ulm university universität uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Konzeption eines generischen Datenmodells für iOS im Kontext akustischer Lokalisation

Bachelorarbeitan der Universität Ulm

Vorgelegt von:
Valentin Knabel
valentin.knabel@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2017

Fassung 12. April 2017

© 2017 Valentin Knabel

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-LaTeX 2_ε

Kurzfassung

Auf mobilen Endgeräten wirft das Genre um akustische Spiele einige interessante Konzepte und Möglichkeiten auf. Dies gilt insbesondere in Kombination mit der Verarbeitung von Bewegungsdaten und dem Identifizieren von Geräuschquellen.

Im Rahmen dieser Arbeit wird ein generisches Datenmodell für Anwendungen dieser Art entwickelt und implementiert. Das erarbeitete Konzept beruht auf funktionalen Anforderungen, welche sich durch eine Analyse von drei im Weiteren ausgeführten, fiktiven Anwendungen und deren Aggregation ergeben.

Diese werden zuerst vorgestellt und voneinander abgegrenzt. Im Anschluss daran wird die Spezifikation erarbeitet, welche vom System und dem Datenmodell erfüllt werden müssen, um alle vorgestellten Anwendungsfälle passend abbilden zu können. Bezüglich dieser Anforderungen wird das Datenmodell als mögliche Lösung vorgestellt und anhand einer Implementierung genauer betrachtet.

Abschließend wird das gefundene generische Datenmodell mit einem problemspezifischen verglichen und bewertet.

Danksagung

An dieser Stelle möchte ich zuerst meinem Betreuer Marc Schickler danken, der mich durch seine Vielzahl an Ideen und Anregungen zu diesem Thema hingeführt und mich im weiteren Verlauf mit seiner fachlichen Kompetenz unterstützt hat.

Ein weiterer großer Dank gebührt meinen Freunden und insbesondere meiner Familie, die mich besonders motiviert und unterstützt haben.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ziele der Arbeit 2

1.2 Aufbau der Arbeit 2

2 Grundlagen **5**

2.1 Anwendungsklasse 5

2.2 Akustische Lokalisation 6

2.3 Definitionen 7

2.4 Anwendungsfälle 8

2.4.1 Geschichte 9

2.4.2 Bewertung 9

2.4.3 Editor 10

3 Anforderungen **13**

3.1 Funktionale Anforderungen für Anwendungen 13

3.2 Funktionale Anforderungen für das Datenmodell 19

3.3 Nichtfunktionale Anforderungen für das Datenmodell 20

4 Konzeption **23**

4.1 Eingesetzte Technologien und Konzepte 24

4.1.1 Programmiersprache 24

4.1.2 Verwendete Entwurfsmuster 25

4.1.3 Lens 26

4.1.4 Validated Type 27

4.1.5 Dependency Injection 28

4.1.6 Functional Reactive Programming 29

4.1.7 Unidirektionaler Datenfluss 31

4.1.8 Sourcery 32

4.2 Architektur 33

4.2.1 Reactive Core 34

4.2.2 Lens Inject 35
4.2.3 Datenmodell 35
4.2.4 Value to Object Translator 40
4.2.5 Ausgabe 41

5 Anforderungsabgleich **43**
5.1 Funktionale Anforderungen 43
5.2 Nichtfunktionale Anforderungen 44

6 Zusammenfassung & Ausblick **47**
6.1 Ausblick 47
6.2 Abschluss 48

A Quelltexte **53**

1 Einleitung

Spiele auf Smartphones haben heutzutage eine große Verbreitung. Gerade unterwegs, sei es zu Fuß oder im Zug, können diese ihre Stärken ausspielen. Für ein kurzes Spiel benötigen diese häufig keine längere Aufmerksamkeit, sondern oft sogar nur eine Minute. Aufgrund dieser zeitlichen Einschränkung sind diese meist weniger komplex als vollwertige Konsolentitel, wodurch der Einstieg hier auch besonders leicht fällt.

Diese Spiele haben meist nur ein Spielprinzip und fokussieren sich nur auf eine Sache oder einen Stil. Diese Spezialisierung ermöglicht es ebenfalls mehrere Apps zu erstellen, die ein nur in Details unterschiedliches oder gar das gleiche Spielprinzip teilen. Diese Ähnlichkeit birgt eine große Variation an Möglichkeiten zur Wiederverwendung der gleichen Codebasis.

Gerade wenn es um Spiele geht, die sich hauptsächlich durch die Inhalte unterscheiden wird dieses Potential gesteigert, was insbesondere für Spiele mit einen starken Fokus auf akustische Effekte gilt.

Ein möglicher Genre basiert auf der Lokalisation von akustischen Signalen, die über Kopfhörer an den Endanwender weitergegeben werden. Gerade durch eine Steuerung durch Bewegungen des Smartphones könnte ein interessantes Spielekonzept hervorgehen.

1.1 Ziele der Arbeit

Das Hauptziel dieser Bachelorarbeit liegt in dem Entwurf und der Implementation eines generischen Datenmodells für mobile Anwendungen zur Lokalisation akustischer Signale durch den Endanwender in einem spielerischen Umfeld. Außerdem soll ein grundlegendes System zur Interaktion und Verwaltung des generischen Datenmodells entworfen und beschrieben werden.

Bezüglich des Entwurfs ist darauf zu achten, dass die Vorteile des Grades der Generizität für eine Vielzahl von Anwendungen gegenüber einer expliziten Implementation überwiegen.

Außerdem soll beispielhaft ein explizites Datenmodell aufbauend auf dem genersichen entworfen werden und grundlegend audio-visuell wiedergegeben werden.

1.2 Aufbau der Arbeit

Im Kapitel 2 werden für das Verständnis der weiteren Arbeit wichtige Grundlagen vermittelt. Darunter sind das Konzept der akustischen Lokalisation und eine Vorstellung der Grundprinzipien und -mechaniken der Anwendungsklasse für das generische Datenmodell. Des Weiteren findet hier eine Einführung in die drei genauer zu betrachtenden Anwendungsfälle statt.

Aufbauend auf den vorgestellten Situationen findet im Kapitel 3 eine Anforderungsanalyse statt. Hier werden zuerst das Datenmodell betreffende funktionale Anforderungen der Anwendungsfälle erarbeitet, die im Anschluss für das generische Datenmodell interpretiert werden, bevor auf die nichtfunktionalen Anforderungen eingegangen wird.

Das angestrebte Konzept wird im Kapitel 4 erarbeitet und vorgestellt. Zuvor werden lösungsspezifische Technologien vorgestellt und deren Wahl begründet. Unter diesen befinden sich die Wahl Swifts als Programmiersprache, die verwendeten Entwurfsmuster, Lenses, die verwendete Validierungsabstraktion, Dependency Injection, der unidirektionale Datenfluss, eine Einführung in Functional Reactive Programming sowie

Codegenerierung. Im Anschluss wird die genaue Architektur der Lösung beleuchtet, sowie auftretende Probleme vorgestellt und gelöst.

Nachdem die Lösung bekannt ist, folgt der Anforderungsabgleich im Kapitel 5, woraufhin ein abschließender Ausblick und eine Zusammenfassung gegeben werden.

2
Grundlagen

Innerhalb dieses Kapitels wird grundlegendes Wissen zum Verständnis der weiteren Arbeit vermittelt. Anfangs werden die Anwendungsklasse und grundlegende Spielmechaniken vorgestellt. Anschließend wird der Begriff der akustischen Lokalisation eingeführt. Aufbauend werden einige für das weitere Verständnis zentrale Begriffe eingeführt und erklärt auf denen die Vorstellung dreier Anwendungsfälle aufbauen wird.

2.1 Anwendungsklasse

Im Rahmen der Anwendungen sollen Benutzer den Ursprung von Tönen räumlich richtig einordnen. Hierfür wird der Ton über Kopfhörer wiedergegeben, um nicht durch den Abstand zum Gerät oder durch Störgeräusche verfälscht zu werden.

Hierfür hält der Anwender das Gerät vor sich, während Töne abgespielt werden. Die Applikation simuliert bei der Wiedergabe der Geräusche jeweils einen Ursprung, um den Benutzer herum. Die Aufgabe ist nun das Lokalisieren der richtigen Töne an den jeweiligen Orten.

Der Anwendung ist es überlassen zu entscheiden, ob die Quellen der Töne auch visuell dargestellt werden. Außerdem beliebig ist die Anzahl an Geräuschquellen und Störquellen oder ob es Hintergrundgeräusche gibt, die keinen zum Anwender relativen Ursprung haben.

Wenn der Anwender den richtigen Standort gefunden hat, obliegt es hier wiederum der expliziten Anwendung wie diese vorgeht. Hier sind sowohl Eingaben über das Display als auch über das Mikrofon möglich.

2.2 Akustische Lokalisation

Das Orten einer Schallquelle durch einen Menschen ist ein komplexes Vorgehen, wobei diese in die Lokalisation von drei Ebenen unterteilt wird.

Die Bestimmung der Richtung, oder auch der Medianebene [1], der Schallquelle findet binaural statt (d.h. unter Verwendung beider Ohren) [2]. Nach Klensch [2] bildet die Grundlage der binauralen Lokalisation der Binauraleffekt, oder auch die Laufzeitdifferenz [1]. Dieser basiert auf der Zeitdifferenz, die der Schall beim Weg vom einen zum anderen Ohr benötigt. Anhand dieser Informationen kann bestimmt werden, ob sich die Quelle auf der rechten oder linken Seite befindet [2].

Wird hier noch die Entfernungswahrnehmung eingerechnet kann die Quelle sogar auf eine Parabel auf der Horizontalebene eingeschränkt werden, in Extremfällen der Seitlichkeit wird diese zu einer Art Strahl [2].

Die Bestimmung der Entfernung basiert nicht vornehmlich auf der Intensität der akustischen Signale, sondern auf deren Qualität [2]. Trotz der Bestimmung der Entfernung und der Richtung der Signalquelle kann es hier allerdings zu einer Vorn-Hintenvertauschung kommen, welche durch die Vorn-Hintenwahrnehmung aufgelöst werden kann [3]. Dazu wird der Kopf um die vertikale Halsachse gedreht [3], wodurch sich die Laufzeitdifferenz verschiebt. Durch diese selbst verursachte Änderung können Rückschlüsse auf die genaue Lokalität geschlossen werden.

Des Weiteren gibt es die Oben-Untenwahrnehmung, welche die Lage der Horizontalebene bestimmt [1]. Diese findet im Rahmen dieser Arbeit allerdings keine Relevanz, da nicht in der Höhe von dargestellten Entitäten unterschieden wird, sondern alle Signale auf einer einzigen Horizontalebene simuliert werden.

Bei einer detailgetreuen Simulation, die auf Kopfhörern basiert, sollte auch auf Unterschiede der Intensität einzelner Tonfrequenzen geachtet werden, da das allgemeine Klangbild bzw. der Klangcharakter als Kombination von Wellenlängen und ihrer Intensität auch Einflüsse auf eine korrekte Lokalisation haben können [2]. Bei langen Wellenlängen (d.h. tiefe Töne) sind die Differenzen der Intensität deutlich geringer als bei hohen Tönen, die in extremen Fällen fast komplett ausgelöscht werden [2].

2.3 Definitionen

Da eine einheitliche Ausdrucksweise für das weitere Verständnis wichtig ist, werden im Folgenden einige für die Anwendungsfälle und das Datenmodell zentrale Begrifflichkeiten erläutert und veranschaulicht.

Eine Entität ist eine Einheit mit einer audio-visuellen Repräsentation und besitzt eine Position.

Eine Entität könnte beispielsweise einen Gegenstand oder ein Lebewesen darstellen. Es kann auch Verhaltensweisen haben, wie beispielsweise eine Bewegung, was den Schwierigkeitsgrad erhöhen kann.

Das Sichtfeld ist der Bereich, in dem Entitäten virtuell dargestellt werden. Sein Zentrum beeinflusst aus welcher Richtung der Anwender später Geräusche wahrnimmt. Es wird typischerweise durch das Bewegen des Smartphones vom Anwender gesteuert.

Ein Ziel muss gelöst werden, um ein Level abzuschließen. Dies könnte das Auffinden einer Entität sein, eine Zeitspanne, in der alle anderen Ziele erreicht werden müssen, oder eine beschränkte Anzahl an Fehlversuchen beim Auffinden der Entitäten.

Ein Ziel kann auch nachdem es gültig ist wieder ungültig werden, wie es beispielsweise bei der Zeitbeschränkung oder den Fehlversuchen der Fall ist.

Ein Level ist eine in sich abgeschlossene Situation und stellt die Kombination von Zielen und Entitäten dar und wird mithilfe von dem Sichtfeld veranschaulicht.

Im Normalfall werden Entitäten nur innerhalb eines Levels dargestellt.

Der Schwierigkeitsgrad existiert lediglich implizit, ist vom Endanwender abhängig und beschreibt als wie schwierig dieser die Lösungsfindung empfindet. Diese Eigenschaft ist stark von der Komplexität des Levels bezüglich seiner akustischen Effekte sowie der Ziele abhängig.

Prinzipiell gilt, dass niedrigere Zeitlimits, mehr Entitäten, ähnlichere und mehr akustische Signale, die sich im Extremfall nicht wiederholen, eine niedrige Anzahl an möglichen Fehlversuchen oder einige Effekte wie Bewegungen von Entitäten jeweils zu einem erhöhtem Schwierigkeitsgrad beitragen.

Besonders hervorzuheben ist, dass durch auch das Ausblenden oder (halbtransparente) Überlagern der Inhalte den Schwierigkeitsgrad signifikant erhöht.

Bei einem niedrigen Schwierigkeitsgrad müsste man beispielsweise die einzige Entität eines Levels lokalisieren ohne durch ein Zeitlimit oder durch eine Begrenzung an Fehlversuchen in der Lösungsfindung eingeschränkt zu sein.

2.4 Anwendungsfälle

Für das zu entwerfende System ergeben sich je nach Anwendungsfall entsprechend unterschiedliche Anforderungen, die alle vom Subsystem unterstützt werden müssen. Hauptsächlich werden die funktionalen Anforderungen betrachtet, da die meisten nicht-funktionalen Anforderungen erst bei der Realisierung und Entwicklung der expliziten Endanwendung Relevanz finden, anstatt der darunterliegenden Bibliothek, die es hier zu entwerfen gilt.

Als Anwendungsbeispiele dienen zum Einen eine Anwendung, die dem Endanwender eine Geschichte erzählen will. Um die Geschichte zeitlich konsistent zu behalten, werden sämtliche Level nacheinander ausgeführt.

Ein weiterer Anwendungsfall für mobile Anwendungen liegt darin nach Beendigung eines Levels eine Wertung zu berechnen: Ziel des Anwenders soll die Maximierung dieser sein.

Als dritten und letzten Anwendungsfall betrachten wir den eines Editors um Level zu erstellen. Da dieser größtenteils unterschiedliche Anforderungen hat, ist das Ziel nicht alle spezifischen Anforderungen des Editors zu implementieren, sondern diese lediglich durch einen modularen Aufbau des Systems zu ermöglichen.

2.4.1 Geschichte

Das Ziel der erzählenden Anwendung liegt in der Unterhaltung des Nutzers durch Veranschaulichung von Situationen und Charakteren. Um die Situationen entsprechend untermalen zu können, muss eine „Geräuschkulisse“ erzeugt werden, die allen voran versucht eine Stimmung zu vermitteln. Einen essentiellen Bestandteil bildet eine austauschbare Hintergrundmusik, die levelbasiert gesteuert werden soll. Zur Entfaltung der Charaktere und ihrer eventuellen Gespräche müssen diese visuell und akustisch repräsentiert werden können.

Einerseits kann dies als Teil der dargestellten Situation und damit als Teil eines Levels geschehen, wenn man beispielsweise die Gesprächspartner suchen muss, andererseits können Charaktere ihre Repräsentation außerhalb des Systems ein Level überlagernd oder als Zwischensequenz zwischen mehreren Level vor einem statischen Hintergrund dargestellt werden.

Des Weiteren können Level nicht ein zweites Mal gestartet werden, da der Teil der Geschichte als abgeschlossen zählt. Denkbar wäre es mehrfach an den erzählerisch gleichen Ort zu kommen, allerdings verschiedene Situationen darzustellen.

Prinzipiell müssen die durch Level dargestellten Situationen nicht an reale angelehnt werden, da der Nutzer das nötige Wissen zum Lösen des Levels durch die Situation bzw. der erzählten Geschichte an sich erhalten wird. Dies bedeutet wiederum, dass sich der Benutzer nach einer erzählerischen Einführung auch auf „fremden Welten“ in denen bestimmte Annahmen nicht gelten zurechtfinden wird.

Einige Level dienen lediglich der Veranschaulichung, und können mitunter sehr einfach gestaltet sein, während andere wiederum deutlich komplexer sind und somit zur Schwierigkeit des gesamten Spiels beitragen.

2.4.2 Bewertung

Der Fokus liegt bei bewertenden Anwendungen nicht auf einer Erzählung sondern bei dem repetitivem Üben von simulierten Situationen oder dient der kürzeren Unterhal-

tung in zeitlich und motorisch einschränkenden Realsituationen des Anwenders, wie beispielsweise einer Zugfahrt. Hier wäre als alternative Steuerung für die Änderung der Hörrichtung, die Neigung des Gerätes in die entsprechende Richtung möglich.

Jedes Level kann in diesem Anwendungsfall mehrfach wiederholt werden. Nach jedem Abschluss eines Levels wird dieses nach eigenen, festen Kriterien bewertet, wobei mögliche Kriterien die Zeit oder die Anzahl der fehlgeschlagenen Versuche beim Lösen des Levels sind. Level werden je nach ihrem Stil in Kategorien unterteilt, die als Szenario oder auch Welt bezeichnet werden.

Denkbar wäre beispielsweise die Freischaltung von Szenarien an Gesamtbewertung oder an In-App-Käufe zu binden. Nach jedem Abschluss eines Levels wird der Endanwender wieder zurück zum Szenario navigiert, um ein neues starten zu können.

Am Anfang eines jeden Levels müssen dem Benutzer alle Ziele und Kriterien zur bestmöglichen Lösung vollständig bekannt sein, welche beispielsweise visuell oder textuell im Vorfeld repräsentiert werden können. Damit der Anwender das Level lösen kann, muss dieser zwischen den akustischen Signalen und den visuellen bzw. textuellen Repräsentationen Assoziationen herstellen können. Andernfalls bliebe dem Anwender lediglich die Lösung des Levels zu erraten, was allerdings mit unnötigem Frust verbunden ist.

Entsprechend versuchen sich die Level inhaltlich an mit dem Spielfortschritt zunehmend komplexeren Realsituationen anzulehnen und diese abzubilden.

2.4.3 Editor

Der Anwendungsfall des Level-Editors ist bezüglich der Nutzerinteraktionen und -zielgruppen grundlegend verschieden zu den zwei vorhergehenden Beispielen, da er nicht die Datenstrukturen für Level und Szenarien präsentieren sondern erzeugen soll. Beispielsweise werden die akustischen Signale nur dann ausgegeben, wenn dies explizit vom Anwender ausgeht, andernfalls ist das dargestellte Level komplett statisch.

Des Weiteren hat jede Änderung am Level durch die erwünschte Persistenz direkte Auswirkungen auf alle weiteren Reinkarnationen. Erst wenn das Level in einem expliziten

Anwendungsfall ausgeführt wird, ändert sich dieses Verhalten, da es im Rahmen anderer Anwendungsfälle im Normalfall erwünscht ist lediglich auf einer Kopie der Level zu operieren, während das Original als konstant und unveränderlich betrachtet wird.

3 Anforderungen

Aus den zuvor beschriebenen Anwendungsfällen sowie der zugrundeliegenden Beschreibung der Anwendungsklasse, ergeben sich sowohl funktionale als auch nichtfunktionale Anforderungen. In erster Linie betreffen diese lediglich die direkten Anwendungsfälle und deren Umsetzung. Erst durch Interpretation dieser können Rückschlüsse auf die Anforderungen für das zugrundeliegende generische Datenmodell gefolgert werden.

3.1 Funktionale Anforderungen für Anwendungen

Im Folgenden werden die funktionalen Anforderungen für die Anwendungen vorgestellt und begründet.

Für alle Anwendungsfälle gilt nach 3.1 und 3.2, dass durch die Darstellung von Leveln auch deren Entitäten dargestellt werden. Außerdem sind hier auch die Facetten der Schwierigkeitsgrade wichtig. Unterschiede liegen innerhalb der Granularität der einzelnen Anpassungsmöglichkeiten innerhalb des Verhaltens und der Darstellung sowie in Möglichkeiten zum Nachladen von Spieldaten und hinsichtlich des Editor auch bei der Persistenz von einem Level.

	Anforderung	Erklärung
FAA#1	Level kann auch nur akustisch dargestellt werden	Wichtig für höhere Schwierigkeitsgrade.
FAA#2	Ein Level kann Entitäten darstellen	Dient der Veranschaulichung. Wichtig bei Aufdeckung des Levels.
FAA#3	Level können durch Daten repräsentiert werden	Wichtig für das Konzept des Level-Editors 2.4.3. Beispielsweise in Form von JSON-Dateien.
FAA#4	Level haben einen Hintergrund	Besonders wichtig für besseres Ambiente bei Geschichten 2.4.1. Dieser kann von Level zu Level verschieden sein.
FAA#5	Level haben richtungsneutrale, akustische Signale	Besonders wichtig für Ambiente bei Geschichten 2.4.1. Diese können von Level zu Level verschieden sein.
FAA#6	Level können sich ändern	Grundlegende Spielmechanik und für höhere Schwierigkeitsgrade wichtig. Beispielsweise könnten sich Entitäten bewegen.
FAA#7	Level können gewonnen und verloren werden	Wichtig für Geschichtsfortschritt 2.4.1 und Bewertung 2.4.2 sowie für Erfolge.
FAA#8	Level können gestartet werden	Grundlegende Spielmechanik.

Tabelle 3.1: Erste funktionale Anforderungen der Anwendungen bezüglich von Leveln

	Anforderung	Erklärung
FAA#9	Level beinhalten Entitäten	Grundlegendes Spielkonzept. Wichtig für das generische Datenmodell.
FAA#10	Level können unabhängig voneinander gespeichert werden	Wichtig für Editor 2.4.3, bewertete Spiele 2.4.2 und für das Nachladen von Inhalten sowie für das generische Datenmodell.

Tabelle 3.2: Weitere funktionale Anforderungen der Anwendungen bezüglich von Leveln

Für das Sichtfeld steht die Art der Benutzerinteraktion im Vordergrund, da alle drei ausgeführten Anwendungsfälle Unterschiede hinsichtlich der Eingabequelle aufweisen, wie es 3.3 zu entnehmen ist. Grundlegend gilt allerdings für alle, dass das Zentrum von dieser Quelle abhängig ist.

	Anforderung	Erklärung
FAA#11	Das Sichtfeld muss verändert werden können	Elementarer Bestandteil des Genre.
FAA#12	Sichtfeldänderung durch Gerätedrehung	Die Standard-Steuerung. Wird von der Geschichtsapplikation benötigt 2.4.1.
FAA#13	Sichtfeldänderung durch Neigen des Geräts	Alternative Steuerung für Bewertete 2.4.2.
FAA#14	Sichtfeldänderung durch Scrollen	Steuerung für Editor 2.4.3.

Tabelle 3.3: Funktionale Anforderungen der Anwendungen bezüglich des Sichtfelds

Bezüglich der Entitäten haben alle Anwendungsfälle sehr ähnlich Anforderungen, wobei vor allem die audio-visuelle Darstellung und eine veränderliche Position wichtig sind (vgl. 3.4). Außerdem sind Möglichkeiten der Benutzerinteraktion wichtig.

	Anforderung	Erklärung
FAA#15	Entitäten haben Positionen	Grundlegende Spielmechanik. Relevant für das generische Datenmodell.
FAA#16	Entitäten haben ein Bild	Grundlegendes Spielkonzept. Relevant für das generische Datenmodell.
FAA#17	Auf Entitäten können Aktionen liegen	Benötigt für das Erreichen von Erfolgen und Zielen.
FAA#18	Entitäten relativ zum Sichtzentrum	Grundlegende Spielmechanik. Wichtig für die Darstellung.

Tabelle 3.4: Funktionale Anforderungen der Anwendungen bezüglich Entitäten

Die Anforderungen für Ziele stimmen nach 3.5 überein, auch wenn der Editor verlangt, dass diese konfigurierbar sein müssen, während den anderen Anwendungsfällen einfacher Programmcode genügen würde, auch wenn diese hiervon profitieren können.

	Anforderung	Erklärung
FAA#19	Es gibt verschiedene Arten von Zielen	Beispielsweise Zeitlimits, mögliche Fehlschläge. Wichtig für späteres System.
FAA#20	Ziele sind einer Reinkarnation eines Levels zugeordnet	Ziele sind entsprechend nicht levelübergreifend gültig. Wichtig für das generische Datenmodell.
FAA#21	Ziele entscheiden, ob ein Level geschafft wurde	Grundlegende Spielmechanik.
FAA#22	Ziele können konfiguriert werden	Wichtig für den Editor 2.4.3 und den Grad der Generizität.

Tabelle 3.5: Funktionale Anforderungen der Anwendungen bezüglich Zielen

Szenarien haben bei den Anforderungen 3.6 wiederum einen größeren Unterschied, da diese vollkommen optional sind. Lediglich bewertende Anwendungen 2.4.2 benötigen ein Szenario. Bei Geschichten 2.4.1 wären anstelle dessen Kapitel denkbar.

	Anforderung	**Erklärung**
FAA#23	Ein Szenario enthält Level	Wichtig für bewertende Anwendungen 2.4.2. Wichtig für das generische Datenmodell.
FAA#24	Ein Szenario ist optional	Geschichtsanwendungen 2.4.1 brauchen keine Szenarien.
FAA#25	Ein Szenario kann freigeschaltet werden	Wichtig für bewertende Anwendungen 2.4.2 und das Subsystem.

Tabelle 3.6: Funktionale Anforderungen der Anwendungen bezüglich Szenarien

Akustische Signale sind zwar komplexer als beispielsweise Szenarien, gehören aber zur grundlegenden Spielmechanik und entsprechend sind Unterschiede höchstens in Details zu finden (siehe 3.7).

	Anforderung	Erklärung
FAA#26	Ein akustisches Signal kann richtungsbehaftet sein	Grundlegende Spielmechanik. Wichtig für Audio-Wiedergabe.
FAA#27	Richtungsbehaftete, akustische Signale sind relativ zum Sichtzentrum	Grundlegende Spielmechanik. Wichtig für Audioausgabe.
FAA#28	Die Position der Quelle eines akustischen Signals kann sich ändern	Wichtig bei höheren Schwierigkeitsgraden und bei dem Subsystem.
FAA#29	Die Lautstärke eines akustischen Signals kann sich ändern	Trägt zu variablerer Schwierigkeit bei. Wichtig für das Subsystem und die Audioausgabe.
FAA#30	Ein akustisches Signal kann wiederholt werden	Wichtig für Hintergrundgeräusche und niedrige Schwierigkeitsgrade. Wichtig für die Audioausgabe.

Tabelle 3.7: Funktionale Anforderungen der Anwendungen bezüglich akustischen Signalen

Erfolge sind entsprechend 3.8 gerade für bewertende Anwendungen 2.4.2 interessant, können aber ebenso bei Geschichten 2.4.1 nützlich sein. Der Editor 2.4.3 benötigt diese lediglich, um diese Anwendungen unterstützen zu können, auch wenn diese hier nur eine Konfiguration anstelle von Code benötigen.

	Anforderung	Erklärung
FAA#31	Ein Erfolg ist levelübergreifend	Wichtig für die Persistenz innerhalb des Subsystems vom generischen Datenmodell.
FAA#32	Erfolge können nach dem Erfüllen von Zielen erhalten werden	Wichtig für Spielmechanik innerhalb des Subsystems.
FAA#33	Erfolge können nach Aktionen erhalten werden	Wichtig für Spielmechanik innerhalb des Subsystems.

Tabelle 3.8: Funktionale Anforderungen der Anwendungen bezüglich Erfolgen

3.2 Funktionale Anforderungen für das Datenmodell

Aufgrund der Art der Unterschiede ist es zu erwarten, dass jede neue Anwendung auch neue Effekte, Verhaltensweisen und Ziele mit sich bringt.

Auch auf Ebene der Persistenz gibt es hier sehr große Unterschiede, beispielsweise müssen bestimmte Daten wie Erfolge, Level oder der aktuelle Spielfortschritt auf unterschiedlichen Ebenen gelöst werden.

Offensichtlich ist ebenfalls, dass sich auch je nach umgesetzter Anwendung das Datenmodell ändern kann, auch wenn die Grundprinzipien die gleichen bleiben.

Wichtig ist gerade im Hinblick auf den Editor, der eine Art Simulation besitzt und bezüglich der Ausführung von Leveln innerhalb mobiler Anwendungen, dass Kopiervorgänge eines Levels einfach und ohne Nebeneffekte geschehen müssen.

Auch im Falle einer Weiterentwicklung des Genre ins dreidimensionale unter Berücksichtigung der Entfernungs- oder gar Oben-Untenwahrnehmung sollte dieses Datenmodell im Idealfall bestand haben.

Aufgrund der hohen benötigten Raten für die Auswertung von Positionsdaten muss auch die Ausführung performant genug sein, dass Endanwender keine störenden Zeitver-

schiebungen bemerken und dass der Akku allerdings auch nicht zu stark beansprucht wird [4].

	Anforderung	Erklärung
FA#1	Eine Hierarchie muss definierbar aber auch optional sein	Aufgrund optionaler Szenarien, siehe 3.6
FA#2	Das Sichtfeld muss viele Quellen zu ändern sein	Aufgrund unterschiedlicher Eingabeszenarien, siehe 3.3
FA#3	Die Anwendung muss die volle Kontrolle über die Oberfläche behalten	Wegen rein-akustischer Ausgabe, siehe 3.1
FA#4	Verschiedene Instanzen eines Grunddatums sind unabhängig	Unabhängige Level, siehe 3.2
FA#5	Auf allen Elementen können beliebige Eigenschaften definiert werden	Unabhängige Level, vgl. 3.4, 3.5, 3.7, 3.8
FA#6	Es können beliebige Elemente definiert werden	Unabhängige Level, vgl. 3.4, 3.5, 3.7, 3.8
FA#7	Erweiterungen können Audioausgabe steuern	Aufgrund der vielen Abhängigkeiten mit akustischen Signalen 3.7

Tabelle 3.9: Funktionale Anforderungen

3.3 Nichtfunktionale Anforderungen für das Datenmodell

Die einzigen im Weiteren betrachteten nichtfunktionalen Anforderungen betreffen lediglich die Performance und das Ressourcenmanagement. Andere beträfen hauptsächlich die Benutzerinteraktion, welche allerdings für das Datenmodell unerheblich ist, sofern das Subsystem keinen übermäßigen Ressourcenverbrauch hat.

Um sowohl einen Editor als auch Spielesoftware erstellen und abbilden zu können, muss das zu entwerfende System trotz der Einschränkung auf die mobile Plattform iOS auch auf macOS-Systemen lauffähig sein. Dies hat zur Folge, dass das System im Kern größtenteils plattformunabhängig sein muss, aber auch das Einbinden plattformspezifischer Bibliotheken erlauben muss. Hieraus resultiert zwangsweise ein modularer Aufbau des Grundsystems, um bestimmte Eigenschaften und Verhaltensweisen sowohl auf den Anwendungsfall als auch auf die Plattform optimieren zu können.

Da die am häufigsten realisierten Anwendungen, die auf dem zu entwerfenden Datenmodell basieren, auf mobilen Endgeräten laufen, muss dieses auch auf aktuellen aber schwächeren Geräten flüssig lauffähig sein. Dies betrifft sowohl den Speicherverbrauch als auch die Leistungsaufnahme. Allgemein ist die Stabilität offensichtlich ein wichtiger, nicht zu vernachlässigender Aspekt, wenn es um Endanwendungen geht. Dies betrifft ganz besonders Bibliotheken, da diese typischerweise von mehreren Anwendungen verwendet werden.

Letztendlich ergeben sich die folgenden nichtfunktionalen Anforderungen für das generische Datenmodell.

	Anforderung	Erklärung
NFA#1	Plattformunabhängiger Kern	Der Kern muss sowohl auf macOS als auch auf iOS lauffähig sein.
NFA#2	Plattformspezifische Erweiterungen möglich	Da der Kern plattformunabhängig sein muss, wird eine Optimierung auf Plattformen benötigt.
NFA#3	Modularer Aufbau	Stellen, bei denen Plattformoptimierungen benötigt werden sind nicht vorhersehbar genug.
NFA#4	Moderater Speicherverbrauch	Der Speicher ist auf mobilen Endgeräten zwar eingeschränkt, allerdings werden nicht viele Daten verarbeitet.
NFA#5	Moderater Leistungsverbrauch	Auf mobilen Endgeräten ist nur eingeschränkt Leistung verfügbar, allerdings sind Inhalte nicht nur statisch.
NFA#6	Hohe Stabilität	Da das System als Bibliothek verwendet wird, läuft diese in entsprechend mehr Applikationen.
FA#7	Einfache Handhabung für Entwickler	Aufgrund des generischen Charakters des Datenmodells.

Tabelle 3.10: Nichtfunktionale Anforderungen

4 Konzeption

Das Grundsystem zur Verarbeitung und Repräsentation eines generischen Datenmodells für die beschriebenen Anwendungsfälle basiert auf der grundlegenden Idee, dass prinzipiell möglichst wenig Vorgaben an ein explizites Datenmodell gestellt werden. Dies ermöglicht es auch Datenmodelle und Anwendungsfälle abbilden zu können, an die bei dem Entwurf des Grundsystems noch nicht gedacht werden konnte.

So wäre zuzüglich zu den zuvor genannten Anwendungsfällen auch das dynamische Nachladen von Leveln möglich die auch eventuell von anderen Benutzern entworfen wurden.

Die gesamte Datenstruktur wird immer als Baumstruktur angesehen, wobei jeder einzelne Knoten beliebige Eigenschaften haben kann, die zur Compile-Time inklusive ihres Typen feststehen. Außerdem gelten unterschiedliche Knoten-Typen auch auf Typebene als unterschiedlich und in ihren Eigenschaften voneinander isoliert.

Diese Baumstruktur repräsentiert den Gesamtzustand und wird als unveränderlich betrachtet, was bedeutet, dass jede schreibende Aktion auf diesem immer in einem neuen Zustand resultiert anstatt den aktuellen Zustand zu verändern. Unerwünschte Seiteneffekte werden durch dieses Konzept verhindert, da das Ausführen eines einzelnen Levels immer wieder zum gleichen Ergebnis führt.

Aufgrund der starken Anforderungen bezüglich der Flexibilität im Hinblick auf die Persistenz, wird lediglich eine Schnittstelle angeboten, die über jede Änderung informiert, damit diese anschließend in einer darunterliegenden Programmschicht angewendet werden kann. Dies bedeutet allerdings auch, dass eben diese plattformabhängige Schicht außerhalb des Grundsystems liegt.

Bezüglich der Ausgabe von Bild- und Audiodaten werden Änderungen zwischen verschiedenen Zuständen extrahiert und an die entsprechenden plattformabhängigen Schnittstellen weitergeleitet.

4.1 Eingesetzte Technologien und Konzepte

An einigen Stellen basiert die Implementierung des generischen Datenmodells auf externen Bibliotheken, Mechanismen oder Konzepten. Einerseits fördert dies die Verständlichkeit des Konzeptes bzw. der Funktionsweise in der Gesamtheit, wenn die verwendeten Konzepte bereits bekannt sind, und andererseits wird die Implementierung vergleichbarer bezüglich Skalierungseffekten und deren Lösung sowie dem Verhalten in Extremfällen bezüglich mangelnder Kapazitäten, wie es bei mobilen Endgeräten im Vergleich zu den meisten Desktopsystemen der Fall ist. Dies hilft insbesondere bei der Validierung vond nichtfunktionalen Anforderungen sowie eventuellen Anwendern der resultierenden Bibliothek bei der Lösungsfindung.

4.1.1 Programmiersprache

Als zugrundeliegende Programmiersprachen kommen nur jene in Frage, die von Apple offiziell und unmittelbar unterstützt werden, um die Interoperabilität mit bereitgestellten Werkzeugen und Software zu gewährleisten.

Dies trifft sowohl auf Objective C als auch auf das noch junge Swift zu. Objective C hat den Vorteil, dass alle integrierten Systembibliotheken auch in dieser Sprache geschrieben wurden. Dies liegt einerseits an dem unterschiedlichen Alter beider Sprachen, aber auch andererseits an der noch fehlenden ABI-Kompatibilität Swifts [5]. Andererseits schließt die Nutzung der einen Sprache die Verwendung der anderen nicht aus. Für Swift spricht allerdings trotz seines jungen Alters die verglichen mit Objective C inklusive der Foundation Bibliothek, die als Ersatz für die mangelnde Standardbibliothek gilt, eine größere Auswahl an Plattformen. Beispielsweise gibt es für die Bibliotheken Foundation und XCTest offizielle Portierungen, die außerhalb der Apple-eigenen Betriebssystemen

lauffähig sind. Des Weiteren bietet Swift den Vorteil modernerer Sprachfeatures sowie einer breiteren und aktiveren Open Source Community.

Diese Vorteile führen in ihrer Gesamtheit zur Wahl der Programmiersprache Swift.

4.1.2 Verwendete Entwurfsmuster

In Swift gibt es viele spezifische strukturelle Entwurfsmuster, die im Rahmen dieser Implementierung verwendet werden. Ihr Ziel ist es vornehmlich einige Laufzeitfehler auf das Typsystem von Swift zu verlagern. Die für das System konzeptionell relevanten Entwurfsmuster werden nun im Folgenden vorgestellt.

Phantom Type

Ein in Swift verbreitetes Konzept ist der Phantom Type, der einzig der Differenzierung von generischen Typen dient und keinerlei eigene Logik enthält [6, Seite 251ff]. Wie in A.1 dargestellt, ist dies im Normalfall eine Enumeration ohne jegliche Fälle, was zu einem Typen führt, von dem es niemals Werte geben kann.

Dies ermöglicht es beispielsweise Interaktionen und Operationen einzuschränken. Denkbar wäre hier die Deklaration von Einheiten, wie es vereinfacht in A.2 dargestellt wird.

Ein einfaches Beispiel wäre es zu verhindern, dass man die Anzahl von Äpfeln nicht mit Birnen vergleichen kann.

So wäre es nach Definition der Typen `Apple` und `Pear` nicht möglich `Amount<Apple>` und `Amount<Pear>` zu vergleichen, da dies vom Typsystem abgefangen wird und zu einem Fehler beim Kompiliervorgang führt.

Andernfalls könnte dies zur Laufzeit unbemerkt zu nicht beabsichtigten Berechnungen und invaliden Ergebnissen führen.

Type Erasure

In einigen Fällen kann das Typsystem von Swift auf direktem Wege nicht mehr genügend Sicherheit bieten, ohne auf zu allgemeine Typen wie beispielsweise `Any` zurückzugreifen. Dies ist vor allem bei der Verwendung von Protokollen mit assoziierten Typen der Fall. So ist es beispielsweise nicht mehr möglich diese in einem nicht generischen Kontext zu verwenden.

Ein einfacher und offensichtlicher Fall stellt wie in A.3 dargestellt die Verwendung eines Protokolls mit assoziiertem Typen in Verbindung mit einem Array dar.

Mittels Type Erasure kann diese Einschränkung im Typsystem umgangen werden [6, Seite 319ff]. Prinzipiell gibt es mehrere Möglichkeiten. Eine gängige Variante ist die Deklaration wie in A.4 eines expliziten und generischen Typen, um in einem nicht generischen Kontext verwendet werden zu können.

4.1.3 Lens

Manchmal muss zur Laufzeit auf Eigenschaften von Werten oder Objekten lesend oder schreibend zugegriffen werden, ohne dass es zur Kompilierzeit innerhalb des Kontexts möglich ist diese aufzulösen. In dynamischen und schwach typisierten Programmiersprachen, wie beispielsweise in JavaScript, liegt es nahe lediglich einen String zu verwenden. Dies erlaubt zur Laufzeit den lesenden und schreibenden Zugriff auf diese Eigenschaft.

Wenn es in die Richtung von statisch typisierten Programmiersprachen wie Swift oder Haskell geht, benötigt man hierfür Hilfsfunktionen, da diese nicht ohne weiteres mithilfe von Strings auf die angefragten Eigenschaften zugreifen können. Insbesondere wenn man hier von unveränderlichen Werten spricht, würde auch eine statisch auflösbare Zuweisung nicht mehr reichen, da als Ergebnis eine Kopie erwartet wird.

Zu diesem Zweck gibt es das Konzept einer Lens [7]. Diese enthält lediglich eine Funktion für den lesenden Zugriff auf die Eigenschaft bei gegebenem Wert, sowie eine weitere für den schreibenden Zugriff, welche den Wert sowie den neuen Wert der betrachteten

Eigenschaft benötigt und eine Kopie mit der neuen Eigenschaft zurück liefert. Eine Implementation des Lens-Typen findet sich im Anhang A.21.

Mit RxLens gibt es eine Implementierung dieses Konzepts, welche außerdem einige Funktionen für eine bessere Interoperabilität mit RxSwift bietet [8].

Beispielsweise finden sich hier innerhalb der Implementation des generischen Datenmodells verwendete Operatoren und Erweiterungen zum Erstellen von Subjects [8].

4.1.4 Validated Type

Sobald mit dynamischen Datenstrukturen gearbeitet wird, steigt die Relevanz von Validieren und dem Extrahieren der benötigten Teildaten, gerade wenn diese trotzdem typsicher verarbeitet werden sollen.

Eine für diesen Anwendungsfall entworfene Bibliothek ist Validated Type [9]. Ziel ist es dieses Validieren statisch und innerhalb des Typsystems zu deklarieren, wofür zu jeder Validation ein eigener Typ angelegt wird, der den eigentlichen Vorgang kapselt.

Nach der Deklaration der Validierungslogik und -typen können die zugrundeliegenden Werte validiert werden [9]. Dies geschieht analog zu A.5 durch die Instanziierung von `Validated` mit dem `Validator` als generisches Argument. Nur im Falle von validen Werten erhält man eine Instanz von `Validated` zurück, andernfalls `nil`, wie es A.6 zu entnehmen ist.

Anschließend kann mittels einer Protokollerweiterung nach A.7 ein berechnetes Attribut hinzugefügt werden, was die spätere Verwendung vereinfacht.

Unter der Bedingung, dass der zugrundeliegende Werte-Typ Value Semantics unterliegt, kann auf Force Casts zurückgegriffen werden, da bereits bekannt ist, dass die definierten Bedingungen erfüllt sind und dass sich diese auch aufgrund der Value Semantics nicht ändern können.

4.1.5 Dependency Injection

Wartbarkeit und Stabilität sind gerade innerhalb der Softwareentwicklung wichtig. Ein Ansatzpunkt, um diese zu verbessern ist das automatisierte Testen, welches allerdings durch explizite Abhängigkeiten erschwert werden kann. Entsprechend nahe liegt es diese zu reduzieren, was durch die Verwendung von Dependency Injection möglich ist.

Abhängigkeiten werden aus dem betrachteten Code entfernt, indem lediglich auf einer definierten Schnittstelle (in Swift ein Protokoll) gearbeitet wird, anstatt auf der direkten Implementierung dieser. Dies führt dazu, dass der resultierende Quellcode weniger Abhängigkeiten auf anderen Abstraktionsniveaus beziehungsweise zu anderen Teilsystemen hat. Offensichtlicherweise vereinfacht dies auf kurze Sicht die Implementierung von Unit Tests und auf lange Sicht werden die Hürden für eine Restrukturierung verringert, insbesondere wenn die darunterliegenden Abhängigkeiten ersetzt werden. Beispielsweise können die Netzwerk- oder Persistenzschichten im Rahmen von Unit Tests durch In-Memory-Varianten ersetzt werden.

Im Rahmen des Datenmodells wird eine Form der Dependency Injection verwendet, die auf typisierten Schlüsseln arbeitet anstatt auf den reinen Typen. Dies hat den Vorteil, dass mehrere Werte des gleichen Typs injiziert werden können, was innerhalb des Konzepts vom generischen Datenmodell verwendet wird.

Diese Art der Dependency Injection wird von der Bibliothek `EasyInject` implementiert, die Container von Dependencies als Datenstruktur betrachtet und als `Injector` bezeichnet, wobei die Kombination aus einem Schlüssel und dem injizierbaren Typen als `Provider` bezeichnet wird [10], wie es aus A.8 zu entnehmen ist.

Die Bibliothek bietet neben dem `StrictInjector`, der bereits bei Instanziierung alle Werte auflöst, auch einen `LazyInjector` mit Lazy-Evaluation, der erst bei Zugriff angefragte Werte auflöst, einen `GloabalInjector`, welcher anstelle von Value-Semantics Reference-Semantics folgt [11], und dem `ComposedInjector`, falls mehrere Injektoren kombiniert werden müssen [10].

Im hier vorgestellten Konzept werden die Daten als Baumstruktur betrachtet, wobei jeder einzelne Knoten einen `Injector` darstellt. Die `Provider` stellen die einzelnen Attribute und Relationen dar.

4.1.6 Functional Reactive Programming

Functional Reactive Programming ist ein Programmierparadigma, in dem versucht wird event-basierte und asynchrone Datenflüsse durch Datenstrukturen zu deklarieren. Diese Datenstrukturen können mittels funktionalen Operatoren kombiniert werden.

Vorteile dieses Paradigmas sind neben einem einheitlichem Datenfluss und, gerade in einem nicht rein funktionalem Umfeld, die explizitere Deklaration von Seiteneffekten [12, Seite 25ff].

Im Bereich der nativen Entwicklung für macOS und iOS in Swift gibt es zwei große Implementationen dieses Paradigmas: ReactiveSwift und RxSwift. Beide Bibliotheken gehen aus ReactiveX hervor, was eine einheitliche und programmiersprachenübergreifende Interpretation des Functional Reactive Programmings ist. Aufgrund der größeren Nähe zu Portierungen anderer Sprachen, wählen wir RxSwift, da die Schnittstellen hier allgemeingültiger sind und in mehr Programmiersprachen existieren [13].

Beispielsweise gibt es ReactiveX-Portierungen für JavaScript, Java, Kotlin, C++, C#, Ruby und Go [13].

Die zentrale Klasse innerhalb ReactiveX und RxSwift stellt die des Observables dar, welcher auch als Stream oder Sequenz bezeichnet wird [12, Seite 55].

Ein `Observable<T>` beschreibt eine Sequenz von Ereignissen, die auch über die Zeit hinweg emittiert werden können.

Seitens der Ereignisse gibt es das Eintreffen eines neuen Wertes, auch `next`, das Ende der aktuellen Sequenz (`completed`), sowie das Auftreten eines Fehlers (`error`), nach dem ebenfalls keine weiteren Ereignisse emittiert werden können [12, Seite 45ff].

Entsprechend ergeben sich die in 4.1 veranschaulichten Fälle. Normalerweise emittieren Streams einige Werte (durch die Kreise `A` und `B` symbolisiert) und werden dann fehlerfrei

beendet, was durch einen senkrechten Strich `|` gekennzeichnet wird. Wie im zweiten Beispiel müssen Streams allerdings nicht beendet werden, sondern können beliebig lange laufen. Im dritten Fall sieht man einen fehlschlagenden Stream, was typischerweise durch ein `X` gekennzeichnet wird. Nach dem Auftreten eines Fehlers kann kein weiteres Ereignis mehr folgen.

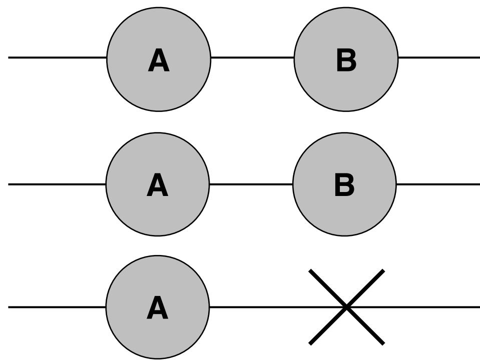

Abbildung 4.1: Lebenszyklus eines Observables

Observables können mittels Operatoren zu komplexeren kombiniert werden, welche ein weiterer, integraler Bestandteil von RxSwift sind. Operatoren sind Methoden von `Observable<T>`, deren Zweck die Manipulation des Datenstroms darstellen und in neuen Streams resultieren [12, Seite 34ff]. Einige Operatoren basieren auf grundlegenden funktionalen Bestandteilen, wie beispielsweise `map`, `filter`, `reduce` und `zip`.

Mittels dem statischem `Observable<T>.of(_:)`, welches die Rolle von `pure` oder auch `return` übernimmt, und dem nicht-statischem `flatMap(transform:)` lassen sich die Monadengesetze erfüllen [11, Seite 185f].

Elementar ist ebenfalls das Konzept des Observers, der Ereignisse entgegennimmt anstatt sie wie ein Stream zu emittieren [11, Seite 31f]. Wenn dieser ebenfalls ein Observable ist, also sowohl auf Ereignisse wartet als auch selbst welche aussendet, wird er auch als Subject bezeichnet [14].

Prinzipiell werden Streams nur dann ausgeführt, wenn sie dies explizit mittels eines Aufrufes seiner Methode `subscribe()` angegeben wurde, was eine sogenannte

`Subscription` zurückgibt. Erst wenn auf dieser die Methode `dispose()` aufgerufen wurde oder der Stream durch `error` oder `completed` terminiert, wird der Lebenszyklus des Streams beendet [12, Seite 55]. Erfolgt dieser Aufruf nie, ist von einem Memory Leak auszugehen [12, Seite 58].

Zum Gruppieren mehrerer Subscriptions gibt es den `DisposeBag` [12, Seite 55], welcher auch bei Deinitialisierung alle darin enthaltenen Subscriptions freigibt [15].

Sobald Code asynchron ausgeführt werden soll, kommt im Bereich der iOS- und macOS-Entwicklung häufig eine Dispatch Queue, die Abstraktion von Threads darstellt, ins Spiel [16]. Das entsprechende Rx-Äquivalent zu den Dispatch Queues ist der Scheduler, der es ermöglicht Teile eines Streams mit einer anderen Nebenläufigkeitsstrategie auszuführen [17]. Im Falle von RxSwift basieren die meisten auf den Dispatch Queues [12, Seite 36f], allerdings können diese auch, wie im Falle von RxJS, auch auch auf einem zeitlichen Versatz beruhen [18, Seite 93].

4.1.7 Unidirektionaler Datenfluss

In zunehmend komplexeren Anwendungen wird der Datenfluss immer komplexer und schwerer zu verstehen. Vor allem ist es häufig nicht möglich den gesamten Zustand eines Systems zu beschreiben.

Ein Ansatz dem entgegenzuwirken basiert darauf für alle Datenflüsse nur noch eine Richtung zu erlauben. Somit entstehen keine rekursiven oder bidirektionalen Abhängigkeiten mehr, die das Gesamtverhalten durch gegenseitige Wechselwirkungen in Form von Seiteneffekten nur schwer vorhersehbar machen.

Nach [19] existiert ein Redux Store, welcher den gesamten Zustand der Applikation enthält. Da dieser Zustand unveränderlich ist, kann es über diesen nicht zu unerwünschten Nebeneffekten kommen.

Die Aufgabe der View ist es wie in 4.2 dargestellt nur den aktuellen Zustand zu repräsentieren und Ereignisse wie den Tap eines Buttons in Form von Aktionen auszusenden [19]. Diese enthalten alle zur Durchführung notwendigen Informationen, enthalten aber explizit keine Logik [19].

Sobald eine Aktion beim Store angekommen ist, wendet dieser alle Reducer nacheinander auf den aktuellen Zustand und die Aktion an [19]. Ein Reducer stellt lediglich eine Funktion dar, die eine Aktion ausführt und eine neue Kopie des Zustands zurückgibt. Nach der Ausführung eines Reducers wird der neue resultierende Zustand dem nächsten übergeben. Am Ende wird der nach Ausführung aller Reducer resultierende Zustand als aktuell gesetzt [19]. Anschließend wird die View wieder über die Änderung benachrichtigt.

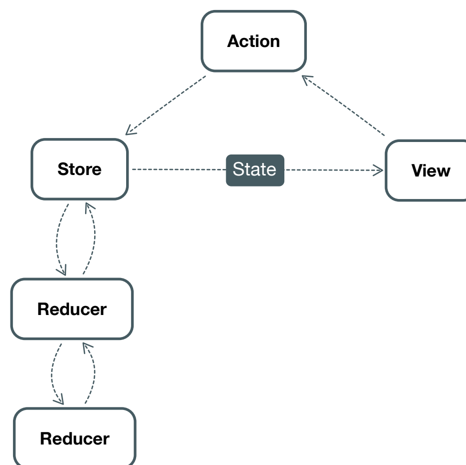

Abbildung 4.2: Unidirektionaler Datenfluss

Im Rahmen der Implementation wird eine eigene Implementierung auf Basis des Functional Reactive Programming verwendet, um die Integration in andere Teile zu erleichtern und zu vereinheitlichen.

4.1.8 Sourcery

In manchen Anwendungsfällen kann viel strukturell ähnlicher Code entstehen, der nicht durch Wiederverwendung von wiederum anderem Code zusammengefasst werden kann. Ziel von Sourcery ist das Generieren dieser Art von repetitivem Code, wodurch

sich wiederholende Aufgaben automatisiert werden können und entsprechend einige Fehlerquellen ausgeschlossen werden [20].

Sourcery extrahiert zu diesem Zweck Typ- und Metadaten aus dem Swiftcode, welche beispielsweise auch über Kommentare, Annotationen genannt, generiert werden können [20]. Die dadurch gewonnenen Daten werden anschließend mittels Templates wieder neuen Swiftcode generieren.

4.2 Architektur

Die Implementierung um das generische Datenmodell ist gemäß den Anforderungen in verschiedene Module unterteilt.

Wie von der Grafik 4.3 visualisiert, gibt es zum einen den plattformunabhängigen ReactiveCore, welcher im Wesentlichen den Redux Store implementiert, und zum anderen das Modul `ReactiveAu3dio`, welches beispielhaft ein Datenmodell implementiert.

Auf diese aufbauend werden ReactiveUI und ReactiveAV implementiert, welche für die audiovisuelle Repräsentation zuständig sind. RxLensHelper und RxValueToObjectTranslator sind Hilfsmodule, die spezifische Probleme lösen, aber nicht auf dem ReactiveCore basieren.

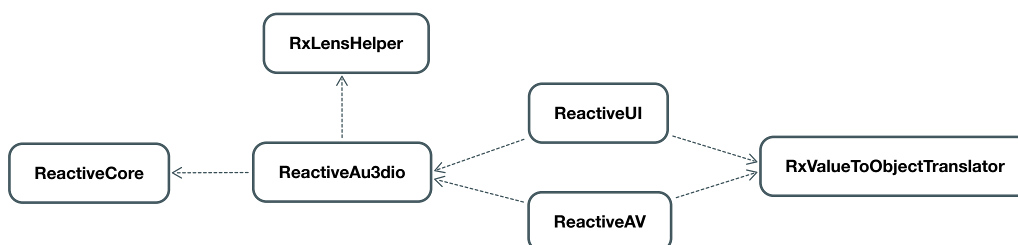

Abbildung 4.3: Struktur und Abhängigkeiten der Module

4.2.1 Reactive Core

Das Modul ReactiveCore enthält den zentralen Kern des Systems in Form eines Redux Stores, der mithilfe von ReactiveX implementiert wurde.

Die zentrale Klasse stellt `Store` dar, welcher vereinfacht im Anhang A.10 zu finden ist. Dieser besitzt einen einzelnen Stream aller valider Zustände, der von der gesamten Applikation verwendet wird. Da dies lediglich eine für den beschriebenen Anwendungsfall spezifische Implementierung des Redux Patterns ist, kann als Wurzel der Baumstruktur ein `Injector` angenommen werden, wodurch sich der Name Single Shared Injector Observable ergibt, der im Folgenden lediglich als SSIO bezeichnet wird. Dieser SSIO ist prinzipiell nicht endlich (vgl. [12, Seite 33]), allerdings wäre es hier sinnvoll bei Beendigung der gesamten Applikation ein `complete`-Event zu versenden, damit die Datenschicht noch vor Beendigung alle Daten schreiben kann.

Auch die aus Redux bekannten Aktionen finden sich hier im Kontext von ReactiveX wieder: alle Aktionen werden von einem Observer entgegen genommen. Sobald ein Ereignis in Form einer Aktion ausgelöst wird, werden nacheinander alle Reducer auf den aktuellen Zustand ausgeführt und das hieraus resultierende Ergebnis wird nun von dem SSIO emittiert. Somit gibt es eine einfache Möglichkeit auf Änderungen des aktuellen Zustands zu reagieren.

Die Reducer müssen wie A.11 zeigt immer bereits bei der Deklaration des Stores angegeben werden und können nicht nachträglich abgeändert werden, was einige Grenzfälle und Fehlerquellen (gerade im Bezug auf Parallelität) eliminiert. Sollte ein solches Verhalten dennoch erwünscht werden, kann einfach eine Funktion in Kombination als Zwischenschritt eingeführt werden. Außerdem werden eigens für Reducer einige Hilfsmittel für die Instanziierung von komplexeren Reducern (z.B. `actionReducer` und `combineReducer`) und Reducer für das Debugging (z.B. `actionLogger` und `stateLogger`) bereitgestellt.

Für komplexere Effekte wird zusätzlich ein Injector angeboten, bei dem Subscriptions abgelegt werden können. Dies kann entweder in Form eines DisposeBags geschehen, wenn sich mehrere Subscriptions die gleiche Lebensdauer teilen (beispielsweise die

Dauer des aktuellen Levels), oder in Form einer einzelnen Subscription. Für andere Arten von Daten ist dies nicht gedacht. Durch dieses Vorgehen können Memory-Leaks durch nicht mehr freigegebene Subscriptions effektiv vorgebeugt werden [12, Seite 58].

4.2.2 Lens Inject

LensInject basiert lediglich auf RxLens und EasyInject. Es ermöglicht eine einfachere und kompakte Deklaration von Lenses, die auf einzelne Attribute eines Knotens abbilden sollen. Dies wird durch die Verwendung eines Injectors als Knoten möglich, da somit die Attribute des Knoten-Typs in Form von Providern auf Ebene von Werten vorliegen.

Diese Erweiterung ermöglicht es nun anhand von `Lens.with(injected:)` in nur einer Zeile eine Lens zu definieren (siehe A.9).

4.2.3 Datenmodell

Eine Beispielimplementation eines möglichen Datenmodells findet sich innerhalb des `ReactiveAu3dio` Moduls wieder.

Darin wurde ein mögliches Datenmodell für die bewertende Anwendung 2.4.2 entwickelt, welche aufgrund des Konzeptes auch die eines Editors 2.4.3 darstellen kann. Entsprechend gibt es dort Deklarationen für Level, Entitäten, Ziele und, im Gegensatz zur Geschichtsanwendung unter 2.4.1, auch ein Szenario. Nicht aufgenommen wurden Erfolge, da diese lediglich für die reine Persistenzschicht einen Unterschied darstellen. Diese genannten Elemente werden durch Sourcery 4.1.8 generiert, was von unnötig repetitiven Code abstrahiert.

Des Weiteren gibt es hier auf klassischem Wege deklarierte Datentypen zur Repräsentation von akustischen Signalen, zur aufbereiteten Darstellung von Entitäten und das Konstrukt einer Position, die wie alle anderen Elementen des Datenmodells Value Semantics folgen.

Alle innerhalb dieses Moduls deklarierten Datentypen sind plattformunabhängig und bieten entsprechend keine Form der audio-visuellen Ausgabe. Theoretisch wäre dieser Teil des Datenmodells somit selbst auf Android lauffähig [21].

Innerhalb des Datenmodells werden Level den Szenarien untergeordnet, denen wiederum die Entitäten unterliegen. Da das Datenmodell als Baumstruktur mit nur einer vorgegebenen Wurzel betrachtet wird, müssen alle Einstiegspunkte für dieses auf den SSI (Single Shared Injector) definiert sein. Im diesem Falle wird also entsprechend zu Abbildung 4.4 eine Liste aller Szenarien mittels Protokollerweiterungen auf einem Injector als Eigenschaft deklariert und hinzugefügt. Das Beispiel A.17 für eine solche Deklaration wird in folgenden Kapitel 4.2.3 genauer erläutert. Neben offensichtlichen Aspekten, wie der Relationen zwischen Leveln, Zielen und Entities, fällt auf, dass sowohl Level als auch Entitäten sowohl eine Position als auch Sounds besitzen. Die Position für Entitäten ergibt sich aus der grundlegenden Spielmechanik. Die Position des Levels repräsentiert den Point of View und damit das Zentrum des Sichtfeldes. Somit kann innerhalb eines Editors der Point of View frei konfiguriert werden. Bezüglich einer Entität gibt es nur eine Art von Signal, welches immer richtungsbehaftet ist, während Level beliebig viele Tonspuren unterstützen. Dies könnte sich aber auch vor allem bei Geschichten 2.4.1 als vorteilhaft erweisen. Außerdem besitzen sowohl Level als auch Entitäten Namen und Bilder in Form eines Strings zur Unterscheidung. Die eigentlichen Bilddaten liegen hier allerdings unter dem angegebenen Namen auf der Festplatte. Bei dem Level wird dies für den Hintergrund verwendet, während das Bild der Entität diese an der angegebenen Stelle innerhalb der View-Hierarchie repräsentiert.

Ziele besitzen hier lediglich einen Namen, erst durch die explizite Implementation dieser erhalten diese zueinander unterschiedliche Attribute. Der Name des Zieles dient vor allem einer einfachen Unterscheidung der Art des Zieles innerhalb von Reducern und vom SSIO abhängigen Streams.

Nach dem Starten eines Levels wird außerdem ein schneller Zugriff auf eine Kopie des ursprünglichen Levels und all seiner Entitäten benötigt. Zu diesem Zweck erhält der SSI außerdem das optionale `currentLevel`-Attribut (vgl. 4.4). Aufgrund der Value Semantics der betroffenen Datentypen wird durch diesen Kopiervorgang innerhalb der

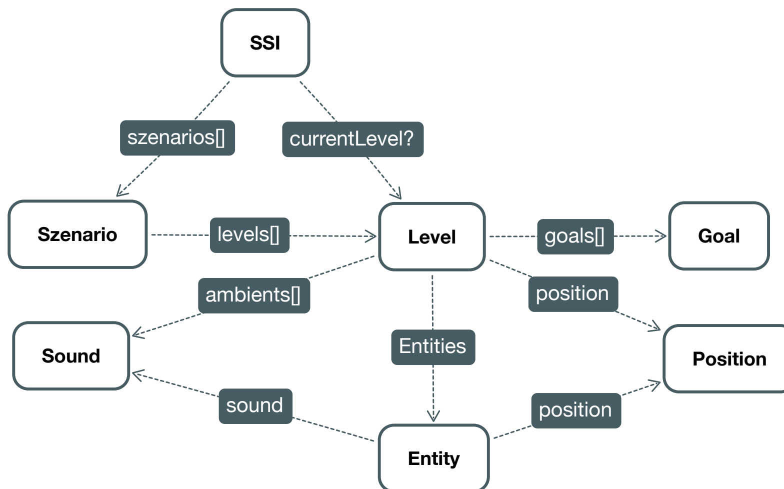

Abbildung 4.4: Relationen innerhalb des Datenmodells

Baumstruktur bereits eine Kopie angelegt, wodurch mehrere Ausführungen eines Levels seriell von einander unabhängig und isoliert stattfinden können.

Der `Sound`-Typ kapselt lediglich den Namen der Tonspur und die Lautstärke in der dieser wiedergegeben werden soll. Der simulierte Ursprung des akustischen Signals findet erst auf einer plattformspezifischen Ebene statt.

Im Rahmen der visuellen Darstellung, kann eine `DisplayEntity` aus einer bestehenden Entität erstellt werden, welche den Namen, das Bild und seine Position in einem Wert bündelt, sofern diese vorhanden sind. Die Position wird durch eine prozentuale Angabe (von 0.0 bis 1.0) auf der X- und Y-Koordinate definiert. Bei einem eventuellen Überlauf während Berechnungen werden diese Angaben wieder normalisiert. Wie unter iOS üblich liegt hier der Koordinatenursprung von `Position(x: 0.0, y: 0.0)` in der oberen, linken Ecke und die Ecke rechts unten somit bei `Position(x: 1.0, y: 1.0)` [22]. Unter macOS weicht dies zwar vom Standard ab, allerdings kann dies einfach übersetzt werden [22]

Aktionen

Erst Interaktionen mit Endanwendern macht das entworfene Datenmodell sinnvoll. Durch die benötigte Plattformunabhängigkeit kann die Logik, die durch die Interaktion ausgelöst werden soll, allerdings nicht mit ihrem Auslöser zusammen deklariert werden, ohne auf mehreren Plattformen implementiert werden zu müssen. Durch den vorher definierten unidirektionalen Datenfluss und die damit einhergehende Trennung zwischen Aktionen und der View-Schicht können diese losgelöst voneinander implementiert werden [19]. Lediglich der plattformabhängige Code benötigt hier Zugriff auf Aktionen. Zugriff auf die Reducer, welche die eigentliche Logik enthalten, werden dadurch nicht mehr innerhalb der View benötigt [19].

Auf Level-Ebene gibt es im Anhang A.19 angegebene mögliche Aktionen für den Start eines Levels, für absolute bzw. relative Bewegungen des Sichtfelds und für die Auswahl einer Entität.

Diese Aktionen werden abhängig von der Oberfläche der Applikation ausgelöst und dem Action Observer übergeben. Dadurch werden alle Reducer ausgeführt [19], wobei die allermeisten keine Änderung vornehmen werden.

Die Implementation eines hierzu passenden Reducer findet sich im Anhang A.20. Diese verwendet mit dem `lensReducer(actionOf:reducer:)` eine Hilfsfunktion aus dem Kern zur vereinfachten Deklaration von Reducern, mithilfe von einer Lens.

Wie beschrieben stellt sich gerade das Starten des neuen Levels aufgrund der Value Semantics und dem darunterliegenden Copy-on-Write Mechanismus als effizienter und trivialer Fall dar [6, Seite 156ff]. Sollte es bereits ein aktives Level geben, wird der aktuelle Zustand nicht geändert, da das weitere Vorgehen innerhalb der Anwendung implementiert werden muss.

Ein weiterer Fall, in dem die Logik stark von der Anwendung abhängt ist die Auswahl einer Entität. In diesem Falle müssen Reducer für Ziele implementiert werden, die abhängig von ihrer entsprechenden Konfiguration sind.

Die weiteren zwei Fälle ändern das aktuelle Sichtzentrum nach gewünschten Kriterien. Hier wird offensichtlich, dass für die Dauer eines Levels ein Action-Stream für

Action Observer definiert werden muss. Mögliche Kandidaten sind hier beispielsweise `CMMotionManager` [23] (bzw. `RxCoreMotion` [24]) oder ein im Rahmen eines Unit Tests künstlich erzeugter Stream. An dieser Stelle ist allerdings auf eine Beschränkung der Updaterate zu achten [4], welche sich allerdings bei Problemen durch eine geschickte Kombination von Operatoren, wie es in A.18 nach [4] definiert wurde, beschränken ließe.

Generierung

Um die Verwendung der sehr allgemeinen Injektoren und ihrer Schlüssel zu vereinfachen und weniger generisch erscheinen zu lassen, werden einige Techniken aus vorherigen Kapiteln angewandt.

Auch wenn sowohl ein Level als auch ein Szenario strikte Injektoren sind, können diese über den Einsatz eines Phantom Types unterschieden werden. Aus dem Phantom Type wird über einen `GenericProvidableKey` mittels Swift Generics ein auf Typebene unterscheidbarer Schlüssel generiert, was in A.12 veranschaulicht wird. Durch Definitionen von Typaliasen wird wieder aus anwendungsspezifischen Funktionen die Komplexität genommen.

Dies ermöglicht es die generischen Datentypen als vermeintlich nicht generische darzustellen.

Dennoch bleibt an weiterer Stelle das Problem der Deklaration und des Zugriffs auf die einzelnen Attribute der Datenstrukturen, was durch die Deklaration von Werten vom Typ `Provider` geregelt wird. Aufgrund der verwendeten Phantom Types ist jeder einzelne Provider explizit mit seinem generischen Datentypen verbunden (vgl. A.13).

Zum vereinfachten Zugriff auf diese Attribute werden nach A.14 statische Validatoren definiert, die Relation zwischen dem Datentypen und seinen Attributen deklariert.

Mittels Protokollerweiterungen und Lenses werden die validierten Eigenschaften nach außen hin leichter zugänglich gemacht (siehe A.15).

Aufgrund der recht großen Menge an Code für ein einzelnes Attribut, werden diese über Sourcery generiert. Hierdurch reicht lediglich die Deklaration des grundlegenden

Datentyps mit all seinen Attributen mit Annotationen für Sourcery bezüglich Namen und Typen. Hierzu wird, wie in A.16 dargestellt, eine Enumeration definiert, die das leere `GenerateInjector` Protokoll implementiert. Diese wird im entsprechenden Sourcery Template als Einstiegspunkt gewählt. Da die Enumerationen hier implizit als `internal` markiert sind und innerhalb des Swift Moduls nicht verwendet werden, werden diese durch die standardmäßig aktivierte Whole-Module Optimization nicht in der resultierenden Binär-Datei zu finden sein [25] und hat entsprechend keine negativen Auswirkungen auf die Laufzeit.

Letztendlich ermöglicht es dieser Ansatz durch einmalige Deklaration einer Enumeration und durch die Generierung der oben genannten Konstrukte, bei jeder Verwendung einen vom Compiler gestützten und typsicheren Zugriff auf die einzelnen Attribute des abzubildenden Datentypen in vielen verschiedenen Situationen.

4.2.4 Value to Object Translator

Im Rahmen der Implementation von plattformspezifischen Schnittstellen zur audiovisuellen Repräsentation von Leveln, mussten Änderungen des Single Shared Injectors an referenz-basierte Schnittstellen propagiert werden. Dies findet innerhalb des Moduls RxValueToObjectTranslator eine generische Lösung, die von der Audio- und Bildausgabe verwendet wird.

Dies erfordert die Extraktion von Änderungen und eine darauf folgende Manipulation von Objekt-Attributen, die Erzeugung oder Freigabe eines entsprechenden Objekts.

Beispielsweise muss bei dem Start eines Levels, für jede Entität und das Ambiente eine View erzeugt und die Audioausgabe initialisiert werden. Bei der Änderung des Sichtfelds, müssen nun Positionen und die Audiokanäle angepasst werden.

Sobald eine Entität gefunden wurde und entfernt werden muss, müssen ihre entsprechenden Objekte entfernt werden.

Auf Programmebene existiert eine Entität allerdings nicht, sondern lediglich der Zustand dieser. Um nun eine Assoziation zwischen der Entität und ihren Objekt-Repräsentationen

(View und Audio Player) zu erzeugen, muss hierfür eine Art primäres Attribut gewählt werden. Bei der View stellt dies das Bild dar, beim Audio Player die Audio-URI.

Bei jeder Zustandsänderung wird nun nach bereits existierenden Assoziationen zwischen den Entitätszuständen und den Ziel-Repräsentationen gesucht. Wenn diese fehlen, werden sie hergestellt. Anschließend wird der Zustand innerhalb jeder gefundenen Assoziationen angewendet. Sollte nach der Zustandsänderung kein Entitätszustand für eine Assoziation mehr existieren, werden die Assoziation und das Ziel-Objekt freigegeben.

Da dieses Problem im Rahmen des Grundsystems generisch gelöst wurde, kann dies auch im Rahmen der Persistenz-Schicht angewendet werden, wenn es benötigt wird. Beispielsweise würde eine Implementierung für ein CoreData- oder Realm-Backend von diesem Mechanismus profitieren, da beide Bibliotheken mit Referenz-Typen arbeiten.

Der vorgestellte Algorithmus benötigt eine Funktion zum Generieren für das primäre Attribut, eine Funktion zum Anwenden des Zustands auf das Objekt und optional zwei Funktionen für Nebeneffekte, die Aufschluss geben, ob ein Objekt erzeugt, gelöscht oder geändert wurde.

4.2.5 Ausgabe

Die audio-visuelle Ausgabe und Darstellung eines Levels findet ihre Implementierung in den Modulen `ReactiveAV` und `ReactiveUI`.

Während eine visuelle Repräsentation von Leveln und Entitäten innerhalb von Anwendungen lediglich optional ist, wird die akustische Ausgabe zwangsweise benötigt.

Die Bibliotheken greifen hier auf `UIKit` und `AVFoundation` zurück und müssen somit vornehmlich mit Referenztypen arbeiten [26] [27]. Aus diesem Grund verwerten beide Module lediglich das `RxValueToObjectTranslator`-Modul 4.2.4. In beiden Fällen werden Level-Daten unmittelbar ausgegeben, während Enitäten von Positionsdaten abhängen.

Im Falle der visuellen Ausgabe wird das Sichtfeld bei einer Drehung lediglich verschoben anstatt alle Entitäten zu verschieben. Entitäten werden anhand der prozentualen Angaben relativ innerhalb der View positioniert.

Bei der akustischen Ausgabe muss bei jeder Änderung des Sichtzentrums die Richtung jeder einzelnen, aktiven Wiedergabe von entitätsbezogenen Tönen angepasst werden. Die hierfür benötigten Berechnungen sind als Methoden der Positionen definiert.

Um die Umsetzbarkeit dieser Ausgaben zu belegen, werden nicht alle Effekte, die bei der Wahrnehmung von Schall innerhalb des Raumes und seiner Lokalisation simuliert 2.2.

5

Anforderungsabgleich

Da die Anforderungen der Anwendungsfälle und die des generischen Datenmodells im Vorfeld bereits erarbeitet wurden und beispielhaft ein explizites Datenmodell dargestellt wurde, können diese Anforderungen nun validiert werden.

Alle Anforderungen werden mittels $+$, $-$ und $\circ$ bewertet, wobei $+$ die betreffende Anforderung besonders gut erfüllt, während $-$ ungenügenden Lösungen vorbehalten ist.

5.1 Funktionale Anforderungen

Aufgrund des hochgradig generischen Grundkonzepts der Implementation sind einige funktionale Anforderungen, die an das Datenmodell und das umgebende Subsystem gestellt wurden, offensichtlich erfüllt.

Dadurch ist es problemlos möglich beliebige Knotentypen zu definieren und allen existierenden weitere Eigenschaften zu verleihen, was in 5.1.

	Anforderung	Wertung	Erklärung
FA#1	Eine Hierarchie muss definierbar aber auch optional sein	o	Das gesamte Datenmodell ist frei wählbar, lediglich der SSI ist vorgegeben.
FA#2	Das Sichtfeld muss viele Quellen zu ändern sein	+	Alle Quellen werden unterstützt. Siehe 4.2.3.
FA#3	Die Anwendung muss die volle Kontrolle über die Oberfläche behalten	+	Aufgrund von der starken Modularität gegeben.
FA#4	Verschiedene Instanzen eines Grunddatums sind unabhängig	+	Gegeben durch Value Semantics und robust gegen Fehler
FA#5	Auf allen Elementen können beliebige Eigenschaften definiert werden.	+	Die verwendeten Injektoren erlauben dies. Siehe A.17.
FA#6	Es können beliebige Elemente definiert werden	+	Auf diese Weise wurde das Beispiel implementiert. Siehe A.16.
FA#7	Erweiterungen können Audioausgabe steuern	+	Auf diese Weise wurde das Beispiel implementiert. Siehe 4.2.5.

Tabelle 5.1: Abgleich funktionaler Anforderungen

5.2 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen sind meist schwerer zu validieren, da sie zum Teil auch subjektiv interpretiert werden können und nicht messbar sind.

Aber gerade der Aufbau auf bereits bestehende und erprobte Konzepte sowie Technologien vereinfacht den Vergleich, denn auf diese Weise sind bereits einige Skalierungseffekte und Optimierungsmöglichkeiten bekannt, wie es in 5.2 veranschaulicht wird.

	Anforderung	Wertung	Erklärung
NFA#1	Plattformunabhängiger Kern	+	ReactiveCore und sogar das Datenmodell inklusive Datenoperationen sind plattformunabhängig.
NFA#2	Plattformspezifische Erweiterungen möglich	+	Für pflattformspezifische Erweiterungen gibt es bereits Beispielimplementationen.
NFA#3	Modularer Aufbau	+	Erüllt drch modularen Aufbau, vgl. 4.3.
NFA#4	Moderater Speicherverbrauch	o	Durch die Value-Types steigt der Speicherverbrauch. Dies wird aber durch Copy-on-Write ausgeglichen.
NFA#5	Moderater Leistungsverbrauch	o	Auch hier gleicht Copy-on-Write Speicheroverhead durch Value-Types aus. Allerdings können Reducer zusammengelegt und optimiert werden [28].
NFA#6	Hohe Stabilität	+	`ReactiveCore` und `ReactiveAu3dio` werden mittels Unit Tests gedeckt.
FA#7	Einfache Handhabung für Entwickler	o	Lediglich das Verwenden von Sourcery Templates macht dies möglich.

Tabelle 5.2: Abgleich nichtfunktionaler Anforderungen

6 Zusammenfassung & Ausblick

Im Rahmen dieser Arbeit wurde die akustische Lokalisation vorgestellt und einige Anwendungsfälle innerhalb einer Anwendungsklasse analysiert. Auf dieser Basis wurde dann das generisches Datenmodell für mobile Anwendungen entworfen und implementiert. Im Verlauf wurden einige Lösungsansätze für anwendungsspezifische Probleme gegeben und umgesetzt. Im Anschluss konnte dieser Lösungsansatz anhand der gestellten Anforderungen bestätigt werden.

6.1 Ausblick

Im Rahmen einer Weiterentwicklung wäre an erster Stelle eine detailliertere Simulation der Wiedergabe akustischer Signale im Raum zur anschließenden Lokalisation, welche beispielsweise auch Abstände simulieren kann, zu erwähnen.

Bezüglich des Schwierigkeitsgrade und Dynamik sind neben Bewegungen von Entitäten auch andere Effekte wie simulierte Störungen während Telefongesprächen und ähnlichem möglich.

Auf Ebene der Anwendungen wären außer den vorgestellten auch medizinische und therapeutische Fälle denkbar. Außerdem kann eine Anwendung auch explizit auf Kinder zugeschnitten werden.

6.2 Abschluss

Im Vergleich zu nicht generischen Datenmodellen gibt es zwar, Nachteile hinsichtlich der Komplexität, diese konnten aber durch den Einsatz von Codegenerierung und die Wiederverwertung bereits bekannter Konzepte und Technologien deutlich verringert werden. Gerade wenn auf Dauer größere Änderungen oder Anpassungen innerhalb der Anwendungsklasse und in mehreren Applikationen stattfinden, überwiegen die Vorteile eines generischen Datenmodells die von bis dahin mehreren nicht generischen Datenmodellen signifikant.

Aufgrund des modularen Aufbaus Gesamtsystems mit seinen Einflüsse aus der funktionalen Programmierung kann die Verwendung dieser Implementation ebenso sinnvoll bei lediglich einer einzelnen Anwendung sein wie sie es bereits bei geteiltem Code ist.

Literaturverzeichnis

[1] Wikipedia: Lokalisation (akustik) – wikipedia. https://de.m.wikipedia.org/wiki/Lokalisation_(Akustik) (2016)

[2] Klensch, H.: Beitrag zur frage der lokalisation des schalles im raum. Technical report, Physiologischen Institut der Universität Bonn (1948)

[3] Mackensen, P., Reichenauer, K., Theile, G.: Einfluß der spontanen Kopfdrehungen auf die Lokalisation beim binauralen Hören. Technical report, Institut für Regelungstechnik der Universität Hannover (1998)

[4] Schickler, M., Pryss, R., Schobel, J., Reichert, M.: An Engine enabling Location-based Mobile Augmented Reality Applications. Technical report, Institute of Databases and Information Systems, Ulm University (2016)

[5] Ilseman, M.: swift/ABIStabilityManifesto.md at master · apple/swift. https://github.com/apple/swift/blob/master/docs/ABIStabilityManifesto.md (2017)

[6] Eidhof, C., Kugler, F., Velocity, A.: Advanced Swift. 2 edn. Kugler and Eidhof GbR (2016)

[7] Eidhof, C.: Lenses in Swift - Chris Eidhof. http://chris.eidhof.nl/post/lenses-in-swift/ (2014)

[8] Knabel, V.: RxLens: Enables reactive read and copy-on-write access. https://github.com/vknabel/RxLens (2016)

[9] Knabel, V., Encz, B., Perpetua, J.M.: vknabel/ValidatedExtension: A Swift Micro-Library for Somewhat Dependent Types. https://github.com/vknabel/ValidatedExtension (2016)

[10] Knabel, V.: EasyInject. https://github.com/vknabel/EasyInject (2017)

[11] Eidhof, C., Kugler, F., Swierstra, W.: Functional Swift. 3 edn. Kugler and Eidhof GbR (2016)

[12] Furrow, A., Belanger, C., Todorov, M., eds.: RxSwift: Reactive Programming with Swift. 1 edn. raywenderlich.com (2017)

[13] ReactiveX: ReactiveX - Languages. http://reactivex.io/languages.html (2017)

[14] Zaher, K., Bernal, J., Li, R.: RxSwift/SubjectType.swift at master · ReactiveX/RxSwift. https://github.com/ReactiveX/RxSwift/blob/master/RxSwift/Subjects/SubjectType.swift (2017)

[15] Zaher, K., Hope, A., Ramezanpoor, M., Pinkham, J., Korolev, Y., Ningen, G.: RxSwift/DisposeBag.swift at master · ReactiveX/RxSwift. https://github.com/ReactiveX/RxSwift/blob/master/RxSwift/Disposables/DisposeBag.swift (2017)

[16] Apple: DispatchQueue - Dispatch | Apple Developer Documentation. https://developer.apple.com/reference/dispatch/dispatchqueue (2017)

[17] ReactiveX: ReactiveX - Scheduler. http://reactivex.io/documentation/scheduler.html (2017)

[18] Mansilla, S.: Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code. Volume 1. The Pragmatic Programmer (2015)

[19] Encz, B.: ReSwift/ReSwift: Unidirectional Data Flow in Swift - Inspired by Redux. https://github.com/ReSwift/ReSwift (2017)

[20] Zablocki, K.: Sourcery. https://github.com/krzysztofzablocki/Sourcery (2017)

[21] Apple: swift/android.md at master · apple/swift. https://github.com/apple/swift/blob/master/docs/Android.md (2017)

[22] Apple: Coordinate System. https://developer.apple.com/library/content/documentation/General/C
CocoaApp/CoordinateSystem.html (2013)

[23] Apple: CMMotionManager. https://developer.apple.com/reference/coremotion/cmmotionmanage
(2017)

[24] García, C., Graham, K., Retamal, A., Zaher, K.: Rxswiftcommunity/rxcoremotion: Provides an easy and straight-forward way to use apple ios coremotion responses as rx observables. https://github.com/RxSwiftCommunity/RxCoreMotion (2017)

[25] Apple: Swift.org - Whole-Module Optimization in Swift 3. https://swift.org/blog/whole-module-optimizations/ (2017)

[26] Apple: Audio systems. https://developer.apple.com/reference/avfoundation/audio_systems (2017)

[27] Apple: UIKit | Apple Developer Documentation. https://developer.apple.com/reference/uikit (2017)

[28] Erikson, M., Yang, S.: redux/Performance.md at master · reactjs/redux. https://github.com/reactjs/redux/blob/master/docs/faq/Performance.md#performance-all-reducers (2017)

A
Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```
enum Phantom { }
```

Listing A.1: Deklaration eines Phantom Types

```
struct Amount<T>: Equatable {
        private var count: Int

        init(_ count: Int, of _: T.Type = T.self) {
                self.count = count
        }

        static func == (lhs: Amount, rhs: Amount) -> Bool {
                return lhs.count == rhs.amount
        }
}
```

Listing A.2: Verwendung eines Phantom Types

```
protocol HasSomething {
        associatedtype Something
        var something: Something
}
let someArray = [Something]() // Won't compile
```

Listing A.3: Protokolle mit assoziiertem Typen

```
struct AnySomething<Something>: HasSomething {
        var something: Something
}
let someArray = [AnySomething]() // compiles fine
```

Listing A.4: Beipsiel für Verwendung von Type Erasure

```
struct DictionaryHasNameValidator: Validator {
        static func validate(value: [String: Any]) -> Bool {
                if let _ = value["name"] as? String {
                        return true
                } else {
                        return false
                }
        }
}
typealias DictionaryWithName =
  Validated<DictionaryHasNameValidator>
```

Listing A.5: Definition eines Validators

```
DictionaryWithName([:]) // nil
// Validated<DictionaryHasNameValidator>
DictionaryWithName(["name": "Valentin Knabel"])
```

Listing A.6: Die Initialisierung von Validated kann fehlschlagen

```
extension ValidatedType where ValidatorType == DictionaryHasNameValidator
        var name: String {
                return value["name"] as! String
        }
}
```

Listing A.7: Erweiterung auf ValidatedType

```swift
import EasyInject

enum ConfigurationKeyType { } // will never be instantiated
typealias Configuration =
  GenericProvidableKey<ConfigurationKeyType>

extension Provider {
  static var baseUrl: Provider<Configuration, String> {
    return Provider<Configuration, String>(for: "baseUrl")
  }
}

var strictInjector = StrictInjector<Configuration>()
strictInjector.provide(for: .baseUrl, usingFactory: { _ in
  return "https://my.base.url/"
})
```

Listing A.8: Dependency Injection von Werten analog zu [10]

```swift
public let nameOfEntityLens: Lens<Entity, String?> =
  .with(injected: entityName)
```

Listing A.9: Erzeugung einer Lens für Injektoren

```swift
/// The central store for all data and subscriptions.
/// Listens for all actions.
struct Store {
  /// Observes for actions.
  /// All actions will be reduced one after another.
  let aao: AnyObserver<Action>
  /// Emits the current `Ssi`
  /// once all actions have been applied.
  let ssio: Observable<Ssi>
```

```
10    /// Stores all disposables associated for keys.
11    let sdi: Sdi
12  }
```

Listing A.10: Auszug der Deklaration eines Stores

```
1  let main = Store(
2    initial: Observable.just(Ssi()),
3    reducers: [
4      LevelAction.reducer
5    ]
6  )
```

Listing A.11: Instanziierung eines Stores

```
1  public enum LevelFieldKey { }
2  public typealias LevelField =
3    GenericProvidableKey<LevelFieldKey>
4  public typealias Level = StrictInjector<LevelField>
5  public typealias LevelProvider<V> = Provider<LevelField, B>
```

Listing A.12: Definition von LevelProvidern

```
1  public extension Provider {
2    public static var levelName: LevelProvider<String> {
3      return .derive()
4    }
5  }
```

Listing A.13: Deklaration eines LevelProviders

```
1  public struct LevelHasName: Validator {
2    public static func validate(_ value: Level) throws -> Bool {
3      _ = try value.resolving(from: .levelName)
4      return true
```

```
    }
}
```

Listing A.14: Validator für den Namen eines Levels

```
public extension ValidatedType
  where ValidatorType == LevelHasName
{
  public var name: String {
    // will only fail if ValidatorType has a bug
    return try! value.resolving(from: .levelName)
  }
}

public typealias LevelWithName = Validated<LevelHasName>
public let nameOfLevelLens: Lens<Level, String?> =
  .with(injected: .levelName)
```

Listing A.15: Computed Property bei Validatoren

```
/// sourcery: name = "level"
enum LevelDeclaration: GenerateInjector {
  /// sourcery: type = "String"
  case name
  // other cases
}
```

Listing A.16: Deklaration von Level für Sourcery

```
/// sourcery: name = "ssi", skipTypedef
enum SsiDeclaration: GenerateInjector {
  /// sourcery: type = "[Scenario]"
  case scenarios
  /// sourcery: type = "Level"
```

```
6    case currentLevel
7  }
```

Listing A.17: Neue Attribute für den SSI

```
1  // defined somewhere else
2  let sourceStream: Observable<Position>
3
4  let limitedSourceStream = Observable.interval(11)
5          .withLatestFrom(sourceStream)
6          .distinctUntilChanged()
```

Listing A.18: Beschränkung der Ereignis-Rate eines Streams

```
1  enum LevelAction: Action {
2    case start(Level)
3    case moveTo(Position)
4    case turnTo(Position.Coordinate)
5    case selectedEntity(Entity)
6  }
```

Listing A.19: Mögliche Aktionen auf dem aktuellen Level

```
1  extension LevelAction {
2    static let reducer = lensReducer(
3      actionOf: LevelAction.self,
4      currentLevelOfSsiLens
5    ) { (level: Level?, action: LevelAction) -> Level? in
6      switch action {
7      case let .start(new) where level == nil:
8        return new
9      case let .start(_):
10       // Keep the state
11       // If required the application
```

```
12        // will handle this case
13        return level
14      case .selectedEntity(_):
15        // Handled by custom reducers of the application
16        return level
17      case let .moveTo(newCenter):
18        guard let level = level else { return nil }
19        return level.providing(newCenter, for: .levelPosition)
20      case let .turnTo(coordinate):
21        guard let someLevel = level,
22        var position =
23          LevelWithPosition(value: someLevel)?.position
24        else {
25          return level
26        }
27        position.x = coordinate
28        return someLevel.providing(position, for: .levelPosition)
29      }
30    })
31  }
```

Listing A.20: Möglicher Reducer für Level-Aktionen

```
1  struct Lens<A,B> {
2    let from : A -> B
3    let to : (B, A) -> A
4  }
```

Listing A.21: Definition vom Lens-Typen

Abbildungsverzeichnis

4.1 Lebenszyklus eines Observables 30
4.2 Unidirektionaler Datenfluss 32
4.3 Struktur und Abhängigkeiten der Module 33
4.4 Relationen innerhalb des Datenmodells 37

Tabellenverzeichnis

3.1 Erste funktionale Anforderungen der Anwendungen bezüglich von Leveln 14
3.2 Weitere funktionale Anforderungen der Anwendungen bezüglich von Leveln 15
3.3 Funktionale Anforderungen der Anwendungen bezüglich des Sichtfelds . 15
3.4 Funktionale Anforderungen der Anwendungen bezüglich Entitäten 16
3.5 Funktionale Anforderungen der Anwendungen bezüglich Zielen 16
3.6 Funktionale Anforderungen der Anwendungen bezüglich Szenarien . . . 17
3.7 Funktionale Anforderungen der Anwendungen bezüglich akustischen Signalen 18
3.8 Funktionale Anforderungen der Anwendungen bezüglich Erfolgen 19
3.9 Funktionale Anforderungen 20
3.10 Nichtfunktionale Anforderungen 22

5.1 Abgleich funktionaler Anforderungen 44
5.2 Abgleich nichtfunktionaler Anforderungen 46

Name: Valentin Knabel Matrikelnummer: 798359

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Valentin Knabel