

Dokumentation über den Neuaufbau der OUTPUT.DD App im WiSe 2020/21

Philipp Matthes

Geboren am: 12. März 1997 in Chemnitz
Studiengang: Diplom Informatik (PO 2010)
Matrikelnummer: 4605459
Immatrikulationsjahr: WiSe 2016/17

Projektarbeit

Betreuer

Dr. Thomas Springer

Eingereicht am: 1. Februar 2021

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich das vorliegende Dokument mit dem Titel *Dokumentation über den Neuaufbau der OUTPUT.DD App im WiSe 2020/21* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in diesem Dokument angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung des vorliegenden Dokumentes beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 1. Februar 2021



Philipp Matthes

Inhaltsverzeichnis

Abbildungsverzeichnis	VI
1 Einleitung	1
1.1 Problemstellung	1
1.2 Ziele der Projektarbeit	2
2 Grundlagen	3
2.1 Swift und SwiftUI	3
2.2 Datenfluss- und Architekturen in SwiftUI	3
2.2.1 Darstellung von View-Daten	3
2.2.2 Coordinator-Pattern	4
2.2.3 Environment-Pattern	5
2.2.4 View-Proxy-Pattern	6
2.2.5 Eventbasierte Datenflüsse	6
3 Analyse	8
3.1 Ermittlung einer Dialoglandkarte	8
3.2 Struktur des Couchbase-Datenbank-Frameworks	8
3.3 Top-Level-Architektur und Muss-Kriterien	11
4 Konzept und Implementation	13
4.1 Konzeption einer Ordnerstruktur	13
4.2 Einführung eines Copyright-Templates	14
4.3 Einführung von Codierungsrichtlinien	14
4.4 Implementation des Frontends	15
4.5 Integration einer CI-Pipeline	17
4.6 Fusion des Frontends mit dem Datenbank-Backend	18
4.6.1 Vorbemerkungen für die Darstellung der SwiftUI-Preview	18
4.6.2 Nutzung des Environment- und Coordinator-Patterns für die Anbindung der Views	19
4.6.3 Nutzung des Proxy-Patterns für die Initialisierung der Datenbank	21
4.6.4 Testen der Datenbankfunktionalität	22
4.6.5 Anbindung der Synchronisationskomponente	22
4.6.6 Testen der Synchronisation	22
4.7 Listener- und Eventbasierte Anbindung der Gamification	23
4.8 Integration des Crowd-Monitoring-Frameworks	26
4.9 Dokumentation der Entwickler- und Distributionsleitfäden	26

5 Zusammenfassung	28
5.1 Vorgehensweise und Ergebnisse	28
5.2 Offene Punkte	29
5.3 Danksagung	30

Abbildungsverzeichnis

2.1	Zwei wesentliche Arten des Datenflusses zwischen View und den in der View dargestellten Daten: unidirektional vs. bidirektional.	4
2.2	Das Coordinator Pattern illustriert am Beispiel eines hypothetischen QR Code Scanners: Der Coordinator realisiert die Initialisierung des Scanners und die Validierung des Codes. Die View stellt den Code lediglich dar, sobald dieser gescannt wurde.	5
2.3	Das Environment Pattern illustriert am Beispiel der Map View: Die Map View erstellt ihr eigenes Environment und gibt dieses an ihre Unteransichten weiter. Somit kann die gewählte Etage von einer Unteransicht modifiziert werden, von anderen Unteransichten wird sie lediglich dargestellt.	6
3.1	Die erstellte Dialoglandkarte zur bestehenden Version der iOS App zeigt die Navigationswege innerhalb der App, sowie die genutzten UI-Elemente.	9
3.2	Eine Übersicht über das zu integrierende Couchbase-Framework. Über eine abstrahierende Wrapper-Programmierschnittstelle wird auf die Datenbank zugegriffen.	9
3.3	Das Couchbase-Framework kapselt die Datenmodelle in voneinander separierte Repositories, auf denen unter anderem grundlegende CRUD-Operationen möglich sind.	10
3.4	Die Repository-Schnittstelle wird generisch durch das Datenmodell und das dazugehörige Repository erzeugt. Der Änderungsaufwand bei einer Änderung des Datenmodells ist somit sehr gering und neue Datenmodelle können ohne größeren Aufwand hinzugefügt werden.	10
3.5	Die TLA der Applikation im Überblick.	11
4.1	Die erstellte Copyright-Template am Beispiel der HomeView.	14
4.2	Die Konfiguration des statischen Codeanalysetools SwiftLint.	15
4.3	Die Hauptansichten des UI der neu implementierten App im Profil.	16
4.4	Die SwiftUI-Preview der CalendarEventCardView innerhalb von XCode.	18
4.5	Das Coordinator-Pattern angewandt am Beispiel der Spiel-Trophäen, um eine schnelle SwiftUI-Preview ohne Daten-Initialisierung zu gewährleisten.	19
4.6	Die Anbindung der Koordinatoren an die View im Beispiel der GameProgressTrophyStackView.	20
4.7	Abstrakte Übersicht über die Initialisierung der Datenbank.	21

4.8 Ein Algorithmus für die Datenbank-Listener-basierte Freischaltung von Trophäen und Aufgaben. Rot hervorgehoben sind Prozesse, bei denen Listener appliziert oder entfernt werden. Violett hervorgehoben ist das auslösende Ereignis für eine Prüfung auf Freischaltung. Grün hervorgehoben ist die Freischaltung der Trophäe oder der Aufgabe.	24
4.9 Ein Algorithmus für die interaktionsbasierte Freischaltung von Trophäen und Aufgaben. Violett hervorgehoben ist das auslösende Ereignis für eine Prüfung auf Freischaltung. Grün hervorgehoben ist die Freischaltung der Trophäe oder der Aufgabe.	25
5.1 Die neu implementierte App umfasst ca. 8000 Zeilen Swift Code.	29

1 Einleitung

OUTPUT.DD ist eine jährlich stattfindende Projektschau, auf welcher in der Fakultät Informatik der Technischen Universität Dresden wissenschaftliche Projekte von Studenten, Firmen und Mitarbeitern präsentiert und ausgestellt werden. Begleitend zur Projektschau wird den Besuchern der Veranstaltung jeweils eine App für iOS und Android zur Verfügung gestellt. Die Apps dienen dabei selbst nicht nur als Plattform, über welche Nutzer den zeitlichen und topografischen Veranstaltungsplan einsehen können, sondern auch als Integrationspunkt für verschiedene Forschungsprojekte aus den Bereichen Mediengestaltung, Application Development und Mobile Computing. So wird die Interaktion der Nutzer beispielsweise durch eine Gamification¹ motiviert und geleitet, oder die Bewegung des Nutzers auf der Messe über ein im Rahmen des Forschungsprojektes „Mapbiquitous“ entstandenes Beacons-Framework verfolgt, um das Besucheraufkommen zu analysieren und dem Nutzer eine interaktive Heatmap auf der Kartenansicht der Veranstaltung anzuzeigen. Außerdem nutzt die App das Offline-First-Prinzip aus dem Forschungsbereich Ubiquitäre Applikationen, bei dem die in der App angezeigten Daten koordiniert von einem Server-Backend synchronisiert werden, um bei einem Netzwerkausfall weiterhin so viele Funktionalitäten wie möglich zu unterstützen und die offline geschriebenen Daten bei der Wiederherstellung der Verbindung im Hintergrund zu aktualisieren. Im Sommer 2020 wurden für die beiden Apps jeweils Datenbank-Frameworks für die NoSQL-Datenbank Couchbase entwickelt, welche architekturell im Repository-Pattern² umgesetzt wurden und zur Synchronisation der Daten so genannte Replikatoren bereitstellen, die über einen Synchronisationsdienst mit dem Couchbase Backend-Server kommunizieren.

1.1 Problemstellung

Da die OUTPUT.DD Apps in den vergangenen Jahren noch auf ein anderes, auf Realm basierendes, Datenbank-Backend aufsetzten, musste eine Substitution des Datenbank-Backends analysiert, geplant, durchgeführt und getestet werden.

Durch die sukzessive funktionelle Erweiterung der Apps über mehrere Jahre unter verschiedenen Teams mit jeweils unterschiedlichen Qualitätsansprüchen, Zeitvorgaben und Kenntnissen bildeten sich außerdem mehrere Probleme in der Implementation heraus, die durch einen Audit der iOS-Codebasis am 5.4.2020 identifiziert werden konnten:

- **Strukturelle Antipatterns**, darunter eine schwer nachvollziehbare Ordner- und Dateistrukturierung, das teils fehlende Separation in Module, sowie die Vermischung von

¹Gamification. Einsatz von Spielementen in einem Nicht-Spiel-Kontext.

²Repository-Pattern. Architekturelles Pattern zur Abstraktion und Separation von Datenbankabfragen durch die Bereitstellung von CRUD-Operationen (Create, Read, Update, Delete).

- abkapselbaren externen und internen Frameworks mit der primären Codebasis
- **Architekturelle Antipatterns**, wie beispielsweise die „Verschmutzung“³ des Codes durch globale Erweiterungen von Datenbankmodellen an unerwarteten Stellen
 - **Weitere Probleme**, darunter die Ignorierung von potentiellen Sicherheitsrisiken durch das Ausschalten von Dependency-Warnings, an einer Stelle auch die Fehlverwendung von View-Life-Cycles, das allgemeine Vorhandensein Code Smells und die technische Alterung des Objective-C Codes.

1.2 Ziele der Projektarbeit

Da OUTPUT.DD 2020 vor dem Hintergrund der Sars-CoV-2 Pandemie nicht stattfand, wurde im Projektteam beschlossen, diese Probleme durch eine Neuimplementation der Applikation zu beheben, mit dem beiläufigen Ziel, auch das neue Datenbank-Backend zu integrieren. Im Projektteam wurde entschieden, die Apps mit den jeweils aktuellsten Technologien neu zu implementieren, darunter eine Reimplementation der Android-App mit der Programmiersprache Kotlin, sowie eine (in dieser Arbeit näher beschriebenen) Reimplementation der iOS-App mit der Programmiersprache Swift und dem 2019 von Apple eingeführten UI-Framework SwiftUI, welches eine deklarative Ansichtserstellung ermöglicht. Hierdurch sollten die Apps technisch und visuell modernisiert werden. Bei der Reimplementation sollten die oben beschriebenen Probleme durch die Einführung klarer Architektur- und Strukturpatterns möglichst vermieden werden, sowie außerdem konkrete Strategien und Richtlinien entworfen werden, um ein Wiedereintreten dieser Probleme zu vermeiden und somit zukünftigen Projektteams die Weiterentwicklung der Apps zu erleichtern. Diese Arbeit soll einen chronologischen Überblick darüber geben, welche Schritte gegangen wurden, um diese Ziele zu erreichen, und die Endprodukte schließlich evaluieren.

³Verschmutzung. Von Englisch „Pollution“, wird oft als Fachbegriff verwendet, um die Degradation der Codequalität durch fehlpositionierte Codefragmente zu verbildlichen.

2 Grundlagen

In diesem Kapitel sollen zunächst wichtige theoretische Grundlagen erklärt werden, die zum Verständnis der in den nachfolgenden Kapiteln beschriebenen Projektbearbeitungsschritte und -entscheidungen notwendig sind.

2.1 Swift und SwiftUI

Eine Festlegung, die bereits vor der Reimplementation der Apps getroffen wurde, ist die Wahl von Swift als Hauptprogrammiersprache der zu erstellenden iOS App. Die Programmiersprache wurde 2014 von Apple veröffentlicht und ersetzt seitdem Objective-C als die von Apple empfohlene Objektorientierte Sprache zur Erstellung von Applikationen für das Apple-Ökosystem mit den Betriebssystemderivaten Mac OS, iOS und bspw. auch Watch OS (Apple Watch). In der aktuellen Version 5 setzt Swift vielseitig etablierte Konzepte aus modernen Programmiersprachen um und ist hierbei leicht verständlich, schnell zu erlernen, sowie sehr kompakt. Zusammen mit Swift 5 wurde SwiftUI als deklaratives UI-Framework 2019 veröffentlicht, ergänzend zum herkömmlichen Constraint-basierten Layouting über UIView(Controller) und XIB/Storyboard-Dateien. SwiftUI orientiert sich an Frameworks wie React, bei denen Ansichten im Code definiert werden können, wodurch die Erstellung und Modifikation von Ansichten direkt im Editor geschehen kann, ohne die Notwendigkeit von etwaiger Controller-Logik, die UI-Elemente mit Code-Elementen (z.B. über die Annotation @IBAction) verbindet. Dies vereinfacht den Implementationsprozess, bringt jedoch auch verschiedene neue Architektur- und Datenflusskonzepte mit sich, die zunächst verstanden werden müssen, um SwiftUI effektiv und zielorientiert nutzen zu können.

2.2 Datenfluss- und Architekturpatterns in SwiftUI

In der nachfolgenden Sektion sollen zunächst die wesentlichsten Grundlagen des Datenflusses in SwiftUI-Ansichten erläutert werden. Anschließend werden verschiedene Patterns vorgestellt, die sich im Implementationskonzept wiederspiegeln und auch zukünftig idealerweise weiter genutzt werden sollten.

2.2.1 Darstellung von View-Daten

Im Unterschied zum imperativen UI-Programmierkonzept, bei dem konkrete Elemente der Ansicht, wie bspw. Labels oder Textfelder imperativ konfiguriert und mit den darzustellenden

Daten ausgestattet werden, referenziert eine SwiftUI Ansicht („View“) die darzustellenden Daten direkt aus dem Code, da sie selbst deklarativ in Form von Code geschrieben werden kann.

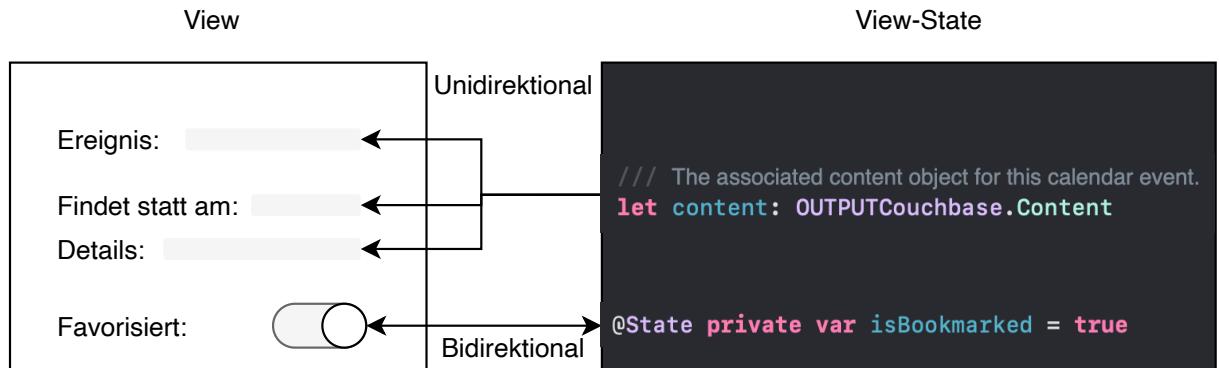


Abbildung 2.1 Zwei wesentliche Arten des Datenflusses zwischen View und den in der View dargestellten Daten: unidirektional vs. bidirektional.

Abbildung 2.1 zeigt die zwei wesentlichsten Varianten, mithilfe derer Daten in der Ansicht dargestellt werden können: **unidirektional** und **bidirektional**. Bei der unidirektionalen Darstellung der Daten werden bei der Initialisierung der View die Daten im Konstruktor der View mitgegeben - da SwiftUI Views so genannte Structs sind, können die Daten im Verlauf der Zeit nicht mehr ändern (Structs sind „immutable“). SwiftUI rendert die assoziierten Elemente der View einmalig und verwendet diese nachfolgend zur Darstellung. Für viele Anwendungsfälle, in Abbildung 2.1 anhang eines „Favorisieren-Buttons“ illustriert, ist es jedoch notwendig, dass durch die View die dahinter liegenden Daten modifiziert werden. Hierfür wird der Property Wrapper `@State`¹ bereitgestellt. Attribute, die hiermit ausgestattet werden, können während der Präsentation der View verändert werden. Eine Besonderheit hierbei ist, dass SwiftUI genau observiert, welche UI-Elemente durch das annotierte Attribut modifiziert oder konfiguriert werden, um bei einer Änderung des Attributs die entsprechenden Elemente neu zu rendern. Wird ein Datenattribut an eine Unteransicht weitergegeben, welche bspw. eine Detailansicht zu einem Objekt darstellt, dann wird das Attribut in der Unteransicht als `@Binding`² referenziert.

2.2.2 Coordinator-Pattern

Um die Geschäftslogik der Anwendung von der Darstellungslogik weitestgehend zu separieren, kann das Coordinator-Pattern genutzt werden. Hierbei instanziert die View ein eigenes Objekt, welches sie durch die `@StateObject`³ Annotation besitzt und kontrollieren kann, bspw. durch die Interaktion des Nutzers oder beim Erscheinen der Ansicht. Das Coordinator-Objekt macht die darzustellenden Datenattribute über die `@Published`⁴ Annotation nach außen verfügbar - so dass die View diese Datenattribute (ähnlich zu `@State` Datenattributen) verwenden und auf deren Änderungen reagieren kann.

¹<https://developer.apple.com/documentation/swiftui/state> (Abgerufen am 1.2.2021)

²<https://developer.apple.com/documentation/swiftui/binding> (Abgerufen am 1.2.2021)

³<https://developer.apple.com/documentation/swiftui/stateobject> (Abgerufen am 1.2.2021)

⁴<https://developer.apple.com/documentation/combine/published> (Abgerufen am 1.2.2021)

```

extension Coordinator: AVCaptureMetadataOutputObjectsDelegate {
    func metadataOutput(
        _ output: AVCaptureMetadataOutput,
        didOutput metadataObjects: [AVMetadataObject],
        from connection: AVCaptureConnection
    ) {
        // When a metadata object is encountered, try decoding its
        // string value (which is interpreted from the code)
        guard
            let metadataObject = metadataObjects.first,
            let readableObject = metadataObject as? AVMetadataMachineReadableCodeObject,
            let code = readableObject.stringValue
        else { return }
        let validatedCode = validateCode(code)
        self.scannedValidCode = validatedCode
    }
}

```

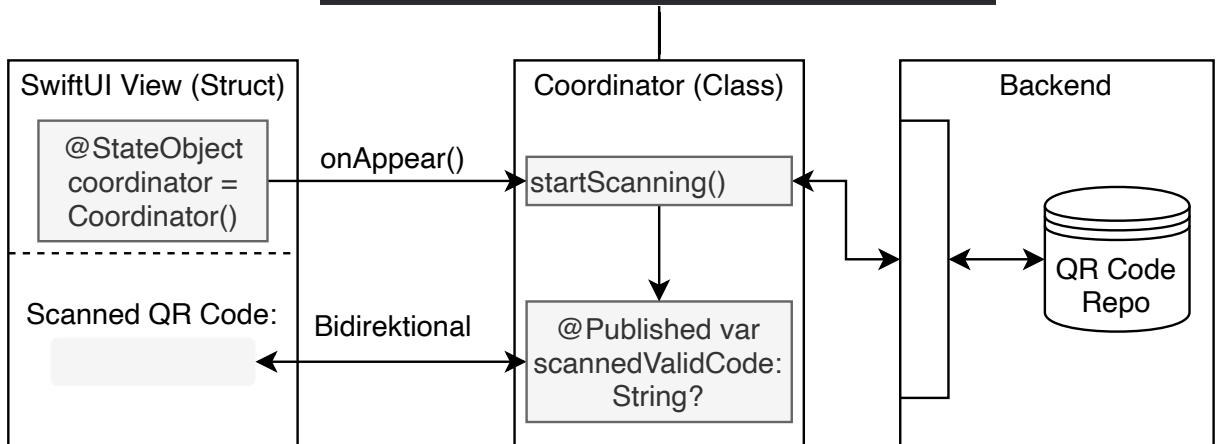


Abbildung 2.2 Das Coordinator Pattern illustriert am Beispiel eines hypothetischen QR Code Scanners: Der Coordinator realisiert die Initialisierung des Scanners und die Validierung des Codes. Die View stellt den Code lediglich dar, sobald dieser gescannt wurde.

In Abbildung 2.2 ist das Pattern am Beispiel illustriert. Auch zu sehen ist hierbei, dass der Coordinator eine Klasse ist. Dies ist nützlich, wenn bestimmte Protokolle (Swift-Interfaces) implementiert werden müssen, in diesem Beispiel `AVCaptureMetadataOutputObjectsDelegate`⁵ zur Reaktion auf gescannte QR Codes. In diesem Beispiel kann die View dieses Protokoll nicht implementieren, da das Protokoll implizit die Implementation durch eine Klasse erfordert.

2.2.3 Environment-Pattern

Als weiteres wichtiges Datenfluss-Pattern dient das Environment-Pattern. Hierbei wird, ähnlich zum Coordinator-Pattern, ein koordinierendes Umgebungsobjekt instanziert und in der Besitz ergreifenden View als `@StateObject` markiert. Anschließend wird das Umgebungsobjekt im Environment der Unteransichten mitgegeben, über den ViewModifier `view.environmentObject(_ :)`⁶. Dies ist zum Beispiel sinnvoll, wenn eine Unteransicht bestimmte Daten aus dem Kontext beziehen und/oder modifizieren möchte.

⁵<https://developer.apple.com/documentation/avfoundation/avcapturemetadataoutputobjectsdelegate> (Abgerufen am 1.2.2021)

⁶[https://developer.apple.com/documentation/swiftui/view/environmentobject\(_:\)](https://developer.apple.com/documentation/swiftui/view/environmentobject(_:)) (Abgerufen am 1.2.2021)

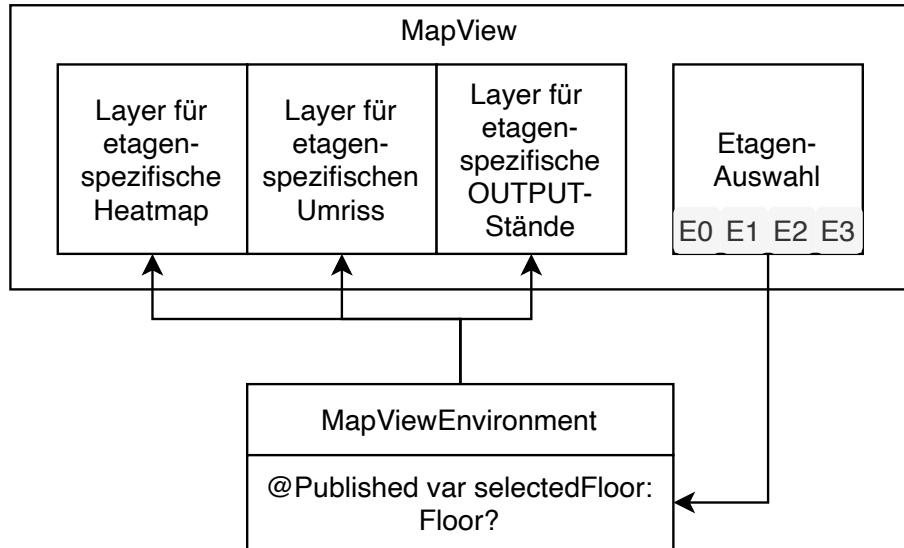


Abbildung 2.3 Das Environment Pattern illustriert am Beispiel der Map View: Die Map View erstellt ihr eigenes Environment und gibt dieses an ihre Unteransichten weiter. Somit kann die gewählte Etage von einer Unteransicht modifiziert werden, von anderen Unteransichten wird sie lediglich dargestellt.

In Abbildung 2.3 ist dieses Pattern am Beispiel illustriert. Ein großer Vorteil des Patterns ist, dass das über den ViewModifier weitergegebene Umgebungsobjekt transitiv an alle weiteren Unteransichten weitergeleitet wird, die sich möglicherweise in dieser Unteransicht befinden. Der Bezug des Objektes findet anschließend über `@EnvironmentObject var object` statt. Ein Problem hierbei ist, dass das Objekt nicht optional weitergegeben werden kann, was die Preview der SwiftUI Unteransichten erschwert. Die hierfür applizierte Lösung hierfür wird später im Konzeptkapitel näher beschrieben.

2.2.4 View-Proxy-Pattern

Die Grundlagen des Environment-Pattern und des Coordinator-Pattern können weiterhin genutzt werden, um bestimmte Bestandteile der Geschäftslogik im View-Proxy-Pattern zu separieren. Hierbei wird eine dedizierte View erstellt, deren Aufgabe es ist, bestimmte Datenverarbeitungsoperationen durchzuführen und bei Beendigung dessen die Produkte dieser Operation als Umgebungsobjekt bzw. Proxy-Objekt der inneren View bereitzustellen. Hierfür wird die innere View der Proxy View als `@ViewBuilder7-Closure` übergeben. Ein klassisches Beispiel für das View-Proxy-Pattern ist der von SwiftUI standardmäßig bereitgestellte `GeometryReader8`, welcher als View genutzt werden kann, um ein Proxy-Objekt für die Geometrie der View zu erstellen, bspw. um die Größe einer Ansicht zu bestimmen. Im Konzept der App wird dies später für die Initialisierung der Datenbank und Replikatoren adaptiert und an dieser Stelle noch einmal genauer beschrieben.

2.2.5 Eventbasierte Datenflüsse

Eventbasierte Datenflüsse stellen ein weiteres wichtiges Pattern für die Weitergabe von Daten dar. Bei den bisherigen Pattern wurden lediglich Methoden vorgestellt, mithilfe derer sich ein Datenfluss zwischen mehreren Views in *derselben* View-Hierarchie realisieren lässt. In manchen

⁷<https://developer.apple.com/documentation/swiftui/viewbuilder> (Abgerufen am 1.2.2021)

⁸<https://developer.apple.com/documentation/swiftui/geometryreader> (Abgerufen am 1.2.2021)

Fällen ist es jedoch notwendig, dass bestimmte Datenverarbeitungsprozesse von der View-Hierarchie entkoppelt werden, beispielsweise bei periodischen Hintergrundprozessen. Um in diesem Fall dennoch bspw. eine entsprechende Meldung im UI anzuzeigen, können globale Events ausgelöst werden, die von Observern innerhalb der View aufgegriffen und zur Anzeige verarbeitet werden können. Hierbei kann das `NotificationCenter`⁹ genutzt werden, um Nachrichten auf einen Event-Bus zu pushen und innerhalb der View über einen so genannten Publisher¹⁰ zu empfangen. Auch dieses Pattern wird später noch an einem konkreten Beispiel im Rahmen des Implementationskonzeptes zu den Spiel-Errungenschaften gezeigt.

⁹<https://developer.apple.com/documentation/foundation/notificationcenter> (Abgerufen am 1.2.2021)

¹⁰<https://developer.apple.com/documentation/combine/publisher> (Abgerufen am 1.2.2021)

3 Analyse

Bevor auf Grundlage der architekturellen Ideen aus Kapitel 2 ein konkretes Lösungskonzept entwickelt werden konnte, musste die bestehende App zunächst analysiert werden. Diese auf den Entwicklungsprozess vorbereitenden, analytischen Aspekte sollen in diesem Kapitel erörtert werden, um im nächsten Kapitel die konkreten Implementationskonzepte vorzustellen.

3.1 Ermittlung einer Dialoglandkarte

Zu Beginn der Analyse wurde die bestehende iOS App getestet, um alle möglichen Navigationswege und Dialoge zu sammeln und innerhalb einer Übersicht zu betrachten. Die in Abbildung 3.1 gezeigte Dialoglandkarte suggeriert bereits die verschiedenen Gruppierungen von Ansichten, welche wir später zur Erstellung von neuen Packages heranziehen konnten. Zu sehen ist beispielsweise, dass die Spiel-Ansichten, der Einführungsassistent, oder bspw. auch die Kalender-Ansicht bereits gut voneinander separierbar sind. Die aufgezeichneten Navigationspfeile zwischen den Ansichten konnten genutzt werden, um die Datenflüsse hierbei vor der Erstellung des Konzeptes besser zu verstehen und die jeweils hierfür am Besten geeigneten Patterns zu finden.

3.2 Struktur des Couchbase-Datenbank-Frameworks

Die Persistierung der Applikationen kann über verschiedene Frameworks geschehen, iOS bietet standardmäßig bereits eine sehr rudimentäre Persistenzschnittstelle unter UserDefaults¹, auf Android analog hierzu SharedPreferences². Da jedoch nicht nur eine größere Anzahl an Daten persistiert werden müssen (Sponsoren, Spieldaten, Kalenderereignisse, Heatmap uvm.), sondern diese auch mit einem Server-Backend nach dem Offline-First-Prinzip synchronisiert werden sollen, kommen nur wenige fortgeschrittene Persistenz-Frameworks in Frage. Ehemals nutzten die Apps eine Realm-Datenbank, welche wegen externer Konditionen jedoch nicht mehr in Frage kam. Daher wurde ein Couchbase-Framework erstellt, inklusive eines Server-Backend-Systems, welches einen Synchronisationsservice und unter anderem auch einen Registrationsservice umfasst.

¹<https://developer.apple.com/documentation/foundation/userdefaults> (Abgerufen am 1.2.2021)

²<https://developer.android.com/reference/android/content/SharedPreferences> (Abgerufen am 1.2.2021)

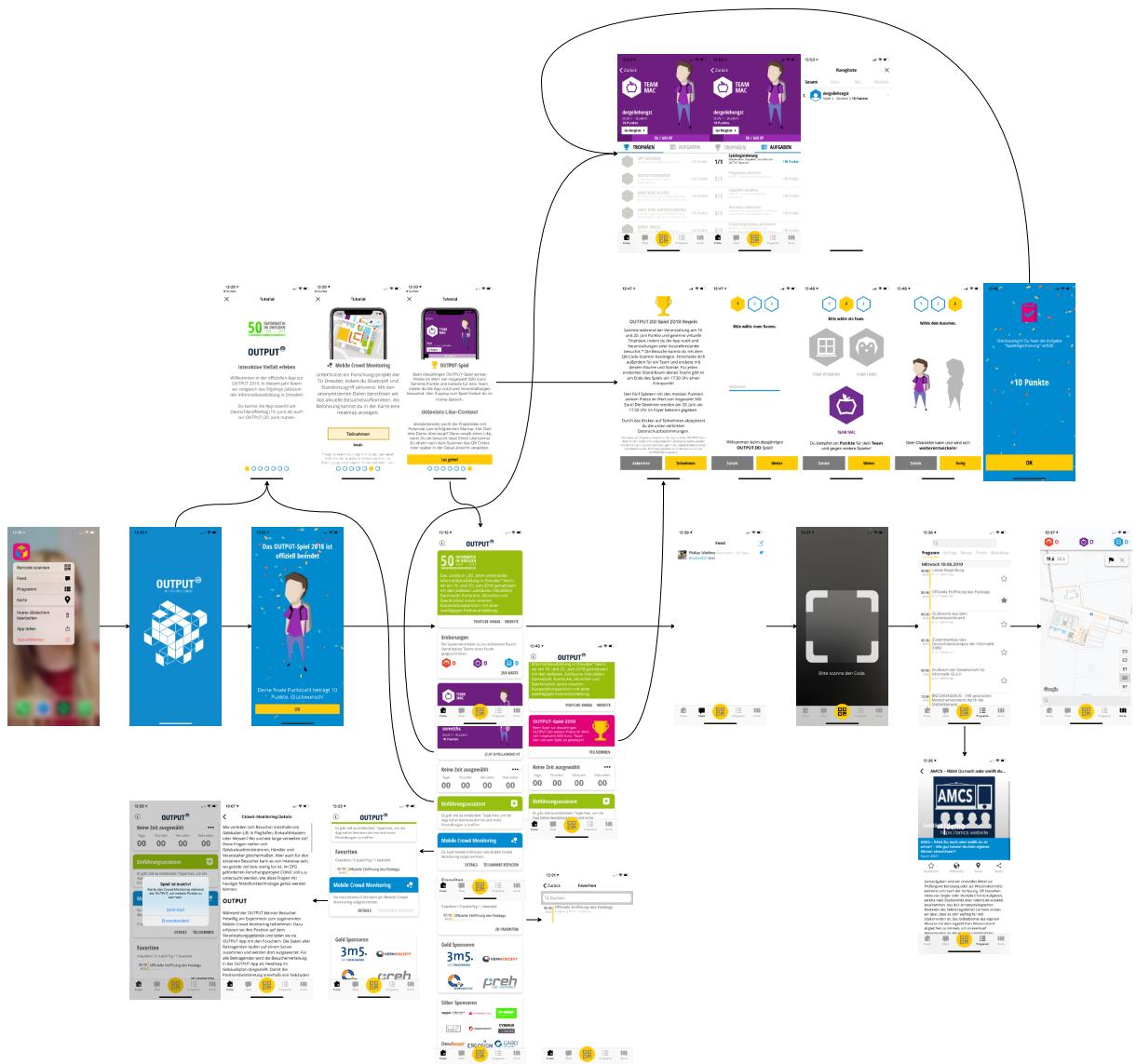


Abbildung 3.1 Die erstellte Dialoglandkarte zur bestehenden Version der iOS App zeigt die Navigationswege innerhalb der App, sowie die genutzten UI-Elemente.

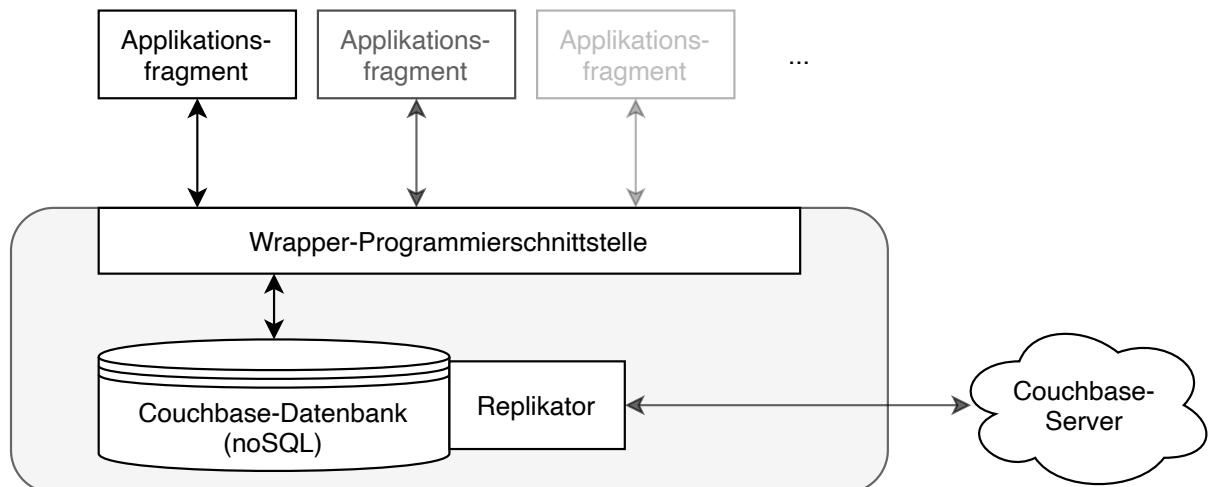


Abbildung 3.2 Eine Übersicht über das zu integrierende Couchbase-Framework. Über eine abstrahierende Wrapper-Programmierschnittstelle wird auf die Datenbank zugegriffen.

Die Metastruktur des Frameworks ist in Abbildung 3.2 gezeigt. Um die Nutzung von Couchbase zu vereinfachen, wurde das Framework in eine abstrahierende Wrapper-Programmierschnittstelle eingebunden, welche die wichtigsten CRUD³-Operationen, On-Change-Listener und auch die Synchronisationskomponente (Replikator) bereitstellt.

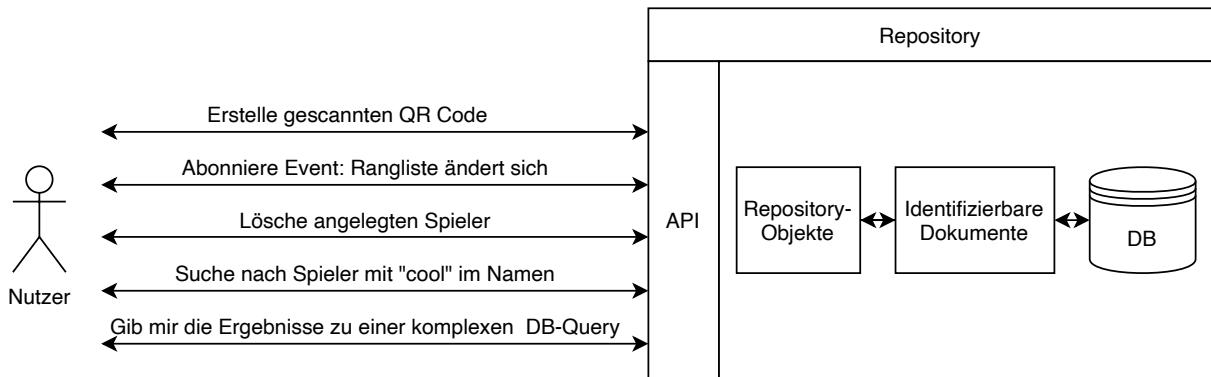


Abbildung 3.3 Das Couchbase-Framework kapselt die Datenmodelle in voneinander separate Repositories, auf denen unter anderem grundlegende CRUD-Operationen möglich sind.

Abbildung 3.3 illustriert, wie die Datenmodelle der Applikation geschrieben und gelesen werden können. Die Repository-Schnittstelle des jeweiligen Datenmodells bietet die dazugehörigen Funktionalitäten an.

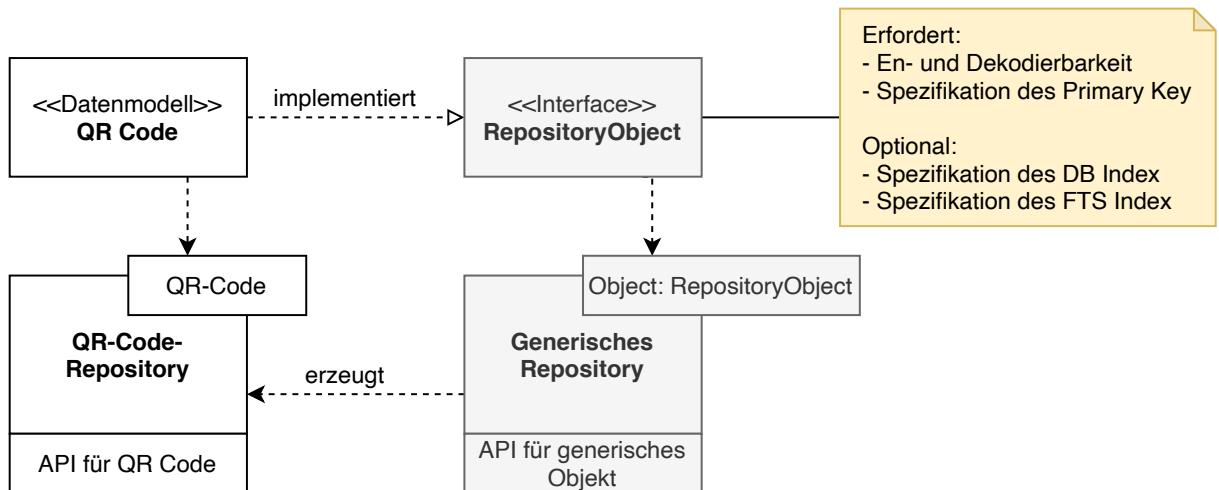


Abbildung 3.4 Die Repository-Schnittstelle wird generisch durch das Datenmodell und das dazugehörige Repository erzeugt. Der Änderungsaufwand bei einer Änderung des Datenmodells ist somit sehr gering und neue Datenmodelle können ohne größeren Aufwand hinzugefügt werden.

Wie in Abbildung 3.4 gezeigt, werden die Repositories vom Framework generisch inferiert und anschließend in Form von standardisierten Schnittstellen bereitgestellt. Die diesen Schnittstellen zugrunde liegenden Datenbankabfragen müssen somit in der App nicht implementiert werden. Gleichzeitig kann das Datenbank-Backend bei Fortbestehen der Schnittstellen auch im Hintergrund ausgetauscht werden, ohne Änderungen in der App zu bedingen.

³CRUD: Create, Read, Update, Delete

3.3 Top-Level-Architektur und Muss-Kriterien

Auf Grundlage der erstellten Dialoglandkarte sowie der bekannten Datenbankstruktur des Couchbase-Frameworks wurde ein Top-Level-Architektur-Modell (TLA) erstellt, um die Applikation in voneinander weitestgehend unabhängige Teilbereiche zu separieren.

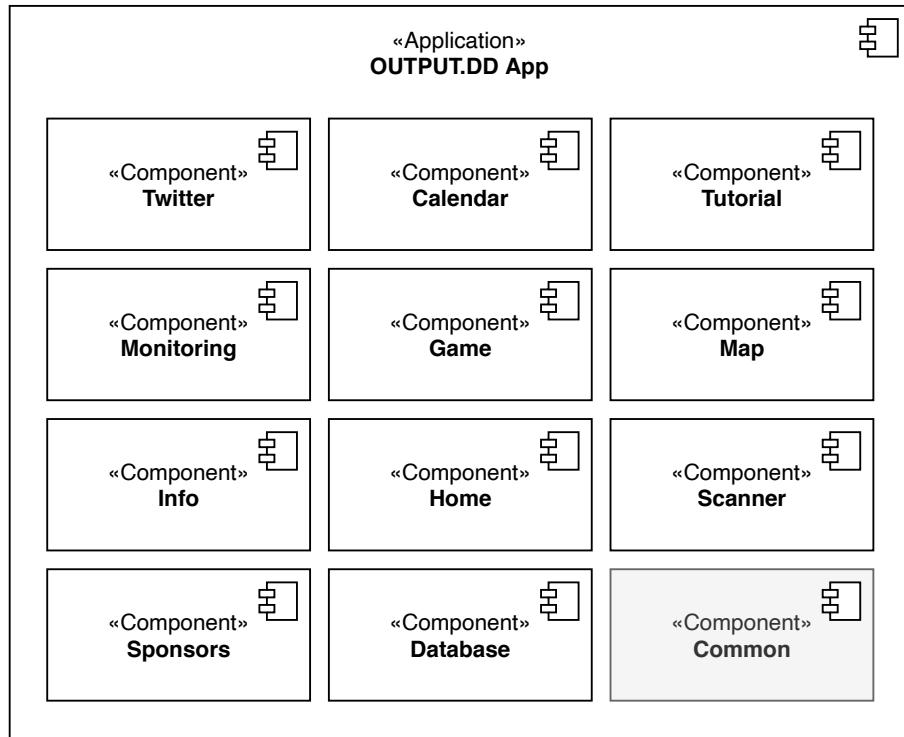


Abbildung 3.5 Die TLA der Applikation im Überblick.

Wie in Abbildung 3.5 zu sehen, wurde die Applikation in 12 Komponenten unterteilt. Im folgenden werden die zu diesen Komponenten gehörenden Muss-Kriterien erläutert:

- **Twitter** stellt alle Feed-Funktionalitäten bereit, inklusive der Feed-Ansicht und der Möglichkeit, neue Tweets zu erstellen.
- **Calendar** realisiert die Kalender-Ansicht des Veranstaltungsplans, inklusive der Detailansichten zu den jeweiligen Ereignissen, einer Möglichkeit, Ereignisse zu favorisieren, sowie nach bestimmten Ereigniskategorien zu filtern und zu suchen.
- **Tutorial** stellt alle Funktionalitäten und Ansichten des Einführungsassistenten bereit, welcher die wesentlichsten Bestandteile der App kurz erläutert.
- **Monitoring** kapselt die Crowd-Monitoring-Funktionalitäten des externen Crowd-Monitoring-Frameworks, sowie die Ansichten, die zur Erklärung und Steuerung dessen notwendig sind.
- **Game** ist die funktionell mächtigste Komponente, da sie alle Ansichten und Funktionalitäten der Gamification beinhaltet, darunter die Spielerregistrierung, die Trophäen- und Errungenschaften-Ansichten, die Fortschrittsansicht, die Rangliste sowie die Eroberungs-Ansicht.
- **Map** beinhaltet die Kartenansichten, darunter die Möglichkeiten der Auswahl bestimmter Etagen und Ansichtsoptionen wie die aktuellen Eroberungen oder die Anzeige des Besucheraufkommens über die Heatmap.
- **Info** stellt wichtige informative Ansichten bereit, darunter die Danksagungen und Lizenzen, aber auch Link-Ansichten zu Datenschutzerklärung und Impressum.

- **Home** dient als Verbinder zwischen verschiedenen Ansichten durch die Bereitstellung einer Home-Ansicht, auf der verschiedene Teilsichten und Links anderer Komponenten gezeigt werden.
- **Scanner** realisiert die QR-Code-Scanner-Funktionalitäten, die benötigt werden, um Errungenschaften für das OUTPUT-Spiel freizuschalten.
- **Sponsors** ist verantwortlich für die Bereitstellung von Ansichten zur Anzeige von Sponsoren in der Home-Ansicht.
- **Database** stellt die zum Couchbase-Framework ergänzend benötigten Ansichten und Schnittstellen bereit.
- **Common** stellt Komponenten und Erweiterungen bereit, die in der gesamten Applikation benötigt werden, beispielsweise eine modulare Card-Ansicht, die sich im UI (siehe Dialoglandkarte) häufig wiederfindet.

Diese TLA soll nachfolgend im Rahmen verschiedener Konzepte verfeinert und implementiert werden.

4 Konzept und Implementation

Im Folgenden soll auf die in Kapitel 2 beschriebenen Patterns und Grundlagen zurückgegriffen werden und, zusammen mit den in Kapitel 3 analysierten Komponenten und Navigationswegen, das erstellte Konzept und die dazugehörige Implementation diskutiert werden.

4.1 Konzeption einer Ordnerstruktur

Die Konzeption einer Ordnerstruktur steht zu Beginn der Bearbeitung und Implementation der Applikation und soll für eine allgemeine Verbesserung der Applikationsstruktur gegenüber den in Kapitel 1 genannten Strukturproblemen sorgen. Die Struktur ist dabei nach folgendem Regelwerk aufgebaut:

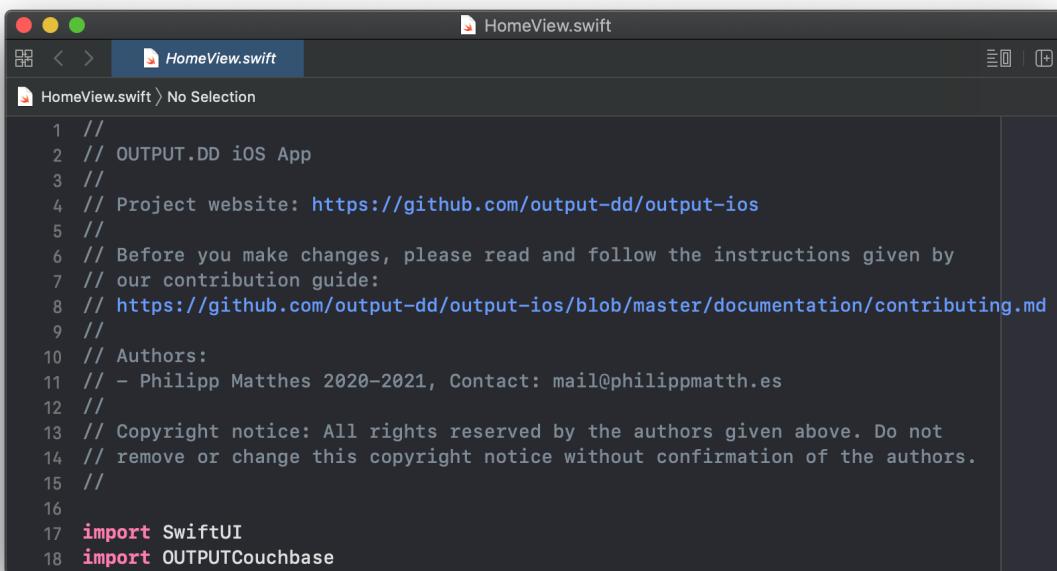
- Im Root-Verzeichnis des Repositories befinden sich Konfigurationsdateien des Package-Managers Cocoapods, Github-Workflows und weitere Hilfsdateien, die nicht direkt zur Codebasis der App gehören. Außerdem befindet sich an dieser Stelle das Applikationsverzeichnis.
- In der obersten Applikationsebene befinden sich die komponentenspezifischen Ordner nach TLA, sowie die Applikationsklasse selbst und global konfigurierende Codefragmente und Artefakte (z.B. Lokalisierungsdateien, Info.plist).
- Außerdem in der obersten Applikationsebene befindet sich ein Ordner Resources, in dem sich die Assets (z.B. AppIcon, UI-Elemente), jedoch auch Fonts, Lizenzen, eventuelle HTML-Templates oder andere Datenfragmente befinden.
- Jede Komponente ist entweder separiert in weitere Teilkomponenten, oder besteht aus einer Auswahl aus den Unterordnern Mocks, Views, ViewModels oder anderen, Backend-Funktionalitäten kapselnden Ordnern.
- Jedes Mocks Verzeichnis dient zur Bereitstellung von Mock-Objekten für die SwiftUI-Previews.
- Jedes ViewModels Verzeichnis dient zur Bereitstellung von Modellen, die selbst nicht als Model Teil des Couchbase-Frameworks sind, sondern zusätzlich von Views zur Darstellung benötigt werden.
- Jedes Views Verzeichnis beinhaltet die Ansichten der Komponente, die zur Darstellung der Applikation benötigt werden.

Die Applikation ist somit auf oberster Ebene vertikal geteilt, um eine bestmögliche funktionelle Separation der Module voneinander zu bestärken. Auf Komponenten-Ebene ist die Applikation horizontal geteilt, um die Ansichten bestmöglich von Datenmodellen und den dazugehörigen Backend-Funktionalitäten zu trennen. Außerdem soll darauf geachtet werden, dass die Klassen

und Dateien innerhalb der jeweiligen Module möglichst als Präfix den Modulnamen tragen, beispielsweise TwitterHTMLView der Komponente Twitter, um sie später im Code besser voneinander unterscheiden und separieren zu können.

4.2 Einführung eines Copyright-Templates

Standardmäßig ergänzt XCode (iOS-IDE von Apple) beim Erstellen einer leeren Codedatei immer einen Copyright-Hinweis. Dieser ist jedoch nicht immer deskriptiv genug bzw. referenziert die dem Projekt beigegebene Lizenz nicht korrekt. Außerdem enthält der Copyright-Hinweis standardmäßig den gewählten Dateinamen, wodurch bei einer späteren Modifikation des Dateinamens auch immer gleichzeitig der Hinweis modifiziert werden muss, um Diskrepanzen zwischen dem Dateinamen und dem Hinweis zu verhindern. Deshalb wurde im Rahmen des Projektes ein Copyright-Template erstellt und in XCode, auf alle Entwickler des Projektes übergreifend, eingebunden.



The screenshot shows the XCode interface with the file 'HomeView.swift' open. The code editor displays the following content:

```
1 //  
2 // OUTPUT.DD iOS App  
3 //  
4 // Project website: https://github.com/output-dd/output-ios  
5 //  
6 // Before you make changes, please read and follow the instructions given by  
7 // our contribution guide:  
8 // https://github.com/output-dd/output-ios/blob/master/documentation/contributing.md  
9 //  
10 // Authors:  
11 // - Philipp Matthes 2020-2021, Contact: mail@philippmatth.es  
12 //  
13 // Copyright notice: All rights reserved by the authors given above. Do not  
14 // remove or change this copyright notice without confirmation of the authors.  
15 //  
16  
17 import SwiftUI  
18 import OUTPUTCouchbase
```

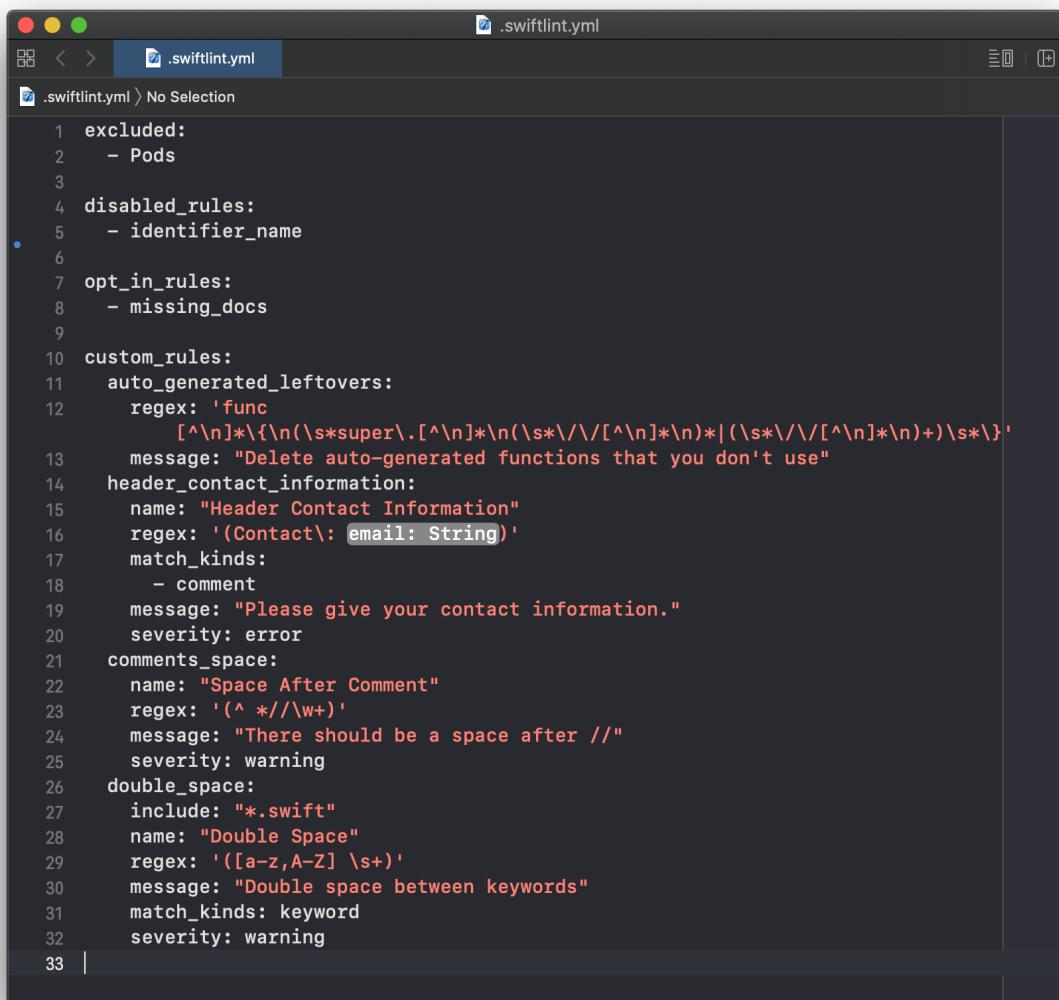
Abbildung 4.1 Die erstellte Copyright-Template am Beispiel der HomeView.

Wie in Abbildung 4.1 sichtbar, enthält die Copyright-Template zusätzlich zu Hinweisen und Links über die Herkunft des Code-Fragmentes auch einen Link zu den Contribution-Guidelines, welche später im Konzept erläutert werden. Die Template ist bewusst so ausgelegt, dass sich mehrere Autoren inklusive ihrer Kontaktinformationen eintragen können. Letzteres ist wichtig, da sich im Projektteam noch nicht auf eine Lizenzierung der App unter einer Open-Source-Lizenz festgelegt wurde und daher die Codefragmente unter das Kopierrecht des Autors fallen. Sollte eine Weiterverwendung der Codefragmetes über den ursprünglich dafür bestimmten Rahmen stattfinden, so können die Autoren hierüber kontaktiert werden.

4.3 Einführung von Codierungsrichtlinien

Um das zusätzlich in der bestehenden Applikation identifizierte Problem mit der allgemeinen Degradation der Codequalität zu mitigieren und dieser präventiv entgegen zu wirken, wurde

als statisches Codeanalysetool SwiftLint eingeführt und konfiguriert.

A screenshot of a code editor window titled ".swiftlint.yml". The file contains configuration for SwiftLint. It includes sections for excluded files (Pods), disabled rules (identifier_name), opt-in rules (missing_docs), and custom rules. One custom rule checks for auto-generated functions and requires header contact information. Another rule ensures there is a space after double slashes. The configuration uses YAML syntax with numbered lines for readability.

```
1 excluded:
2   - Pods
3
4 disabled_rules:
5   - identifier_name
6
7 opt_in_rules:
8   - missing_docs
9
10 custom_rules:
11   auto_generated_leftovers:
12     regex: 'func
13       [^\n]*{\n(\s*super\.[^\n]*\n|\s*\//[^^\n]*\n|(\s*\//[^^\n]*\n+)\s*)\s*\}'
14     message: "Delete auto-generated functions that you don't use"
15   header_contact_information:
16     name: "Header Contact Information"
17     regex: '(Contact: email: String)'
18     match_kinds:
19       - comment
20     message: "Please give your contact information."
21     severity: error
22   comments_space:
23     name: "Space After Comment"
24     regex: '^( *//\w+'
25     message: "There should be a space after //"
26     severity: warning
27   double_space:
28     include: "*.swift"
29     name: "Double Space"
30     regex: '([a-z,A-Z] \s+)'
31     message: "Double space between keywords"
32     match_kinds: keyword
33     severity: warning
34
35 |
```

Abbildung 4.2 Die Konfiguration des statischen Codeanalysetools SwiftLint.

In Abbildung 4.2 ist die Konfiguration von SwiftLint gezeigt. Neben gewöhnlichen Konfigurationen des Frameworks beinhaltet die Konfiguration auch eigene Regeln, darunter eine Regel, die auf das Vorhandensein des Copyright-Templates prüft. Das Tool wurde anschließend im Buildprozess von XCode integriert, so dass bei einer Verletzung der Codierungsrichtlinien diese innerhalb des Editors als Warnung (oder bei schwerwiegenden Verletzungen als Error) markiert wird und refaktorisiert werden kann.

4.4 Implementation des Frontends

Nach der Konzeption einer klaren Ordnerstruktur, sowie der Vorbereitung der Implementation mithilfe der Einführung des Copyright-Templates und der Installation eines statischen Code-Checkers wurde mit der Implementation des Frontends begonnen. Die Implementation fand hierbei entkoppelt vom Datenbank-Backend statt, indem für jede Ansicht temporäre ViewModels entworfen wurden, die zur Darstellung benutzt werden konnten. Dies ergab wesentliche

Vorteile im Entwicklungsprozess:

- Die Ansichten konnten schnell mithilfe der SwiftUI-Preview erstellt werden, ohne eine aufwändige Anbindung des Datenbank-Backends oder eine entsprechende Daten-Initialisierung.
- Die Ansichten konnten durch diesen Zwischenschritt leichter vom Datenbank-Backend separiert werden.

Außerdem konnte auf diese Weise in wenigen Wochen ein voll funktionsfähiges UI erstellt werden, lediglich mit einer fehlenden Anbindung an das Datenbank-Backend. Das UI konnte so bereits getestet und zwischen der iOS- und der Android-Version abgeglichen werden.

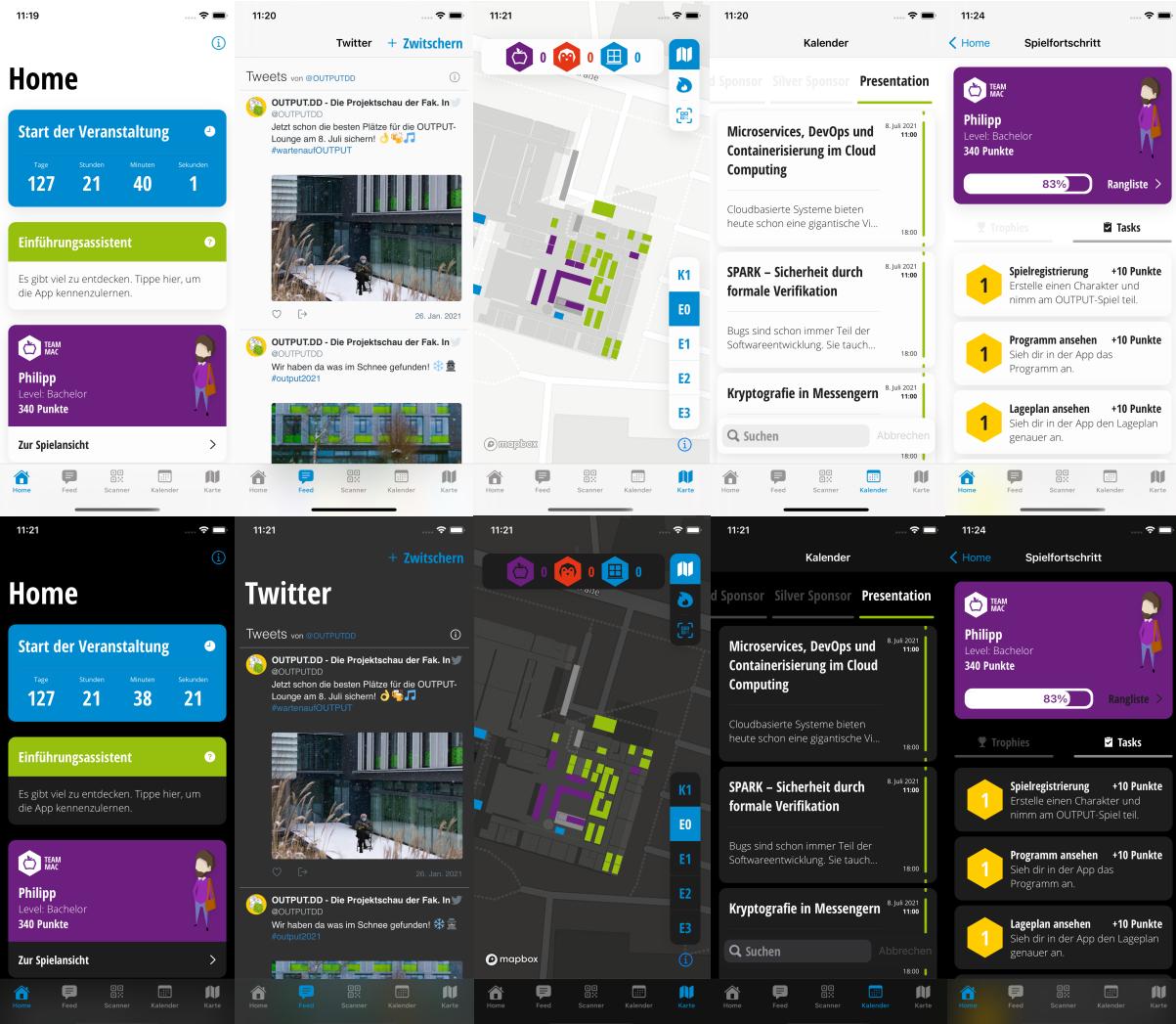


Abbildung 4.3 Die Hauptansichten des UI der neu implementierten App im Profil.

In Abbildung 4.3 werden die 5 Hauptansichten der App gezeigt, außerdem wurden noch einige weitere Ansichten (analog zur Dialoglandkarte) implementiert. Alle Ansichten orientieren sich hierbei am Aufbau der bisherigen Ansichten in der bestehenden App. Gleichzeitig wird das UI an vielen Stellen modernisiert, modularisiert und nutzerfreundlicher gestaltet. Zur Modernisierung gehört bspw. eine Implementation eines Dark-Mode Features, welches sich an der Systemeinstellung des Nutzers orientiert und bei Dunkelheit die Augen vor zu viel Helligkeit schützt. Außerdem wurden an verschiedenen Stellen bestimmte Animationen implementiert, bspw. bei dem Berühren eines Buttons, um die Benutzung der UI noch intuitiver und zufriedenstellender

zu gestalten. Gleichzeitig wurde insbesondere auch auf Grundprinzipien des Usability Engineering geachtet, sodass Buttons immer eine bestimmte Größe je nach Platzierung im UI besitzen und die meisten Interaktionen in der App in der unteren Hälfte des Bildschirms mit dem rechten Daumen getätigt werden können. Bei wichtigen Interaktionen erhält der Nutzer zudem ein haptisches Feedback. Zur Modularisierung des UI tragen bestimmte UI-Komponenten bei, die innerhalb verschiedener Ansichten mehrfach wiederverwendet werden können, wie eine (auch in Abbildung 4.3 sichtbare) CardView oder ein eigener ViewPaginator. Nicht nur bei letzterem, sondern auch bei der Auswahl der Formen, Farben und Fonts wurde auf die Vorgaben aus der OUTPUT.DD Corporate Identity geachtet, so dass beispielsweise die Farbpalette übernommen wurde und ein wiederverwendbarer ViewModifier für die Applizierung des OpenSans-Fonts innerhalb von SwiftUI implementiert wurde. Außerdem wurde die Applikation vollständig internationalisiert, auf Grundlage der von SwiftUI bereitgestellten Funktion zur String Interpolation in Localizable.strings Dateien.

Reimplementation von weiteren Teilansichten: Zusätzlich zu einer Reimplementation des gesamten UI, inklusive komplexerer Ansichten wie die Reward-Ansicht mit einer Konfetti-Animation, wurde in der Map-Ansicht das Google Maps Framework durch das MapBox Framework ersetzt. Vor dieser Entscheidung wurde anhand einer internen Team-Diskussion abgewägt, welches der beiden Frameworks zukünftig genutzt werden soll. Aufgrund der Nutzerfreundlichkeit, der leichten Anbindbarkeit und der Verwendung des Frameworks in anderen Projekten der Professur entschieden wir uns für MapBox. In der Twitter-Ansicht wurde während der Implementation des Frontends ein weiteres Problem identifiziert, welches darin bestand, dass das von der bestehenden App genutzte Twitter-Framework von Twitter nicht länger unterstützt wurde und damit die Gefahr bestand, dass in Zukunft diese UI-Komponente nicht mehr funktionieren könnte. Daher konzipierten wir als Lösung hierfür eine rein auf HTML basierende TwitterHTMLView, welche das von Twitter bereitgestellte Twitter Widget nutzt. Dies hat nicht nur den Vorteil, dass das nicht mehr unterstützte Framework entfernt werden konnte, sondern auch, dass kein API-Schlüssel mehr für die Verwendung notwendig ist.

4.5 Integration einer CI-Pipeline

Um die Applikation an verschiedene Tester über die von Apple bereitgestellte App „TestFlight“ zu verteilen, bietet XCode die Möglichkeit, die Applikation zu signieren, in einem Archiv zu verpacken, dieses an die Plattform AppStore Connect zu übermitteln, und von dort (unter Vorausfüllung von Testinformationen) an die Tester zu übermitteln. Dieser Prozess ist jedoch sehr zeitaufwändig und dauert, je nach Tageszeit ca. 20-30 Minuten. Daher bat es sich an, die Applikation in einer Continuous-Integration-Pipeline einzubinden, welche diesen Prozess übernimmt. Hierzu wurde ein GitHub-Workflow adaptiert, der bereits für ein anderes iOS-Projekt („Peerbridge“ Blockchain Messenger) erstellt wurde. Der Workflow nutzt unter anderem das Framework Fastlane, um die Applikation zu archivieren und an die Tester vollautomatisch zu übermitteln. Hierfür sind einige technische Details von Relevanz, beispielsweise das Importieren des Signatur-Zertifikats in die CI-Pipeline, welche jedoch an dieser Stelle nicht weiter erläutert werden sollen. Für die Distribution wurden die technisch notwendigen Schritte in einem Delivery-Guide zusammengefasst, welcher im App-Repository unter <https://github.com/output-dd/output-ios/blob/master/documentation/delivery.md> (Zuletzt abgerufen am 31.1.2021) zu finden ist. Aus der Integration des Workflows bildet sich eine signifikante Zeitsparnis bei der Distribution der Applikation an die Tester. Die Applikation muss lediglich vom Entwickler mit einer Versionsnummer getaggt werden und die Bestimmungen von Apple erfüllen. Letztere sind konkret, dass die Build- und Versions-Nummer vor dem Taggen inkrementiert werden muss, sowie, dass die eventuell notwendigen Berechtigungen in der Info.plist Datei beschrieben

sind.

4.6 Fusion des Frontends mit dem Datenbank-Backend

Auf Grundlage des implementierten Frontends und der CI-Pipeline konnten nun sukzessiv die einzelnen Komponenten und Ansichten um eine Backend-Persistierung erweitert und damit die übergangsweise genutzten ViewModels an geeigneten Stellen substituiert werden. Hierzu wurde das Datenbank-Backend über den Package-Manager Cocoapods als privates Pod hinzugefügt und die CI-Pipeline um die entsprechende Berechtigung ergänzt, damit diese auf das private Repository zugreifen kann. Anschließend wurden noch kleinere Änderungen am Datenbank-Backend, vor allem der Datenbank-Modelle vorgenommen, welche sich um Laufe der Zeit leicht geändert hatten. Mithilfe dieser Änderungen konnte nun auf Grundlage der in Kapitel 2 beschriebenen Patterns die Integration beginnen.

4.6.1 Vorbemerkungen für die Darstellung der SwiftUI-Preview

Wie in Kapitel 2 bereits angemerkt, ist ein zentrales Feature von SwiftUI die schnelle deklarative Erstellung von Ansichten. Dies wird unter anderem ermöglicht durch eine Echtzeit-Preview der Ansicht in XCode.

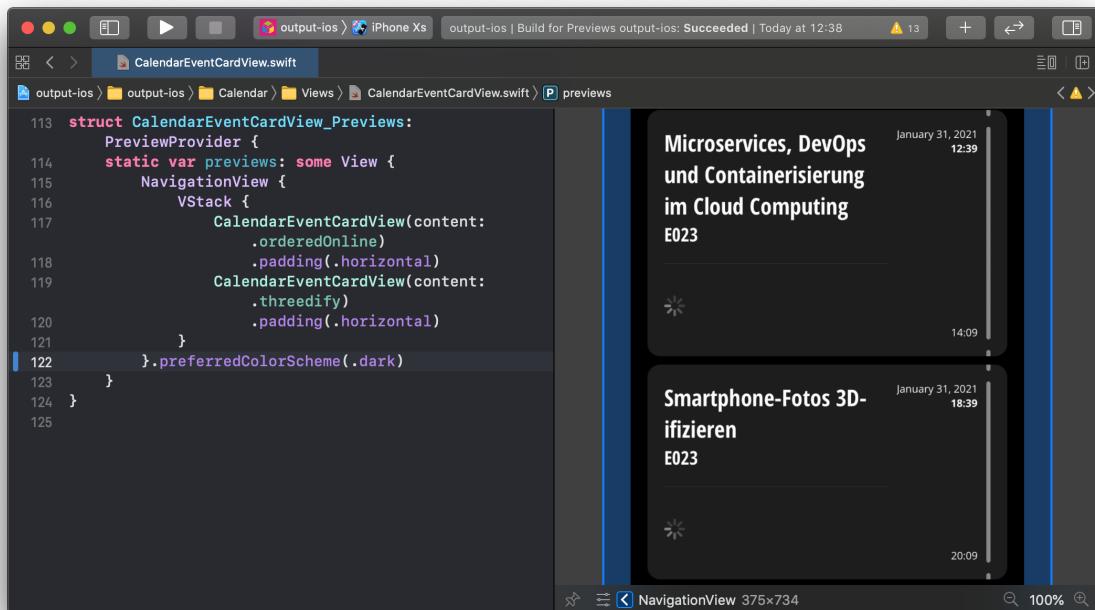


Abbildung 4.4 Die SwiftUI-Preview der `CalendarEventCardView` innerhalb von XCode.

Abbildung 4.4 zeigt eine solche Preview innerhalb XCode und wie diese konfiguriert werden kann. Hieraus entwickelt sich jedoch eine weitere Herausforderung bei der Integration des Datenbank-Backends in den Views, denn die Views sollten während der Preview nicht auf ein existierendes Datenbank-Backend angewiesen sein. Eine Vorinitialisierung des Datenbank-Backends in der Preview wäre einerseits mit sehr viel zusätzlichem Boilerplate-Code verbunden, andererseits könnte die Preview dann nicht schnell editiert werden, weil bei jeder Änderung der Ansicht das

Datenbank-Backend wieder initialisiert werden müsste. Im Folgenden Abschnitt wird die hierfür konzipierte Lösung gezeigt.

4.6.2 Nutzung des Environment- und Coordinator-Patterns für die Anbindung der Views

Um in der Preview der Views auf die zuvor erstellten Mocks zur schnellen Darstellung und Änderung der Ansicht zugreifen zu können und gleichzeitig beim Testen der Applikation auf dem Simulator oder dem physischen Gerät die initialisierte Datenbank zu nutzen, werden die anzubindenden Views mit jeweils einem `DemoCoordinator` oder einem `PersistedCoordinator` ausgestattet. Die jeweiligen Koordinatoren erben hierbei von der selben Klasse `Coordinator` - welche die benötigten Schnittstellen zur Verfügung stellt. Die zwei unterschiedlichen Subklassen-Koordinatoren unterscheiden sich infolge dessen nur darin, ob zum Prozessieren bzw. Holen der Daten die Datenbank genutzt wird, oder lediglich Mock-Objekte für die Preview zurückgegeben werden.

```
20 private class TrophyCoordinator: ObservableObject {  
21     @Published var trophies: [OUTPUTCouchbase.Trophy]?  
22  
23     /// Initialize the coordinator.  
24     required init() {}  
25  
26     func loadTrophies(repos: RepositoryStack?) {  
27         fatalError("ERROR: loadTrophies has not been implemented.")  
28     }  
29 }  
30  
31 private class DemoTrophyCoordinator: TrophyCoordinator {  
32     override func loadTrophies(repos: RepositoryStack?) {  
33         DispatchQueue.main.asyncAfter(deadline: .now() + 1) {  
34             self.trophies = [.example]  
35         }  
36     }  
37 }  
38  
39 private class PersistedTrophyCoordinator: TrophyCoordinator {  
40     override func loadTrophies(repos: RepositoryStack?) {  
41         guard let repos = repos else { return }  
42         repos.trophyRepository.getAllObjects { response in  
43             switch response {  
44                 case .failure(let error):  
45                     print("WARNING: Error during trophy fetch: \(error)")  
46                 case .success(let trophies):  
47                     self.trophies = trophies  
48             }  
49         }  
50     }  
51 }
```

Abbildung 4.5 Das Coordinator-Pattern angewandt am Beispiel der Spiel-Trophäen, um eine schnelle SwiftUI-Preview ohne Daten-Initialisierung zu gewährleisten.

Zur Illustration ist dieses Pattern beispielhaft in Abbildung 4.5 an der Implementation gezeigt. Der `TrophyCoordinator` gibt die Schnittstellen vor, an dieser Stelle die geladenen Trophäen (bidirektional referenzierbar durch die Ansicht) sowie eine Funktion zum Laden der Trophäen.

Der `DemoTrophyCoordinator` lädt lediglich gemockte Beispiel-Trophäen nach einer kurzen Verzögerung, um den Ladeprozess zu simulieren. Der `PersistedTrophyCoordinator` greift auf die Repositories und damit auf die Datenbank zu, um die tatsächlich persistierten Trophäen zu laden.

```

54 private struct GameProgressTrophyStackViewInternal<C>: View
55     where C: TrophyCoordinator {
56         @EnvironmentObject private var repos: RepositoryStack
57
58         @StateObject private var coordinator = C()
59
60         init(_ t: C.Type) { /* Generic requirement */ }
61
62         var body: some View {
63             ScrollView {
64                 VStack {
65                     if let trophies = coordinator.trophies {
66                         ForEach(trophies) { trophy in
67                             CardView {
68                                 GameProgressTrophyView(trophy: trophy)
69                             }
70                             .padding(.horizontal)
71                         }
72                     } else {
73                         ProgressView()
74                     }
75                 }
76                 .padding(.vertical)
77                 .padding(.bottom, 64)
78                 .edgesIgnoringSafeArea(.bottom)
79             }.onAppear(perform: {
80                 coordinator.loadTrophies(repos: repos)
81             })
82         }
83     }
84
85 struct GameProgressTrophyStackView: View {
86     @Environment(\.isPreview) private var isPreview
87
88     var body: some View {
89         if isPreview {
90             GameProgressTrophyStackViewInternal(DemoTrophyCoordinator.self)
91         } else {
92             GameProgressTrophyStackViewInternal(PersistedTrophyCoordinator.self)
93         }
94     }

```

Abbildung 4.6 Die Anbindung der Koordinatoren an die View im Beispiel der `GameProgressTrophyStackView`.

Die Integration der Koordinatoren erfolgt über die Übergabe als generischen Typparameter im Konstruktor der View. Um die Koordinatoren außerhalb der View-Code-Datei unsichtbar zu machen und nicht den globalen Scope zu verschmutzen, in dem die Koordinatoren keine Bedeutung haben (da sie sich nur auf die einzelne View beziehen) wird die Entscheidung, welcher Koordinator genutzt wird, durch ein Umgebungsattribut `isPreview` inferiert. Dieses Umgebungsattribut wird aus der Prozessumgebung bezogen, welche Informationen beinhaltet, ob es sich bei der aktuellen Ausführung um eine SwiftUI-Preview-Umgebung handelt, oder

nicht. Entsprechend wird der `DemoCoordinator` nur genutzt, wenn es sich um eine SwiftUI-Preview-Umgebung handelt. Dieses Pattern wurde auf alle Views angewandt, welche bei der Ausführung auf dem Gerät (physisch oder Simulator) mit der Datenbank kommunizieren.

4.6.3 Nutzung des Proxy-Patterns für die Initialisierung der Datenbank

Um die Datenbank zu initialisieren, wird auf das in Kapitel 2 erläuterte Proxy-Pattern zurückgegriffen.

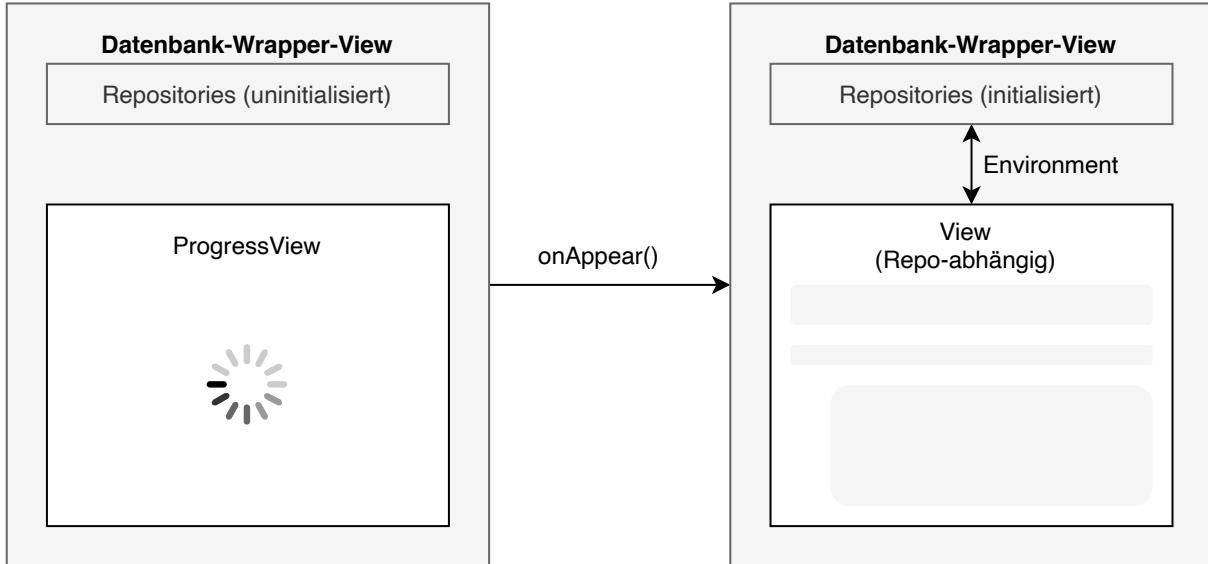


Abbildung 4.7 Abstrakte Übersicht über die Initialisierung der Datenbank.

Abbildung 4.7 zeigt die Initialisierung der Datenbank. Hierzu werden die datenbankabhängigen Ansichten als innere View mitgegeben. Sobald nun die Gesamtansicht erscheint (in diesem Fall beim Start der App) wird zunächst ein Loading-Indicator angezeigt und die Datenbank initial erstellt bzw. geladen. Anschließend werden die Repositories auf der Datenbank initialisiert und über das Environment der Ansicht an die Unteransichten übergeben. Die Unteransichten besitzen hierdurch die Möglichkeit, über die Umgebung auf die Repositories und somit auch auf die persistierten Objekte zuzugreifen. Gleichzeitig wird die Datenbank auch wieder deinitialisiert, sobald die Applikation geschlossen wird. Dies ist in SwiftUI sehr einfach möglich über die Nutzung von `onAppear()`¹ und `onDisappear()`². Es handelt sich infolgedessen insgesamt um einen eher SwiftUI-orientierten Lösungsansatz. Alternativ könnte als nicht direkt an SwiftUI gekoppelte Lösung auch bspw. ein Singleton-Pattern genutzt werden, indem die Datenbankrepositories auf einem Singleton statisch initiiert und verfügbar sind. Hierbei könnte das Singleton jedoch auch innerhalb der Proxy-View initialisiert werden, oder unabhängig von SwiftUI innerhalb der Methoden eines `AppDelegate`³. Beide Varianten, also einerseits die Nutzung des Environment zusammen mit einer Proxy-View und andererseits die Nutzung eines Singletons, wurden getestet und gegeneinander im Rahmen der Entwicklung abgewägt. Wegen der allgemeinen SwiftUI-Orientierung der Neuimplementation wurde sich in der iOS-Implementierung für ersteres entschieden.

¹[https://developer.apple.com/documentation/swiftui/view/onappear\(perform:\)](https://developer.apple.com/documentation/swiftui/view/onappear(perform:)) (Abgerufen am 1.2.2021)

²[https://developer.apple.com/documentation/swiftui/view/ondisappear\(perform:\)](https://developer.apple.com/documentation/swiftui/view/ondisappear(perform:)) (Abgerufen am 1.2.2021)

³<https://developer.apple.com/documentation/uikit/uiapplicationdelegate> (Abgerufen am 1.2.2021)

4.6.4 Testen der Datenbankfunktionalität

Um die Anbindung der Datenbank zu testen, wurden im Rahmen der Implementation zu Testzwecken auch mehrere Daten-Initialisierer implementiert. Hierbei wurden in die Repositories beim erstmaligen Start der Applikation teils vorliegende und teils zufällig generierte Objekte geschrieben, um die Darstellung der jeweiligen Daten innerhalb der angebundenen UI-Komponenten zu testen. Unter anderem wurden hierzu auch Test-Skripte entwickelt, bspw. um die Heatmap auf Grundlage der vorliegenden GeoJSON-Umrisse der Etagen des Andreas-Pfitzmann-Baus zu generieren. Somit konnte für die angebundenen Ansichten sichergestellt werden, dass unter Existenz von Objekten in der Datenbank diese korrekt behandelt und dargestellt werden. Gleichzeitig konnte auch die Performanz der Applikation getestet werden, beispielsweise durch die Generierung von mehreren tausenden Einträgen im Leaderboard der Gamification.

4.6.5 Anbindung der Synchronisationskomponente

Ähnlich zur Integration der Datenbank konnte für die Anbindung der Synchronisationskomponente auch das View-Proxy-Pattern genutzt werden. Hierbei wird zusätzlich zur Couchbase-Datenbank, die (neben ihrer Schnittstellen) lediglich eine Referenz zu einer physisch auf dem Dateisystem vorliegenden Datei darstellt, zwei verschiedene Replikatoren für unterschiedliche Repositories implementiert. Aus Datenschutz- und Datensicherheitsgründen wurden die Repositories des Couchbase-Datenbank-Frameworks in `personal` und `shared` separiert, wobei ersteres nur individuell für den Nutzer selbst verfügbar ist und letzteres für alle Nutzer. Die `shared` Repositories beinhalten öffentlich verfügbare Daten, wie bspw. die Sponsoren und Kalenderereignisse. Innerhalb der `personal` Repositories befinden sich vorrangig Daten zum individuellen OUTPUT.DD-Spieler und zur Favorisierung von Ereignissen. Entsprechend benötigt der Replikator für die `personal` Repositories eine zufällig generierte Authentifikation, welche zum jeweiligen Spieler des OUTPUT.DD-Spiels gehört, bei einem zum Server-Backend gehörenden Registrierungsservice erstellt und in den `UserDefaults` der Applikation persistiert wird. Somit wird der `personal` Replikator erst verbunden, wenn der Nutzer einen Spieler im OUTPUT.DD-Spiel (und damit seine Authentifikation) registriert hat. Gleichzeitig bleibt der `shared` Replikator stets auch ohne Authentifikation aktiv, um die öffentlichen Daten mit dem Server-Backend zu synchronisieren. Diese Funktionalität wurde innerhalb eines `Couchbase Managers` implementiert, welcher durch eine eigene Proxy-View initialisiert wird. Lediglich für die Registrierung des Spiels wird der `CouchbaseManager` an die Unteransichten weitergegeben, damit an dieser Stelle, bei erfolgreicher Registrierung und Authentifikation, der `personal` Replikator gestartet werden kann und unverzüglich alle Daten der `personal` Repositories synchronisiert werden.

4.6.6 Testen der Synchronisation

Zum Testen der Replikatoren wurde ein im Rahmen der Implementation des Couchbase-Datenbank-Frameworks entstandenes Docker-Test-Setup genutzt. In diesem Test-Setup sind alle notwendigen Services enthalten, welche für die Realisation der Synchronisation notwendig sind. Darunter:

- Der Couchbase Server, in welchem eine zentrale Referenz der Datenbank gespeichert wird.
- Das Couchbase Sync Gateway, welches eine Synchronisationsschnittstelle für die Replikatoren bereitstellt.
- Der Registrierungsservice, welcher zur Registrierung eines neuen Spielers genutzt wird.

- Der Sync Service, welcher die Couchbase-Daten mit der OUTPUT.DD-Website synchronisiert und bestimmte Daten aggregiert.
- Der Couchbase Update Notifier, welcher den Sync Service von bestimmten Aktualisierungen in der Datenbank notifiziert.

Dieses Test-Setup kann mithilfe von Docker-Compose lokal auf dem Entwicklersystem gestartet werden und als Endpunkt für die Replikatoren und die Spielerregistrierung referenziert werden. Mithilfe des Docker-Test-Setups konnte die Replikator-Funktionalität erfolgreich sichergestellt werden. Außerdem konnten einige Probleme in der Implementation des Test-Setups identifiziert werden. Darunter ein Problem bei der Registrierung eines neuen Spielers, bei dem der Registrierungsservice mit einem internen Server-Error (Code 500) antwortet, wenn der Spielername bereits vergeben ist, sowie eine fehlerhafte Aggregierung des Player-Modells im shared Repository, welches dazu führt, dass kein Leaderboard dargestellt werden konnte. Die Probleme wurden im Projektteam besprochen und die entsprechenden Fixes beim zuständigen Entwickler eingeleitet.

4.7 Listener- und Eventbasierte Anbindung der Gamification

Nachdem das Frontend der Applikation, sowie die Datenbank und die Synchronisationsfunktionalität implementiert werden konnte, wurden anschließend die in der Gamification der App erreichbaren Trophäen und Aufgaben analysiert und eine Implementation der jeweils dahinter stehenden Logik geplant. Hierzu wurde zunächst eine Übersicht über die vorhandenen Trophäen und Tasks erstellt und anhand dessen die Kriterien für die Freischaltung festgelegt, damit diese Kriterien möglichst einheitlich zwischen der iOS- und Android-Version der App gehabt werden. Außerdem wurden die Trophäen und Aufgaben in jeweils zwei Untergruppen separiert:

- **Datenbasierte** Errungenschaften, welche durch die Änderung oder Erstellung von bestimmten Objekten in der Datenbank erreicht werden können. Beispiele hierfür sind das Speichern eines gescannten QR Codes oder die Registrierung eines neuen Spielers.
- **Interaktionsbasierte** Errungenschaften, die durch eine bestimmte Interaktion mit Elementen der App erreicht werden können, welche jedoch nicht direkt eine Änderung der innerhalb der Datenbank persistierten Daten zur Folge hat. Ein Beispiel hierfür ist der Aufruf einer bestimmten Ansicht.

Die Freischaltung bestimmter Errungenschaften kann hierbei als Hintergrundprozess geschehen werden - der Nutzer führt bestimmte Aktionen aus, welche observiert werden und bei Zutreffen des jeweiligen Kriteriums der Errungenschaft eine Freischaltung zur Folge hat. Die observierenden Komponenten der Applikation entscheiden also selbstständig bei der Benachrichtigung durch das jeweilige Ereignis (datenbasiert oder interaktionsbasiert), ob die jeweilige Trophäe oder die Aufgabe freigeschaltet werden soll. Nach diesem Paradigma konnten die Errungenschaften gut von der sonstigen Darstellungs- und Geschäftslogik der App separiert werden.

Implementation der datenbasierten Errungenschaften

Da sich die Freischaltung von datenbasierten Errungenschaften direkt von der Änderung bestimmter Datenmodelle innerhalb der Datenbank ableitet, wurden für die Implementation Datenbank-Listener für die jeweiligen Datenmodelle genutzt, welche durch das Couchbase-Repository-Framework bereitgestellt wurden. Hierbei durchläuft jeder Observer nach der Initiation der Repositories folgenden Algorithmus:

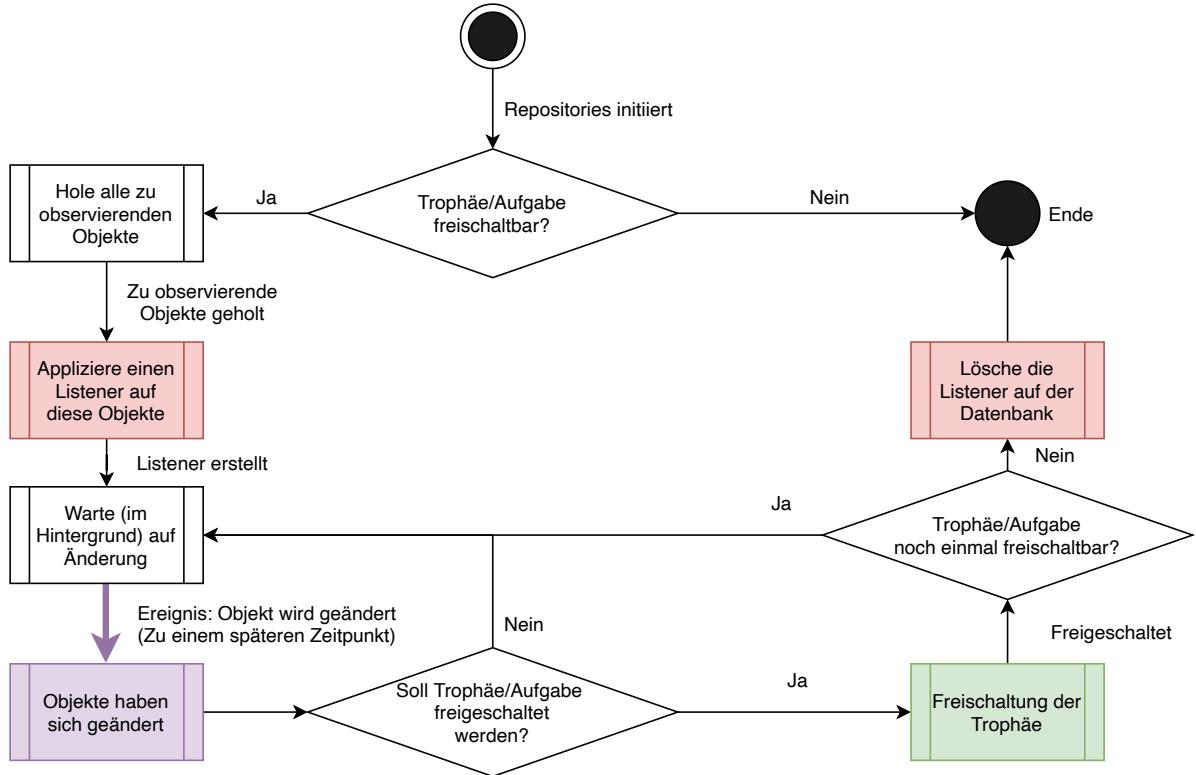


Abbildung 4.8 Ein Algorithmus für die Datenbank-Listener-basierte Freischaltung von Trophäen und Aufgaben. Rot hervorgehoben sind Prozesse, bei denen Listener appliziert oder entfernt werden. Violett hervorgehoben ist das auslösende Ereignis für eine Prüfung auf Freischaltung. Grün hervorgehoben ist die Freischaltung der Trophäe oder der Aufgabe.

Wie in Abbildung 4.8 illustriert, werden im Algorithmus bei einer Freischaltbarkeit der Trophäe bzw. Aufgabe die entsprechenden, zu observierenden Datenbankobjekte zunächst geholt. Wenn die Aufgabe oder die Trophäe bereits freigeschaltet wurde und nicht mehrmals freigeschaltet werden kann, werden keine Datenbank-Listener appliziert, um Ressourcen zu sparen. Wurden die zu observierenden Objekte geholt, wird ein Listener über das Repository-Interface der Datenbank auf diesen Objekten erstellt, welcher notifiziert wird, sobald die Datenbank eine Änderung der Objekte feststellt. Bei einer Änderung der relevanten Objekte, die im Hintergrund während der gesamten Laufzeit der App durch bestimmte Aktionen stattfinden kann, wird anschließend geprüft, ob die Trophäe oder Aufgabe freigeschaltet werden soll, oder nicht. Infolgedessen kann der Listener weiter fortexistieren, wenn bspw. die Trophäe oder die Aufgabe nicht freigeschaltet wurde oder mehrmals freigeschaltet werden können. Sonst wird der Listener vom Repository entfernt, um Ressourcen zu sparen, da er nicht weiter benötigt wird. Durch diesen Algorithmus konnten alle datenbasierten Errungenschaften implementiert werden. Beispielsweise existiert die Aufgabe „Spielregistrierung“, bei der ein Nutzer einen Spieler erstellen muss. Hierzu wird auf alle Objekte des LocalPlayerRepository ein Listener appliziert, und sobald ein Nutzer einen Spieler registriert und dieser im LocalPlayerRepository gespeichert wird, wird die Freischaltung der Aufgabe ausgelöst. Gleichzeitig lassen sich hierdurch QR-Code-basierte Aufgaben und Trophäen auch sehr leicht implementieren, indem die zu observierenden QR-Codes aus der Datenbank geholt werden, hierauf ein Listener erstellt wird, und beim Scannen eines neuen QR Codes geprüft wird, ob hierdurch die jeweilige Aufgabe oder Trophäe freigeschaltet werden soll.

Implementation von interaktionsbasierten Errungenschaften

Für bestimmte Aktionen, welche keine direkte Änderung von Objekten in der Datenbank induzieren, lässt sich der Algorithmus aus der vorigen Sektion nicht applizieren. Daher wurde hierfür ein weiterer Algorithmus entworfen:

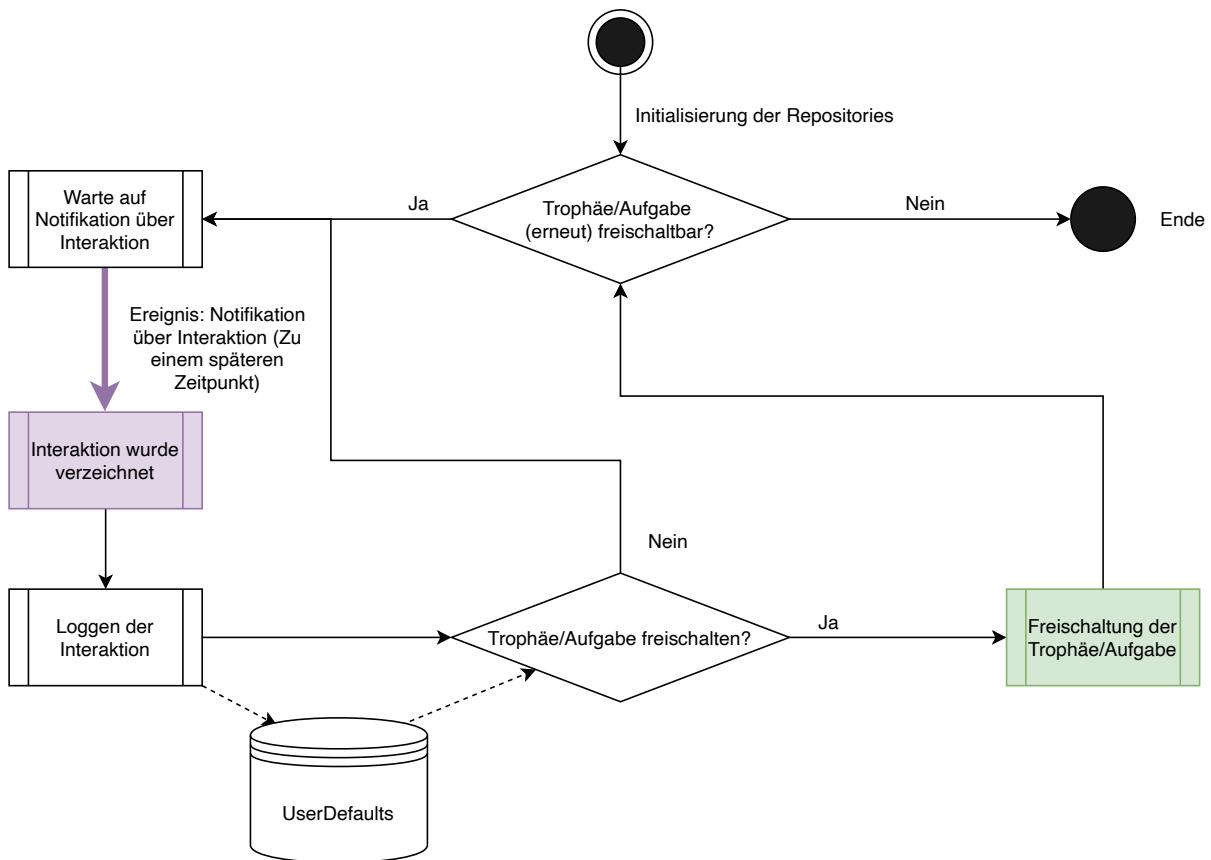


Abbildung 4.9 Ein Algorithmus für die interaktionsbasierte Freischaltung von Trephäen und Aufgaben. Violett hervorgehoben ist das auslösende Ereignis für eine Prüfung auf Freischaltung. Grün hervorgehoben ist die Freischaltung der Trephäe oder der Aufgabe.

Der in Abbildung 4.9 gezeigte Algorithmus ähnelt dem der datenbasierten Freischaltung, unterscheidet sich jedoch in wesentlichen Punkten. Statt Listener auf der Datenbank zu erstellen, wird eine Interaktion direkt aus dem UI an den Observer übergeben. Anschließend wird die Interaktion in den UserDefaults der App gespeichert, um anhand der vergangenen Interaktionen und der aktuellen Interaktion schließlich zu entscheiden, ob die Interaktion zu einer Freischaltung der jeweiligen Trephäe bzw. der jeweiligen Aufgabe führt. Beispielsweise existiert eine Trephäe „APP EXPLORER“, bei der die Besuche der 6 wichtigsten Ansichten als Interaktionen zwischengespeichert werden - wurden alle Ansichten besucht, wird die Trephäe freigeschaltet. Infolgedessen kann der Observer weiterhin bspw. von Besuchen neuer Ansichten informiert werden - durch die Freischaltung der nicht wiederholbaren Trephäe jedoch wird keine Prüfung auf Freischaltung mehr stattfinden, um Ressourcen zu sparen.

Freischaltung und eventbasierte Darstellung der Freischaltungsansicht

Sobald ein interaktions- oder datenbasierter Observer feststellt, dass eine Trephäe oder eine Aufgabe freigeschaltet werden soll, wird dies in der Datenbank persistiert (unter den Mo-

dellen AchievedTrophy und CompletedTask) und der Spieler erhält die damit verbundenen (Erfahrungs-)Punkte. Zuletzt wird die App über dieses Ereignis benachrichtigt, hierbei kommt ein in Kapitel 2 beschriebener eventbasierter Datenfluss zum Einsatz. Durch das Absenden des Events (beinhaltet Informationen über die Freischaltung) aus dem Hintergrundprozess der Freischaltung wird im Vordergrund die Darstellung einer Konfetti-Ansicht mit den entsprechenden Informationen ausgelöst.

Erweiterbarkeit und Änderbarkeit der Errungenschaften: Durch die zuvor konzipierte Separation der Observer konnte deren Implementation hauptsächlich in generalisierten Oberklassen durchgeführt werden, um die konkrete Implementation der jeweiligen Freischaltungslogik der Trophäe oder Aufgabe so kompakt und robust wie möglich zu halten.

Testbarkeit: Um die Implementation der Trophäen und Aufgaben besser testen zu können, wurde im QR-Code-Scanner (als Ausgangspunkt für verschiedene Trophäen und Aufgaben) zusätzlich eine Debug-Ansicht implementiert, mithilfe derer die verfügbaren QR Codes ausgewählt und ein Scannen simuliert werden kann.

4.8 Integration des Crowd-Monitoring-Frameworks

Zusätzlich zur Anbindung des Couchbase-Repository-Frameworks sollte auch die Reintegration der Crowd-Monitoring-Frameworks durchgeführt werden. Hierzu wurden die Komponenten, welche in der bestehenden Applikation zum Crowd-Monitoring-Framework gehörten, vom restlichen Teil der Applikation isoliert. Anschließend wurde das Crowd-Monitoring-Framework zu Testzwecken an die entwickelte UI der neuen Applikation angebunden. Hierzu mussten bestimmte Veränderungen und Bugfixes am Framework vorgenommen werden, damit das Framework kompiliert. Anschließend wurde getestet, inwiefern das Framework beim Aktivieren des Crowd-Monitorings innerhalb der App die entsprechenden Funktionalitäten weiterhin ausführt. Beim Testen konnte keine fehlerhafte Funktionsweise des Frameworks festgestellt werden, jedoch war die Testbarkeit des Frameworks eingeschränkt, da keine (für das Crowd-Monitoring benötigte) iBeacons vorhanden waren und Bluetooth auf dem iOS-Simulator nicht verfügbar ist. Während der Integration des Crowd-Monitoring-Frameworks wurden weitere technische Probleme, ähnlich zu den in Kapitel 1 Beschriebenen, innerhalb des Frameworks identifiziert, beispielsweise der Zugriff auf einen nicht mehr verfügbaren Endpunkt und das Vorhandensein von Google Maps Schnittstellen und GeoJSON-Loadern, die in der neuen App durch die Repositories und MapBox ersetzt und bewusst vom Crowd-Monitoring-Framework separiert wurden. Günstigerweise wurde zu diesem Zeitpunkt eine Forschungsarbeit am Crowd-Monitoring-Framework durchgeführt bzw. sollte durchgeführt werden, so dass dem dazugehörigen Projektteam diese Probleme kommuniziert werden konnten. Die Reimplementation und die Refaktorisierung des Crowd-Monitoring-Frameworks wird in diesem Rahmen außerhalb des Kontextes dieser Arbeit weiter durchgeführt. Um dies zu gewährleisten, wurde das Crowd-Monitoring-Framework in ein externes Pod-Repository (ähnlich zum Couchbase-Framework) ausgelagert und dort zur Weiterbearbeitung zur Verfügung gestellt. Außerdem wurden isolierte klare Schnittstellen vereinbart, über welche die Reintegration des Crowd-Monitoring-Frameworks mit sehr wenig Änderungs- bzw. Erweiterungsaufwand stattfinden kann.

4.9 Dokumentation der Entwickler- und Distributionsleitfäden

Zum Abschluss der Projektbearbeitung wurden außerdem noch ein Entwickler- und ein Distributionsleitfaden erarbeitet. Wesentlicher Bestandteil des Entwicklerleitfadens (Contribution

Guidelines) sind auch die technischen Erläuterungen des Konzeptes dieser Arbeit. Zukünftige Projektteams, die an der OUTPUT.DD-App weiter arbeiten, können über den Entwicklerleitfaden ein klares Verständnis der verwendeten Patterns und Strukturen erlangen, mit dem langfristigen Ziel, die Degradation der Codequalität und -struktur präventiv zu verhindern. Weitere Dokumente, wie die Auflistung aller Trophäen und Aufgaben, sowie derer Kriterien für die Freischaltung, sollen eine Hilfe darstellen, um die Konsistenz zwischen der iOS- und Android-Applikation zu gewährleisten. Im Distributionsleitfaden befinden sich weitere Beschreibungen, wie die CI-Pipeline für eine Distribution in einem iOS-Entwickler-Account genutzt werden kann. Außerdem wurden im GitHub-Repository der iOS-App Issue-Templates angefertigt, um diese zu vereinheitlichen.

5 Zusammenfassung

Im Rahmen einer Projektbearbeitung im Sommersemester 2020 wurde ein Framework für Couchbase erstellt, welches das bestehende Realm-Framework in der OUTPUT.DD App ersetzen sollte. Bei der Erstellung wurden im Rahmen eines Audits zahlreiche, teils strukturelle oder architekturelle, Probleme und Antipatterns in der bestehenden OUTPUT.DD App identifiziert. Die Problemstellung dieser Arbeit war es somit, nicht nur das neue Datenbank-Framework in der OUTPUT.DD App zu integrieren, sondern auch einen allumfassenden Neuaufbau zu planen und durchzuführen. Hierdurch sollten die identifizierten Probleme behoben und verschiedene Strategien entworfen werden, mit denen das Wiedereintreten dieser Probleme verhindert werden soll.

5.1 Vorgehensweise und Ergebnisse

Hierzu wurden verschiedene Software-Patterns beschrieben, sowie deren Anwendungsbereiche und weshalb deren Nutzung einen signifikanten Vorteil für die Separabilität, Funktionalität und Erweiterbarkeit als zentrale Bestandteile der Codequalität bietet. Außerdem wurde die Struktur der bestehenden App anhand einer Dialoglandkarte analysiert und anschließend, mit Hinblick auf die Struktur des neuen Couchbase-Frameworks, in eine neue Top-Level-Architektur überführt, die eine Separation der unterschiedlichen Komponenten vorschlägt. Mithilfe konkreter Konzepte für die Strukturierung von Verzeichnissen anhand der horizontalen und vertikalen Teilung wurde die Neuimplementierung der iOS-App konzipiert und durchgeführt. Hierbei wurde als statisches Code-Analysetool SwiftLint eingeführt, welches zur Vermeidung von Code Smells beitragen soll. Begonnen wurde mit der Neuimplementation des Frontends auf Grundlage einer Modularisierung und Wiederverwendung von UI-Komponenten, sowie der Orientierung an Prinzipien der Usability und der OUTPUT.DD Corporate Identity. Hierbei wurden unter gemeinsamer Absprache bestimmte Teile der App auch optisch modernisiert und durch ein Dark-Mode-Feature ergänzt. Um den redundanten Arbeitsaufwand bei der Übermittlung der iOS-Applikation an App Store Connect zu reduzieren und gleichzeitig die iOS-Applikation über TestFlight testen zu können, wurde eine CI-Pipeline konzipiert, dokumentiert und auf GitHub integriert.

```

philippmatthes@Philips-MacBook-Pro: ~/output/output-ios
~/output/output-ios > master > cloc output-ios
  237 text files.
  220 unique files.
   33 files ignored.

github.com/AlDanial/cloc v 1.74 T=1.45 s (140.3 files/s, 11757.1 lines/s)
-----
          Language      files      blank     comment      code
-----
Swift                  127       1312       2458      8073
JSON                   73        0          0      5041
Python                 1         23         11         97
HTML                   2         2          2         54
C/C++ Header           1         2          18         0
-----
SUM:                  204       1339      2489     13265
-----
```

Abbildung 5.1 Die neu implementierte App umfasst ca. 8000 Zeilen Swift Code.

Für die Integration des Datenbank-Backends wurden konkrete Patterns ausgewählt und verwendet, um gleichzeitig die Persistierung der Datenbankmodelle auf simulierten oder physischen Geräten und eine performante SwiftUI-Preview zu ermöglichen. Hierfür wurden insbesondere das Coordinator-Pattern, das Environment-Pattern, sowie das Proxy-Pattern verwendet. Durch die konsistente Nutzung dieser Architekturpattern konnte die Geschäftslogik der einzelnen Ansichten weiter voneinander separiert werden. Außerdem konnten die genannten Pattern genutzt werden, um zusätzlich zur lokalen Datenbank auch die Synchronisationskomponenten an diese anzubinden. Zum Testen der Datenbank- und Synchronisationsfunktionalitäten wurden konkrete Teststrategien angewandt, darunter die Generierung und Initialisierung von Daten zum Testen der Darstellung innerhalb der UI-Komponenten, sowie die Nutzung eines containerisierten Docker-Test-Setups für das Testen der Couchbase-Replikatoren. Abschließend wurden für die Implementation der Gamification-Errungenschaften konkrete Modelle und Basisalgorithmen entwickelt, mithilfe derer alle Errungenschaften weitestgehend unabhängig und separiert von den beteiligten UI-Komponenten implementiert werden konnten. Für die Fortführung des Projektes wurde diese Dokumentation neben weiteren Leitfäden mit besonders technischem Fokus erstellt, um als wesentlicher Teil des Entwicklerleitfadens zu dienen und zukünftig den Einstieg in die Entwicklung an der OUTPUT.DD App zu vereinfachen und zu vereinheitlichen.

5.2 Offene Punkte

Die Reimplementation der OUTPUT.DD App konnte vollständig durchgeführt werden und beinhaltet neben den bereits vorher existierenden Funktionalitäten, einer technischen, architekturellen und optischen Modernisierung auch allgemeine Verbesserungen in der Usability. Lediglich im Crowd-Monitoring-Framework konnten weitere Probleme identifiziert und kommuniziert werden, welche mithilfe klarer Integrationspunkte im Rahmen einer anderen Forschungsarbeit behoben werden sollen.

Bis zu OUTPUT.DD 2021 am 8. Juli 2021 wird die App weiterhin getestet und ggf. Verbesserungen und Optimierungen umgesetzt, welche sich aus dem Test-Feedback ergeben. Auch die Distribution der Apps und das Deployment des Couchbase-Backend auf einem Produktionssystem für OUTPUT.DD 2021 wird noch im Rahmen des Supportprozesses zukünftiger Teil dieser Arbeit sein.

5.3 Danksagung

Die Teilnahme an dieser Projektarbeit hat mir persönlich sehr viel Spaß gemacht und ich konnte meine Kenntnisse im Bereich Application Development weiter ausbauen. Ich danke meinem Betreuer, Dr. Thomas Springer, für das Vertrauen in meine Kompetenzen, mich mit einem sehr komplexen und vielseitigen Projekt wie diesem zu betrauen. Außerdem möchte ich B.Sc. Felix Kästner danken für die sehr kompetente Unterstützung bei der Implementation und Lösungsfindung auf Android-Seite.