information sendet und nur wenig Information empfängt beziehungsweise verarbeitet, würde ein Server mit eher geringer Verarbeitungsleistung genügen. Allerdings birgt dieses Design zugleich große Sicherheitslücken, denn das Mehrspielerspiel läuft lediglich auf dem eigenen Mobilgerät. Daher können Daten nahezu beliebig manipuliert werden. So könnten beispielsweise Ereignisse des Erzählers gefälscht oder direkt die gespeicherten Daten geändert werden. Da ein solches Spiel gerade durch seine Interaktion mit

anderen Spielern besticht, können selbst Einzelfälle von Betrug zu einer schlechten Spielerfahrung führen. Somit müssten die Daten verschlüsselt oder anderweitig unzugänglich gemacht werden und darüber hinaus bräuchte es Erzählerereignisse, die nicht zu fälschen sind, also zumindest eine Art Sicherheitsschlüssel besitzen.

Eine aufwendige Verschlüsselung und darüber hinaus die Prüfung auf Richtigkeit nach einem Datenaustausch sind jedoch keine trivialen Angelegenheiten. Falls nicht zwingend notwendig, sollte dies daher vermieden werden. Die beste Variante, um Datenmanipulation effektiv verhindern zu können, ist es, Nutzern den Zugriff von vornherein zu verwehren. Sind Daten und Verarbeitung auf einem Server, kann der Nutzer am ehesten keinen Einfluss darauf nehmen, außer er dringt direkt in das System ein oder findet anderweitig Lücken. Mittels Server ließe sich zudem der Migrationsprozess von einer Datenversion zu nächsten zentral realisieren. Der Migrationsprozess müsste also nicht nach und nach auf die verteilten Daten angewendet werden und wäre dadurch wiederum weniger fehleranfällig.

Da die Vorteile im Sinne der ARS überwiegen, fällt die Entscheidung zugunsten einer Trennung zwischen Darstellung auf einem Mobilgerät oder PC und Datenspeicherung sowie Verarbeitung auf einem geschützten Server. Diese recht grundlegende Aufteilung in Datenspeicherung, Darstellung und Verarbeitung wird in der Softwarearchitektur gemeinhin als *Model View Controller*, kurz MVC, beschrieben.

Das auf dem Smalltalk-76 erstmals erwähnte Modell *Vgl.* (Reenskaug, The Smalltalk-76 Programming System Design and Implementation, 1979) wurde über die Jahre hinweg immer wieder überarbeitet, denn einige der damaligen Vorstellungen haben sich durch neue Technologien erübrigt oder wurden an sie angepasst. Somit entstanden in den 40 Jahren, in denen das Modell im Einsatz ist, nicht nur neuere Versionen, sondern auch solche, die an bestimmte Problemstellungen angepasst wurden. Zum Teil treten unterschiedliche Vorgehensweisen dieses Modells unter der gleichen Bezeichnung auf. Das erschwert es, die verschiedenen Versionen zu unterscheiden. Um die Übersicht zu gewährleisten, soll daher zunächst Klassisches MVC und anschließend in aktuellen Versionen die Modelle MVP, MVVM und Clean Architecture für den Einsatz am Projekt YARPG überprüft werden.

4.1.2.1 Klassischer Model View Controller (MVC)

Die ursprüngliche Idee hinter MVC war die Realisierung einer Anwendung mit Oberfläche. Die Entwickler entwarfen dafür inhaltlich voneinander getrennte Verantwortungsbereiche und definierten zwischen diesen entsprechende Arbeitswege. Dadurch sollte nicht nur die Portierbarkeit unterstützt werden, sondern auch die Pflege des Softwareprodukts und seine Testbarkeit. *Vgl.* (Reenskaug, The Smalltalk-76 Programming System Design and Implementation, 1979) Die so konzipierten Bestandteile Model, View und Controller haben dabei folgende Aufgaben:

Model

Das Model ist gemeinhin die Datenhaltung des jeweiligen Softwareprodukts. Dabei geht es jedoch nicht nur zwangsläufig um Dateien, Datenbanken oder ähnliches, sondern auch um die Repräsentation der Daten als Objekt in der Informatik. In der objektorientierten Softwareentwicklung handelt es sich dabei um Klassen mit Attributen, welche zumeist die Datenhaltung widerspiegelt, also eine Abstraktion dieser ist. Ergänzt werden solche Klassen mit Methoden, um Manipulationen der Daten zu realisieren, Berechnungen auf Basis des Inhalts auszuführen oder Kombinationen zu erstellen.

Beispiel:

Gegeben sei das Model *Person* – bestehend aus Name, Alter, Geschlecht, Anschrift und Beruf. Die Daten sind dabei in einer SQL-Datenbank als Tabellen gespeichert. Die in der Software entwickelte Klasse besteht aus den Attributen einer solchen Person. Wird nun der Datensatz zur Person Max Mustermann benötigt, wird eine Instanz der Klasse *Person* erzeugt, indem eine Transaktion auf der Datenbank ausgeführt wird, um den entsprechenden Tabelleneintrag zu laden. Das so erzeugte Objekt kann nun vom Programm entsprechend der Informatik verwendet werden. Änderungen der Attribute müssen jedoch zurück in die Datenbank geschrieben werden, um inkonsistente Zustände zu vermeiden. Das heißt: Es muss wiederum eine Transaktion auf der Datenbank ausgeführt werden, um den Tabelleneintrag entsprechend des Objekts zu ändern. Auf diese Weise ist das Datenobjekt an die Datenbank gekoppelt. Solche Klassen können dabei auch Geschäftslogik beinhalten. Sollte es beispielsweise beim Erzeugen eines neuen Objekts *Person* ein Mindestalter geben, kann dieses bereits mit geprüft werden, bevor der Datensatz in die Datenbank geschrieben wird.

Ein solches Datenobjekt muss dabei nicht zwangsläufig die genaue Abbildung eines Tabelleneintrags sein, sondern kann aus verschiedenen Tabelleneinträgen, Datenquellen oder sogar hierarchischen Objekten bestehen. Das Model in der MVC-Architektur ist also die Summe von Objekten und Methoden (z. B. Geschäftslogik), die an eine Datenhaltung gekoppelt sind. *Vgl.* (Nolan, 2016)

View

Der View-Programmteil ist verantwortlich für die Anzeige und die Benutzereingaben. Es spielt dabei keine Rolle, ob es sich um eine Terminalausgabe oder eine grafische Oberfläche handelt. Der Nutzer sendet in beiden Fällen eine Anfrage an das Programm, welche zur Anzeige gebracht werden soll. Für das Model *Person* kann das beispielsweise eine Liste mit allen Personen sein. Dafür braucht der View-Programmteil die Datenobjekte aus dem Model, welche vom Controller bereitgestellt werden. Somit muss sich der View-Programmteil nicht damit beschäftigen wie Datenobjekte initialisiert werden. Die so erhaltene Liste mit Personenobjekten kann dann entsprechend der Informatik genutzt werden, um eine Anzeige mit relevantem Inhalt zu füllen. Soll nun der Inhalt eines Tabelleneintrags geändert werden, kann dies über das Datenobjekt selbst geschehen, vorausgesetzt es gibt dafür eine Methode, die eine Transaktion auf der Datenhaltung auslöst.

Der View-Programmteil nutzt somit den Controller, um Datensätze aus der Datenhaltung zu erlangen und arbeitet anschließend auf den Objekten des Models selbst, um Einträge zu ändern oder aus deren Kombination weitere Information zu erstellen. Des Weiteren verarbeitet der View-Programmteil alle Benutzereingaben und Fehlermeldungen, die vom Model oder Controller geliefert werden. *Vgl.* (Reenskaug, THING-MODEL-VIEW-EDITOR an Example from a planningsystem, 1979)

Controller

Der Controller hat die Aufgabe, Objekte des Models zu laden und für den View-Programmteil bereit zu stellen, das bedeutet, für jede Aufgabe (Features), die das Programm haben soll, müssen Methoden bereitstehen.

Es sei wieder das im Abschnitt *Model* beschriebene Beispiel *Person* gegeben. Soll nun der View-Programmteil eine Liste an Namen anzeigen, die jünger als 18 sind, muss es im Controller eine Methode geben, die eine Liste an Personen mit diesem Parameter zurückgibt.

Dabei könnte es sich um eine SQL-Anfrage mit entsprechender *Where*-Bedingung handeln. Der Controller lädt somit alle gesuchten Tabelleneinträge und initialisiert die Datenobjekte. Diese werden wiederum als Liste an den View-Programmteil weitergereicht, der sie zur Anzeige bringt.

Es ist auch möglich, dass ein Controller keine Datenobjekte des Models herausgibt, zum Beispiel wenn es sich um das Ergebnis einer Berechnung handelt. Wird die Summe aller unter 18-Jährigen abgefragt, muss nicht zwangsläufig jeder Tabelleneintrag dafür initialisiert werden. Außerdem kann es Methoden geben, um neue Datensätze zu erstellen. In diesem Fall werden alle notwendigen Parameter eines Datenobjekts gesetzt und in einer Transaktion auf die Datenhaltung übertragen.

In der nachfolgenden Grafik ist ein typischer Ablauf des MVC Modells abgebildet. Der Ablauf beginnt mit einer Nutzereingabe über den View-Programmteil. Dies löst über den Controller eine Anfrage zu entsprechenden Datensätzen aus. Gemäß der Datenhaltung sucht der Controller nach Einträgen und initialisiert diese anhand des Models. Das so erzeugte Objekt wird an den View-Programmteil übergeben und anschließend geändert. Wie im Abschnitt *Model* beschrieben, löst die Methode eine entsprechende Transaktion auf die Datenhaltung aus und gibt deren Ergebnis zurück. *Vgl.* (Nolan, 2016)

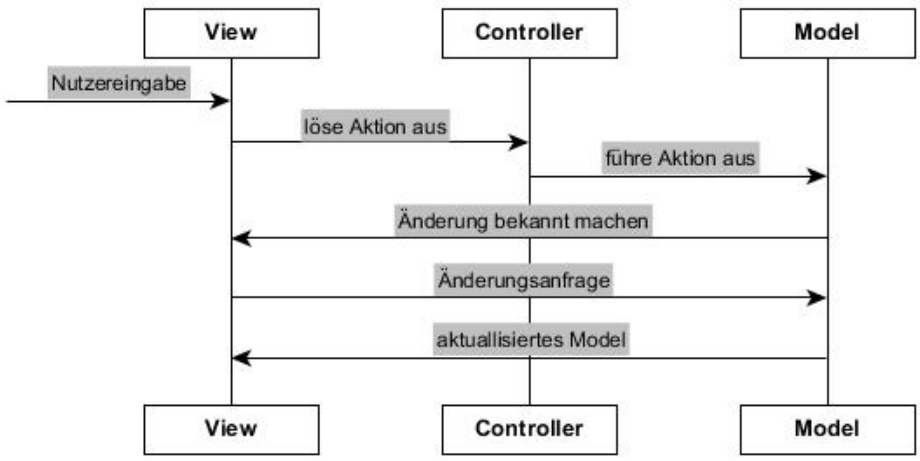

Abbildung 1: MVC Programmablauf einer Nutzereingabe

Dieser Ablauf ist jedoch nur dann möglich, wenn der View-Programmteil Zugriff auf das Model hat, so zum Beispiel bei nativen Apps oder Programmen. Im Bereich der Webentwicklung ist der View-Programmteil jedoch physisch vom Rest getrennt, da Controller und Model auf einem Server liegen und der View-Programmteil auf einem Client ausgeführt wird. Somit müsste der Controller Datenobjekte an den Client schicken, welche direkten Zugriff auf die Datenhaltung haben. Da der View-Programmteil in der Webentwicklung jedoch zumeist nicht einmal die gleiche Programmsprache nutzt, müsste der View-Programmteil eigene Objekte erzeugen und diese mit einer Transaktionslogik versehen. Wenngleich ein solches Vorgehen möglich wäre, würde dies nicht mehr dem klassischen MVC-Prinzip entsprechen. Änderungen in der Datenhaltung würden demnach nicht über das Datenobjekt selbst eingepflegt werden. Stattdessen müssten im Controller zusätzliche Methoden bereitstehen, um dies zu erreichen.

YARPG kann somit nicht in der klassischen MVC-Architektur entwickelt werden. Die Architektur ist jedoch ein guter Ausgangspunkt, um vergleichbare Modelle zu verstehen.

4.1.2.2 Model View Presenter (MVP)

Die MVP-Architektur ist wie MVC-Architektur in drei Arbeitsbereiche gegliedert, um Datenhaltung, Verarbeitung und Anzeige voneinander zu trennen. Aufgabe und Aufbau des *Models* sind dabei identisch. Unterschiede gibt es hingegen beim View-Programmteil. Diesem wird die Geschäftslogik entzogen, um dem *Separation of Concerns (SoC) Prinzip* zu entsprechen. Auf diese Weise wird das Programm in eindeutig voneinander getrennte Aufgaben unterteilt. In der MVC-Architektur erhält der View-Programmteil ein Objekt des Models und benutzt dieses, um eine Anzeige zu generieren. Das Datenobjekt kann jedoch mehr Information beinhalten, als für die Anzeige notwendig sind oder der View-Programmteil muss Daten selbst kombinieren. Somit stecken nicht nur im *Controller* und *Model* Geschäftslogik, sondern auch in der Darstellung. Des Weiteren können diese zusätzlichen Daten zu einem Sicherheitsrisiko führen, weil Nutzer dadurch Zugriff auf Inhalte erlangen, die eigentlich nicht einsehbar sein sollten. Jede verarbeitende Logik in der Darstellung, die sich mit dem Inhalt der Datenobjekte beschäftigt, wird daher in den Presenter ausgelagert. Der Arbeitsablauf ähnelt der Funktionsweise des MVC-Controllers. Der Presenter wird vom View-Programmteil aufgerufen und erarbeitet entsprechend seiner Aufgabe die gesuchte Information mittels des Models. Der Unterschied ist, dass die gefundenen Datenobjekte nicht zur Formatierung an den View-Programmteil weitergereicht, sondern für die Anzeige bereits fertig formatiert werden. Als Beispiel kann wieder das Model *Person* genutzt werden. Für den View-Programmteil soll eine Liste mit Namen bereitgestellt werden. Entsprechend der Parameter lädt der Presenter dafür die Liste

aus der Tabelle und erstellt daraus eine neue Liste, die um alle nicht notwendigen Information beschnitten ist. Dieses neue Datenobjekt entspricht somit keinem Objekt des Models und ist somit auch nicht an die Datenhaltung gekoppelt. Der View-Programmteil erhält diese fertige Liste und zeigt sie lediglich an. Soll der Inhalt dieser Liste verändert werden, kann dies nicht über das mitgelieferte Datenobjekt geschehen. Stattdessen muss es eine zusätzliche Schnittstelle im Presenter geben, welche die Änderung in die Datenhaltung eingepflegt. *Vgl.* (Nolan, 2016)

Der Ablauf sieht dabei wie folgt aus:

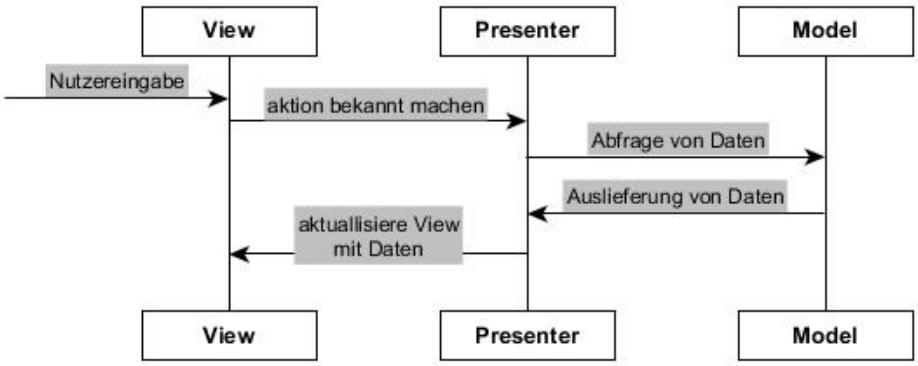

Abbildung 2: MVP Programmablauf einer Nutzereingabe

Durch eine Benutzeraktion wird der Presenter angesteuert (Siehe Abbildung 2), der die Daten an den View-Programmteil zurückgeben soll. Entsprechend des *Models* lädt der *Presenter* dafür Datenobjekte aus der Datenhaltung, überarbeitet diese und übergibt sie dem View-Programmteil. Somit gibt es keine direkte Kommunikation mehr zwischen *View* und *Model*.

Martin Fowler, ein führender Autor zum Thema MVP, unterscheidet in seinem Blog in zwei verschiedene View-Arten, *Passiv View* und *Supervising Controller*. *Vgl.* (Fowler, Model-View-Presenter (MVP), 2006)

Passiv View reicht alle Benutzeraktionen an den Presenter weiter, wartet auf dessen Rückmeldung und zeigt dessen Ergebnis lediglich an. Es erfolgen keine zusätzlichen Bearbeitungsschritte, weder in Bezug auf die Daten noch auf die Benutzeraktion. Einzige Aufgabe ist die Handhabung der Anzeige.

Der *Supervising Controller* hingegen arbeitet mit einem *Observer Pattern*. Eine Instanz des *Presenters* selbst ist dabei Teil der Anzeige. Der View-Programmteil wird somit aktiv vom

Presenter gesteuert. Der View-Programmteil wartet also nicht auf das Ergebnis, sondern wird direkt angesprochen. Diese Funktionsweise ist jedoch nur dann möglich, wenn sich *View* und *Presenter* auf demselben System befinden.

In der Webentwicklung sind *Model* und *Presenter* jedoch, wie bereits erwähnt, physisch vom View-Programmteil getrennt. Daher kann in diesem Zusammenhang lediglich *Passiv View* zum Einsatz kommen.

Insgesamt betrachtet ist die MVP-Architektur für das Projekt YARPG deutlich relevanter als das klassische MVC-Prinzip. Da der View-Programmteil keinen direkten Zugriff auf die Datenhaltung hat, bietet MVP mehr Sicherheit und streicht zudem die Geschäftslogik aus der Anzeige.

Benutzer hätten, wie im Projektplan vorgegeben, darüber hinaus die Möglichkeit, eigene Views zu entwickeln. Alle notwendigen Darstellungen und Interaktionen wären über den *Presenter* erreichbar. Lediglich Formatierungsvorschriften dürften nicht Teil der vom *Presenter* erstellten Datenobjekte sein, um Entwicklern notwendige Freiheiten in der Darstellung zu geben. Es dürfte also kein fertiges HTML-Dokument ausgeliefert werden, obwohl dies im Zuge der MVP-Architektur möglich wäre.

In dieser flexiblen Ausgabemöglichkeit steckt im Übrigen ein weiterer Vorteil des *Presenters*. Er lässt sich dadurch in kleine Aufgabenpakete zerlegen und so besser wiederverwenden. Deutlich wird das wieder anhand des Beispiels *Person*. Soll eine Personenliste dargestellt werden, könnte der *Presenter* ein HTML-Dokument ausliefern, in dem die Liste mitsamt Darstellung tabellarisch vordefiniert ist. Soll nun dieselbe Liste in einer anderen Darstellung verwendet werden, müsste es einen zusätzlichen *Presenter* geben, der lediglich eine andere Formatierung mitliefert. Ist die Formatierung dagegen nicht Teil der vom *Presenter* erstellten Rückgabe, kann der View-Programmteil selbst entscheiden wie die Anzeige auszusehen hat.

4.1.2.3 Model View ViewModel (MVVM)

Die MVVM-Architektur ist der MVP-Architektur sehr ähnlich. Unverändert ist nicht nur das *Model*, sondern auch das Ziel, keine Geschäftslogik in den View-Programmteil zu integrieren, diesen also nur mit den nötigsten Daten zu versorgen. Die verarbeitende Logik steckt dagegen im *ViewModel*, das für den View-Programmteil die nötige Information erarbeitet und entsprechend der Vorgaben zusammenstellt. Der große Unterschied zum *Presenter* der MVP-

Architektur besteht darin, wie die erstellten Datenobjekte an den View-Programmteil weitergereicht werden. Im MVVM werden Elemente des View-Programmteils direkt an einzelne *ViewModels* geknüpft, das heißt, die Objekte werden miteinander verbunden. Dieses Vorgehen wird auch als *Databinding* bezeichnet. Ziel ist es, das Senden und Erhalten von Updates zu ermöglichen. *Vgl.* (Fowler, Presentation Model, 2004) Dies ist jedoch nur dann möglich, wenn sich *ViewModel* und *View* auf demselben System befinden und dieses System *Databinding* unterstützt. *Windows Phone* unterstützt beispielsweise diese Technologie mittels Bibliotheken in der C#-Umgebung. Für *Android* (Java) und *iOS* (Objectiv C) hingegen müssten dafür Methoden selbst geschrieben werden. Da es sich dabei jedoch um wiederkehrende Methoden mit wenig Logik handelt, sogenannten *Boilerplate code*, ist es ratsam, den Aufwand durch eine automatische Quellcodegenerierung zu reduzieren. Anders als bei MVC und MVP hängt es also vom Zielsystem ab, wie hoch der Aufwand ist, eine solche Architektur umzusetzen. Wird die automatische Quellcodegenerierung unterstützt, liegt der Vorteil in der Reduktion des für die Kommunikation notwendigen Quellcodes, welcher vom Entwickler geschrieben werden muss. *Vgl.* (Verma, 2019)

Für das Projekt YARPG ist MVVP letztlich nicht geeignet, da erstens View und Back-End getrennt werden sollen und zweitens alle Zielumgebungen mit möglichst geringem Aufwand entwickelt werden sollen.

4.1.2.4 Clean Architecture

Die Clean Architecture ist eine Kombination zahlreicher verschiedener Ansätze, die in den letzten Jahren aufgekommen sind. Nennenswert sind die folgenden:

- **Hexagonal Architecture** (auch bekannt als Ports and Adapters) von Alistair Cockburn, adaptiert von Steve Freeman und Nat Pryce
- **Onion Architecture** von Jeffrey Palermo
- **Screaming Architecture** von Robert C. Martin
- **Data, context and interaction (DCI)** von James Coplien und Trygve Reenskaug
- **Bountry Control Entity (BCE)** von Ivar Jacobson

Vgl. (Martin, 2012)

All diese Architekturen verfolgen mit *Separation of Concerns (SoC)* dasselbe Ziel. Sie nutzen dabei ähnliche Mittel und unterscheiden sich lediglich im Detail. Die *Trennung der Zuständigkeiten* erreichen alle, indem die Anwendung in Schichten unterteilt wird, darunter mindestens eine für Geschäftslogik und eine für Schnittstellen.

Die so produzierten Systeme sind

- unabhängig vom Framework,
- unabhängig vom Front-End,
- unabhängig von der Datenbank,
- unabhängig von jedem externen Dienst und somit auch
- unabhängig von Front-End, Datenbank und Webserver testbar.

Vgl. (Martin, 2012)

Diese Unabhängigkeit wird durch die *Dependency Rule* (Abhängigkeitsregel) der einzelnen Schichten erreicht. Um diese zu erklären, soll nachfolgende Abbildung dienen, in der die zuvor genannten Architekturen zusammengefasst sind.

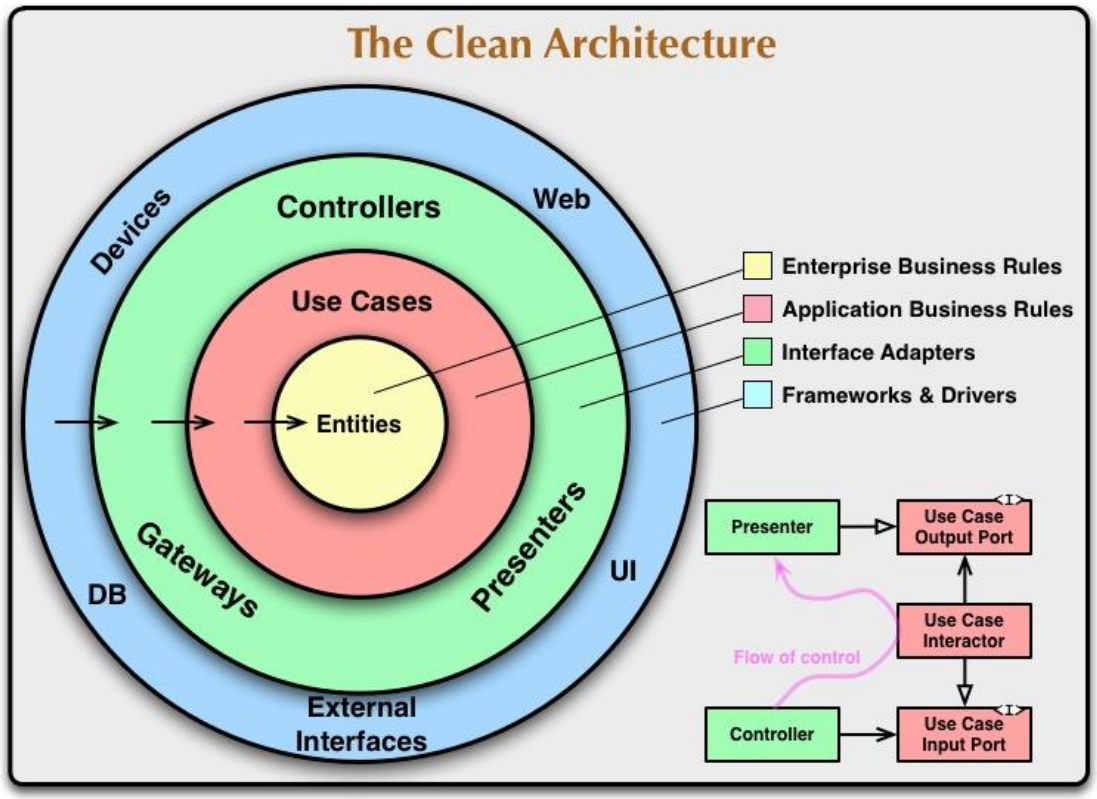

Abbildung 3: Aufbau der Clean Architecture Vgl. (Martin, 2012)

Die konzentrischen Kreise repräsentieren unterschiedliche Bereiche der Software. Je weiter innen ein Kreis ist, desto generalisierter ist die Software, das bedeutet, im Inneren steckt die grundlegende Strategie oder Geschäftslogik. Die äußeren Kreise hingegen beinhalten Mechanismen. Die *Dependency Rule* besagt dabei, dass alle Quellcodeabhängigkeiten nach innenzeigen. Kein Kreis kann daher etwas über den darüberliegenden Kreis wissen. Wird also beispielsweise eine Datenbankanbindung über das *Spring Framework* in Java realisiert, gibt es diesbezüglich im darunterliegenden Kreis keinen Hinweis.

Wie in der Abbildung 3 zu erkennen, erfolgt eine solche Verbindung im äußersten Kreis und muss zunächst auf die im Projekt verwendeten Entities transformiert werden. Dabei können sich Struktur und Bezeichnungen gegebenenfalls ändern. Die Anwendung ist damit nicht von *Spring* abhängig. Struktur und Funktionsweise des Models bleiben *im Kern* unberührt, selbst dann, wenn *Spring* durch ein anderes Framework ersetzt wird.

Die Transformation hingegen findet in einem der äußeren Kreise statt und hat die Aufgabe, von einer Datenstruktur zur im Projekt verwendeten Datenstruktur zu übersetzen. Dazu gehört auch, Daten in das Projekt zu laden und wieder heraus zu schreiben. Dieser Vorgang wiederholt sich mit allen Abhängigkeiten. Es existiert somit zu jedem externen Softwarepaket, das in das Projekt integriert wird, eine Transformationsschicht. Auf diese Weise wird sichergestellt, dass alle Abhängigkeiten nach innen zeigen.

Darüber hinaus gilt diese Regel auch für den eigenen Quellcode. Wird im Projekt beispielsweise eine Weboberfläche benötigt und auf REST-Schnittstellen gesetzt, soll sich diese Verbindung nicht in den inneren Kreisen widerspiegeln. Da die Geschäftslogik unabhängig von der Darstellung ist, muss es eine Schicht geben, die das Ergebnis der Verarbeitung transformiert und für die gegebene REST-Schnittstelle verwendet. Der Vorteil ist auch hier wieder, dass von einer Weboberfläche auf eine Konsolenanwendung oder sogar auf mehrere Darstellungen umgestellt werden könnte, ohne dabei die Geschäftslogik zu verändern. Dieser Aufbau hilft nicht nur, im späteren Verlauf der Entwicklung andere Abhängigkeiten in das Projekt einzubeziehen oder zu ändern. Er garantiert außerdem maßgeblich, dass sich die üblichen Softwareänderungen, wie etwa neue Features, größtenteils in einer der äußeren Schichten befinden.

Änderungen haben somit weniger Einfluss auf andere Komponenten, sind weniger fehleranfällig und weniger aufwendig zu testen. Eine externe Bibliothek muss beispielsweise nicht getestet werden, lediglich ihre Transformation zu den eigenen Entities. Da sich alle notwendigen

Objekte im Projekt selbst erstellen lassen, entfällt auch die Notwendigkeit, einen Datenbankkontext zur Überprüfung von Programmkomponenten zu erstellen. Dies verringert sowohl die Komplexität der notwendigen Tests als auch der zu schreibenden Softwarekomponenten und steigert gleichzeitig die Stabilität des Produkts. *Vgl.* (Martin, 2012)

Entities

Die Entities der Clean Architecture kapseln die anwendungsweiten Regeln in Form von Objekten mit Methoden oder auch in Form eines Sets aus Datenstrukturen und Funktionen. Sie bilden die grundlegenden Objekte für die Anwendung, auf die jede Verarbeitung zurückführt und bleiben während üblicher Erweiterungsarbeiten an den weiter außen liegenden Schichten in der Regel unverändert.

Eine Anpassung der Entities wird jedoch dann erwartet, wenn die bestehenden Objekte die Anwendungsproblematik nicht mehr abbilden können. Der elementare Unterschied zu den anderen Architekturen besteht in der Abgrenzung externer Softwarepakete und somit auch in der direkten Objektverknüpfung mit einer Datenbank.

Es ist unrealistisch, in einem Projekt ohne externe Bibliotheken auszukommen und alle Datenbankoperationen selbst zu entwickeln. Wie bereits in Kapitel *2 Grundlagen* erwähnt, würde dies nicht nur den Aufwand enorm steigern, sondern das Produkt auch anfälliger für Fehler machen. Der Dateninhalt der Entities muss also aus der darüberliegenden Schicht kommen, welche die Objekte initialisiert. *Vgl.* (Martin, 2012)*:Entities*

Use Cases

Diese Schicht beinhaltet weitestgehend die Geschäftslogik der Anwendung. Sie ist dabei deutlich spezifischer als die eher rudimentären Regeln der Entities. Darüber hinaus bestimmt die Schicht den Datenfluss von und zu den Entities. Es wird nicht erwartet, dass sich eine Änderung in dieser Schicht auf die Entities auswirkt oder sich eine Änderung der äußeren Schicht auf den Use Case auswirkt. Es wird jedoch erwartet, dass eine Erweiterung oder Anpassung bestehender Use Cases diese Schicht verändert. *Vgl.* (Martin, 2012)*:Use Case*

Interface Adapter

Der Interface Adapter transformiert das Datenformat der Use Cases und Entities in externe Datenformate oder Webanwendungen. Die Schicht ist zudem für die Darstellung der Anwendung verantwortlich und könnte demzufolge in Bezug auf das Front-End eine MVC ähnliche Struktur aufweisen. Ein Beispiel für einen solchen Adapter wäre ein REST-Controller für die Webanwendung. Dieser nutzt einen Use Case, um seine Aufgabe zu erfüllen. Der REST-Controller würde dabei ein Objekt als Ergebnis erhalten und es an den View-Programmteil einen Ring weiter außen zurückgeben. Das vom REST-Controller genutzte Model in dem Beispiel wäre unter Umständen lediglich ein *Data Transfer Object (DTO)*, also ein Datenobjekt, das für die Kommunikation benutzt wird. Dieses Objekt muss in eine für die Webanwendung verständliche Form gebracht werden, beispielsweise in das JSON-Format. Die Transformation besteht somit darin, aus Use Case beziehungsweise Entity ein JSON-String zu erstellen und über den REST-Controller für die Webanwendung zugänglich zu machen. *Vgl.* (Martin, 2012)*:Interface Adapters*

Auf die gleiche Weise werden externe Datenbanken bedient. Kommt beispielsweise eine SQL-Datenbank zum Einsatz, befindet sich der gesamte zugehörige Quellcode ausschließlich in der Schicht des Interface Adapters, so zum Beispiel das Lesen und Schreiben der Datenbank sowie das Initialisieren der Entities aus der gelesenen Information. In den Entities selbst befindet sich keine Referenz zu einer SQL-Datenbank. Der Quellcode ist inhaltlich getrennt voneinander. Auf diese Weise könnten auch diverse Datenquellen unterschiedlichster Formate in das Projekt eingebunden werden, ohne Use Case oder Entity ändern zu müssen.

Außerdem befindet sich in dieser Schicht jeder andere Adapter, der einen externen Service nutzt und die gegebene Information in ein internes Format transformiert.

Frameworks und Treiber

Im äußersten Ring befinden sich alle externen Frameworks, Datenbanken oder ähnliches. In diesem Bereich wird in der Regel nicht mehr Quellcode geschrieben als für den externen Service notwendig. Ebenfalls enthalten sind Web- oder andere Front-End-Darstellungen, da diese keinen Einfluss auf das Softwareprodukt selbst haben sollen. Im Falle von YARPG wäre demnach die App, in der das Spiel dargestellt werden soll, Teil dieser Schicht. *Vgl.* (Martin, 2012)*:Frameworks and Drivers*

Grenzen überqueren

In der Abbildung 3 ist unten rechts aufgezeigt wie die Grenzen der jeweiligen Ringe überschritten werden. Dabei wird deutlich wie ein Controller mit einem Presenter und dem zugehörigen Use Case zusammenarbeitet. Die Aktivität beginnt im Controller, der auf einen Use Case verweist. Dieser wiederum zeigt auf den Presenter. Gelöst wird diese Grenzüberschreitung durch das *Dependency Inversion Principle*. Dabei werden, zum Beispiel in Java, Interfaces erstellt, die im Use Case Anwendung finden, jedoch eine Schicht darüber durch den Presenter implementiert sind. Die Abhängigkeit des Use Case zeigt somit nicht auf den Presenter, sondern auf ein Interface der gleichen Ebene. An jeder Grenzüberschreitung dieser Architektur kommt dieses Prinzip, wenn nötig, zum Einsatz. Somit wird die *Dependency Rule* nicht gebrochen. *Vgl.* (Martin, 2012)*:Crossing boundaries*

4.1.3 Entscheidung

Klassischer MVC und MVVM sind für die Anwendung mit Server und Mobilgeräten nicht geeignet. Grund hierfür ist vor allem der Umstand, dass eine Aufspaltung in server- und clientseitig ausgeführte Programmteile nicht ohne Weiteres möglich ist. Aufgrund des Databindings (MVVM) beziehungsweise des direkten Zugriffs auf das Model (Klassischer MVC) sehen diese Architekturen keine zusätzliche Kommunikationsebene vor.

MVP hingegen ermöglicht eine solche Ebene durch die Entkopplung des Models zum eigentlichen View-Programmteil. Der Presenter verarbeitet und transformiert entsprechend seiner Aufgabe Daten, die generalisiert auch auf ein anderes Gerät übertragen werden können. Somit wäre eine Kommunikation zwischen Server und Mobilgerät leichter möglich als mit MVVM und klassischem MVC. Die Architektur beinhaltet jedoch keine besonderen Maßnahmen für Stabilität, Testbarkeit oder Erweiterbarkeit. Wie gut sich ein System mit dieser Architektur in den genannten Kategorien schlägt, läge also fast ausschließlich an Detailentscheidungen der Komponenten. Nichtsdestotrotz ist MVP eine solide Architektur, die für YARPG eingesetzt werden könnte.

Die Clean Architecture ist in ihrem Grundaufbau etwas umfangreicher als die anderen Architekturen. Sie besticht jedoch durch das einfache Prinzip, Abhängigkeiten immer nach innen

zeigen zu lassen. Das macht eine Anwendung unabhängig von externen Quellen, sehr einfach testbar, langfristig stabiler und sehr einfach erweiterbar. Viele der wichtigen ARS sind damit erfüllt. Dank der effizienten Erweiterbarkeit lässt sich eine solche Anwendung auch zu einem späteren Zeitpunkt leichter skalieren. So können externe, limitierende Faktoren, wie beispielsweise die Datenbank, vollständig ausgetauscht werden, ohne das gesamte Projekt neu entwickeln zu müssen. Relevant ist das vor allem für langlebige und dynamische Projekte wie YARPG, die sich in verschiedene Richtungen entwickeln können und bei denen nicht zu erwarten ist, dass alle zu Beginn getroffenen Entscheidungen bestehen bleiben. Diese Veränderbarkeit stellt daher einen immensen Vorteil dar. Darüber hinaus ist diese Architektur auf jedem Zielsystem ohne besondere Technologien umsetzbar. Nicht zuletzt ist auch der View-Programmteil als alleinstehende Komponente vorgesehen, wodurch jede Variante genutzt werden kann, die der entsprechenden Aufgabe gerecht wird. Die Entscheidung fällt somit eindeutig zugunsten der Clean Architecture aus.

4.1.4 Mobilgeräte

Durch das Geschäftsmodell ist eine Reihe an Zielen vorgegeben, die sich gegenüberstehen. Zielsystem sind einerseits GPS-fähige Mobilgeräte sowie die gängigsten Browser. Andererseits soll die Anwendung als App von nur einer Person entwickelt werden. Da jedes Betriebssystem eine andere Programmiersprache nutzt, ist jedoch eine Portierung als native App von einem System zum anderen nicht ohne Weiteres möglich. Zumindest Teile der Anwendung müssten für jedes System spezifisch entwickelt werden, nicht nur einmalig, sondern für jedes neue Spielelement aufs Neue.

Web App

Eine beliebte Möglichkeit ist die Verwendung einer Web App. Im Unterschied zur nativen App werden sie nicht spezifisch für eine Plattform entwickelt. Sie nutzen stattdessen einen Webbrowser, der während der Nutzung alle nötigen Inhalte aus dem Internet herunterlädt. Da ein solcher Browser auf jedem modernen Mobilgerät zur Grundausstattung gehört, wäre dies in jedem Fall ein gangbarer Weg. Die Anwendung bräuchte dann nur einmalig für Browser entwickelt werden und könnte folglich plattformunabhängig genutzt werden. Allerdings gehen mit einer Web App auch einige Nachteile einher. Gerätefunktionen, wie Kamera, lokaler Speicher oder GPS sind damit entweder gar nicht oder nur eingeschränkt nutzbar. Zwar ist die Entwicklung nicht stehen geblieben und über HTML5 sind einige dieser Gerätefunktionen

ebenfalls nutzbar, allerdings besitzen die Browser der Zielsysteme auch einige spezifische Vor- und Nachteile. Daher kann es passieren, dass auch im Rahmen der Web App einige Funktionen Browser-spezifisch entwickelt werden müssen. Trotz mancher Unterschiede interpretieren alle Browser HTML, CSS und JS jedoch weitgehend identisch. Der Aufwand für einige Browser-spezifische Funktionen wäre also vertretbar. Vgl. (Chaffee, 2012)

Deutlich negativer ins Gewicht fällt die Reaktionszeit einer Web App. Weil alle Daten erst aus dem Internet bezogen werden müssen, wartet eine solche Anwendung entsprechend der Verbindung bis sie reagieren kann, typischerweise gefolgt von einem Seitenaufbau. Moderne Browser speichern Websiteinhalte zwar gezielt im Cache, allerdings ist dieser Vorgang vom Browser abhängig und daher kaum zu steuern. Die native App ist hier eindeutig im Vorteil, da alle relevanten Inhalte bereits auf dem Gerät zur Verfügung stehen, wie beispielsweise Bilder, Ansichten oder die Abläufe der Benutzeraktionen. Lediglich veränderbare Information muss nachgeladen werden, wie beispielsweise Lebenspunkte oder Position der Monster auf der Karte.

Adobe PhoneGap (Hybrid App)

Eine weitere Möglichkeit wäre die Nutzung von *Adobe PhoneGap*, einem Framework zur Erstellung sogenannter hybrider Apps. *PhoneGap* hat den Fokus, die Entwicklungsarbeit über mehrere Plattformen hinweg zu unterstützen. Das Framework nutzt dafür einen Browser, der als natives Programm für die jeweiligen Mobilsysteme entwickelt und um weitere Funktionen erweitert wurde. Um den Schwächen einer Web App entgegenzuwirken, bietet *PhoneGap* einerseits die Möglichkeit, Betriebssystem-spezifische Plug-ins zu schreiben, die vollen Zugriff auf die Gerätefunktionen haben. Andererseits ermöglicht es, die Anwendung mit allen Web-spezifischen Dateien auszuliefern. Das wiederum erspart das Cachen oder Nachladen. Lediglich die wechselnden Inhalte müssen wie bei nativen Apps heruntergeladen werden. Somit ist nicht nur die benötigte Datenmenge verringert, sondern auch die Reaktionszeit vergleichbar mit einer nativen App. Zudem sind viele Plug-ins frei zugänglich, da sie durch eine breite Interessensgemeinschaft bereits entwickelt wurden. Dadurch ist es möglich, GPS, Kamera und ähnliches direkt über die Betriebssystemschnittstelle anzusteuern, anstatt über die Schnittstelle des Browsers, wie im Fall einer einfachen Web App. Vgl. (Ratnottar, 2019)

Natürlich hat diese Vorgehensweise auch Nachteile. So muss beispielsweise ein vollständiger, wenn auch für den Nutzer nicht sichtbarer, Browser mit der eigentlichen App ausgeliefert werden. Dadurch sind selbst kleine Anwendungen deutlich größer als vergleichbare native

Apps. Eine leere *PhoneGap* App mit einigen ausgewählten Plug-ins ist in etwa 5 MB groß. Eine leere native App kommt im Vergleich dazu mit nur wenigen KB aus. Die für die Anwendung benötigten Dateien werden natürlich in beiden Fällen gebraucht. Je umfangreicher die Anwendung ist, desto größer wird daher auch die native App und zwar in ähnlichem Maß wie die PhoneGap App. Da die Speicher moderner Mobilgeräte immer größer werden, sollte es also auf einige MB mehr oder weniger nicht ankommen.

Nachfolgend werden die vorgestellten Möglichkeiten anhand verschiedener Parameter verglichen. Darüber hinaus wird bewertet wie wichtig diese Parameter für das Projekt sind.

Tabelle 1: Vergleich Native, Hybride und reine Web Application

	Entwicklungsart			
Vergleichsparameter	**Native App**	**Reine Web App**	**Adobe PhoneGap (Hybrid App)**	**Gewichtung**
Entwicklungsaufwand, um alle Zielsysteme zu erreichen	Hoch, Neuentwicklung für jedes System notwendig	Mittel, Browserspezifische Anpassungen notwendig	Mittel, Browserspezifische Anpassungen für PC notwendig	**Sehr wichtig**
Nutzbarer Umfang der Gerätefunktionen	Uneingeschränkt	Teilweise eingeschränkt	Uneingeschränkt	**Wichtig**
Versionierung der Anwendung, Clientupdates nötig	Ja, nach jeder View-Änderung	Nein, alle Daten werden immer neu geladen	Ja, nach Änderung der Plug-ins	**Weniger wichtig**
Reaktionszeit nach Benutzerinteraktion	Schnell, Ablauf ist auf dem Gerät vorhanden	Langsam, Ablauf muss geladen werden	Schnell, Ablauf ist auf dem Gerät vorhanden	**Sehr wichtig**
Benötigter Gerätespeicher	Niedrig	Keiner	Hoch	**Weniger wichtig**

	Entwicklungsart			
Vergleichsparameter	**Native App**	**Reine Web App**	**Adobe PhoneGap (Hybrid App)**	**Gewichtung**
Benötigter Arbeitsspeicher	Niedrig, nur die Daten der Anwendung sind im Speicher	Hoch, Daten von Browser und Anwendung sind im Speicher	Hoch, Daten von Browser und Anwendung sind im Speicher	**Wichtig**
Benötigte Datenbandbreite	Niedrig, nur Spielinformation nötig	Hoch, alle Daten nötig	Niedrig, nur Spielinformation nötig	**Weniger wichtig**

Entscheidung

Eines der wichtigsten Punkte ist der Entwicklungsaufwand. Besonders die native App sticht dabei negativ heraus und scheidet damit von vornherein aus. Übrig bleiben damit die reine Web App und die hybride App. Die größten Unterschiede zwischen beiden ergeben sich in puncto Reaktionszeit und Datenverbrauch. Die Reaktionszeit bezeichnet allgemein die Zeit zwischen Impuls und Antwort, sowohl bei einem Organismus als auch einem Softwaresystem. In Bezug auf ein Programm kann es die Zeit zwischen Benutzereingabe und Anzeige einer Reaktion sein. Laut Jakob Nielsen Vgl. (Nielsen, 1993) gibt es 3 wichtige Fristen für eine Anwendung: Unter 0,1 Sekunden, unter 1 Sekunde und 10 Sekunden. Bei unter 0,1 Sekunden scheint die Anwendung für Nutzer sofort zu reagieren. Der Nutzer erlebt also keinerlei Einschränkungen. Ladezeiten von bis zu 1 Sekunde sind für Nutzer bereits ersichtlich. Gedankenfluss und Handlungsablauf werden dabei jedoch nicht gestört. 10 Sekunden oder mehr stellen die Grenze für die Aufmerksamkeit eines Nutzers dar. Um die Wartezeit zu überbrücken, wird er sich einer anderen Aufgabe widmen. Laut Nielsens Empfehlung sollte ein Programm so schnell wie möglich reagieren oder zumindest ein Lade-Symbol einblenden. Damit sind nicht zwangsläufig Ladebalken oder Prozentanzeigen gemeint. Ein „In Arbeit“-Symbol für unter 10-sekündige Prozesse ist ausreichend. Eine Anwendung, die Daten aus dem Internet bezieht, ist natürlich von der zur Verfügung stehenden Datenbandbreite und dem Ping zum Server abhängig. Auf diesen Teil der Übertragung kann kein Einfluss genommen werden. Jedoch kann die Datenmenge verringert werden, um die App auch bei schlechter Bandbreite

nutzen zu können. Die hybride App bietet diesbezüglich die Möglichkeit, Bilder, HTML-, CSS- und JavaScript-Dateien in die App zu integrieren. Dies macht in der Regel den Großteil der nötigen Daten aus. Außerdem lässt sich die Anwendung damit so realisieren, dass sich die Ansicht bereits aufbaut, während die aktuellen Spieldaten noch geladen werden. Im Zweifel kann auf diese Weise sogar ein Symbol eingeblendet werden. Sinnvoll ist das beispielsweise auch, wenn der Nutzer auf eine andere Seite wechselt. Bei einer reinen Web App wäre dies nur dann möglich, sofern mit einer *Single Page Application* gearbeitet wird, das bedeutet, die Anwendung wechselt nicht zu einer neuen Ansicht, sondern arbeitet mit Fenstern, Tabs oder Überlagerungen, um Inhalte im gleichen Bild anzuzeigen. Dafür müssten jedoch auch alle darstellungsrelevanten Inhalte beim ersten Aufruf geladen werden. Somit braucht eine reine Web App in jedem Fall mehr Datenbandbreite und somit meist auch mehr Zeit, um auf eine Benutzereingabe zu reagieren. Auch bei einem Ausfall der Internetverbindung ist die reine Web App im Nachteil, denn sie kann lediglich eine Fehlermeldung des Browsers anzeigen. Mit einer hybriden App gibt es in jedem Fall bessere Möglichkeiten, den Nutzer zu informieren, ohne ihm dabei das Gefühl zu geben, die Anwendung verlassen zu haben. Der View-Programmteil auf den Mobilgeräten wird aufgrund dieser Vorteile mit *Adobe PhoneGap* implementiert.

4.1.5 Aufbau des Systems

In den vorangegangenen Kapiteln wurde entschieden, die Anwendung mit der Clean Architecture zu entwickeln und zur Anzeige eine App mit *Adobe PhoneGap* zu schreiben. Das System besteht demzufolge aus einem Server mit Datenbank, einer Datenverarbeitung und den Schnittstellen für die Kommunikation. Die App fungiert lediglich als View-Programmteil und nutzt dafür eine Schnittstelle. Ihr Aufbau ist somit für das System nicht entscheidend und kann folglich in dieser Betrachtung vernachlässigt werden.

Wie zuvor beschrieben besteht eine solche Software aus Entities, Use Cases, Adapter und Schnittstellenimplementierungen. Um die Übersichtlichkeit zu gewährleisten, ist es sinnvoll, die Anwendung in Themen oder auch Teilprojekte zu unterteilen. Entities und Use Cases bilden das Herzstück oder auch den Kern der Anwendung und sollten dementsprechend auch in der Projektstruktur auftreten. Die zwei anderen großen Bereiche sind der Einstiegspunkt (Entry Point) in die Anwendung, sowohl für den Nutzer als auch für den View-Programmteil und natürlich die Datenbankanbindung. Daher wird das Projekt in Core, Entry Point und Dataprovider aufgeteilt. In der nachfolgenden Abbildung ist dies veranschaulicht.

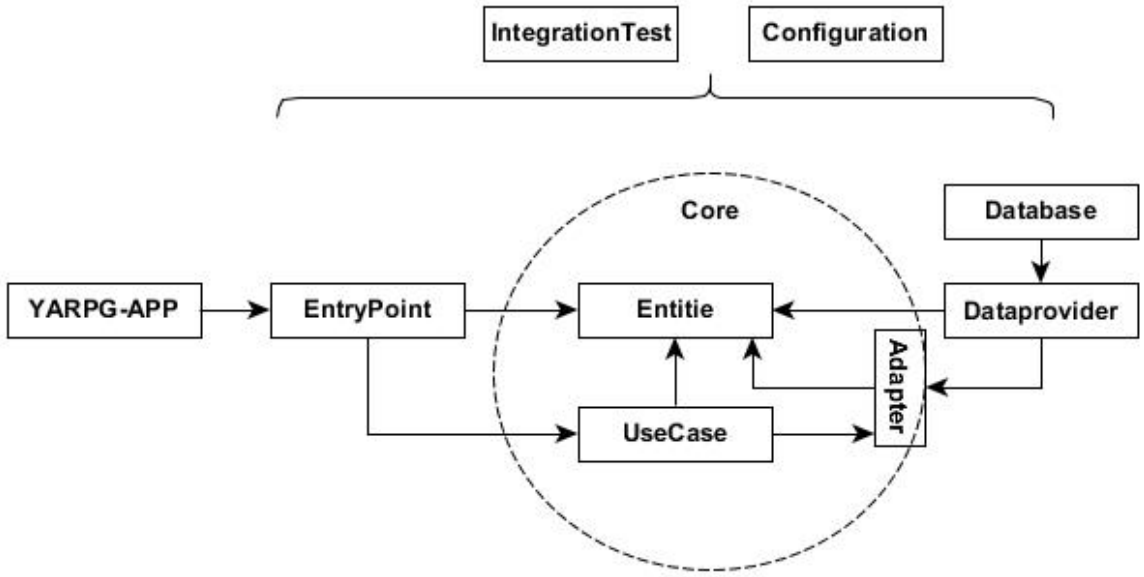

Abbildung 4: Aufbau und Abhängigkeiten der Teilprojekte in YARPG

Die Pfeile in dieser Abbildung zeigen die Abhängigkeiten der einzelnen Abschnitte zueinander. Sie alle zeigen nach innen oder auf die gleiche Ebene. Zu erkennen ist ebenso, dass die Entities im Core von allen Teilbereichen benötigt werden. Davon ausgenommen sind lediglich die Datenbank und die App. Die Schnittstelle der App kommuniziert mit *Data Transfer Objects (DTO)*, die zum Teilprojekt Entry Point gehören. Die Datenbank ist durch die Implementierung des Teilprojekts Dataprovider an die Anwendung gekoppelt. Im Rahmen dieser Implementierung werden die Entities später mittels entsprechender Datenbankeinträge initialisiert.

Unit Tests sind ein integraler Bestandteil eines jeden Teilprojekts. Sie müssen daher nicht gesondert aufgeführt werden. Einen gesonderten Baustein bilden dagegen die Integration Tests. Sie erstrecken sich über zwei oder mehr Teilprojekte. Aufgrund der Projektstruktur ist dies jedoch nur dann möglich, wenn diese Tests ein eigenes Teilprojekt bilden, da ansonsten die Abhängigkeiten zu den einzelnen Teilprojekten nicht aufgelöst werden können. Die Teilprojekte Entry Point und Dataprovider besitzen keine Abhängigkeit zueinander. Übergreifende Tests können somit nicht Teil dieser Teilprojekte sein. Das gleiche gilt für die Konfiguration des Systems. Da das Teilprojekt Core über keine Information zu den Teilprojekten Entry Point und Dataprovider verfügt, muss zum Start der Anwendung festgelegt werden, welche Provider und welche Entry Points vorhanden sind. Aus diesem Grund muss das Spiel durch das Teilprojekt Configuration gestartet werden. Demzufolge sind Cofiguration und Integration Test eigene Teilprojekte, die auf alle Teilprojekte zugreifen können. Obwohl das Gesamtprojekt von

der Konfiguration abhängig ist, haben die einzelnen Klassen keine Abhängigkeit dorthin. Somit ist die Abhängigkeitsbedingung weiterhin erfüllt. Die App muss nicht durch das System konfiguriert werden, da sie lediglich eine Schnittstelle verwendet und ein eigenes Teilprojekt bildet.

4.2 Interaktive Karte

4.2.1 Grundlegende Entities

Gemäß Kapitel *3.2 Aufbau und Grundidee* ergeben sich grundlegende Objekte, die zwingend vorhanden sein müssen. Neben einem Spielermodel und dessen Nutzerdaten gibt es die georeferenzierten Objekte und Charakter-Models, wie Monster und Helden. Damit die Übersichtlichkeit gewahrt bleibt, ist eine Auswahl an Entities sinnvoll, um letztlich einzelne Use Cases zu implementieren. Das nachfolgende Klassendiagramm soll daher lediglich einen Ausschnitt darstellen, der für die interaktive Karte beziehungsweise den Erzähler notwendig ist (siehe Kapitel *3.3 Auswahl der Spielelemente für die Softwareanalyse*).

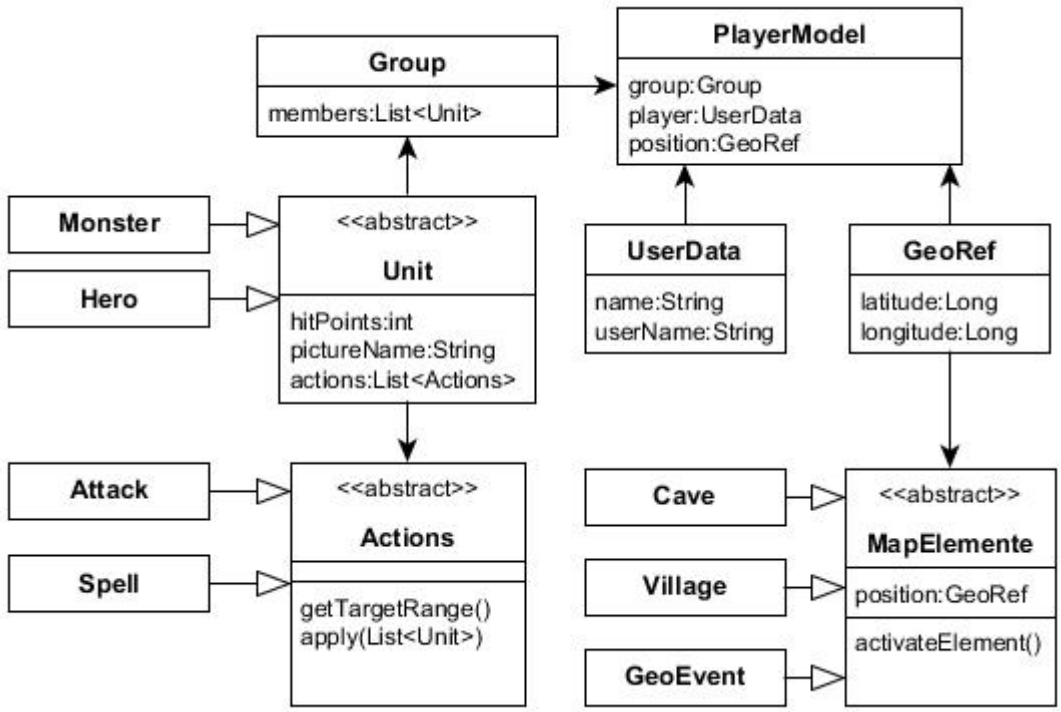

Abbildung 5: Ausgewählte Entities als Klassendiagramm

PlayerModel

Wie in Abbildung 5 zu erkennen ist, verbindet das PlayerModel den Spieler mit seinen Helden, seiner Position und seinen Nutzerdaten. Dieses Objekt ist der Dreh- und Angelpunkt des gesamten Spiels. Das Objekt bewegt den Spieler, löst Ereignisse durch den Erzähler aus und speichert alle spielrelevanten Daten, wie Lebenspunkte, Inventar oder ähnliches. Letzteres ist in der Grafik nicht zu erkennen, da sie für die im Rahmen dieser Masterarbeit ausgewählten Spielelemente unwichtig sind. Für das vollständige Spiel müsste das PlayerModel also noch deutlich erweitert werden.

MapElements

Außerdem zu erkennen ist das Objekt MapElements. Es ist als abstrakte Klasse aufgeführt, die dazu dient, jegliche georeferenzierte Punkte darzustellen. Als Beispiel sind Höhle, Siedlung oder ganz allgemein GeoEvent aufgeführt, welche sich aus der im Anhang beschriebenen erweiterten Spielidee ergeben. Im Zuge der kompletten Spielumsetzung werden noch weitere Implementierungen hinzukommen, dargestellt als Methode activateElement(). Diese unterscheiden sich jedoch lediglich durch ihre Auswirkung.

Unit

Die letzte Objektgruppe ist die Unit. Sie umfasst sämtliche Aktionen aller Einheiten, darunter Monster und Helden. Aktionen sind Fähigkeiten, die im Kampf eingesetzt werden können. Da dieser Bereich später stark erweitert werden soll, ist er zunächst als generalisierte Liste dargestellt. Jede Aktion wird eine eigene Implementierung erhalten, die zudem in verschiedene Aktionsarten unterteilt werden können. In der Abbildung 5 ist dies lediglich durch Spell und Attack angedeutet.

4.2.2 Spielerposition

Für die interaktive Karte benötigt die Anwendung die Möglichkeit, die Spielerposition einzutragen und abhängig von dieser Kartenelemente anzuzeigen beziehungsweise GeoEvents auszulösen. Dazu dient der Use Case *MovePlayer*. In diesem werden sowohl die Position des

Spielers in der Datenbank geändert als auch die Kartenelemente für seinen Ausschnitt geladen. Dafür werden mindestens zwei Datenbank-Adapter benötigt sowie eine Kommunikationsschnittstelle, um den Vorgang anzustoßen. Somit ergibt sich folgendes Klassendiagramm.

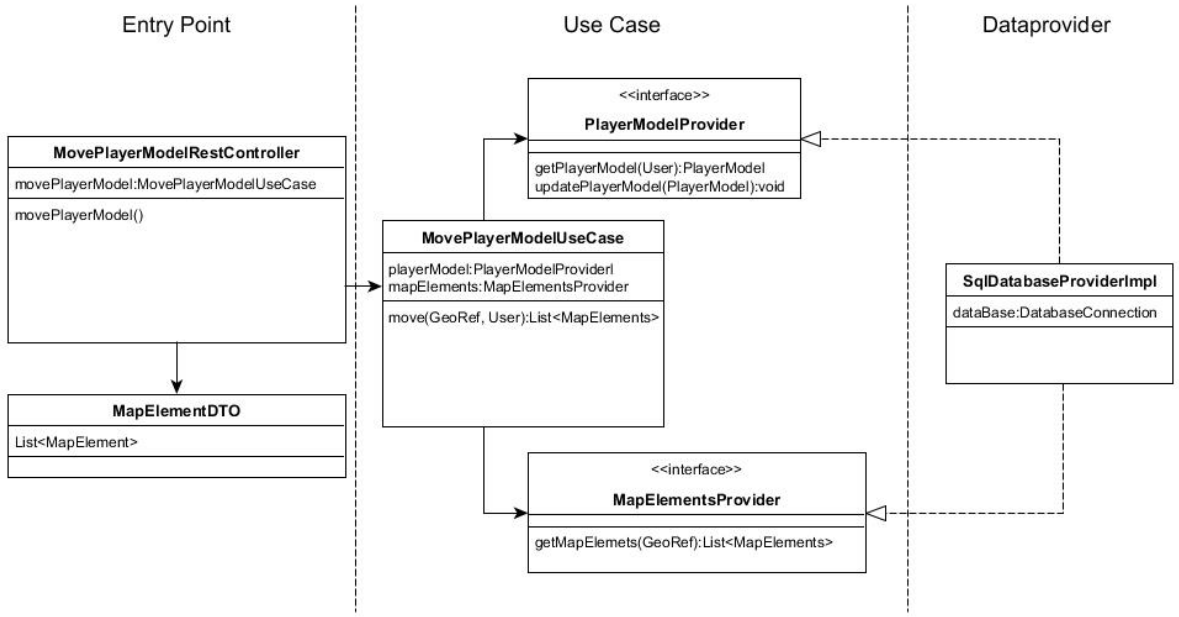

Abbildung 6: Klassendiagramm des Spielelements Move Player inclusive der Teilprojektgrenzen

In der Mitte von Abbildung 6 ist der Use Case *MovePlayerModelUseCase* zu erkennen. Dieser greift auf die beiden Adapter *PlayerModelProvider* und *MapElementsProvider* zu. Die durch ein Interface dargestellten Adapter besitzen Methoden, notwendige Information aus der Datenbank zu laden und werden im Use Case in der Methode *move()* verwendet. Im Sinne der Übersichtlichkeit, sind Entities jedoch nicht in der Abbildung 6 enthalten.

Weil die Information aus einer Datenbank geladen wird, müssen die Adapter implementiert werden. Diese Implementierung ist die Datenbankschnittstelle, in der Abbildung 6 dargestellt als *SqlDatabaseProviderImpl*. Die gestrichelte Linie zwischen Adapter und Implementierung deutet die Grenze der Teilprojekte Core und Dataprovider an. Der Dataprovider implementiert die Adapter. Diese Implementierung ist übrigens auch genau derjenige Teil, der später geändert werden kann, ohne dabei den Kern anzupassen. Soll also von der hier angedeuteten SQL-Datenbank auf etwas anderes gewechselt werden, muss lediglich die Implementierung geändert werden, nicht jedoch die Methoden oder die verwendeten Objekte im Use Case selbst.

Auf der linken Seite von Abbildung 6 befindet sich der Entry Point. In diesem Fall ist er als REST-Schnittstelle angedeutet. Dieser Controller initiiert die Spielerbewegung und übermittelt seine neue Position. Die gestrichelte Linie deutet auch hier wieder den Übergang zwischen zwei Teilprojekten an, dem Entry Point und dem Core. Zu erkennen ist außerdem die fehlende Verbindung zwischen Entry Point und Dataprovider, das bedeutet, der Controller arbeitet über die Use Cases, kommuniziert also nicht direkt mit der Datenbank. Dadurch befindet sich keine verarbeitende Logik im Controller und es wird sichergestellt, dass alle Entry Points dieselben Quellcodepfade für ähnliche Aufgaben nutzen.

Ebenfalls Teil des Entry Points ist ein Data Transfer Object (DTO), in der Abbildung 6 als *MapElementDTO* aufgeführt. Das Objekt bündelt die durch den Use Case erzeugten Daten in ein für den Controller nutzbares Format und übermittelt dieses anschließend als JSON-String.

Die Interaktivität der Karte entsteht nicht nur durch die Bewegung des Spielers, sondern vor allem durch die Interaktion mit den Kartenelementen (MapElements). Dies ist jedoch eine Sache des View-Programmteils, welcher die von der REST-Schnittstelle bereitgestellte Information anzeigt. Soll ein Element bestimmte Handlungen des Spielers zulassen oder sogar erzwingen, muss dies im View-Programmteil sichtbar zum Ausdruck kommen. Die dafür notwendige Information befindet sich im JSON-String. Jede dieser Handlungen stellt einen eigenen Use Case dar. Weil sich dessen Aufbau jedoch nicht von der Bewegung des Spielers unterscheiden, sind die verschiedenen Handlungen nicht näher erläutert. Hervorzuheben ist in diesem Zusammenhang lediglich, dass jede Handlung, die neu hinzugefügt wird, keinerlei Einfluss auf die bestehenden Use Cases hat, da sie lediglich die vorhandenen Provider nutzt oder gegebenenfalls neue benötigt.

4.3 Erzähler

Im Unterschied zur interaktiven Karte ist der Entry Point für den Erzähler keine REST-Schnittstelle, die durch den Nutzer aufgerufen wird, sondern ein Thread, der parallel zur Anwendung läuft und in geregelten Abständen Use Cases ansteuert. Ein solcher Use Case ist beispielsweise das Erzeugen, Löschen oder Verschieben von georeferenzierten Objekten. Dieser Thread zählt somit ebenfalls als Entry Point für die Anwendung und kann im gleichnamigen Teilprojekt Platz finden. Der grundlegende Aufbau entspricht ansonsten dem der interaktiven Karte.

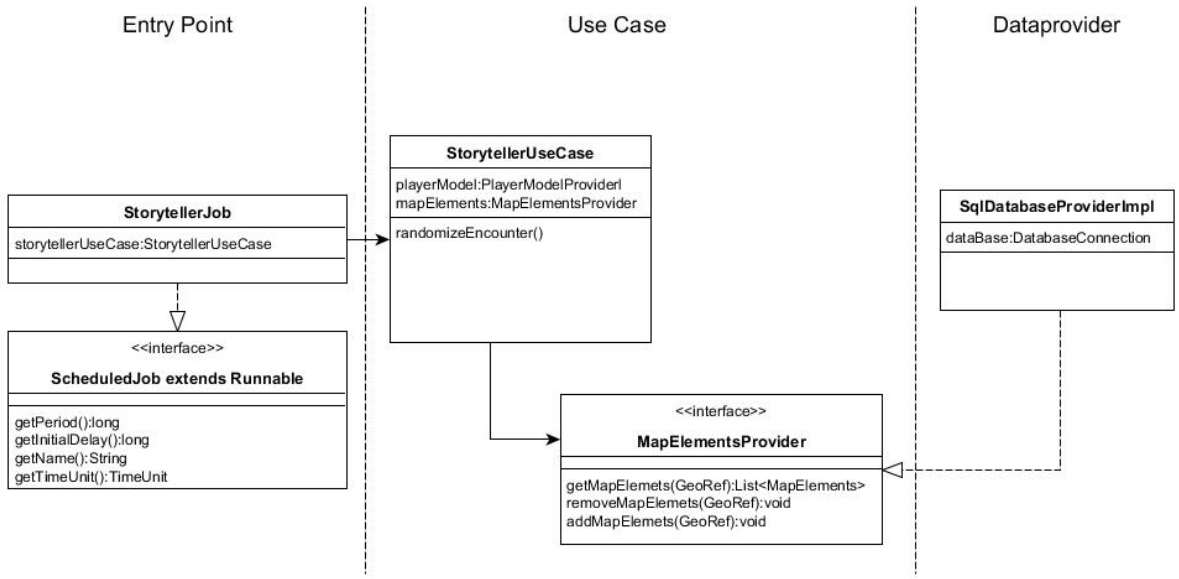

Abbildung 7: Klassendiagramm des Spielelements Erzähler inclusive der Teilprojektgrenzen

In Abbildung 7 ist der Erzähler (*StorytellerJob*) auf der linken Seite zu erkennen. Der Erzähler implementiert ScheduledJob und dieser wiederum erweitert Runnable. Die Klasse besitzt somit eine Methode run(), in welcher der Use Case aufgerufen wird. Der Use Case selbst erzeugt vorerst lediglich randomisiert eine bestimmte Anzahl an Kartenelementen oder verschiebt diese, falls sie bereits vorhanden sind. Der Adapter *MapElementsProvider* ist derselbe wie schon bei der interaktiven Karte. Es werden lediglich weitere Funktionen hinzugefügt, welche dann ebenfalls im *SQLDatabaseProvider* implementiert werden. Damit der Thread laufen kann, muss im Konfigurationsprojekt eine Startmethode geschrieben werden. Demnach startet der Erzähler mit der Anwendung.

5 Umsetzung

5.1 Projektaufbau

Durch die Festlegungen in Kapitel *4.1.5 Aufbau des Systems* sind die grundlegende Projektstruktur und deren Teilprojekte vorgegeben. Nicht festgelegt wurden bisher die zu verwendende Entwicklungsumgebung und die Programmsprache. Die Clean Architecture selbst ist diesbezüglich mit keinen Einschränkungen behaftet. Somit eignet sich jede moderne objektorientierte Sprache, die es erlaubt, eine Netzwerkkommunikation aufzubauen, vernünftige Tests zu schreiben und die darüber hinaus keine größeren Nachteile für die Entwicklung hat.

YARPG wird in diesem Fall mit Java umgesetzt. Es ist derzeit eine der weitverbreitetsten Entwicklungssprachen und kann mittels virtueller Maschine auf jedem System einfach eingesetzt werden. *Vgl. (Hasan, 2018)* Java hat im Hinblick auf dieses Projekt nicht nur keine besonderen Nachteile, sondern bietet vielmehr eine große Gemeinschaft und dementsprechend gepflegte Softwarebibliotheken, die für die Umsetzung bestens geeignet sind. Zudem besitzt der Entwickler für YARPG grundlegende Kenntnisse in dieser Sprache, was den Entwicklungszeitraum gegenüber anderen Sprachen deutlich verkürzt.

Damit ein Softwareprojekt nicht nur von einer Person entwickelt werden kann und nicht an ein bestimmtes Computersetup gebunden ist, bedarf es zwingend eines Projektmanagement-Werkzeugs. Dieses muss unter anderem dazu in der Lage sein, Softwareabhängigkeiten aufzulösen, Teilprojekte zu einem fertigen Projekt zusammenzufügen und für die Entwicklungsumgebung vorzubereiten. Ein solches Werkzeug ist beispielsweise *Gradle*. Durch Konfigurationsdateien kann es angewiesen werden, die zuvor genannten Aufgaben zu erfüllen. So kann das Projekt auf einen anderen Computer überführt und dort an die vorhandenen Gegebenheiten angepasst werden. Dabei werden die virtuelle Maschine festgelegt, Projektpfade angepasst, fehlende Softwarebibliotheken heruntergeladen und schlussendlich notwendige Information in die Projektdatei der Softwareumgebung eingetragen. Mit einem solchen Werkzeug lassen sich außerdem unterschiedliche Entwicklungsumgebungen unterstützen. Über entsprechende Plug-ins unterstützt *Gradle* beispielsweise die beliebten Java-Umgebungen *Eclipse*, *IntelliJ IDEA* und *Netbeans*. Für das Projekt YARPG fällt die Wahl auf *Eclipse*, da es die vom Entwickler präferierte Umgebung ist.

Die vorerst letzte wichtige Grundsatzentscheidung ist die Verwendung eines geeigneten Versionierungswerkzeugs. Dieses sollte nicht nur die Kooperation zwischen Entwicklern gewährleisten, sondern auch die Möglichkeit bieten, den Verlauf des Projekts und den Grund einer Änderung nachvollziehen zu können. Hierfür bietet sich zum Beispiel *Git* an. Das Werkzeug unterstützt die zuvor genannten Anforderungen und ist zudem kostenfrei.

Die Umsetzung erfolgt somit in Java, mittels des Projektmanagementwerkzeugs *Gradle* und der Entwicklungsumgebung Eclipse, versioniert durch *Git*.

Server-Projektordner

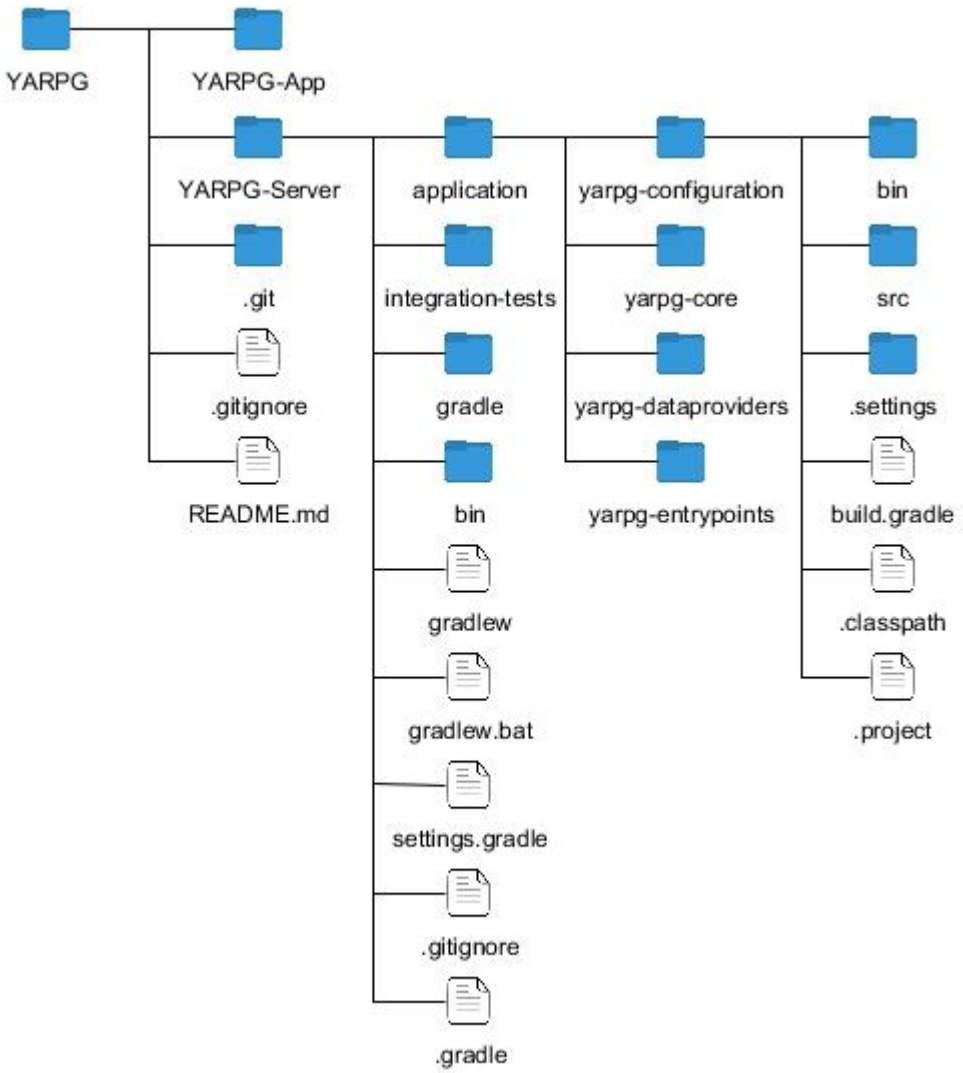

Abbildung 8: Ordnerstruktur des YAPRG-Server Projekts

Wie in Kapitel *4.1.5 Aufbau des Systems* festgelegt gibt es die folgenden Teilprojekte: Core, Entry Point, DataProvider, Configuration und IntegrationTest.

Wie in Abbildung 8 zu sehen, befinden sich diese Teilprojekte im Ordner *application*. Parallel dazu existiert ein Ordner mit den Integration Tests. Dieser Ordner ist bewusst separat platziert. Zum einen wird dadurch deren übergreifende Bedeutung sichtbar und zum anderen sind Anwendung und Integration Tests dadurch eindeutig voneinander getrennt. Die Unit Tests wiederum befinden sich in den Unterordnern der Teilprojekte. Das fertige Produkt kann selbstverständlich ohne diese Tests ausgeliefert werden. Dies lässt sich ohne Schwierigkeiten in *Gradle* einstellen.

Da sowohl App als auch Server im selben Repository gespeichert werden sollen, muss es dazu einen gemeinsamen Ordner geben, in dem sich die *Git*-relevanten Dateien befinden. Mittels der *Gradle*-Konfigurationsdateien *build.gradle* und *settings.gradle* werden schlussendlich die einzelnen Abhängigkeiten der Teilprojekte und andere Eigenschaften definiert.

Wie in Kapitel *4.1.2 Diskussion möglicher Architekturen* beschrieben erfolgt eine Trennung zwischen der Darstellung auf einem Mobilgerät oder PC und der Datenspeicherung sowie der Verarbeitung auf einem geschützten Server. Dies spiegelt sich auch in der Ordnerstruktur wider. Der Projektordner YARPG enthält daher auf der obersten Ebene die beiden Projekte für Server und App samt der zugehörigen *Git*-Projektdateien. Der Ordner *.git* beinhaltet das versionierte Repository. Die Datei *.gitignore* ist eine Konfigurationsdatei, um Dateien mit bestimmten Namen oder Endung aus der Versionierung auszuschließen. Die Datei *Readme.md* ist eine optionale Textdatei. Sie enthält üblicherweise grundlegende Information und Anweisungen zum Projekt. Der Inhalt dieser Datei wird auf der *Git*-Webseite des Projekts angezeigt und kann somit genutzt werden, um sich einen Überblick über das Projekt zu verschaffen.

Der Ordner *YARPG-Server* enthält neben den bereits erwähnten Projektordnern *application* und *integration-tests* auch den Projektordner *gradle* sowie alle dazugehörigen Dateien. Die Dateien *gradlew* und *gradlew.bat* sind wichtig, um das Projektwerkzeug entweder auf einem Linux- oder einem Windows-System nutzen zu können.

Außerdem befindet sich im Ordner *YARPG-Server* eine weitere *.gitignore*-Datei, um bestimmte Dateien auszuschließen, die möglicherweise im *YARPG-App* Projekt versioniert werden müssen. Im Ordner *bin* werden die *gebauten* Projektdateien gespeichert, welche generell nicht versioniert werden.

Das Teilprojekt *yaprg-configuration* dient in der Abbildung 8 als Beispiel für den Aufbau aller Teilprojekte, inklusive des Integration Tests. So beinhaltet jedes dieser Projekt eine Datei namens *build.gradle*, in der die Abhängigkeiten und die nötigen Plug-ins definiert werden. Die Dateien *.classpath* und *.project* sowie der Ordner *.settings* sind *Eclipse*-spezifische Dateien, die von Gradle angelegt werden. Sollte eine andere Entwicklungsumgebung genutzt werden, können Inhalt und Dateiendung variieren. Die zu schreibenden Java-Klassen werden im Ordner *src* mit entsprechenden Unterordnern angelegt, dazu mehr in den nachfolgenden Kapiteln.

Um das Projekt mit *Gradle* für *Eclipse* vorzubereiten, muss über die Kommandozeile (Konsole, Terminal, Windows shell) im Ordner *YAPRG-Server* das *Gradle*-Kommando mit zugehörigen Befehlen aufgerufen werden. Für Linux kann dies wie folgt aussehen:

```
./gradlew cleanEclipse eclipse
```

Dieses Kommando reinigt zunächst den Projektordner von alten *Eclipse*-spezifischen Dateien und legt diese in aktualisierter Form neu an. Diese Befehlskette ist jedoch nur dann notwendig, wenn alte Dateien vorhanden sind. Andernfalls genügt das Kommando *eclipse*. Auf diese Weise werden die Teilprojekte in *Eclipse*-Projekte überführt, können in *Eclipse* importiert werden und die Entwicklung kann starten.

5.2 Entities und Use Case

In Kapitel *4.2 Interaktive Karte* ist eine Auswahl notwendiger Entities vorgegeben. Bei diesen handelt es sich um sogenannte Plain Old Java Objects (POJO), also um gewöhnliche Klassen ohne spezifische Restriktionen bezüglich des Pfads oder der verwendeten Technologie. Somit lassen sich die Klassen ohne Annotation oder zusätzliche Bibliotheken einfach anlegen.

Im Teilprojekt *yaprg-core* sind zwei src Pfade angelegt, *src/main/java* und *src/test/java*, die in den Projekteinstellungen als sogenannte *Source Folder* eingetragen werden. Dieser Vorgang wird für alle anderen Teilprojekte im *application*-Ordner wiederholt. In diese *Source-Folder* werden die sogenannten *packages* eingefügt, also die gesamte Ordnerstruktur der Klassen aus der Entwicklungsumgebung.

Damit die Quellabhängigkeit bei der Verwendung einer Klasse in der späteren Entwicklung bereits beim Import auffällt, erhalten alle Klassen den Namen des entsprechenden Teilprojekts in der Paketbezeichnung, zum Beispiel *com.yarpg.core.entitiy*. Der Präfix *com.yarpg* wurde gewählt, um einen eindeutigen Pfad für dieses Spiel zu erhalten und wird innerhalb alle Teilprojekt verwendet. Die Eindeutigkeit des Pfades hilft nicht nur, Kollisionen von gleichnamigen Klassen zu verhindern, sondern ermöglicht auch das leichte Filtern oder Suchen durch Projektwerkzeuge beziehungsweise Skripte, die im späteren Projektverlauf gegebenenfalls zum Einsatz kommen.

Nach dem Anlegen der Entities können bereits die Use Cases *MovePlayer* und *Storyteller* mitsamt Adapter geschrieben werden, da alle notwendigen Abhängigkeiten vorhanden sind. Anzumerken ist hier, dass weder eine Klasse im Konfigurationsprojekt existiert noch festgelegt wurde, wie die Entities überhaupt erzeugt werden. Dennoch ist möglich, den Use Case vollständig abzubilden. Dafür werden Adapter erzeugt, die den Use Case mit notwendigen Daten füllen sollen. Im Beispiel des *MovePlayerUseCases* existieren die Adapter *PlayerModelProvider* und *MapElementsProvider*. Diese sind lediglich Interfaces, das heißt, sie beinhalten keinerlei Implementierung, sondern nur Festlegungen in Bezug auf die Methodenbezeichnungen und Rückgabewerte. Mittels dieser Methoden kann der Use Case *MovePlayer* bereits vollständig geschrieben und durch Unit Tests auch bereits vollständig getestet werden. Dieses Beispiel verdeutlicht sehr gut die Stärke der Clean Architecture. Die Projektstruktur ist noch nicht einmal zur Hälfte fertiggestellt, um die Anwendung starten zu können. Dennoch ist es möglich, sich bereits mit dem Inhalt, also dem Kern der Anwendung, zu beschäftigen und diesen auch zu testen.

Die Adapter müssen für den Unit Test *gemockt* werden, das bedeutet, dem Compiler wird vorgetäuscht, es gäbe ein initialisiertes Objekt dieses Typs, wobei jede aufgerufene Methode dieses Typs festgelegte Rückgaben benötigt. Auf diese Weise kann die zu testende Methode mit Daten gefüllt werden, den Use Case durchlaufen und zu einem Ergebnis führen. Dies steht im Kontrast zu üblichen Projektstrukturen, in denen Entities von Softwarebibliotheken abhängig sind oder Annotationen besitzen, die das Erzeugen verkomplizieren können. Analog zum *MovePlayerUseCase* kann auch der *StorytellerUseCase* erzeugt und getestet werden.

Somit ist das Teilprojekt *yarpg-core* bereits fertig und muss nachfolgend nicht mehr verändert werden, es sei denn die Use Cases sollen noch einmal angepasst werden.

Die Paketstruktur sieht zu diesem Zeitpunkt wie folgt aus:

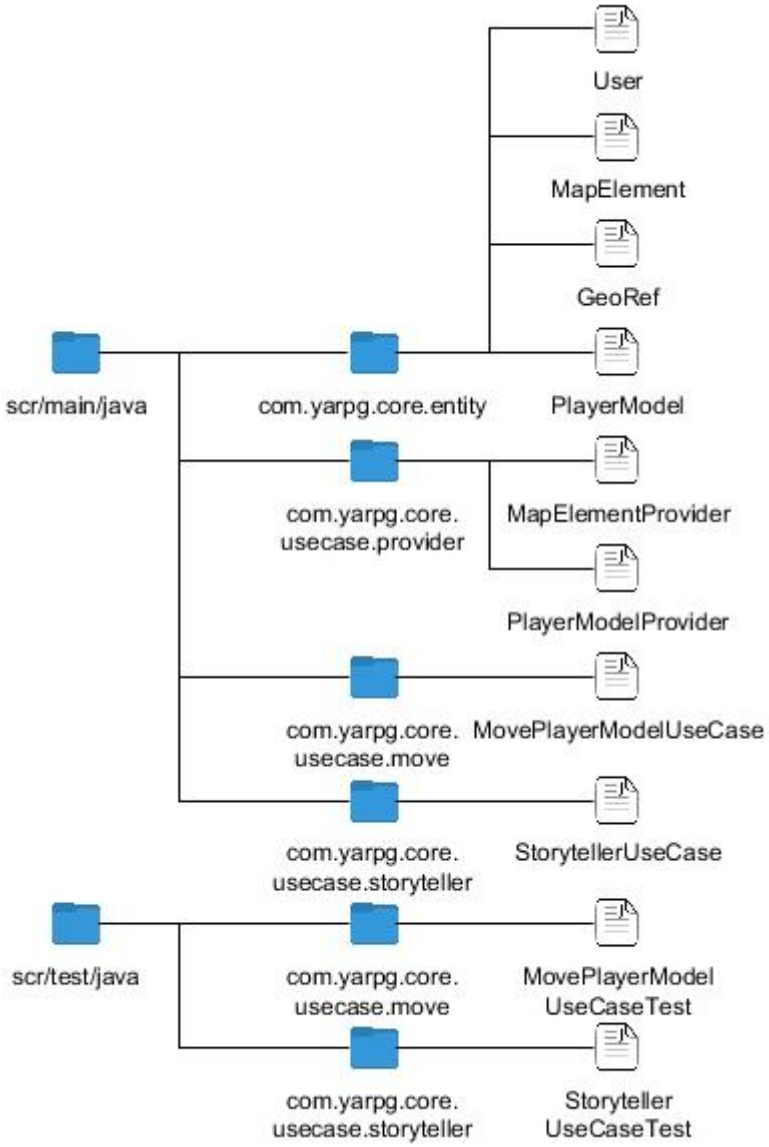

Abbildung 9: Ordnerstruktur der Java-Klassen im Core-Projekt

Die Unit Test Pakete für die Use Cases befinden sich im Ordner *scr/test/java* und sind nach den getesteten Klassen benannt. So können die Tests auch private Methoden der Pakete aufrufen, sind über den SourceFolder einfach zu identifizieren und lassen sie sich später aus dem durch *Gradle* durchgeführten Erstellungsprozess ausschließen. Die Abhängigkeiten und Inhalte der einzelnen Klassen sind identisch zu den in Kapitel *4.2 Interaktive Karte* und *4.3 Erzähler* festgelegten Strukturen. Die Implementation dieser Methoden ist trivial. Sie wird daher im Rahmen dieser Masterarbeit nicht erläutert.

5.3 Data Provider

Die Implementation der Adapter ist von der zu verwendeten Datenbank abhängig und kann stark variieren. Für dieses Projekt genügt zur Veranschaulichung eine einfache In-Memory-SQL-Datenbank. Die YARPG-Entities ließen sich leicht in separate Tabellen unterteilen, so zum Beispiel in eine Tabelle für die Georeferenz. Dies ist jedoch gar nicht notwendig. Es genügen wie folgt die Tabellen *User*, *Player_Model* und *Map_Elements*:

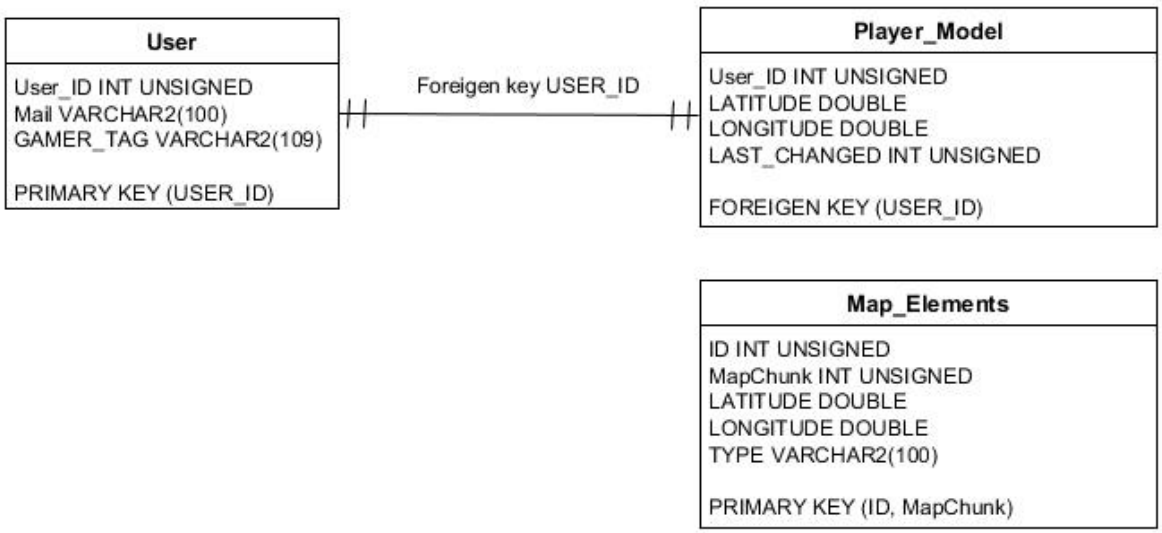

Abbildung 10: Datenbankmodel

Es wäre auch möglich, *User* und *PlayerModel* zusammenzufassen, da sie ohnehin in einer Eins-zu-Eins-Beziehung zueinanderstehen. Allerdings gewähren mehrere miteinander verbundene Tabellen eine bessere Übersichtlichkeit als wenige nicht verbundene. Andererseits entspräche ein Zusammenschluss dieser Tabellen der Clean Architecture, denn die Entities existieren unabhängig von der Datenbank. So sind die Objekte zwar durch Klassen getrennt, könnten jedoch in der Datenbank als eine Information geführt werden. Das Beispiel zeigt letztlich, dass sich Struktur und Aufbau der Datenbank beliebig wählen lassen.

Die in Abbildung 10 gezeigte Struktur muss durch eine SQL-Datei definiert werden. Diese Datei wird beim Starten der Anwendung von der Konfiguration geladen und die Datenbank entsprechend initialisiert. Die Provider-Implementation arbeitet mit einem Objekt, das auf die so erstellte In-Memory-Datenbank zugreifen kann. Die einzelnen zu implementierenden Methoden müssen mittels SQL-Anweisungen auf die entsprechenden Tabelleneinträge zugreifen.

Theoretisch ist es sinnvoll, auch die Provider-Implementation durch Unit Tests abzudecken. Dies sollte jedoch nur dann erfolgen, wenn die Datenbankanbindung umfangreicher ist und nicht der gesamte Quellcode dafür *gemockt* werden muss.

Das Projekt *yaprg-dataprovider* erhält für die SQL-Datei einen weiteren Source Folder unter *src/main/ressource/*. Die Implementation des Providers kommt hingegen in das Verzeichnis *src/java/main/* und trägt die Paketbezeichnung *com.yarpg.dataprovider.jdbc.*

Somit enthält das Teilprojekt *Dataprovider* lediglich zwei Dateien und nur eine davon ist eine Java-Klasse.

5.4 Entry Point

Für die beiden Use Cases *MovePlayerModel* und *Storyteller* müssen zwei unterschiedliche Einstiegspunkte existieren. Der eine ist eine REST-Schnitstelle, die eine URL implementiert und als solche auch von einem Webserver angeboten werden muss. Die andere ist lediglich ein Thread, der zeitgesteuert immer wieder dieselbe Funktion anstößt.

REST-Controller

Um einen REST-Controller anbieten zu können, braucht es eine Softwarebibliothek, die den Webserver starten und den Controller entsprechend seiner URL eintragen kann. Eine solche Implementierung wird durch *Spring Frameworks* wie beispielsweise *Spring Boot* bereitgestellt. Die Softwarebibliothek ermöglicht weitaus mehr als nur das Anbieten von URLs durch einen Webserver. Im Zuge der Clean Architecture soll die Verwaltung der Entities jedoch nicht durch externe Quellen übernommen werden. Somit wird nur ein kleiner Teil von *Spring* verwendet. Trotzdem ist die Verwendung einer solchen Bibliothek vorteilhaft, denn sie kann später leichter durch eine andere ausgetauscht werden, sollte sie den Anforderungen nicht mehr genügen.

Spring Boot unterstützt die Annotation *@RestController*. Sie wird vor die Klassendefinition gesetzt, während des Starts der Anwendung durch *Spring* gesucht und entsprechend konfiguriert. Darüber hinaus steht die Annotation *@RequestMapping* zur Verfügung. Sie wird zu jeder Methode hinzugefügt, die als URL angeboten werden soll. Im Use Case *MovePlayer-Model* werden dadurch der Pfad und das Übertragungsprotokoll GET festgelegt. Für den Rückgabetyp wird ein neues DTO-Objekt angelegt, das durch *Spring* automatisch in eine

JSON-Zeichenfolge konvertiert und übermittelt wird. Das DTO ist ein POJO, das heißt, für diesen Vorgang werden keine weiteren Annotationen benötigt.

Somit entstehen für den REST-Controller zwei Klassen, zum einen der Controller selbst und zum anderen das DTO-Objekt. Darüber hinaus muss der *Spring Context* in der Konfiguration erstellt werden, um den Webserver zu starten (siehe Kapitel *5.5 Konfiguration*).

Erzähler

Der Erzähler ist im Grunde nur eine Implementation der Klasse *Runnable*, die standardmäßig Teil der Java-Umgebung ist. Zusätzlich zum Erzähler wäre es denkbar, noch weitere Jobs neben der laufenden Anwendung auszuführen. Dadurch ließe sich beispielsweise die Datenbank gelegentlich aufräumen oder ähnliches. Um konsistent zu bleiben, wäre es zudem sinnvoll, das Verwalten solcher Jobs der Konfiguration zu überlassen. Aus diesem Grund wird ein Interface *ScheduleJob* erstellt, das neben den grundlegenden Eigenschaften wie *Name* oder *Periode* auch die Klasse *Runnable* erweitert. Der Erzähler, als Beispiel eines solchen Jobs, implementiert das Interface und muss neben den dortigen Methoden auch die Methode *run()* der Klasse *Runnable* implementieren. In dieser Methode kann der Use Case für den Erzähler aufgerufen werden. Ein solcher Job muss jedoch nicht wie beim Erzähler einen eigenen Use Case besitzen. Es ist auch durchaus denkbar, Logik in den Job zu integrieren und diverse Use Cases zu nutzen. Unerwünscht ist dagegen, wie beim REST-Controller, der direkte Zugriff auf die Datenbank.

Auch hier entstehen vorerst zwei Klassen, eine für das Interface und eine zweite für dessen Implementation. Der Inhalt ist dabei identisch zu den in Kapitel *4.3 Erzähler* definierten Klassen.

5.5 Konfiguration

Die Aufgabe im Rahmen der Konfiguration besteht darin, dass die Klassen initialisiert werden müssen. Für den Konstruktor der Klasse *MovePlayerModelUsecase* werden daher beispielsweise Instanzen der Adapter benötigt, die wiederum die Datenbankverbindung benötigen. Damit auf die so erstellten Objekte zugegriffen werden kann, müssen sie überall in der Anwendung bekannt gemacht werden. Für diese Aufgabe kann *Spring Bean* genutzt werden.

Spring verwendet dafür sogenannte *Inversion of Control Container*. Diese lösen die Abhängigkeiten der verketteten Klassen auf und erzeugen aus den Klassen Instanzen. Für das Projekt YARPG werden demnach Konfigurationsklassen mit der Annotation *@Configuration* erzeugt. Diese wiederum beinhalten Methoden, die mit der Annotation *@Bean* gekennzeichnet sind. Für jede Klasse, die durch *Spring* erzeugt werden soll, muss es eine Methode geben, welche die entsprechende Klasse als Rückgabewert besitzt. Zum derzeitigen Stand sind das die Klassen *MovePlayerModelUseCase*, *MovePlayerModelRestController*, *StorytellerUseCase*, *SQLDataProviderImpl* und *Storyteller*. Zum Instanziieren der Klassen werden üblicherweise andere Klassen benötigt, die noch nicht erzeugt wurden. Diese werden ganz einfach als Parameter der Methode übergeben. Dadurch entsteht eine Verkettung, die durch *Spring* aufgelöst werden muss. *Spring* beginnt dabei mit denjenigen Klassen, die keine Abhängigkeiten benötigen. Eine solche Methode sieht beispielsweise wie folgt aus:

```
@Configuration
public class UseCaseConfiguration {

    @Bean
    public MovePlayerModelUseCase getMovePlayerModelUseCase(PlayerModelProvider playerModelProvider,
            MapElementsProvider mapElementsProvider) {
        return new MovePlayerModelUseCase(playerModelProvider, mapElementsProvider);
    }
}
```

Abbildung 11: Quellcodeausschnitt: zur Konfiguration eines Use Cases

Beim Versuch, alle zuvor genannten Klassen auf diese Weise zu konfigurieren, fallen jedoch drei Probleme auf. Die Klasse *SQLDataProviderImpl* benötigt eine Datenbankverbindung in Form eines *JdbcTemplate*-Objekts und der Erzähler muss an irgendeiner Stelle seinen Prozess starten. Außerdem muss die Anwendung angewiesen werden, beim Programmstart *Spring* auszuführen.

5.5.1 Konfiguration der Datenbank

Das Objekt *JdbcTemplate* wird in der Provider-Implementation für die Datenbankzugriffe verwendet. Der Aufbau der Datenbank wurde jedoch bisher nur in einem SQL-Skript festgehalten, das zudem bislang nirgends aufgerufen wird. Um die Klasse *JdbcTemplate* zu erzeugen, wird daher eine *DataSource* benötigt, welche wiederum durch ein *ConnectionString* und mitgelieferte SQL-Skripte erzeugt werden kann. *DataSource* ist somit eins der wenigen Objekte, die keine zusätzliche Parameter benötigen. *Spring* muss demnach auch keine Parameter auflösen. Dadurch kann eine *Bean* für *DataSource* erzeugt werden. Die dafür notwendige Information wird entweder direkt in die Methode implementiert oder aus einer Datei geladen. Die

Methode, um die In-Memory-Datenbank zu erzeugen und als Datenquelle bereitzustellen, sieht dabei wie folgt aus:

```
@Bean
public DataSource createDataSource() throws IOException {
    // in Memory
    String createDatabase = "runscript from 'classpath:/" + SCHEMA_INITIALISATION_SCRIPT + "'";

    StringBuilder connectionStringBuilder = new StringBuilder();
    connectionStringBuilder.append("jdbc:h2:mem:test;MODE=Oracle;INIT=");
    connectionStringBuilder.append(createDatabase);

    String jdbcUrl = connectionStringBuilder.toString();
    String username = "yarpg";
    String password = "";
    return JdbcConnectionPool.create(jdbcUrl, username, password);
}
```

Abbildung 12: Quellcodeausschnitt: der Datenbankkonfiguration

Die Variable *createDatabase* beinhaltet die Anweisung, das in Kapitel *5.3 Data Provider* erzeugte SQL-Skript beim Starten der Verbindung auszuführen. Sofern eine externe SQL-Datenbank zum Einsatz kommt, müsste es in dieser Methode eine Quellcodeweiche geben, um nicht durch einen Neustart die Datenbank versehentlich zu überschreiben. Die Methode *createDataSource* müsste dann so gestaltet werden, dass der *Connectionstring* aus einer Konfigurationsdatei ausgelesen werden kann, um IP, Port, Datenbankname, Benutzername und Passwort bereit zu stellen. Nach dem Erstellen der *DataSource* kann nun die *Bean* für *JdbcTemplate* erzeugt werden, die wiederum in der zuvor geschriebenen *Bean* der Klasse *SQLDataProviderImpl* benötigt wird. Die Abhängigkeitskette ist somit aufgelöst.

Erwähnenswert ist an dieser Stelle, dass die *Bean* mit den Use Cases lediglich die *ProviderAdapterInterfaces* als Parameter nutzt (siehe Abbildung 11 Konfiguration), nicht jedoch eine spezielle Implementation dieser. Trotzdem erkennt *Spring*, dass es ein initialisiertes Objekt gibt, das die Adapter implementiert und diese als Parameter verwendet, in diesem Fall das Objekt *SQLDataProviderImpl*. Existiert hingegen mehr als eine Implementation der Adapter, kann *Spring* die Abhängigkeiten nicht mehr eindeutig auflösen und unterbricht den Startprozess. Soll es mehrere Datenbank-Implementationen geben muss in einer Konfigurationsdatei vor dem Start der Anwendung definiert sein, welche davon initiiert werden soll. Dafür kann es beispielsweise ein Quellcodeweiche geben in der ein Parameter aus der Konfiguration verglichen wird. So wäre es möglich, je nach Startumgebung eine andere Datenbank anzusteuern, die unter Umständen eine völlig andere Implementation der Schnittstellen erfordert.

5.5.2 Konfiguration der ScheduledJobs

Ziel ist es, alle *ScheduledJobs* aufzunehmen und sie einem *ExecutionService* zu übergeben. Wie zuvor die Klassen können die Jobs dazu erzeugt und von einer *Bean* als Parameter *eingesammelt* werden. Allerdings wird in Bezug auf die Jobs mehr als eine Implementation gefunden. Somit kann *Spring* nicht mehr zuordnen, welches Objekt wohin gehört. Dieses Problem kann umgangen werden, sofern die *Bean* eine unbestimmte Anzahl der gleichen Objekttypen als Parameter akzeptiert.

Der Erzähler (Storyteller) und jeder weitere Job wird in einer Bean wie folgt erzeugt:

```
@Bean
public ScheduledJob createStoryteller(StorytellerUseCase storytellerUseCase) {
    return new Storyteller(storytellerUseCase);
}
```

Abbildung 13: Quellcodeausschnitt: Servicekonfiguration anhand des Storytellers

Die so erzeugten *ScheduledJobs* können anschließend wie folgt *eingesammelt* und einem *ExecutionService* übergeben werden:

```
@Bean
public ScheduledExecutorService scheduledExecutorService(ScheduledJob... jobs) {
    ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(5);

    for (ScheduledJob job : jobs) {
        long initialDelay = job.getInitialDelay();
        long period = job.getPeriod();
        TimeUnit timeUnit = job.getTimeUnit();
        LOGGER.info("Scheduling job {} to run every {} {}, after an initial {} {}", job.getName(),
                period, timeUnit, initialDelay, timeUnit);
        scheduledExecutorService.scheduleAtFixedRate(job, initialDelay, period, timeUnit);
    }
    return scheduledExecutorService;
}
```

Abbildung 14: Quellcodeausschnitt: einsammeln und starten aller Services

Die im *ScheduledJob-Interface* festgelegten Methoden werden hier benötigt, um den Job in einem bestimmten Zeitraum ablaufen zu lassen. Der *ScheduledExecutionService* ist ein Java Standard Interface, welches das Starten und Warten der mitgelieferten *Runnables* reguliert. Das Objekt wird durch *Spring Bean* erzeugt, jedoch nirgends von der YARPG-Anwendung als Parameter benötigt. Somit existiert das hier erzeugte Objekt im Inversion of Control Container von *Spring Bean* und verwaltet sich selbst. Es ist also davon auszugehen, dass für den Service selbst ebenfalls ein Thread gestartet wird. Auf diese Weise wird letztlich jeder Job erzeugt, eingesammelt und verwaltet, so zum Beispiel auch der Erzähler.

5.5.3 Starten mit *Spring*

Wichtig ist nicht zuletzt der Einstiegspunkt in die Java-Anwendung. Dessen Aufgabe besteht nicht nur darin, die Anwendung zu starten, sondern vor allem darin den *Spring Context* zu laden. Dafür müssen die Konfigurationsdateien eingesammelt und der Webserver gestartet werden. Im Teilprojekt *yarpg-configuration* wird zu diesem Zweck eine Klasse mit der statischen main()-Methode erzeugt, die mit lediglich einer Zeile auskommt. Diese Zeile startet *Spring* und den zuvor erwähnten Prozess. Die notwendigen Startbedingungen sind als Annotation an die Klasse selbst angefügt. Die Anwendung startet somit über folgende Klasse:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan(basePackages = "com.yarpg.application.configuration")
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Abbildung 15: Quellcodeausschnitt: main()-Methode von YARPG-Server

5.6 Integration Test

Um Integration Tests ausführen zu können, muss eine Umgebung geschaffen werden, in der die entsprechend notwendigen Objekte vorhanden sind. Soll beispielsweise die Datenbank getestet werden, muss die Verbindung zu dieser gestartet werden. *Spring* bietet auch hierfür eine Lösung. So kann die Klasse *AbstractTransactionalJUnit4SpringContextTests* erweitert und mit den nötigen Konfigurationsklassen geladen werden. Da diese Klasse nur für den Test zur Verfügung stehen soll, kann sie im Teilprojekt *integration-test* implementiert werden. Sie steht damit im restlichen Projekt nicht zur Verfügung und ist vor allem kein Teil der fertigen Anwendung. Wäre sie dagegen Teil der fertigen Anwendung, entstünde dadurch ein gewisser Angriffsvektor. Grund dafür sind die zumeist weitreichenden Funktionen, die eine solche Testumgebung benötigt, um Datenbanken oder Objekte manipulieren können.

Um die Datenbank-Implementation testen zu können, muss die entsprechende Klasse generiert werden, die wiederum vom *jdbcTemplate* benötigt wird. Wie in Kapitel *5.5 Konfiguration*

beschrieben wird diese Abhängigkeit durch die Konfigurationsklassen als *Bean* in *DatasourceConfiguration* und *DataProviderConfiguration* bereitgestellt.

Auf diese Weise entsteht eine Klasse, die wie folgt die notwendigen Objekte erzeugt:

```
@ContextConfiguration(classes = { DatasourceConfiguration.class, DataProviderConfiguration.class })
@RunWith(SpringJUnit4ClassRunner.class)
public class DatabaseIntegrationTest extends AbstractTransactionalJUnit4SpringContextTests {
```

Abbildung 16: Quellcodeausschnitt: Datenbankkonfiguration für Integrationtests

Durch *AbstractTransactionalJUnit4SpringContextTests* ist im *Scope* der Klasse *DatabaseIntegrationTest* nun auch die Datenbankverbindung *jdbcTemplate* verfügbar, nicht als *Bean*, sondern als öffentliche Eigenschaft der Klasse. Somit können in der Klasse *DatabaseIntegrationTest* Methoden geschrieben werden, die auf die Datenbank zugreifen. Für den Test könnten beispielsweise die Methoden *createUser* oder *createPlayerModel* zum Einsatz kommen.

Um den Kontext herzustellen, muss eine *Bean* als globale Variable eingebunden werden und die Annotation *@Autowire* erhalten. Damit kann die eigentliche Testklasse *DatabaseIntegrationTest* erweitern und erhält über genanntes *Autowire* das Objekt. Die so entstandene Testklasse sieht wie folgt aus:

```
public class SqlDataProviderImplIntegrationTest extends DatabaseIntegrationTest {

    @Autowired
    SqlDataProviderImpl _sqlDataProviderImpl;

    @Before
    public void setUp() throws Exception {
        cleanUpDatabase();
    }

    @Test
    public void getPlayerModel_byId() {
```

Abbildung 17: Quellcodeausschnitt: Aufbau eines Integrationtests

Der Test sei hier nur als Methode angedeutet. Im Rahmen dieser Methode kann die Datenbank-Implementation verwendet werden. Eine Klasse, welche die gesamte Implementation der Datenbank testen soll, wird schnell unübersichtlich. Daher ist zu empfehlen, für jeden Adapter eine separate Testklasse anzulegen, in der lediglich die Methoden des entsprechenden Adapters getestet werden. Somit entsteht eine Vielzahl an Klassen. Jede dieser Klassen erweitert die Klasse *DatabaseIntegrationTest* und erhält somit Zugriff auf die Datenbank. Häu-

fig gebrauchte Funktionen zum Erstellen von Daten sollten daher auch Teil von *DatabaseIntegrationTest* sein. Auf die gleiche Weise lassen sich projektübergreifende Tests schreiben. Sie benötigen dafür lediglich die passende Konfiguration.

5.7 Der View-Programmteil als App

Der View-Programmteil für YARPG soll wie bereits erwähnt mit *Adobe PhoneGap* erstellt werden. Dabei handelt es sich vereinfacht gesagt um einen Browser, der sich an die verschiedenen Betriebssysteme der Mobilgeräte anpassen lässt.

Ein *PhoneGap*-Projekt besteht hauptsächlich aus Plug-ins sowie einigen Konfigurationsdateien und dem Ordner für die Webseite. Die Plug-ins beinhalten jeweils die Implementationen für die verschiedenen Plattformen, für welche die Anwendung *übersetzt* werden kann. Da bereits alle grundlegenden Funktionen wie Kamera, GPS, Dateisystem und ähnliches vorhanden sind und somit allen Plattformen zur Verfügung stehen, sinkt der Entwicklungsaufwand enorm.

Im JavaScript der entstehenden Webseite kann daher auf bereits initialisierte Variablen der Plug-ins zurückgegriffen werden, um so Zugriff auf das entsprechende Feature des Mobilgeräts zu erhalten. Somit besteht lediglich die Aufgabe, eine Webseite zu erstellen, die das Spiel darstellt und Funktionen zur Interaktion mit dem Server beinhaltet. Dazu zählen die Darstellung einer Karte mit Spielerposition und das Anzeigen von Kartenelementen, mit denen sich interagieren lässt.

5.7.1 Karte und Bedienelemente

Da der View-Programmteil losgelöst vom Server existiert und lediglich über REST-Calls arbeitet, lässt sich die Darstellung frei wählen und vor allem auch zu einem späteren Zeitpunkt beliebig verändern. Die nachfolgend diskutierte Darstellung soll dazu dienen, existierende Funktionalitäten des Servers aufzuzeigen und mit diesen zu arbeiten. Als Beispiel dienen die vom Erzähler zufällig erzeugten Monster, die als Kartenelemente lediglich rund um die Position des Spielers angezeigt werden sollen.

Damit keine laufenden Kosten anfallen, vor allem nicht in diesem frühen Entwicklungsstadium des Spiels, wird auf eine lizenzfreie Karte gesetzt. Eine gute Wahl dafür ist *OpenStreetMap*

(OSM). Diese bietet nicht nur entsprechendes Kartenmaterial, sondern gleichzeitig auch eine JavaScript-Bibliothek, um die Karte im Sinne des Spiels nutzen zu können. Des Weiteren soll es Bedienelemente für GPS und das Optionsmenü geben. Diese Elemente sind lediglich durch CSS formatierte HTML-Objekte mit entsprechender Grafik und einem onClick-Event. Die nachfolgende Abbildung zeigt einen Screenshot der so entstandenen Darstellung. Oben rechts ist der Button für die Optionen zu sehen, unten rechts der für GPS. Das kleine blaue Plus oben links kann genutzt werden, um Kartenelemente ein- und auszublenden.

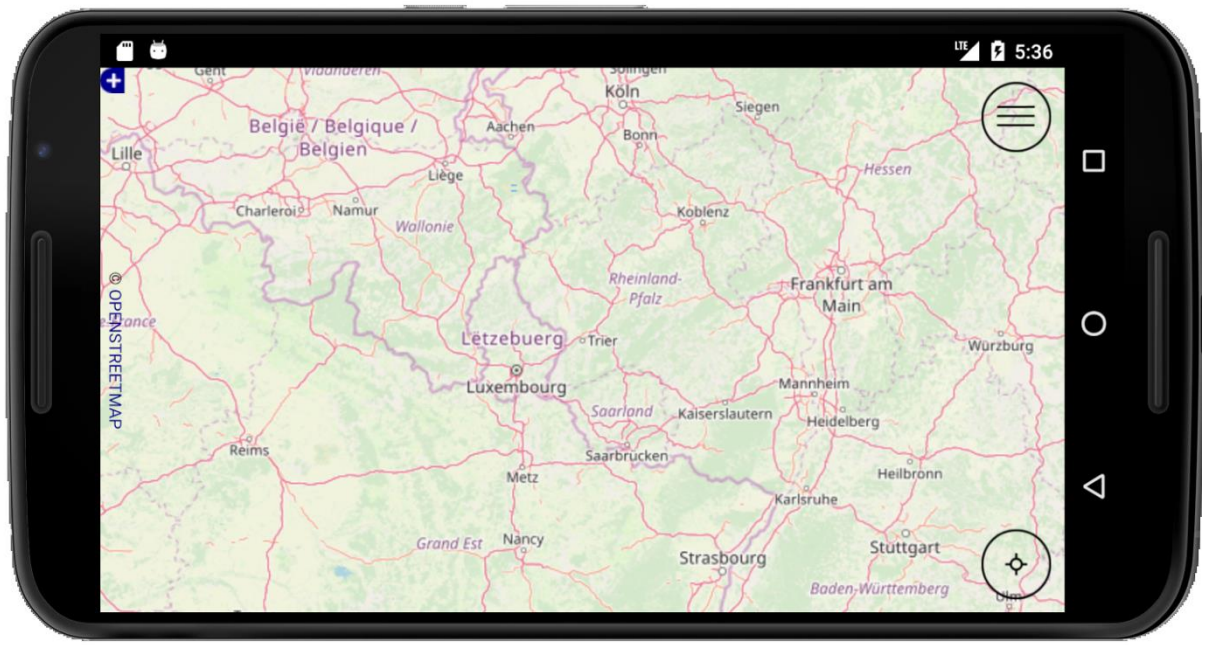

Abbildung 18: YAPRG-App: Kartenansicht mit Steuerelementen

5.7.2 GPS und Serverkommunikation

Der GPS-Button startet das Auslesen der Position des Mobilgeräts durch ein Plug-in und sendet diese Information mittels HTTP-Request an den Server. Die darauffolgende Antwort des Servers besteht aus den Kartenelementen, die sich im Umkreis des Spielers befinden. Mittels der OSM-JavaScript-Bibliothek können diese Kartenelemente in einer dafür angelegten Ebene angezeigt werden. Die rote Markierung stellt dabei die Position des Spielers dar, die Pfoten symbolisieren die Kartenelemente für die Monster. Jedes der so angezeigten Elemente erhält ein onClick-Event, um mit ihnen interagieren zu können, das bedeutet, der Nutzer kann diese Elemente antippen und erhält daraufhin via Pop-up weitere Information oder Optionen. Eine solche Option könnte beispielsweise ein Kampf, eine Aufgabe oder die Möglichkeit zum

Handel sein. Da die Implementierung der entsprechenden Use Cases den Rahmen dieser Masterarbeit sprengen würde, zeigt das Pop-up in der nachfolgenden Abbildung lediglich den Namen des Elements an. Dies soll jedoch genügen, um die Interaktionsmöglichkeit exemplarisch zu demonstrieren.

Abbildung 19: YAPRG-App: Kartendarstellung mit Spielerposition und Monstermarker sowie Kontextfenster

5.7.3 Optionen der App

Der Optionen-Button startet eine Plattform-spezifische Ansicht mit zuvor definierten Parametern, das bedeutet, der Nutzer befindet sich nicht mehr auf der Webseite, sondern in einem Menü des Mobilgeräts. Die dort getätigten Einstellungen werden gespeichert und sind auch nach dem Schließen und erneuten Öffnen der Anwendung noch aktiv. Die nachfolgende Abbildung zeigt das Menü in einer *Android*-Umgebung. Zu sehen ist unter anderem die Option *Track*. Sofern diese aktiviert ist, wird die Position des Spielers verfolgt. Der Spieler ist dabei mittig auf dem Bildschirm ausgerichtet und um den Spieler herum wird das entsprechende Kartenmaterial geladen. Über die Option *GPS-Timeout* lässt sich für den Fall einer ergebnislosen Positionsabfrage festlegen, nach welcher Zeit die GPS-Funktion deaktiviert werden soll. Über den Menüeintrag *Information* erhalten Nutzer weiterführende Information über die Anwendung.

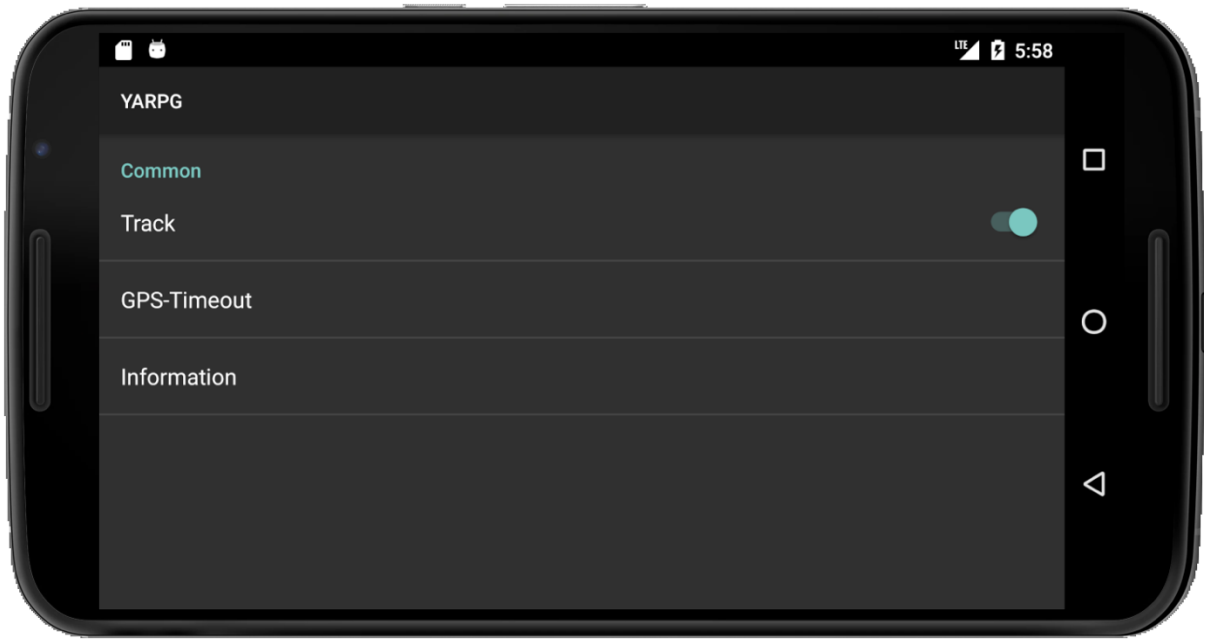

Abbildung 20: YAPRG-App: Konfigurationsansicht

5.8 Bauen und Starten der Anwendung

Da View und Server zwei voneinander vollkommen getrennte Anwendungen sind, müssen diese auch separat gebaut und gestartet werden. Im Falle des Servers kann dies auf jedem Computer erfolgen, der *Gradle* und zugehörige Werkzeuge unterstützt. Die App hingegen ist diesbezüglich stärker limitiert. So kann auf einer *Linux*-basierten Umgebung für *iOS* und *Android* gebaut werden, nicht jedoch für *Windows*. Grund ist eine dafür notwendige Datei, die nur *Windows* öffnen kann. Das gleiche Problem gibt es für *iOS* auf einem *Windows*-PC. Soll die Anwendung für alle Plattformen gebaut werden, sind also immer zwei unterschiedliche Betriebssysteme notwendig. Die Quelldateien für *Android* und *Windows* sind kostenlos, für iOS wird hingegen eine Entwicklungslizenz verlangt, um auf diese zugreifen zu können. Im Rahmen dieser Masterarbeit wurde das Bauen der App exemplarisch lediglich für Android realisiert.

5.8.1 YARPG-Server

Im Serverprojekt existieren zwei Varianten, die Anwendung zu bauen und zu starten. Die erste Variante ist wichtig für die Entwicklung. Der Server kann damit aus der Entwicklungsumgebung heraus gestartet werden. Änderungen lassen sich auf diese Weise schnell überprüfen. Das Bauen der Quelldateien übernimmt dabei die Entwicklungsumgebung automatisch. Wie in Kapitel *5.1 Projektaufbau* beschrieben ermöglicht es *Gradle*, die Quelldateien in eine für die Entwicklungsumgebung verständliche Projektform zu bringen, sodass diese die Anwendung übersetzen kann. Da im Teilprojekt *yarpg-configuration* eine Main-Methode existiert, kann diese auch zum Starten genutzt werden.

Die zweite Variante, um den Server zu starten, ist das Bauen der Jar-Datei durch *Gradle*. Dabei wird keine Entwicklungsumgebung benötigt, sondern lediglich ein Befehl auf der Kommandozeile. Da die Abhängigkeiten der Teilprojekte in *Gradle* definiert sind, können diese dort auch aufgelöst werden. Die so entstandene JAR-Datei kann über die virtuelle Maschine des jeweiligen Computers gestartet werden. Im Rahmen einer künftigen Weiterentwicklung soll es diesbezüglich noch eine Konfigurationsdatei geben, um Datenbank, Port und ähnliches vor dem Starten festlegen zu können. Diese Konfigurationsdatei wird dann während des Startvorgangs beziehungsweise beim Konfigurieren der *Beans* gelesen und entsprechend angewendet. Ist der YARPG-Server wie zuvor beschrieben gestartet, steht der Webserver und dessen REST-Calls für den View-Programmteil zur Verfügung.

5.8.2 YARPG-App

Auch für die App existieren zwei wichtige Varianten, um das Projekt zu starten. *PhoneGap* kann entweder ein App für die verschiedenen Mobilgeräte bauen oder einen Webserver auf der Entwicklungsumgebung starten. Ersteres wird in *PhoneGap* als Plattform *Android*, *iOS* oder *Windows Phone* bezeichnet, letzteres als Plattform *Browser*.

Im Falle der Plattform *Browser* startet *PhoneGap* selbst einen Webservice, der die Website und die Plug-ins beinhaltet. Einige dieser Plug-ins sind jedoch für diese Plattform nicht fertig implementiert. Das betrifft zum Beispiel das Plug-in für die Optionen, womit diese Möglichkeit derzeit noch entfällt.

Im Falle der Plattform *Android* kann die gebaute App in einem Emulationsprogramm verwendet werden. Allerdings muss das Emulationsprogramm separat konfiguriert werden, da es kein Bestandteil von *PhoneGap* ist. Für die Auslieferung der App stellt *PhoneGap* einen Befehl zum Bauen zur Verfügung. So lässt sich für jedes Zielsystem eine entsprechende App-Datei erstellen. Diese kann auf ein entsprechendes Mobilgerät geladen und dort gestartet werden. Das Mobilgerät muss sich dafür jedoch im Entwicklermodus befinden, da die App andernfalls vom System geblockt wird. Das Hochladen in die entsprechenden Shops der Hersteller wurde im Rahmen dieses Projekts nicht getestet.

Damit die App mit dem Server kommunizieren kann, müssen sich beide in derselben Netzwerkumgebung befinden. Entweder ist der Server über eine URL oder IP-Adresse im Internet erreichbar oder beide befinden sich in einem lokalen Netzwerk. Bis dato wurde das Projekts lediglich gegen das Emulationsprogramm entwickelt, das bedeutet, es wurde eine feste IP-Adresse verwendet. Bevor die App ausgeliefert werden kann, muss also die URL zum Server bekannt sein und entsprechend eingetragen werden. Alternativ könnte dies auch als Option hinzugefügt werden. Dann ließe sich die URL später einfach ändern.

6 Auswertung

In den Grundlagen zur Softwarearchitektur wurden acht Punkte genannt, die eine gute Softwarearchitektur ausmachen. Diese sollen nun im Nachgang noch einmal bezüglich des entstandenen Projekts YARPG betrachtet werden. Da die meisten dieser Punkte jedoch schwer oder gar nicht zu messen sind und das Projekt als solches nicht vollständig vorliegt, muss hier auf die Erfahrung aus der Entwicklung zurückgegriffen werden, um eine Tendenz zu erkennen. Anschließend soll ein Fazit gezogen sowie ein Ausblick gegeben werden, welche Bereiche in der Praxis noch zu ergänzen wären.

Das Projekt besteht derzeit nur aus einem Grundgerüst der verschiedenen Softwarekomponenten. Es wurde jedoch darauf geachtet, die Spielelemente so auszuwählen, dass jede Architekturkomponente mindestens einmal implementiert werden musste. Somit entstanden zwei verschiedene Einstiegspunkte, einer für den View-Programmteil und ein zweiter für den parallel ablaufenden Erzähler-Thread. Zudem wurden geeignete Use Cases mit entsprechenden Datenbankabfragen entwickelt.

Funktional:

Eine Architektur kann als gescheitert gelten, wenn unabhängig von allen anderen Punkten die eigentliche Funktion der Software nicht erfüllt wurde. Von einem solchen Scheitern kann jedoch nicht ausgegangen werden. Andernfalls hätten sich die Clean Architecture und andere kaum durchgesetzt. Die Frage besteht also nicht darin, ob alle Funktionen umsetzbar waren, sondern vielmehr darin, wie aufwendig oder umständlich der gewählte Aufbau gewesen ist.

Schon sehr früh während der Planungsphase des ersten Use Case wurde klar wie einfach eine Anwendung aufgebaut werden kann, wenn mit POJO gearbeitet wird. Durch eine nahezu freie Wahl der Entities, die auf den Spielelementen basieren, war auch deren Verwendung eindeutig. So konnte der erste Use Case entwickelt werden, ohne dass andere Rahmenbedingungen gesetzt waren. Dabei ergaben sich auch klare Vorgaben, welche Datenbankabfragen existieren müssen. Von der Planung der Entities über die Konstruktion eines Use Cases wurden also die dafür notwendige Datenbankabfragen als solche vorgegeben. Jede weitere Funktion, die als Use Case dargestellt werden musste, fügte sich gleichermaßen in dieses Schema ein. Die bereits entstandenen Quellcodepfade behinderten sich dabei nicht, sondern konnten sogar zum Teil wiederverwendet werden. Es war somit eine enorme Vereinfachung,

Spielelemente unabhängig von View, Entry Point oder Datenbank zu entwickeln. Die so entstandenen Adapter und deren für die Use Cases notwendigen Methoden stellten keine Hürde dar. Eine größere Herausforderung war die Konfiguration des Projekts selbst, da die Nutzung des *Spring Frameworks* eine gewisse Einarbeitungszeit erforderte. Die von der Clean Architecture vorgeschriebene Abhängigkeitsregel vereinfachte letztlich die Entwicklung funktionaler Quellcodepfade, obwohl dies in der Theorie zunächst umfangreicher und komplexer erschien.

Robust:

Während der Entwicklung und des Testens der Anwendung wurde wiederholt eine unerwartete Exception ausgelöst. Zu keinem Zeitpunkt führte dies zu einem vollständigen Programmabbruch oder einem unumkehrbaren Zustand. Dies wurde nicht durch die Clean Architecture selbst erreicht, sondern durch die Verwendung von *Spring* und dessen Webserver. Jede Anfrage, die durch einen Entry Point entsteht, läuft in einem eigenen Thread. Stürzt dieser ab, wird er von *Spring* aufgefangen, ohne die anderen Vorgänge zu beenden. Im Falle eines REST-Calls wird dessen Scheitern über den Webserver als Response bekannt gegeben.

Eine Schwäche ist jedoch bei den Datenbankoperationen festzustellen, da noch ein Konzept fehlt, Use Cases als zusammenhängende Transaktion zu verstehen. Für den Prototyp wurde auf eine sehr einfache Get- und Set-Schnittstelle gesetzt, die durch die Adapter im Core-Projekt vorgegeben werden. Somit ist nicht die Implementierung der Datenbankabfragen das Problem, sondern die vereinfachten Core-Komponenten. Im derzeitigen Projektstand kann ein Softwarefehler zwischen zwei Set Querys zu einer fehlerhaften Datenbank führen. Damit dies nicht geschieht, müsste der Ablauf eines Use Case als zusammenhängende Transaktion gehändelt werden. Im Falle eines Fehlers ließe sich der Ablauf damit rückgängig machen. Eine alternative Lösung wäre es, für jeden Use Case jeweils nur eine Set Query zu verwenden beziehungsweise zusammenzustellen. Dies liegt jedoch nicht an der Clean Architecture, sondern am derzeitigen Konzept der existierenden Adapter.

Messbar:

Dieses Qualitätsmerkmal kann anhand des derzeitigen Projektstands nicht eindeutig untersucht werden. So ist es beispielsweise noch nicht möglich, mit mehreren Nutzern zu arbeiten

oder die Datenbank einem sinnvollen Stresstest zu unterziehen. Sollten bei steigendem Projektumfang Probleme auftreten, muss analog zur Datenbank untersucht werden, welche Konzepte die Last besser verteilen oder händeln können. Es handelt sich hierbei also um diverse Programmstellen, die im Zuge dieser Masterarbeit noch nicht entwickelt oder zur Diskussion standen. Kritische Punkte werden unter anderem dadurch entstehen, dass sehr viele Nutzer auf ein und dieselben Datensätze zugreifen wollen. Eine Überlegung wäre es, Nutzer und Datensätze regional so zu trennen, dass die ausgewählte Datenbank den Anfragen gerecht werden kann. Gleiches gilt für den Webserver selbst. So muss es unter Umständen mehrere Server geben, die über Schnittstellen einen Nutzerwechsel zwischen den Regionen organisieren können. Dies liegt jedoch nicht im Aufgabenfeld der Clean Architecture, denn diese gibt lediglich den grundlegenden Rahmen der Anwendung vor. Sollte also auf regionale Server und/oder eine verteilte Datenbank gesetzt werden, können diese genau wie das hier entstandene Projekt aufgebaut sein. Ändern müssten sich dafür der Aufbau einiger Adapter oder deren Implementierung und einige Use Cases. Da eine so große Änderung auch im Rahmen der Clean Architecture umfangreich ist, wäre deren Planung der nächste Schritt in der Projektumsetzung.

Debuggable:

In dieser Betrachtung muss klar zwischen View und Server unterschieden werden, denn die während der Entwicklung gesammelte Erfahrung unterscheidet sich stark.

Der Server war durch den Aufbau der Clean Architecture zu jedem Zeitpunkt einfach zu debuggen. Quellcodepfade sind nachvollziehbar und es gibt keine Seiteneffekte, die ein Objekt unerwartet und aus dem Quellcode nicht ersichtlich verändern. Fehlerquellen wurden außerdem klar durch den Webserver und das eingebaute Logging kommuniziert.

Wahrscheinlich fehlerverursachende Programmzeilen können so in der Exception direkt oder im Umfeld der angezeigten Fehlerquelle gefunden werden. Einzig die Konfiguration bildet hierbei eine Ausnahme, da die dortigen Fehler im Zusammenhang mit *Spring* häufig unter den zuvor genannten Seiteneffekten leiden. So werden dort Objekte erzeugt, aufgerufen und verändert, ohne dass Quellcodepfade dafür klar nachvollziehbar sind.

Die App hat anders als der Server keine so leicht zugängliche Fehlersuche. Es ist zwar möglich, den *PhoneGap Browser Log* über die USB-Schnittstelle des Mobilgeräts auszulesen, jedoch ermöglicht dies kein schrittweises Debugging. Wird jedoch die Kompilierung für die

Plattform *Browser* genutzt, bietet dieser seinerseits alle üblichen Werkzeuge und Möglichkeiten für die Webentwicklung. Da jedoch im Browser nicht alle Plugins funktionsfähig sind, ist dies für YARPG nicht möglich gewesen. Dieses Problem ist allerdings nicht im Rahmen der Clean Architecture entstanden, sondern ist dem Mobilgerät beziehungsweise den verwendeten *PhoneGap* Plugins geschuldet. Diese Plugins müssten dahingehend erweitert oder ausgetauscht werden.

Somit ist der Server zwar sehr gut zu debuggen, der View-Programmteil ermöglicht jedoch derzeit nur eingeschränkten Zugriff.

Wartbar:

Die Wartbarkeit ergibt sich aus einer klaren Struktur der Anwendung, einer guten Testabdeckung sowie einer guten Modularisierung. Die klare Aufteilung der Anwendung in die verschiedenen Teilprojekte und deren wiederkehrende Bezeichnungen für Klassennamen und Aufgaben verbessern den Überblick wesentlich. Allein durch den Import von Pfad und Klassenname ist fast immer der Inhalt der Klasse vorgegeben. Dadurch lässt sich der Quellcode schnell lesen und verstehen, ohne dass alle aufgerufenen Methoden näher untersucht werden müssen. Gerade mit Anwachsen des Projekts durch mehr Spielelemente und Entwickler, bietet diese klare Modularisierung einen leichten Einblick beim Verstehen, Ändern oder Erweitern der vorhandenen Programmzeilen.

Darüber hinaus sind alle Komponenten durch Unit Tests und Integration Tests leicht zu überprüfen. Zwar können umfangreiche Tests auch in anderen Architekturen umgesetzt werden, jedoch kann die Einfachheit in diesem Projekt auf die Abhängigkeitsregel der Clean Architecture zurückgeführt werden. So können die Entities an jeder Stelle des Testcodes erstellt oder Methoden leicht gemockt werden. Bereits während der frühen Entwicklungsphase konnten dadurch alle Komponenten eine vollständige Testumgebung erhalten, die nicht auf umständliche oder umfangreiche Konfiguration angewiesen war. Auf diese Weise konnte von Anfang an eine hohe Testabdeckung für komplette Komponenten und jede sinnvolle Methodenkonstellation aufgebaut werden. Die Auslagerung der Integration Tests in ein eigenes Teilprojekt hilft dabei, die für diesen Testcode notwendigen Klassen klar vom Produkt zu trennen.

Prinzipiell ist der View-Programmteil schwerer zu debuggen als der Server. Als Vorteil erweist sich die Trennung zwischen Server und App. Dadurch kann dieser Teil getrennt vom Server

weiterentwickelt oder angepasst werden. Somit können sowohl Server als auch View als gut wartbar angesehen werden.

Wiederverwendbar:

Die Use Cases und Datenbankadapter sind so aufgebaut, dass sie bestimmte Aufgaben erfüllen sollen. Darüber hinaus haben diese Komponenten keine weitere Aufgabe. Der Use Case erarbeitet also keine Objekte oder Ansichten für eine spezielle View, sondern löst lediglich ein spezielles Problem. Treten Aufgaben im Zusammenhang mit mehreren REST-Calls oder einem anderen Use Case auf, können diese ineinander entsprechend verknüpft und somit wiederverwendet werden. Gleiches gilt natürlich auch für die Adapter. Datenbankabfragen sind entsprechend generischer *Get-* oder *Set-Methoden* aufgebaut. Sie können an jeder Stelle, an denen Entities erzeugt oder aktualisiert werden müssen, zum Einsatz kommen.

Die REST-Calls im Teilprojekt *Entry Point* verwenden hingegen lediglich ein oder mehrere Use Cases. Sie beinhalten darüber hinaus nur Logik, um die Daten für den View-Programmteil vorzubereiten. Bei der Entwicklung vollkommen anderer Views könnten die Entry Points zwar nicht wiederverwendet werden, dafür jedoch die Gesamtheit aller Use Cases und damit ein Großteil des Softwareprojekts. Viele Inhalte der Teilprojekte *Core* und *Dataprovider* können für wiederkehrende Probleme häufiger verwendet werden.

Erweiterbar:

Durch eine klare Modularisierung und die Abhängigkeitsregel ist eine Erweiterung der Anwendung jederzeit möglich. Die einfachsten Veränderungen der Anwendung können an den Entry Points und Dataprovidern vorgenommen werden. Dafür muss weder ein Use Case noch eine Entity angepasst werden. Demnach lässt sich die gesamte Datenhaltung inklusive des Tabellenaufbaus und deren Verknüpfungen ändern, ohne dass eine einzige Quellcodezeile im *Core* umgeschrieben werden müsste. Dies könnte beispielsweise im Rahmen einer Optimierung der Abfragezeiten geschehen.

Zudem könnten neue REST-Calls problemlos geschrieben werden und vorhandene Use Cases nutzen. Sind genügend verschiedene Use Cases zur Lösung eines Problems vorhanden, ist durch den kombinierten Einsatz dieser oftmals kein zusätzlicher Use Case erforderlich. Änderungen am Core haben hingegen immer große Folgen und könnten auch von keiner anderen in dieser Masterarbeit diskutierten Architektur besser behandelt werden. Solange

vorhandene Entities unverändert bleiben oder lediglich durch weitere Use Cases ergänzt werden, sind Änderungen leicht zu integrieren. Wird hingegen eine Entity verändert oder erweitert, ist mit wesentlich umfangreicheren Programmänderungen zu rechnen.

Sicher:

Jede Klasse und jedes Teilprojekt von YARPG hat jeweils nur Zugriff auf diejenige Information, die zur Lösung der jeweiligen Aufgabe unmittelbar notwendig ist. Darum sind die Spielelemente auf verschiedene Use Cases, REST-Calls oder Datenadapter verteilt. Ein Zugriff auf das Gesamtsystem ist nicht möglich. Damit ist das *Seperation of Concern (SoC)* Prinzip erfüllt.

Da das Projekt noch keinen Login besitzt, kann noch keine Aussage zur Sicherheit vor Unbefugten getroffen werden. Um die REST-Calls nur eingeloggten Nutzern zur Verfügung zu stellen, kann *Spring Secure* für das Spiel zum Einsatz kommen. Darüber hinaus wird in jedem Use Case der Nutzer identifiziert, um die Daten, die für ihn sichtbar sein sollen, zuzuordnen.

Wie schon im Abschnitt *Robust* erwähnt, gibt es derzeit keine Mechanik, um die Konsistenz der Datenobjekte zu gewährleisten, das heißt, mit dem derzeitigen Stand ist YARPG nicht sicher. Jedoch ist es möglich, die genannten Kritikpunkte im weiteren Entwicklungsverlauf zu beseitigen. Sobald ein Login hinzugefügt ist und die Datenbankabfragen eine Mechanik für gekapselte Operationen beinhalten, wird die Anwendung relativ sicher gegenüber unbefugten Zugriffen und fehlerhaften Datenbankoperationen sein.

7 Fazit und Ausblick

Die ausgewählte Clean Architecture konnte in diesem noch sehr prototypischen Projekt überzeugen. Die Teilprojekte und die konsistente Benennung der Softwareteile sorgen für einen übersichtlichen und nachvollziehbaren Quellcode. Im fortlaufenden Entwicklungsprozess ist dies vor allem dann hilfreich, wenn Erweiterungen und Änderungen anstehen oder eine Fehlersuche durchgeführt werden muss.

Der theoretische Ansatz mit den kreisförmigen Ebenen und den zunächst abstrakt wirkenden Adaptern stellte sich in der Umsetzung als unkompliziert heraus. Anders als erwartet, waren die Teilprojekte in Kombination mit der Abhängigkeitsregel sogar hilfreich, das Projekt zu starten.

Da die Entities nicht von externen Bibliotheken abhängig sind, konnte das Core-Projekt mitsamt Use Cases entwickelt und getestet werden, bevor Datenbank oder Webschnittstelle betrachtet wurden. Aus diesem Grund war es möglich, sich zuerst auf den Spielinhalt und dessen funktionalen Aufbau zu konzentrieren.

Die entstandene Softwarequalität weist derzeit noch Mängel auf. Insbesondere die Robustheit der Datenbankoperationen kann noch als problematisch betrachtet werden. Für den Prototyp ist das ausgewählte Datenbankverfahren in jedem Fall ausreichend. Jedoch sollte im Rahmen der Weiterentwicklung ein verlässlicheres Verfahren gewählt werden, das stets konsistente Datensätze erzeugt. Zur Auswahl eines angemessenen Datenbankkonzepts müssten allerdings umfangreiche Betrachtungen für mögliche Lösungsvarianten vorgenommen werden. Dies würde jedoch den Rahmen dieser Masterarbeit überschreiten.

Dieser Umstand zeigt, dass die Clean Architecture allein keine Garantie zur Entwicklung einer vollumfänglich guten Software liefern kann. Gleichzeitig wird deutlich wie wichtig eine gründliche Planung der Core-Komponenten ist, da diese einen signifikanten Einfluss auf alle nachfolgenden Komponenten haben. Je früher diese Planung im Projektverlauf berücksichtigt wird, desto geringer ist der Aufwand, alle Qualitätsmerkmale zu erfüllen. Ein wie in dieser Masterarbeit entwickelter Prototyp kann sich in diesem Zusammenhang sogar als hilfreich erweisen, um derartige Schwachstellen frühzeitig aufzudecken und entsprechende Maßnahmen zu ergreifen.

Der mittels *PhoneGap* entstandene View-Programmteil ist aus Sicht der Clean Architecture nur eine externe Anwendung und somit nicht durch dessen Aufbau bestimmt. Jedoch konnte veranschaulicht werden wie einfach eine plattformübergreifende App mittels HTML, CSS und JS entwickelt werden konnte. Da für die in der App verwendete Website jedoch die gleichen Qualitätsansprüche gelten wie für die anderen Programmteile, ist für deren Weiterentwicklung ebenso eine geeignete View-Architektur zu erarbeiten.

Trotz der genannten Mängel des Prototyps konnte gezeigt werden wie lohnenswert die sorgfältige Recherche und Auswahl einer geeigneten Software-Architektur sein kann. Selbst unerfahrene Entwickler sind damit in der Lage, schnell und zuverlässig gute Ergebnisse zu erzielen. Wie das Beispiel YARPG zeigt, konnte mit überschaubarem Aufwand durch lediglich einen Entwickler ein solides Fundament errichtet werden, das den Ausbau eines beliebig wachstumsfähigen Produkts ermöglicht. Insofern fällt das Fazit dieser Masterarbeit rundum positiv aus.

8 Literaturverzeichnis

Chaffee, A. (2012). *What is a web application (or "webapp")?* Von jGuru: http://www.jguru.com/faq/view.jsp?EID=129328 abgerufen

Fowler, M. (2004). *Presentation Model*. Von Martin Fowler Blog: https://martinfowler.com/eaaDev/PresentationModel.html abgerufen

Fowler, M. (2006). *Model-View-Presenter (MVP)*. Von Martin Fowler Blog: https://www.martinfowler.com/eaaDev/uiArchs.html abgerufen

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Prentice Hall.

Hasan, M. (2018). *Top 20 Most Popular Programming Languages To Learn For Your Open-source Project*. Von Ubuntu Pit: https://www.ubuntupit.com/top-20-most-popular-programming-languages-to-learn-for-your-open-source-project/ abgerufen

Koffmann, & Wolfgang. (2013). *Introduction to Software Design.* Von College of Information and Computer Sciences. abgerufen

Martin, R. C. (2012). *The Clean Architecture*. Von The Clean Code Blog: http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html abgerufen

McCall, & Boehm. (1991). *ISO 9126 Software Quality Characteristics.* International Organization for Standardization.

Nielsen, J. (1993). *Usability Engineering Kapitel 5: "Response Times: The 3 Important Limits".* Morgan Kaufmann. Von Nielsen Norman Group. abgerufen

Nolan, G. (2016). *Android Design Patterns.* Von Google Developer Group talk at Detroit GDG: https://www.youtube.com/watch?v=JV63czrUpbI abgerufen

Poort, E. R., & Vliet, H. v. (2012). *RCDA: Architecting as a risk- and cost management discipline.* Journal of Systems and Software. 85 (9).

Posch, T., Gerdom, M., & Birken, K. (2012). *Basiswissen Softwarearchitektur: Verstehen, entwerfen, wiederverwenden.* dpunkt Verlag.

Professor Gielen, F. (2018). *Attribute-Driven Design.* Von Software Engineering Institute: https://resources.sei.cmu.edu/asset_files/FactSheet/2018_010_001_513930.pdf abgerufen

Ratnottar, S. (2019). *What is PhoneGap & Why You Should Use It For App Development*. Von The Startup: https://medium.com/swlh/what-is-phonegap-why-you-should-use-it-for-app-development-5ac5e19eb19a abgerufen

Reenskaug, T. (22. März 1979). *The Smalltalk-76 Programming System Design and Implementation*. Von Trygve Reenskaug Blog: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html abgerufen

Reenskaug, T. (12. Mai 1979). *THING-MODEL-VIEW-EDITOR an Example from a planningsystem.* Von Trygve M. H. Reenskaug Blog: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf abgerufen

Rouse, M. (2016). *augmented reality (AR)*. Von WhatIs: https://whatis.techtarget.com/definition/augmented-reality-AR abgerufen

Shvets, A. (2014). *Alexander Shvets Design Patterns Explained S.6.* Goodreads Author.

Stephen. (2016). *Software Architecture 101: What Makes it Good?* Von codementor: https://www.codementor.io/learn-development/what-makes-good-software-architecture-101 abgerufen

Takahashi, D. (2018). *Pokémon Go studio Niantic rekindles its first game with launch of Ingress Prime*. Von venture beat. abgerufen

Verma, A. (2019). *Introduction to MVVM*. Von MindOrks: https://medium.com/mindorks/introduction-to-mvvm-836b1f3b7f61 abgerufen

9 Abkürzungsverzeichnis

ADD
Attribute-driven design 10, 11, 18
App
Application software 17, 31, 33, 34, 35, 36, 37, 38, 39, 46, 58, 60
AR
Augmented Reality 12, 13, 15, 17
ARS
Architecturally significant requirements 10, 18, 20, 33
BDD
Behavior-driven development 11
CSS
Cascading Style Sheets 34, 37, 59
DTO
Data Transfer Object 31, 38, 42, 51, 52
GPS
Global Positioning System 12, 13, 14, 15, 16, 17, 18, 33, 34, 58, 59, 60
HTML
Hyper Text Markup Language 26, 34, 37, 59
HTTP
HyperText Transfer Protocol 59
JAR-Datei
Java Archive File 62
JdbcTemplate
Java Database Connectivity Template 53, 54
JS
JavaScript 34
JSON
JavaScript Object Notation 31, 42, 52
MDD
Material-driven design 11
MVC
Model View Controller 8, 9, 20, 21, 22, 23, 24, 26, 27, 31, 32
MVP
Model View Presenter 20, 24, 25, 26, 27, 32
MVVM
Model View ViewModel 20, 26, 27, 32
NSC
Nicht Spieler Charakter 16
OSM
OpenStreetMap 59
PDD
und Process-driven development 11
POJO
Plain Old Java Object 47, 52
PvE
Player versus Environment 79
PvP
Player versus Player 79
REST
Representational state transfer 29, 31, 42, 51, 52, 58
RSS
Rich Site Summary 19
SoC
Separation of Concerns 24, 28
SQL
Structured Query Language 21, 23, 31, 41, 50, 51, 53, 54
TDD
Test-driven development 11
URL
Uniform Resource Locator 51
YARPG
Yet Another Rollplay Game 3, 12, 13, 15, 20, 24, 26, 27, 31, 32, 33, 38, 44, 46, 50, 53, 55, 58, 62

10 Abbildungs- und Tabellenverzeichnis

Anhang A: Spielkomponenten im Detail

Abenteurergruppe

Die vom Spieler gesteuerte Abenteurergruppe ist vermutlich das wichtigste Spielelement als Dreh- und Angelpunkt jeder Interaktion zwischen Spielern und der Umgebung. Sie ist vergleichbar mit der Spielfigur eines Videospiels, wobei ihre Bewegung nicht durch eine Tastatur oder ein Gamepad gesteuert wird, sondern durch die Bewegung des Mobiltelefons. Die Position des Spielers in der realen Welt wird dabei auf die Virtuelle so abgebildet, dass die Distanzen identisch sind. Das heißt, um mit anderen Spielern oder Spielelementen interagieren zu können, muss der Spieler sich in deren Nähe befinden. Geplant ist ein Interaktionsradius von 100 bis 200 Metern, damit der Spieler nicht zu weit von einer Straße abweichen muss und mehr Handlungsfreiheit hat. Zudem ist damit auch eine ungenaue GPS Position ausreichend. Die Position der Gruppe und andere Spielelemente werden auf einer interaktiven Karte durch Symbole angezeigt. Befindet sich ein solches Symbol innerhalb des Interaktionsradius, kann die entsprechende Spielhandlung gestartet werden.

Damit eine gewisse Spieltiefe und Handlungsfreiheit entsteht, besteht eine Gruppe aus Charakteren mit Geschlecht, Name, Rasse, Klasse, Inventar oder andere Eigenschaften, um sie zu individualisieren oder zu spezialisieren. Ausgewählte Spielelemente brauchen unter Umständen einen Charakter mit bestimmten Fähigkeiten oder Gegenständen, damit diese Handlung überhaupt ausgewählt werden kann. Solche Handlungen können der Kampf gegen Monster, dem Handel der Gegenstände mit anderen Spielern, dem Untersuchen von Höhlen oder ähnliches sein. Die Gruppenzusammensetzung hat deshalb großen Einfluss auf mögliche Handlungen oder den Kampfverlauf und steht somit im Zentrum des Spiels für fortlaufende Anpassung und Verbesserung.

Siedlung

Die Siedlung repräsentiert die Heimat der Abenteurergruppe. Sie wird vom Spieler an einem von ihm gewählten Ort gesetzt. In der Siedlung kann der Spieler seinen Gruppenaufbau ändern, neue Gruppenmitglieder anwerben und Gegenstände herstellen. Damit sich der Spieler nicht immer in der realen Welt bewegen muss, bietet dieser Ort reichliche Interaktionsmöglichkeiten. Dabei ist die virtuelle Ortschaft mit Nicht-Spieler Charakteren, sogenannten NSC's, gefüllt, die nachfolgend Einwohner genannt werden. Diese Einwohner handeln im Auftrag des Spielers, bauen neue Gebäude, forschen und erwirtschaften Abgaben. Investiert der Spieler diese Abgaben in die Siedlung, werden neue Gegenstände, bessere Charaktere und weitere Spielhandlungen freigeschaltet. Eine ausgebaute Siedlung wird durch Gebäude, Bewohner und den Ruf der Abenteurergruppe charakterisiert. Durch diese Kombination werden neue Bewohner angelockt und natürlich auch neue Abenteurer, welche der Spieler in seine Gruppe integrieren kann.

Entsprechend der Einwohnerzahl hat eine Siedlung eine bestimmte Menge freier Arbeiter, die für Projekte eingesetzt werden können. Dabei wird zwischen Ressourcenbeschaffung und Umsetzung eines Bauprojekts unterschieden. Soll beispielsweise eine Stadtmauer gebaut werden, müssen dafür Steine, Holz und Arbeitsstunden berechnet werden. Die Siedlung produziert die Ressourcen in einer festgelegten Quantität, die abhängig von den eingesetzten Arbeitern in dem entsprechenden Bereich ist. Umso größer oder hochwertiger ein Bauprojekt desto mehr Ressourcen und Arbeitsstunden müssen bereitgestellt werden. Sind in der Siedlung bessere Gebäude oder Forschung verfügbar, kann die geförderte Ressourcenmenge je Arbeiter gesteigert oder auch Arbeitsstunden für ein Projekt reduziert werden.

Ab einer bestimmten Ausbaustufe kann eine Siedlung auch mit anderen Siedlungen Handel betreiben, um fehlende oder überschüssige Ressourcen zu tauschen.

Der genaue Standort einer Siedlung soll jedoch nicht für andere auf der Karte sichtbar gemacht werden, um die Privatsphäre des Spielers zu schützen. Im Spiel soll lediglich angezeigt werden, wie viele Siedlungen sich in der aktuellen Umgebung befinden und welche Eigenschaften sie haben. Der Interaktionsradius ist dabei von Gebäuden und dem Ruf abhängig. Es handelt sich um eine stetig wachsende Spielmechanik, welche auch nach einer Veröffentlichung des Spiels erweitert werden soll.

Handwerk und Handel

Die in der Siedlung freigeschalteten Produktionsgebäude und die durch die Bewohner der Siedlung gegebene Arbeitsleistung kann zur Gütererzeugung genutzt werden. Die Art der Güter ist dabei abhängig von dem was alles in der interaktiven Welt existieren soll.

Waffen und Rüstungen sind einfache und übliche Gegenstände eines Rollenspiels, jedoch können auch Verbrauchsmaterialien existieren. Schweres Gerät, Bandagen und Tränke sind nur eine kleine Auswahl der Möglichkeiten, welche nicht nur das Handwerk und den Handel beeinflussen, sondern auch Kämpfe oder Gruppenzusammensetzungen. Es ist angedacht zu Beginn nur einen Grundstock an Gütern zu implementieren und diese stetig zu erweitern. Alle Güter sollen dabei vom Spieler frei handelbar sein, entweder über Tausch oder über eine Währung wie Münzen.

Die Zahlungsmittel werden in den Siedlungen durch Besteuerung der Bevölkerung geschaffen. Dabei sind Münzen als universelles Zahlungsmittel neben dem Handel auch für den Unterhalt der Gruppe, der Siedlung oder anderen Aufgaben notwendig. Langfristig soll die im Umlauf befindliche Menge an Münzen begrenzt werden, um den Markt und die Interaktion der Spieler in ihrer spielfördernden Weise nicht zu behindern. Somit ist es erforderlich, die Generierung von Münzen in Einklang mit deren Löschung zu bringen.

Der einfachste Weg ist es, Güter in Stufen zu untergliedern. Die höchste Stufe besitzt die besten Eigenschaften bei deutlich höheren Kosten. Da Güter nur eine begrenzte Lebensdauer haben sollen, muss der Spieler auch aus ökonomischen Überlegungen entscheiden in welcher Stufe er ein Gut erwirbt, um damit sein Ziel zu erreichen. Beispielsweise könnte er ein Schwert der Stufe 1 bis 5 erwerben, wobei die Kampfkraft des Schwertes linear ansteigt, die Kosten für dessen Beschaffung jedoch exponentiell.

Das Handwerk muss so gestaltet werden, dass es nicht dazu in der Lage ist, alle Spieler dauerhaft mit Stufe 5 Gütern für alle Aktionen zu versorgen. Um das zu erreichen, muss die Menge an Zahlungsmitteln aber auch an anderen Ressourcen begrenzt werden.

Stufe 1 bis 3 Güter können in Siedlungen ohne besondere Anforderungen hergestellt werden. Die dafür benötigten Gebäude oder Forschungen entwickeln sich im Laufe der Spielzeit ohne Probleme in ausreichender Menge.

Zur Herstellung der Güter in Stufe 4 und 5 müssen hingegen besondere Ressourcen gesammelt oder andere Anforderungen erfüllt werden. Dafür könnten gefundene Gegenstände bei Player versus Environment (PvE), Questbelohnung oder ähnliches verwendet werden. So kann die Qualität der hochstufigen Güter gesichert und deren Menge begrenzt werden.

Stattfindender Handel ist immer an die Position der Ortschaft oder des Spielers gebunden. Es ist jederzeit möglich, Kauf- und Verkaufsangebote in den Markt zu stellen. Diese Angebote stehen jedoch nur den Spielern zur Verfügung, welche sich im Interaktionsradius des Anbieters befinden. Der Umkreis sollte im Anfangsstadium ca. 1 km betragen. Je nach Ausbaustufe und Spielerdichte sollte er später vergrößert oder verkleinert werden. Es besteht aber auch die Möglichkeit, dass der Spieler mit seiner Gruppe umherzieht und so Angebote aus Interaktionsradien anderer Spieler wahrnehmen kann. Kauf und Verkauf ist somit immer regional abhängig.

Tauschhandel ist eine Möglichkeit Güter ohne Zahlungsmittel zu handeln. Solche Angebote sind schwer in einem Markt oder einer Liste anzuzeigen und wurden deshalb vorerst nicht berücksichtigt.

Kampf: PvE und PvP

Bei Player versus Environmen (PvE) wird gegen Monsterhorden, Banditen oder ähnliche nicht Spieler gesteuerte Figuren gekämpft. Während der Spieler mit seiner Gruppe durch die Gegend zieht, begegnet er diesen Wesen zufällig. Seine vorher zusammengestellte Gruppe kämpft dann automatisch, ohne dass der Spieler eingreifen kann. Dabei werden vom Spieler festgelegte Handlungen, Zauber und ähnliches abgearbeitet.

Ein Kampflog hilft während oder nach einem Gefecht mögliche Schwächen oder Probleme zu erkennen. Für die meiste Zeit wird der Log jedoch für den Spieler uninteressant sein. Wichtig sind gewonnene Erfahrung und gefundene Gegenstände. Jeder so simulierte Kampf benötigt etwas Zeit. Die Spielzeit wird so etwas gestreckt und der Spieler kann bei Bedarf den Log beim Entstehen beobachten.

Ein Kampf ist rundenbasiert mit Aktionen, welche Ausdauer oder Manapunkte brauchen. Ist eine Gruppe erschöpft, kämpft sie je nach Erschöpfungsgrad deutlich schwächer. Dadurch können Kämpfe, die sonst ohne Probleme gewonnen werden könnten, auch verloren gehen.

Damit wird erreicht, dass der Spieler an einem Tag nicht endlos viel Erfahrung sammelt oder beliebig viele Kämpfe führt.

Ist die Ausdauer verbraucht, muss sich die Gruppe regenerieren. Diese Regeneration kann über Verbrauchsgüter beschleunigt werden. Wird ein Kampf verloren, muss der Spieler pausieren. Seine Helden sind dabei in einer Phase der Zwangsregeneration, um ihre Wunden zu heilen. Darüber hinaus sind noch andere temporäre Effekte denkbar. Diese sind abhängig vom Gegner oder anderen jeweiligen Situationen.

Ist der Kampf hingegen gewonnen, bekommt der Spieler Gegenstände in sein Inventar. Diese sind abhängig von der besiegten gegnerischen Gruppe. So ist Drachenhaut denkbar sowie Horn, Krallen oder andere Teile von Tieren und Monstern, sowie magische bzw. alchemistische Güter wie Feenstaub oder Essenzen.

Der Spieler kann abhängig von der Zusammensetzung seiner Abenteurergruppe gezielt Aufgaben suchen, wie das Sammeln von Kräutern oder das Erforschen von Höhlen. Höhlen gehören zum Erzählsystem, welches im Anhang Erzähler erklärt wird.

Kräuter sind weitere alchemistischen Gegenstände, welche für bestimmte Handwerke benötigt werden. Für den Beginn sollte sich das Spiel jedoch nur auf das Suchen und Bekämpfen von Monstern beschränken, um es gerade in der Anfangsphase nicht zu überladen.

Werden genügend Kämpfe ausgefochten, steigt das Level der einzelnen Gruppenmitgliedern an. Diese erhalten mit jedem Aufstieg zusätzliche Eigenschaften oder vorhandene steigern sich.

Der Spieler soll die Möglichkeit haben jedes Gruppenmitglied zu individualisieren. Er entscheidet selbst welche Eigenschaften jedes Gruppenmitglied erhalten wird. Dafür kann ein Fähigkeitenbaum dienen oder er muss sich zwischen zwei Optionen seiner Klasse entscheiden.

Player versus Player (PvP) funktioniert nach einem ähnlichen Prinzip wie bei PvE. Der Gegner ist dabei die Gruppe eines anderen Spielers. Diese Kämpfe dienen in erster Linie dazu sich zu messen oder ein Event zu schaffen. Es steht dabei nicht das Sammeln von Erfahrungen oder Gegenständen im Vordergrund. Es sind keine zufälligen Ereignisse. Die Spieler müssen

sich entweder zu einem Kampf verabreden oder sie gehen zu Events, die derartige Kampfspiele anbieten. Eines dieser Events ist beispielsweise der Kampf um die Karte.

Weiterhin soll es vom Spieler generierte Festen oder Burgen geben, die das jeweils umliegende Gebiet kontrollieren. Dabei wird jede Feste nicht von nur einem Spieler gebaut und gehalten, sondern immer von einer Spielergemeinschaft. Dazu könnten Gilden oder andere Allianzen ins Spiel kommen. Ein so kontrolliertes Gebiet gibt entsprechend Bonus oder Malus und gehört zum langfristigen Spielspaß. Es stellt einen Spielinhalt dar, der den Spieler neben dem Aufbau seiner Siedlung und dem Erleben von Abenteuern endlos beschäftigt.

Aufgaben und Erzähler

Neben dem Aufbau der Siedlung kann der Spieler auch andere Aufgaben erhalten oder auf zeitlich beschränkte Ereignisse treffen. Aufgaben werden über die eigene Siedlung vergeben und stehen solange zur Verfügung bis sie erfüllt wurden. Diese Aufgaben dienen zur Einführung in das Spiel und zeigen was alles getan werden kann. Zum anderen bietet die Erfüllung dieser Aufgaben eine Möglichkeit kleine Boni für die Siedlung oder die Gruppe zu erhalten. Die zufälligen Ereignisse hingegen gehören zum sogenannten Erzähler. Dieser schafft zeitlich beschränkte Punkte auf der Karte, bei denen der Spieler eine Handlung ausführen kann. Dies kann das Untersuchen einer Höhle sein, das Sammeln besonderer Ressourcen oder das Bekämpfen von Monstern. Als Belohnung für das Erfüllen dieser Aufgaben gibt es Erfahrungspunkte, Ruf und Gegenstände abhängig von dem was erledigt wurde.

Der Erzähler ist eine interne Bezeichnung für das automatisierte und zufällige Eingreifen in das Spielgeschehen der Welt. Für den Spieler tritt der Erzähler nicht in Erscheinung.

Darüber hinaus kann es Zeiträume geben, in denen der Erzähler wechselt oder ein zusätzlicher Erzähler das Spielgeschehen bestimmt. Typischerweise sind Feiertage wie Weihnachten prädestiniert, das Spiel mit besonderen Ereignissen oder Belohnungen zu ergänzen. Diese Ereignisse müssen von Erzählern mit den dafür nötigen Eigenschaften erzeugt werden. Das heißt, intern bekommen die Erzähler Namen für welche Art von Ereignissen sie zuständig sind. Dadurch können sie später ausgetauscht oder erweitert werden.

Auch denkbar ist es, die Art des Erzählers an die aktuelle Tageszeit anzupassen. Somit könnten von der Tageszeit abhängige Monster erschaffen werden.

Der Schwierigkeitsgrad der zufällig erzeugten Ereignisse ist dabei immer an den Spieler und den Erfahrungslevel seiner Gruppe angepasst.

Im Verlaufe der Entwicklung sollte dieses System erweitert werden. Anfänglich sind es nur Kämpfe, später eine Kombination aus Aufgaben und Ereignissen. Diese sollen den Spieler möglichst vielfältig beschäftigen.