Anpassbarkeit und Erweiterbarkeit von mobilen Software-Anwendungen

Sarah Troschl

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2015

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 28. September 2015

Sarah Troschl

Inhaltsverzeichnis

Kurzfassung

Diese Arbeit beschäftigt sich mit der Umsetzung einer Android-Anwendung, die flexibel angepasst und erweitert werden kann. Sowohl während der Entwicklung als auch nach Release soll die Anwendung sowohl während der Entwicklung als auch bei der Nutzung änderungsfreundlich sein. Zur Umsetzung einer solchen Anwendung werden gängige Software Architektur Muster und Software Design Muster hergezogen. Außerdem müssen Eigenheiten der Plattform berücksichtigt werden und eine passende Entwicklungsumgebung gewählt werden. Eine flexible Modell-Architektur wird dementsprechend vorgestellt und an einem konkreten Projekt getestet, um Stärken, Schwächen und mögliche Verbesserungspunkte des Modells aufzudecken.

Abstract

This work takes a look at the implementation of a dynamically adaptable and extensible Android application. It should be easy to change and maintain while still in development as well as after its release. Different popular software architecture patterns and software design patterns are determined that can contribute to the implementation of such an application. Furthermore the particularity of the Android platform itself has to be considered and a suitable development environment has to be chosen. According to this a flexible architecture model is presented and effectively tested on a concrete project to disclose the model's strengths, weaknesses and potential improvements.

Kapitel 1

Einleitung

1.1 Motivation

Mobile Technologien bieten Software Entwicklern und Entwicklerinnen eine zusehends wachsende neue Plattform an, die sich ständig verändert und weiterentwickelt. Ständig kommen neue Anwendungen für mobile Geräte auf den Markt. Diese müssen flexibel sein und schnell auf neue technologische Möglichkeiten und neue Trends reagieren, um mit der gewaltigen Konkurrenz mitzuhalten. Bessere Hardware und neue Updates für deren Betriebssysteme eröffnen neue Möglichkeiten für neue Applikationen, ebenso wie für bereits Existierende. Um das Interesse der Nutzer und Nutzerinnen auch aufrechtzuerhalten werden viele Anwendungen, die bereits am Markt sind ständig aktualisiert und erweitert um neue Funktionen hinzuzufügen und bereits Existierende zu verbessern.

Auch das Aussehen von Applikationen, deren grafische Darstellung sowie die Erscheinung der Benutzeroberfläche und Interaktion machen ständige Veränderungen durch. Das Design und Interface sollte stets modern und ansprechend sein und sich aktuellen Trends und Standards anpassen. Software-Firmen für verschiedene Plattformen bieten oft User Interface Guidelines an, wie zum Beispiel für Windows [25], Android [24, 53] oder iOS [37]. Nutzer und Nutzerinnen haben oft bestimmte Erwartungen an die Interaktion mit Applikationen und deren Interface und es gibt diverse Usability-Standards, die beachtet werden sollten, um benutzerfreundliche und leicht bedienbare Anwendungen zu erstellen. Für Webseiten gibt es die Standards der International Standards Organisation wie ISO 9241 [40] oder ISO 14915 [39]. Verschiedene Webseiten wie usability.gov [19] oder usabilitynet.org [52] bieten ebenfalls Artikel zu relevanten Standards, Guidelines und Trends zum Thema Usability. Standards und Trends ändern sich, weshalb längere Zeit laufende Anwendungen wie Facebook oder GMail ihr Interface öfters überar-

beiten [48]. Einige Anwendungen wie Line[1], Dashclock[2] oder CMLauncher[3] ermöglichen ihren Nutzern und Nutzerinnen außerdem diese zu personalisieren, um möglichst den Wünschen und dem Geschmack der einzelnen Person zu entsprechen. Möglichkeiten in diese Richtung beinhalten Plugins und Themes, die heruntergeladen werden können um das Aussehen der Applikation zu verändern oder auf zusätzliche Funktionen und Inhalte zuzugreifen. Grafische Komponenten sollten leicht austauschbar sein. Neue Plugins sollten mit der Hauptanwendung stets kompatibel sein und Schnittstellen für solche Erweiterungen bereitstellen. Neue Software in der Form von Plugins muss erkannt und von der Hauptanwendung verwendet werden, wobei auch das Aktualisieren der Anwendung und der Plugins berücksichtigt werden muss, was Versionskontrolle zu einer wichtige Komponente bei der Software Entwicklung macht. Eine offene und flexible Software Architektur kann das Hinzufügen neuer Komponenten erleichtern und helfen zu einem späteren Zeitpunkt mit möglichst geringem Aufwand dynamisch Inhalte und Funktionen zu der existierenden Anwendung hinzuzufügen.

Architekturmuster, welche auch auf diversen anderen Plattformen genutzt werden, können auch für mobile Plattformen verwendet werden um Anwendungen flexibel zu gestalten, jedoch müssen die speziellen Anforderungen der Plattform und der von ihr genutzten Betriebssysteme bei der Entwicklung von mobilen Anwendungen beachtet werden.

1.2 Ziele

Diese Arbeit wird den Einfluss verschiedener Ansätze und Methoden auf die Anpassbarkeit und Erweiterbarkeit von mobilen Software Anwendungen diskutieren. Es werden Architektur- und Softwaremuster identifiziert und wie diese zur Erstellung eines mobilen Software Clients beitragen können, der einfach verändert werden kann und die Möglichkeit bereitstellt, dynamisch durch Inhalte und Funktionen erweitert zu werden. Teile der Anwendung sollen erweitert oder ausgetauscht werden können, ohne Einfluss auf andere Komponenten, die nicht direkt von der Veränderung betroffen sind. Die Absicht der gewählten Architektur ist, die Erweiterung des Clients durch Plugins oder ähnliche Services zu unterstützen und zu ermöglichen, weiteren Inhalt zu dem Client hinzuzufügen, ohne die gesamte Anwendung aktualisieren zu müssen. Die grafische Darstellung, die Logik, sowie die Datenspeicherung sollen unabhängig sein. Eine Anforderung ist die Möglichkeit, grafische Komponenten ohne jeglichen Einfluss auf die Logik der Anwendung auszutauschen.

Geeignete Methoden sollen dazu beitragen, dass die Anwendung leicht

[1] http://line.me/de/
[2] https://github.com/romannurik/dashclock
[3] https://www.cmcm.com/en-us/cm-launcher/

durch den Entwickler oder die Entwicklerin verändert und erweitert werden kann. Es wird erwartet, dass diverse Module unabhängig voneinander modifiziert werden können und neue Inhalte und Funktionen dynamisch zu der Anwendung hinzugefügt werden können. Sowohl das Ändern der Basisapplikation durch Updates als auch das Hinzufügen neuer Module in Form von Plugins in eigenen Applikationen soll von der Architektur unterstützt werden. Es ist beabsichtigt, dass für Nutzer und Nutzerinnen die Möglichkeit besteht neue Module und Funktionen in der Form von Plugins herunterzuladen. Diese sollen in die Basis Applikation integriert werden und aktualisiert werden können ohne die Funktionen der Basis Anwendung zu beeinflussen. Ziel ist es, Methoden zu finden, mit denen sowohl die Erweiterbarkeit der Basis Applikation als auch deren Erweiterung durch Plugins bestmöglich abgedeckt wird.

1.3 Methoden

Um solche Methoden zu bestimmen und zu evaluieren, werden im Zuge dieser Arbeit diverse existierende Architektur und Software Muster als Grundlage dienen. Der Einfluss dieser Muster auf die Anpassbarkeit und Erweiterbarkeit von Software-Anwendungen wird diskutiert werden. Dabei werden sowohl Architektur Design Muster, als auch Interface Design Muster und Software Design Muster berücksichtigt werden. Weiters werden verschiedene Ansätze, die es einer Anwendung ermöglichen Erweiterungen zu unterstützen, diskutiert. Die speziellen Anforderungen der Android Plattform und deren Auswirkung auf die Struktur von Projekten wird ausführlich betrachtet.

Als Basis für eine konkrete Implementierung werden die Vor- und Nachteile verschiedener Implementationsansätze und Methoden diskutiert und verglichen. Dabei werden verschiedene Situationen und Anforderungen betrachtet, welche beachtet werden müssen, um sich für einen Ansatz zu entscheiden. Einige dieser Methoden werden schließlich als Grundlage für ein generelles Architekturmodell für die Android Plattform verwendet. Anhand dessen wird ein konkretes Projekt umgesetzt und getestet. Schlussendlich findet eine Analyse des Ansatzes statt um zu bewerten wie gut und unter welchen Umständen die gewählten Methoden verwendet werden können.

1.4 Aufbau

Kapitel 2 wird sich mit der grundlegenden Thematik sowie einigen Begriffen, die für die Arbeit relevant sind, beschäftigen. Kapitel 3 diskutiert die verschiedenen Möglichkeiten zur Umsetzung einer erweiterbaren Anwendung und welche Architektur für eine gewählte Problemstellung in Frage kommt. Kapitel 4 zeigt eine konkrete Implementation einer solchen Architektur am

Beispiel eines Studienprojekts. In Kapitel 5 wird die umgesetzte Anwendung diskutiert und anhand von Feedback der am Projekt Beteiligten analysiert. Kapitel 6 fasst die Schlussfolgerungen zusammen, die sich aus dieser Arbeit und dem damit verbundenen Projekt ergeben.

Kapitel 2

Hintergründe zu Software-Architektur und Plattform

Es gibt viele verschiedene Muster und Richtlinien für die Entwicklung von Software Applikationen, die entwickelt wurden, um deren Qualität und Organisation zu verbessern. Verschiedene Muster können abhängig von der Situation verwendet werden, um die generelle Struktur des Projekts zu verbessern oder ein bestimmtes Problem zu lösen. Neben der Anwendung von Mustern bezüglich der Architektur der Software gibt es auch Muster zur Handhabung der Benutzer-Interaktion. Bei der Entwicklung von Software und der Verwendung solcher Muster spielt jedoch auch das Betriebssystem, auf dem die Anwendung laufen soll, eine wichtige Rolle. Es wirkt sich darauf aus, welche Möglichkeiten bezüglich Input, Interface Design und User Interaktion zur Verfügung stehen, ebenso wie darauf welche Entwicklungsumgebungen und welche Programmiersprachen zur Verfügung stehen.

2.1 Erweiterbarkeit

Die Erweiterbarkeit ist ein wichtiges Qualitätsmerkmal in der Softwareentwicklung. Einige der Prinzipien und Techniken die zu dieser beitragen sind laut [4]:

- *Interface Segregations-Prinzip*: Es besagt, dass technische Schnittstellen (eng. Interfaces) so aufgeteilt werden sollen, dass sie den Anforderungen verschiedener Clients genügen. Ein Client hat somit nur die Schnittstellen, die er benötigt. Somit haben Änderungen an nicht benötigten Schnittstellen keine Auswirkungen auf den Client. Mit dem Extension Interface Designmuster wird dies dadurch erfüllt, das eine Komponente mehrere Schnittstellen exportiert und dem Client zur

Verfügung stellt. So werden existierende Schnittstellen nicht zerstört, wenn sich im Hintergrund etwas ändert oder erweitert wird [1, 4].

- *Loose Coupling*: Die Kopplung (coupling) bezeichnet die Stärke der Wechselwirkungen zwischen den Teilsystemen eines Entwurfs. Ein Entwurf gilt als gut, wenn innerhalb der Teilsysteme starke Bindung und dazwischen nur schwache Kopplung herrscht. Ziel ist es die künstlichen Abhängigkeiten von unterschiedlichen Komponenten so gut wie möglich zu reduzieren und das System so simpel wie möglich zu halten. Loose Coupling führt zu erhöhter Flexibilität durch Spezialisierung der Teilsysteme und erleichterter Austauschbarkeit mittels Schnittstellen zur Kopplung [4, 35].
- *Liskovsches Substitutionsprinzip*: In Bezug auf die Polymorphie von Klassen ist zu beachten, dass eine Referenz einer abgeleiteten Klasse die Referenz einer Basisklasse ersetzen können soll. Überschriebene Methoden der abgeleiteten Klasse müssen die Anforderungen anderer Klassen an die Basisklasse erfüllen [4].
- *Open-Closed-Prinzip*: Ein Modul soll sowohl offen sein, sodass Erweiterungen möglich sind, als auch geschlossen, sodass es zur Benutzung für andere Module unverändert zur Verfügung steht. Ein für Änderungen geschlossenes Modul kann zum Beispiel durch Subklassen erweitert werden, ohne die Schnittstelle des ursprünglichen Moduls zu beeinflussen [42, 13].
- *Dependency Inversion-Prinzip*: Es handelt sich dabei um die Umkehr von Abhängigkeiten in hierarchischen Systemen. Statt dem direkten Aufruf von Methoden von untergeordneten Elementen, welche zu Abhängigkeiten führen, werden Schnittstellen von höheren Ebenen vorgegeben, die erfüllt werden müssen. Das führt dazu, dass untere Ebenen von diesen abhängig werden und somit die Abhängigkeiten umgekehrt werden [4, 42].
- *Inversion of Control*: Ähnlich dem Dependency Inversion-Prinzip handelt es sich hier um die Umkehr von Abhängigkeiten zwischen gleichberechtigten Komponenten. Die Steuerung des Kontrollflusses wird nach dem Prinzip „Don't call us, we'll call you" an ein Modul abgegeben, zum Beispiel durch Dependency Injection – dabei wird die Referenz auf ein benötigtes Objekt zur Laufzeit von außen mittels eines Injektors übergeben [29, 4].

Die Erweiterbarkeit kann außerdem auch durch die gewählte Architektur beeinflusst werden. Dabei ist jedoch zu beachten, dass Performance und Flexibilität meist zwei verschiedene Zielrichtungen sind, da erhöhte Flexibilität z.B. durch Zwischenschichten erhöht, die Performance aber durch den höheren Rechenaufwand verschlechtert wird [4].

2.2 Architekturmuster

Die Softwarearchitektur eines Systems umfasst laut [4]:

- die statische Zerlegung des Systems in seine physischen Bestandteile (Komponenten) und bei verteilten Systemen die Verteilung dieser Komponenten auf die einzelnen Rechner (Deployment),
- die Beschreibung des dynamischen Zusammenwirkens aller Komponenten und
- die Beschreibung der Strategie für die Architektur, d. h. wie die Architektur in Statik und Dynamik funktionieren soll,

mit dem Ziel, alle nach außen geforderten Leistungen des Systems erzeugen zu können.

2.2.1 Schichtenarchitektur

Das Schichtenarchitekturmuster dient zur Strukturierung einer Anwendung durch die Aufteilung einzelner Funktionalitäten auf Schichten, die in einer linearen Hierarchie angeordnet sind. Dabei dienen die Schichten als Abstraktionen der in ihnen enthaltenen Funktionalitäten. Die Funktionalität einer Schicht darf nur auf Funktionen in derselben Schickt und darunterliegenden Schichten aufbauen. Der niedrigste Abstraktionslevel dient als Basis des Systems, auf dem höhere Schichten aufbauen bis hin zum obersten Grad an Funktionalität. Jede Schicht bietet eine Schnittstelle zur Kommunikation mit angrenzenden Schichten. Dieser Ansatz ermöglicht die getrennte Anpassung einzelner Schichten und deren Austausch unter Berücksichtigung der Schnittstelle. Dabei werden jedoch höhere Schichten bevorzugt, da diese weniger Abhängigkeiten zu anderen Schichten aufweisen. Es wird hierbei zwischen Strict Layering, wobei eine Schicht nur Zugriff auf Funktionen der direkt unterliegenden Schicht erhält und Layer Bridging, wobei auf Funktionen aller unterliegenden Schichten zugegriffen werden kann, unterschieden. Abhängigkeitsbeziehungen zwischen den Schichten können reduziert werden, indem jede Schicht nur auf die nächsttiefere Schicht zugreift und mit dieser eine Art Client-Server Beziehung eingeht. Tiefere Schichten sollen eine Funktionalität für höhere Schichten bereitstellen, auf die diese zugreifen können und ein Ergebnis an diese zurückliefern, umgekehrt dürfen tiefere Schichten jedoch nicht auf höhere Schichten zugreifen [1, 4, 12]. Die Vorteile sind laut [1, 4]:

- Die leichtere Verständlichkeit einzelner Schichten als Abstraktionen bestimmter Funktionalitäten,
- die Möglichkeit zur Wiederverwendbarkeit von Schichten,
- die Schichten sind stabil und können standardisiert werden,

- die Austauschbarkeit von Schichten bei unveränderter Schnittstelle,
- Code-Abhängigkeiten können minimiert werden, dabei sollten Änderungen am Quellcode wenige Ebenen betreffen und lokal gehalten werden,
- nach Festlegung der Schnittstellen können die einzelnen Schichten parallel entwickelt werden,
- die Schichten können gut bottom-up integriert und getestet werden.

Die Nachteile sind laut laut [1, 4]:

- Der Mehraufwand in der Implementierung,
- Strict Layering ist oft zu einschränkend, dies kann aber durch Layer Bridging umgangen werden,
- Effizienzverluste z.B. durch Weiterreichung von Anfragen zwischen Schichten,
- Änderungen können sich über Schichten hinaus auswirken,
- es kann schwierig sein, die richtige Anzahl an Schichten zu finden.

2.2.2 Kanäle und Filter (Pipes and Filters)

Diese Architektur strukturiert eine Anwendung in eine Kette von Prozessen (Filter) die sequentiell verarbeitet werden. Sie eignet sich zur Strukturierung von Systemen, die einen Datenfluss verarbeiten. Der Datenfluss erfolgt über die Reihung der Prozesse – die Ausgabe eines Prozesses dient dem darauffolgenden als Eingabe. Bei diesem System entsteht eine Kette, wobei jeder Prozess das Ergebnis der vorherigen weiterverarbeitet und eine Pipe zur Verbindung zweier Prozesse dient. Daten werden so mittels Pipes zwischen angrenzenden Filtern weitergereicht. Diese Architektur ist nur für Systeme geeignet, die sich in einzelne Verarbeitungsschritte einteilen lassen können. Ein Vorteil dieser Architektur ist ihre Flexibilität bezüglich des Austauschs der Filter innerhalb einer Pipeline, jedoch ist dies stark vom Datenformat, das zum Austausch verwendet wird, abhängig. Durch die Rekombination von Filtern können Familien von verwandten Systemen gebaut werden [1, 4]. Die Vorteile sind laut [1, 4]:

- Flexibilität gegenüber Änderungen oder Erweiterungen,
- Flexibilität von Filteraustausch und Neukombinierung,
- Vorteil bei Rapid Prototyping von Pipelines,
- nicht benachbarte Verarbeitungsstufen teilen keine Information und sind daher entkoppelt,
- das Speichern von Zwischenergebnissen ist nicht nötig aber möglich,
- Effizienz durch Parallel Processing – Aufgaben der Filter können bis zu einem gewissen Grad parallel abgearbeitet werden,
- die Wiederverwendbarkeit von Filtern in anderen Filterketten.

Die Nachteile sind laut [1, 4]:

- Schwierigkeiten bei der Fehlerbehandlung, da im System kein gemeinsamer Zustand existiert,
- der langsamste Filter in der Kette bestimmt die Arbeitsgeschwindigkeit – keine vollständige Parallelisierung, da Filter aufeinander warten müssen,
- Overhead durch Datenkonvertierungen – Format Spezifizierung der Datenkanäle,
- Weitergabe von Statusinformation ist aufwändig / unflexibel.

2.2.3 Blackboard Muster

Das Blackboard Muster wird für Probleme genutzt, für die es keine deterministische Lösung gibt. Die Architektur besteht aus mehreren spezialisierten Untersystemen, die ihr Wissen kombinieren um eine eventuell nur teilweise oder annähernde Lösung zu finden. Es beruht auf der Idee, dass mehrerer unabhängige Programme kooperativ an einer gemeinsamen Datenstruktur arbeiten [1].

2.2.4 Komponentenbasierte Architektur

Bei diesem Ansatz wird die Architektur aus unabhängigen Komponenten zusammengesetzt. Eine Definition für eine solche Komponente gibt [15, S. 589]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Komponenten werden durch Komposition zu Entitäten zusammengefügt und kommunizieren über Schnittstellen. Dabei wissen sie nichts von anderen Komponenten und sind von diesen unabhängig. Dies ermöglicht flexible und anpassungsfähige Systeme deren Komponenten jederzeit ausgetauscht werden können. Komponenten können beliebig hinzugefügt und entfernt werden, was das Erweitern von Funktionalität ermöglicht. Dieser Ansatz führt zu sehr geringer Kopplung der einzelnen Funktionalitäten [12, 17]. Die Vorteile sind laut [12, 49, 54, 17]:

- Die Verständlichkeit der Architektur,
- Wartbarkeit durch bessere Strukturierung,
- separate Testbarkeit von Teilen des Systems,
- Komponenten können unabhängig voneinander entwickelt werden,
- Komponenten können mehrfach eingesetzt, kombiniert und wiederverwendet werden,

- solange die Schnittstelle der Komponente sich nicht ändert, kann die dahinterliegende Implementierung dieser Komponente dynamisch ausgetauscht werden.

Die Nachteile sind laut [12, 54, 17]:

- Die komplexe Implementierung,
- eine Architektur zur Verwaltung und Kommunikation kann Overhead mit sich bringen.

Einsatzgebiete: Eine Komponentenarchitektur ist sinnvoll, wenn bereits Komponenten vorhanden sind oder von Drittanbietern erworben werden können oder Komponenten die in verschiedenen Programmiersprachen verfasst sind kombiniert werden sollen. Auch bei Anwendungen mit prozeduralen Funktionen und wenig oder keinem Dateninput kann eine solche Architektur gut umgesetzt werden. Außerdem sind Komponenten geeignet um eine Kompositions- oder Plugin-Architektur umzusetzen, bei der einzelne Teile leicht erweitert oder ersetzt werden sollen [51].

2.2.5 Serviceorientierte Architektur

Die Serviceorientierte Architektur (SOA) beschreibt ein System von voneinander unabhängigen Services, die miteinander kommunizieren können. Dieser Ansatz ist dem der komponentenbasierten Architektur sehr ähnlich. Ein Service ist in sich geschlossen und hat stets dieselbe eine Funktionalität, unabhängig von anderen Services. Ein Service wird definiert und dem System zur Verfügung gestellt. Dieses erkennt Services, die es nutzen kann und interagiert dann mit diesen. Dieser Ansatz findet vor allem bei Webservices Anwendung, die von Applikationen aufgerufen werden können. Dadurch, dass Services unabhängige Funktionalitäten bieten, funktionieren sie stets in jedem System. Sie sind in keinem Fall von anderen Services abhängig. Eine solche Anwendung kann jederzeit durch neue Services erweitert werden [9, 16]. In der Serviceorientierten Architektur werden Komponenten oder Teilkomponenten, als Services zur Verfügung gestellt, wobei diese für gewöhnlich Anwendungsfällen oder Teilen von Anwendungsfällen aus Sicht der Systemanalyse entsprechen sollen. Das System soll bei Änderungen von Geschäftsprozessen so flexibel bleiben. Dadurch entsteht eine Kapselung in Form von Diensten und eine Rollenverteilung in Form von Serviceanbieter und Servicenutzer [4]. Die Vorteile sind laut [4]:

- Übersicht über alle benötigten Dienste und Schnittstellen,
- die Komplexität von verteilten Systemen wird reduziert durch Aufteilung in Komponenten von Services bzw. elementaren Services,
- Services können mehrfach eingesetzt und wiederverwendet werden,
- solange die Schnittstelle des Services sich nicht ändert, kann die dahinterliegende Implementierung dieses Services dynamisch ausgetauscht

werden.

Die Nachteile sind laut [4]:

- Eine zu feine Granularität der Services erzeugt komplexe Strukturen,
- die üblicherweise eingesetzten Protokolle stellen hohe Anforderungen an den Durchsatz der Netzwerkverbindungen,
- Geschäftsprozesse müssen klar definiert und dokumentiert sein.

Einsatzgebiete: Diese Architektur ist besonders für die Umsetzung von Client-Server Systemen sinnvoll und zur Vereinfachung komplexer Systeme. Bestehende Systeme können gekapselt und plattformunabhängig genutzt werden. Außerdem können Kunden oder Lieferanten durch SOA in die Geschäftsprozesse eingebunden werden [4].

Zur Verwendung von Services müssen laut [9, 16] folgende Schritte erfolgen:

- Bereitstellung des Services,
- Erkennen des Services durch die Anwendung,
- Verwendung des Services durch die Anwendung.

2.2.6 Plugin Entwurfsmuster

Dieses Muster dient zur Umsetzung von Erweiterungen einer Anwendung. Dabei wird eine Plugin Schnittstelle definiert, die zur Verfügung stellt, mit welchen Teilen der Anwendung die Plugins kommunizieren können. Die Schnittstellen dienen als Erweiterungspunkte der Software und können von Plugins implementiert werden. Die Anwendung ist in diesem Fall unabhängig und funktioniert allein. Ein Plugin kann jedoch von der Basis-Anwendung und der von ihr definierten Schnittstelle abhängig sein und muss nicht allein funktionieren. Es dient dazu, der Basisanwendung zusätzliche Features oder Funktionalitäten bereitzustellen. Es kann auch einer dritten Partei die Möglichkeit bereitgestellt werden, die Software zu erweitern, ohne deren Quellen zu kennen. Ein Plugin-Manager dient dazu, Plugins, die Schnittstellen des Clients implementieren, zu instanziieren und einzuhängen. Die Anwendung dieses Musters dient zur Umsetzung von Systemen, die im laufenden Betrieb flexibel erweiterbar sein sollen [4]. Die Vorteile sind laut [4]:

- Jedes Plugin besitzt seine eigene Zuständigkeit,
- Plugin-Architekturen zeigen in der Regel robustes Verhalten,
- im Gegensatz zu Bibliotheken und Frameworks kann eine dritten Partei unabhängig Zusatzfunktionalität für eine vorhandene Software entwickeln, ohne Kenntnis über deren Programmcode zu haben,
- durch den modularen Aufbau eines Systems kann dieses „schlank“ gehalten werden, indem nur wirklich genutzte Funktionen geladen werden,

- der Wartungsaufwand sinkt, da in Plugins ausgelagerte Funktionslogik für einen Anwendungsfall angepasst und als neue Version eines Plugins ausgeliefert werden kann,
- die Entwicklung komplexer Software kann bequem aufgeteilt werden,
- Plugins können in mehreren verschiedenen Versionen vorhanden sein und auch gleichzeitig ausgeführt werden,
- jedes Plugin kann getrennt von den anderen Plugins getestet werden.

Die Nachteile sind laut [4]:

- Der initiale Aufwand bei der Implementierung einer Anwendung ist höher, da die Architektur Schnittstellen bereitstellen muss, welche von den Plugins implementiert werden,
- der Verwaltungsaufwand während der Ausführung einer speziellen Anwendung steigt,
- für Erweiterungen muss eine gemeinsame Schnittstelle gefunden werden und Plugins haben zu dieser Schnittstelle eine Abhängigkeit.

Einsatzgebiete: Diese Architektur ist sinnvoll, wenn es mehrere Nutzer und Nutzerinnen mit verschiedenen Anforderungen gibt. Auf diese Weise kann jeder Nutzer und jede Nutzerin nur die für ihn oder sie nötigen Funktionen erhalten. Außerdem kann die Architektur sinnvoll für die Erweiterung einer Anwendung sein, falls diese nicht alle Anforderungen abdeckt die benötigt werden. Durch die Verwendung von Schnittstellen kann eine Anwendung sogar vom Anwender oder der Anwenderin selbst um Funktionalität erweitert werden. Dabei kann die Art der Erweiterung durch die Schnittstelle eingeschränkt werden. Ein Beispiel für ein Plugin System sind Webbrowser, bei denen Plugins zur Wiedergabe von medialen Inhalten benutzt werden [4].

2.2.7 Microkernel Muster

Dieses Muster ähnelt dem Plugin Muster – der Microkernel übernimmt die Kommunikation zwischen Komponenten eines Systems und bietet den Komponenten dazu ein Interface. Er verwaltet zusätzlich jedoch auch Ressourcen des Systems und steuert, welche Komponenten Zugriff erhalten. Das Muster bezieht sich jedoch hauptsächlich auf die Entwicklung von Betriebssystemen [4].

2.2.8 DAO Entwurfsmuster

Dieses Muster verwendet sogenannte Data Access Objects um den Zugriff auf die Daten einer Applikation zu erledigen. Ein eigener Layer trennt die konkrete Implementation der Datenbank einer Anwendung von dem Modell in dem diese gespeichert werden. So muss das Modell der Anwendung nichts

von der Umsetzung der Datenspeicherung wissen. Das DAO Layer dient dazu Daten aus dem Modell in die Datenbank zu speichern und Daten aus dieser zu lesen. Diese gibt es an das Datenmodell der Anwendung weiter. Um die konkrete Implementation der Datenspeicherung austauschbar zu gestalten, kann eine Schnittstelle für den DAO Layer zur Verfügung gestellt werden. Das Datenmodell selbst bleibt von der Änderung unbeeinflusst [14].

2.3 Architekturmuster für interaktive Systeme

Diverse Muster zur Strukturierung von interaktiven Systemen sind sowohl bei der Umsetzung von Desktop-Anwendungen, als auch bei Webanwendungen sehr weit verbreitet. Auch auf mobilen Plattformen können solche Muster als Grundlage für die strukturierte Umsetzung der Benutzerinteraktion hilfreich sein. Einige der populärsten Muster, sowie die Umsetzung mittels von diesen vorgeschlagenen Komponenten, werden folgend vorgestellt [14].

2.3.1 Model-View-Controller (MVC)

Dieses Muster für interaktive Systeme trennt eine Anwendung in Model, View und Controller auf (siehe Abb. 2.1). Das Model dient zur Verwaltung und Verarbeitung der Daten und enthält Grundfunktionalitäten der Anwendung. Die View dient zur Darstellung und Eingabe. Es können mehrere Views zu einem Model existieren, wobei jede View eine zugehörige Controller-Komponente hat. Der Controller übernimmt die Logik, verarbeitet die Eingaben des Nutzers oder der Nutzerin und gibt diese aus. Das Model ist nicht von der Ein- und Ausgabe abhängig und somit sind View und Controller leicht austauschbar, ohne die Datenstruktur zu beeinflussen. Dadurch wird eine Trennung der Oberflächenkomponenten von der Verarbeitung erzielt. Benutzerschnittstellen in Systemen werden oft geändert und sollen leicht austauschbar sein, eventuell sogar zur Laufzeit, ohne das restliche Programm zu ändern. View und Controller bilden zusammen das User Interface der Anwendung. Die beiden Komponenten sind abhängig vom Model und kennen den Aufbau der Daten. Die View nutzt Informationen über Daten des Models um diese darzustellen, während der Controller Zugriff auf die Daten des Models benötigt um diese ändern zu können. Außerdem kommunizieren Controller und View miteinander, da die View den Controller über Interface Eingaben benachrichtigen und der Controller den Zustand der View bestimmt und deren Ansicht ändern kann. Wenn das Model vom Controller einer View geändert wird, so sollen alle Views, die von diesen Daten abhängig sind, die Änderungen übernehmen. Daher werden diese vom Model benachrichtigt, wenn dessen Daten sich ändern. Die Views fragen dann die neuen Daten beim Model ab und aktualisieren die dargestellten Informationen [1, 4]. Um Abhängigkeiten zwischen View und Model zu reduzieren, kann eine abgewandelte Form des MVC Musters – das Passive

Model MVC Muster – genutzt werden. Bei diesem Ansatz erfolgt die Kommunikation zwischen Model und View ausschließlich über den Controller. Das Model benachrichtigt den Controller bei Änderungen. Dieser wiederum benachrichtigt die View, die schließlich die Änderungen an der Ausgabe umsetzt [14]. Die Vorteile sind laut [1, 4]:

- Model-Klassen können unabhängig von der Gestaltung der Benutzeroberfläche entworfen werden,
- die Benutzeroberfläche kann geändert werden, ohne eine Veränderung an den Model-Klassen vornehmen zu müssen,
- das Model kann unabhängig von der Oberfläche getestet werden,
- es können verschiedene Benutzeroberflächen für dieselbe Anwendung entworfen werden (Austauschbarkeit von „Look and Feel“),
- verschiedene Views für ein Model,
- synchronisierte Views,
- Hinzufügen von Views und Controllern,
- Framework Potential.

Die Nachteile sind laut [1, 4]:

- Häufige Aktualisierungen können die Performance verschlechtern, besonders wenn mehrere Aufrufe der View nötig sind, um Daten vom Model zu erhalten,
- der erhöhter Implementierungsaufwand durch die Unterteilung in viele Klassen ist bei kleineren Anwendungen nicht gerechtfertigt,
- erhöhte Komplexität,
- potentiell exzessive Anzahl von Updates,
- enge Verbindung zwischen View und Controller,
- Close Coupling von Views und Controllern zu einem Model,
- Ineffizienz des Datenzugriffs in der View,
- Unvermeidbarkeit von Änderungen an View und Controller bei der Portierung (keine Trennung von plattformabhängigem Code),
- Schwierigkeit beim Nutzen von MVC mit modernen User-Interface Tools.

2.3.2 Model-View-Presenter (MVP)

Das Model-View-Presenter Muster ist eine Variation des bereits erwähnten Model-View-Controller Musters, die auf moderne ereignisgesteuerte Systeme abgestimmt ist. Das System besteht aus drei Komponenten: Model, View und Presenter. Die View ist die visuelle Repräsentation der Daten im Model in Form des gesamten Bildschirms und den verwendeten Widgets und behandelt Events von Interface Aktionen. Der Presenter ist für die Präsentationslogik verantwortlich und die Interaktion mit dem Model. Das Model

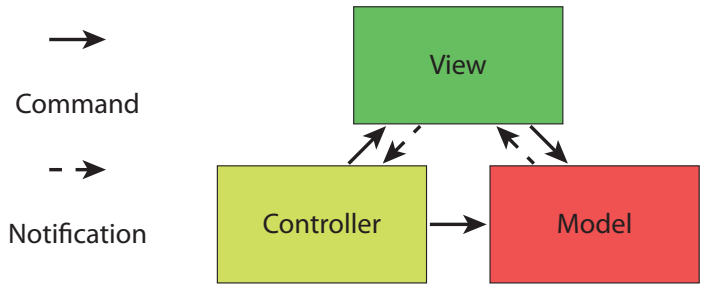


Abbildung 2.1: Illustration des Model View Controller Musters.

ist ein Domain Model und beinhaltet die Daten und Business-Funktionalität der Anwendung. Wie bei dem MVC Muster, wird auch hier die View benachrichtigt, wenn Änderungen am Model erfolgen und reagiert indem es die relevanten Teile des Bildschirms aktualisiert. Es gibt außerdem spezielle Abwandlungen des MVP Musters, Supervising Controller und Passive View. Diese beiden Muster beschäftigen sich mit der Präsentationslogik unabhängig von spezifischer Domain-Logik. Obwohl sie in diesem Fall im Kontext des Model-View-Presenter Musters vorgestellt werden, sind sie nicht konkret nur in Zusammenhang mit diesem einsetzbar. Die Supervising Controller Version nutzt das Observer Muster zur Kommunikation zwischen Model und View (siehe Abb. 2.2). Die View kann direkt mit dem Model interagieren um Daten zu speichern, wenn keine Änderung am Bildschirm erfolgen soll. Ansonsten erfolgt die Kommunikation zwischen Model und View ausschließlich über den Presenter. Die Anliegen der Präsentation und der Präsentationslogik werden aufgeteilt, wobei die View simple Präsentationslogik übernimmt, während der Presenter (oder Controller) die Zuständigkeit auf Benutzerinput zu interpretieren und komplexe Präsentationslogik zu handhaben übernimmt. Views delegieren User Events an den Presenter, der wiederum mit der Business-Domain der Anwendung interagiert. Für simple Präsentationslogik nutzt die View Data Binding Techniken und das Observer Muster um sich selbst zu updaten, wenn Änderungen in der Anwendung auftreten. Im Fall der Passive View Version erfolgt die Interaktion zwischen View und Model ausschließlich über den Presenter (siehe Abb. 2.3). Die Anliegen der Präsentation und der Präsentationslogik werden aufgeteilt, wobei der Presenter (oder Controller) die Zuständigkeit auf Benutzerinput zu reagieren und Präsentationslogik zu handhaben übernimmt. Die View beinhaltet die visuellen Komponenten der Anwendung, wie Bildschirme und Widgets. Views delegieren User Events an den Presenter, der wiederum mit der Business-Domain der Anwendung interagiert und/oder die View aktualisiert. Die View hat keinerlei Verbindung zum Model und ist für mit der Präsentation verbunde-

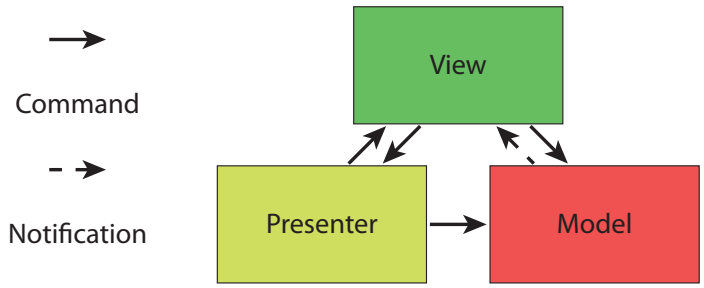


Abbildung 2.2: Illustration des Model View Presenter Musters (Supervising Controller).

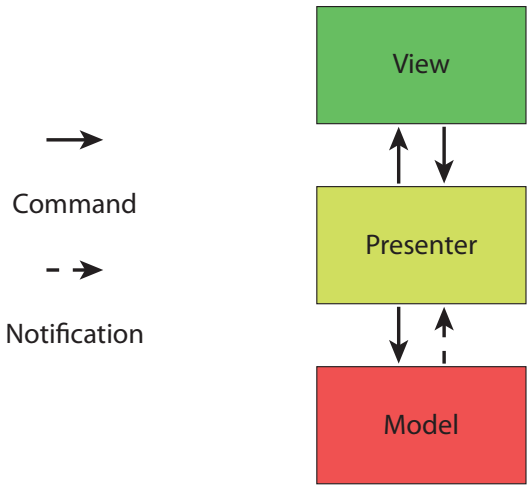


Abbildung 2.3: Illustration des Model View Presenter Musters (Passive View).

ne Logik vollständig auf den Presenter angewiesen. Presenter übernehmen in diesem Fall eine vermittelnde Rolle zwischen den Views und der Domain Logik Strategie [36, 14, 55].

2.3.3 Model-View-ViewModel (MVVM)

Model-View-ViewModel ist ein weiteres Muster zur Trennung von Präsentation und Business Logik (siehe Abb. 2.4). Das ViewModel ist die verbindende Komponente zwischen View und Model. Dies wird durch Binding zwischen Komponenten realisiert, was nicht in allen Entwicklungsumgebungen stan-

dardmäßig unterstützt wird. Dieses Design Pattern wird unter anderem in Microsoft Systemen und modernen Web Frameworks eingesetzt. Es ist eine spezielle Abwandlung des MVC für UI Entwicklungsplattformen wie Silverlight und hat seinen Ursprung in .Net von Microsoft. Das Model in MVVM sollte keine View-spezifische Logik enthalten, sondern nur die Logik, die notwendig ist, um die minimale API dem ViewModel bereitzustellen. Jegliche Event Logik und die Delegation von Informationen zwischen View und Model wird von dem ViewModel übernommen. Das Model verwaltet Daten, repräsentiert das Domain Model sowie die Kern Business Logik, während die View das Benutzer-Interface repräsentiert mit dem Nutzer und Nutzerinnen interagieren. Das ViewModel erhält Referenzen von der View und nutzt diese, um die UI zu aktualisieren. Es stellt die Verbindung zwischen Model und View dar und stellt der View Daten zur Darstellung zur Verfügung. Außerdem bietet es Befehle an, die die View nutzen kann um Events an das ViewModel zu kommunizieren. Die View ist dabei direkt via Binding mit dem ViewModel verbunden, wodurch Änderungen in der View automatisch im ViewModel reflektiert werden und umgekehrt. Einer der Vorteile der Umsetzung des MVVM Musters ist die klare Trennung vom Verhalten der Anwendung und der Präsentation der Anwendung. Dies führt zu klarerem und wartbarerem Code. Außerdem ist es möglich das ViewModel über mehrere Plattformen hinweg zu testen und wiederzuverwenden, da das Verhalten und der Status der Anwendung von dessen Repräsentation durch die View isoliert sind. Dadurch kann viel Zeit und Aufwand bei der Entwicklung für mehrere Plattformen gespart werden. Views müssen zwar angepasst werden, doch das Model und das generelle Verhalten bleiben oft über mehrere Plattformen hinweg konsistent. Durch das Schreiben von Unit Tests kann auch das Testen der Anwendung erleichtert werden. Der Ansatz des MVVM kann in Situationen verwendet werden, in denen Data Binding möglich ist. Beispiele dafür sind WPF und Javascript Projekte die Knockout nutzen. In Android muss ein Framework, wie zum Beispiel MvvmCross, genutzt werden, um dieses Muster umzusetzen [10, 50, 14, 55].

2.3.4 Presentation-Abstraction-Control (PAC)

Das Presentation-Abstraction-Control Muster definiert die Struktur für interaktive Software Systeme als eine Hierarchie von kooperierenden Agenten (siehe Abb. 2.5). Dabei ist jeder Agent für einen bestimmten Aspekt der Funktionalität der Anwendung zuständig. Ein Agent besteht aus drei Komponenten: Präsentation, Abstraktion und Kontrolle. Diese Unterteilung trennt Aspekte der Human Computer Interaction des Agenten von seinem Funktionskern und seiner Kommunikation mit anderen Agenten. Die Anwendung selbst ist als baumartige Hierarchie solcher PAC Agenten strukturiert. Die Präsentationskomponenten stellen das sichtbare Verhalten des PAC Clients dar. Die Abstraktionskomponenten verwalten das Datenmodell das dem

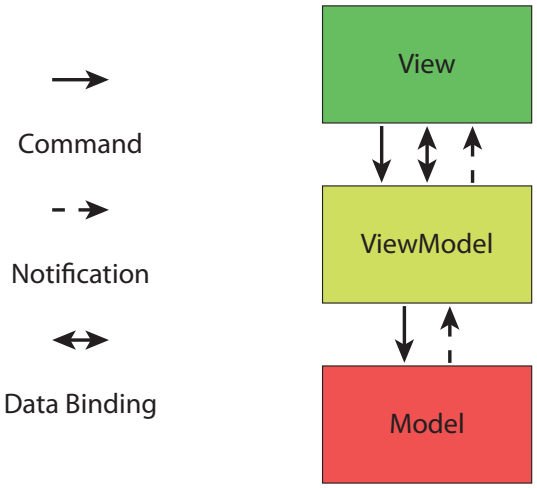


Abbildung 2.4: Illustration des Model View ViewModel Musters.

Client zugrunde liegt und bieten Funktionalität, die mit den Daten arbeitet. Die Kontrollkomponenten verbinden die Präsentations- und die Abstraktionskomponenten und bieten Funktionalität die es dem Agenten ermöglich mit anderen PAC Agenten zu kommunizieren [1]. Die Vorteile sind laut [1]:

- Separation of Concerns,
- Unterstützung für Änderung und Erweiterung – Änderungen an Komponenten von PAC Agenten beeinflussen keine anderen Agenten, neue Agenten sind einfach zu integrieren,
- Unterstützung von Multitasking.

Die Nachteile sind laut [1]:

- Erhöhte Systemkomplexität,
- komplexe Kontrollkomponenten,
- Effizienz – Kommunikations-Overhead,
- Anwendbarkeit – je kleiner die atomaren semantischen Konzepte einer Anwendung und je größer die Ähnlichkeit der User Interfaces, umso weniger anwendbar ist dieses Muster, da dies sonst zu einer feinkörnigen Struktur führt, die schwer zu verwalten ist.

2.4 Plattform Anforderungen

Bei der Entwicklung einer Anwendung, ist die Zielplattform ein wichtiger Einflussfaktor. Je nach Plattform stehen bestimmte Entwicklungsumgebun-

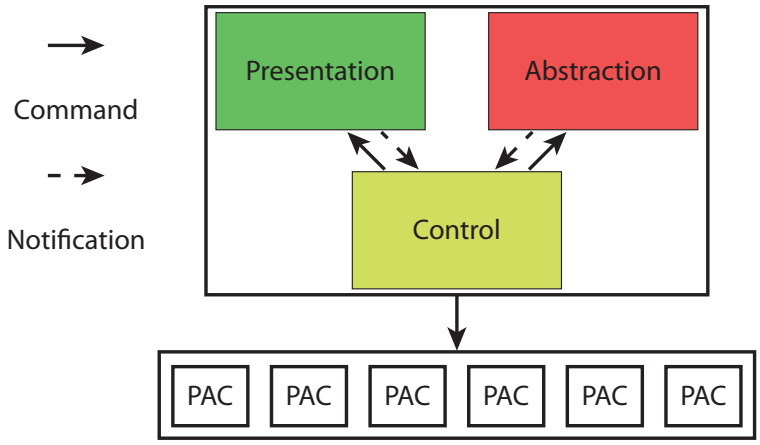


Abbildung 2.5: Illustration des Presentation Abstraction Control Musters.

gen und Frameworks zur Verfügung. Soll in der Standard Entwicklungsumgebung programmiert werden, kann die Programmiersprache, ebenso wie Teile der Projektarchitektur bereits vorgegeben sein.

2.4.1 Android Projekt Architektur

Auf offiziellen Webseite des Android Operating Systems auf [21] wird die Plattform wie folgt angeworben:

> Android is the customizable, easy to use operating system that powers more than a billion devices across the globe—from phones and tablets to watches, TV, cars and more to come.

Neben Infos zur aktuellen API und diversen Design Guidelines für Android gibt es auch einen Developer Guide, mit Informationen zur Entwicklung von Software für die Android Plattform. Zur Entwicklung wird die Installation von Android Studio empfohlen. Es kann jedoch auch mit Eclipse für Android entwickelt werden wenn das Android Developer Tools (ADT) Plugin installiert wurde. Neben der Android Studio IDE werden die Android SDK Tools benötigt. Android Studio ist die offizielle IDE zur Android Entwicklung und basiert auf IntelliJ IDEA. Es bietet ein flexibles auf Gradle basiertes Build-System, Code Templates, einen Layout Editor und viele weitere Features. Um in Android Studio für Android zu entwickeln, sollte die Struktur des Projekts beachtet werden. Neben Source Code Dateien benötigt jedes Projekt eine Manifest XML Datei, in dem allgemeine Informationen

zu der Applikation und ihrer Struktur verwaltet werden. Außerdem werden Build Dateien, Ressourcen und Layout XML Files benötigt. Weiters werden einige Tools zur Verfügung gestellt die beim Debugging der Anwendung von Nutzen sein können [23]. Für die Entwicklung stehen in Android laut [23, 2, 31, 7] folgende Komponenten zur Verfügung:

- *Activity* – Sie repräsentiert einen einzelnen Bildschirm einer Anwendung sowie dessen User Interface. Einzelne Activities einer Anwendung sind unabhängig. Sie können von anderen Komponenten gestartet werden. Sie können beendet werden. Es kann immer nur eine Activity im Vordergrund laufen.
- *Service* – Er läuft im Hintergrund um lang-andauernde Operationen oder Arbeit für Remote Prozesse durchzuführen und bietet kein Interface. Eine andere Komponente kann eine Service starten und stoppen, ihn laufen lassen oder sich an diesen binden um mit ihm zu interagieren. Intent Services stoppen automatisch wenn sie ihre Aufgaben erledigt haben. Services laufen standardmäßig auf demselben Thread wie die Activity.
- *Content Provider* – Er verwaltet ein gemeinsam genutztes Set an App Daten. Mit den korrekten Permissions können andere Anwendungen auf die Daten zugreifen oder diese ändern.
- *Broadcast Receiver* – Er reagiert auf System-weite Broadcasts, welche vom System selbst oder von Anwendungen ausgesendet werden können. Er bietet kein Interface, kann jedoch Statusleisten-Notifikationen erzeugen.

Komponenten müssen im Manifest der Anwendung deklariert werden, um von dieser genutzt werden zu können. Das Manifest dient außerdem unter anderem dazu Permissions, ebenso wie Software und Hardware Features, die von der Anwendung benötigt werden, zu verwalten. Das minimal geforderte API Level muss ebenfalls im Manifest angegeben werden [23]. Weiters bietet Android diverse Mechanismen zur Interaktion von Komponenten an. Ein wichtiger Bestandteil der Entwicklung sind Intents. Intents sind asynchrone Nachrichten, die individuelle Komponenten zur Laufzeit aneinander binden, auch wenn diese nicht zur selben Applikation gehören. Neben Intents können Broadcasts genutzt werden um Daten und Informationen über das ganze System auszuschicken [23, 31, 7].

Inter Process Communication in Android: Zur Inter Process Communication (IPC) in Android eignen sich laut [23, 31, 7] folgende Methoden:

- Intents,
- Broadcasts,
- Messenger,
- AIDL,

- Meta-data.

Intents können zur Kommunikation mit Activities und Services genutzt werden. Explizite Intents können eine Komponente mittels deren vollen qualifizierten Klassennamen starten. So können Activities direkt gestartet werden, wenn deren Name der Anwendung bekannt ist. Implizite Intents hingegen rufen eine Komponente nicht über den Namen, sondern über spezielle Eigenschaften auf. Durch diese Umsetzung entsteht Loose Coupling zwischen den Komponenten, da eine Anwendung nicht den Empfänger des Intents kennt, sondern lediglich eine bestimmte „action“ spezifiziert, die dieser erfüllen soll. Wenn mehr als eine Komponente gestartet werden könnte, wird entweder eine gespeicherte Standardanwendung gestartet oder der Nutzer oder die Nutzerin kann aus der Liste der verfügbaren Anwendungen auswählen. Services können über Intents nicht nur gestartet sondern auch wieder gestoppt werden [23, 31, 7, 44]. Eine Kommunikationsmöglichkeit, die sich auch für IPC eignet ist die Nutzung von Broadcasts und Broadcast Receivern. Eine Komponente kann einen Intent via Broadcast versenden, während ein Broadcast Receiver auf diesen hört und ihn empfängt. Dazu können Intent Filter genutzt werden, die einschränken, welche Intents der Broadcast Receiver empfangen soll. Eine solche Kommunikation kann jedoch Security Probleme mit sich ziehen, da Informationen über das gesamte Gerät verschickt werden, und andere Anwendungen diese empfangen könnten, auch wenn sie nicht für diese gedacht sind. Sollten Broadcasts nur innerhalb einer Anwendung verschickt und empfangen werden, kann ein lokaler Broadcast versendet werden, den nur lokale Broadcastreceiver erhalten können [23, 31, 7, 44]. Wenn statische Daten mit anderen Anwendungen geteilt werden sollen, eignet sich dafür die Umsetzung eines Content Providers. Ein Content Provider verwaltet den Zugriff auf Daten in einem separaten Prozess. Daten werden in den Provider gespeichert, sodass aus anderen Prozessen auf diese zugegriffen werden kann. Es ist aufwändig eigene Content Provider korrekt und sicher zu implementieren, jedoch die beste Möglichkeit, Daten zu speichern die von mehreren Anwendungen gemeinsam genutzt werden. Permissions sollten beachtet werden, damit nur Anwendungen mit den korrekten Berechtigungen Zugriff auf die gespeicherten Daten haben [23, 31, 44]. Auch die Meta-Data im Manifest einer Anwendung kann von anderen Anwendungen ausgelesen werden, um Informationen zu erhalten. Dies wird hauptsächlich für Grundinformationen wie etwa der Anwendungsversion genutzt [23, 44]. Wenn eine andere Komponente mit einem Service kommunizieren möchte, besteht auch die Möglichkeit mittels des von Android bereitgestellten Binder Systems an den Service zu binden. Bei einer rein lokalen Kommunikation kann mittels einer Implementation der Binder Klasse durch den Client direkt auf die public Methoden des Services zugegriffen werden. Wenn die Kommunikation jedoch auch für Remote Prozesse möglich sein soll, muss die Kommunikation entweder über einen Messenger,

oder über eine AIDL-Schnittstelle (Android Interface Definition Language) erfolgen. Bei der Umsetzung eines Messengers werden die Nachrichten der Reihe nach abgearbeitet, Multi-Threading wird nicht unterstützt, dadurch ist die Kommunikation Thread-sicher, bei mehreren synchronen Anfragen kann es jedoch zu Verzögerungen kommen. AIDL dient als Basis für den Messenger, kann jedoch auch direkt von Entwicklern und Entwicklerinnen verwendet werden. Die Implementation muss Thread-sicher umgesetzt sein, was mehr Entwicklungsaufwand bedeutet, dafür wird Multi-Threading unterstützt und Anfragen können blockieren und gleichzeitig abgearbeitet werden. Wenn Multi-Threading nötig ist, sollte mittels AIDL gearbeitet werden, insbesondere wenn mehrere Services gleichzeitig mit einem Client interagieren sollen. Ansonsten ist ein Messenger die bevorzugte Lösung für IPC [22, 23, 30, 31, 7, 46].

Kapitel 3

Theoretische Architektur

Dieses Kapitel beschäftigt sich mit der Strukturierung einer Architektur, die die gestellten Anforderungen erfüllt – sowohl für Änderungen anpassbar als auch für Erweiterungen offen zu sein. Während der Entwicklung soll diese Architektur es ermöglichen, die Anwendung möglichst einfach anzupassen. Auch nach dem Release der Anwendung sollen Änderungen und Erweiterungen möglichst einfach implementiert werden können. Es soll dabei sowohl neuer oder aktualisierter Content in Form von Ressourcen beinhaltet sein, als auch das Ändern, Austauschen oder Hinzufügen von neuem User Interface. Auch sollen neue Features entsprechend eingefügt und existierende Features bei Änderungen leicht aktualisiert werden können. Die Architektur sollte neben der benutzerfreundlichen Aktualisierung der Anwendung auch deren flexible Erweiterung und Personalisierung durch den Nutzer oder die Nutzerin unterstützen.

3.1 Mögliche Methoden und Techniken

Die gewählte Architektur basiert auf einigen der zuvor erklärten Muster und ist auf die Android Plattform zugeschnitten. Je nach Anforderungen sollten gewisse Anpassungen an der Architektur gemacht werden. Die Vor- und Nachteile der verschiedenen Möglichkeiten zum Aufbau einer solchen Architektur werden unter diesem Gesichtspunkt diskutiert.

3.1.1 Entwicklungsumgebung

Zur Umsetzung einer Android Anwendung gibt es verschiedene Möglichkeiten. Eine davon ist die Umsetzung einer nativen Anwendung mit von der Plattform empfohlenen Entwicklungsumgebungen wie Eclipse oder Android Studio.

Neben einer nativen Umsetzung existiert auch die Möglichkeit eine reine Web App umzusetzen, also die Anwendung mittels Websprachen zu imple-

mentieren. Es handelt sich um eine Webseite die für mobile Geräte angepasst ist, wobei dies aber zu starken Einschränkungen im Usability Bereich führt. Plattformspezifische Konventionen und Normen werden nicht unterstützt. Frameworks können Möglichkeiten bieten, den „Look and Feel“ dem einer nativen Anwendung anzugleichen. Zwar ist die plattformübergreifende Natur von Web Apps von Vorteil, jedoch kann daher nicht auf alle unterschiedlichen Konventionen und Normen angeglichen werden. Von Vorteil ist auch, dass Web Apps über eine eindeutige URL identifiziert und geteilt werden können. Nachteile von Web Apps sind Defizite bezüglich Device-Features, eine schlechtere Geschwindigkeit im Vergleich zu nativen Lösungen und Performanceschwierigkeiten. Es sollte auch beachtet werden, dass Webapps oft eine Internetverbindung benötigen. Auch sicherheitskritische Anwendungen sind sehr schwer umzusetzen, außerdem werden Web Apps nicht über Stores vertrieben [11]. Für die gewählte Architektur wurde eine Web App ausgeschlossen, da sie nicht ausreichend Kontrolle über die Anwendung gibt und der große Vorteil der Plattformunabhängigkeit für diese Arbeit nicht von Bedeutung ist.

Hybride Apps bedienen sich ebenfalls diverser Frameworks um Vorteile von nativen Anwendungen und Web Apps zu verschmelzen. Dadurch können Anwendungen plattformübergreifend mittels Websprachen entwickelt werden, und erhalten dennoch Zugriff auf diverse plattformspezifische Features. Um dies zu ermöglichen wird von einer Web App mittels nativem Wrapper auf das Betriebssystem des Geräts zugegriffen. Davon abgesehen unterliegt die eigentliche Webanwendung jedoch nach wie vor den gleichen Einschränkungen. Durch plattformspezifisches Design, kann an die Usability einer nativen Umsetzung angenähert werden, wodurch jedoch die Plattformunabhängigkeit reduziert wird, die einer der Vorteile dieses Ansatzes ist. Nachteile dieses Systems sind ähnlich derer von Web Apps, viele Funktionen die der Wraper bereitstellt funktionieren nicht auf allen Geräten oder verhalten sich verschieden, die Performance kann nicht mit nativen Anwendungen mithalten, verschiedene Browser und sicherheitskritische Funktionen können auch hier Probleme bereiten. Dafür ermöglicht ein hybrides Framework, durch die Umsetzung mit bekannten Websprachen, eine einfachere und schnellere Umsetzung als ein nativer Ansatz und bietet gleichzeitig Plattformunabhängigkeit. Die Qualität der Frameworks ebenso wie ihre Popularität hat in den letzten Jahren stark zugenommen [26, 11]. Beispiele für Hybride Frameworks sind Phonegap[1], Sencha Touch[2] und Appcelerator Titanium[3].

Eine andere Möglichkeit ist die Verwendung eines nicht Web-basierten Frameworks. Diese nutzen bestimmte Laufzeitumgebungen und Bibliotheken um die Umsetzung mobiler Anwendungen zu erleichtern. Je nach Framework

[1] http://phonegap.com/
[2] https://www.sencha.com/products/touch/
[3] http://www.appcelerator.com/

wird in einer bestimmten, bevorzugt einfachen und bekannten, Sprache programmiert, die nicht von der gewählten Plattform abhängig ist. Diese Sprache wird dann entweder zu nativem Code kompiliert und ausgeführt oder in einen Container als hybride App verpackt und interpretiert. Prinzipiell können Framework Apps dieselben Eigenschaften wie native Apps haben und sogar native Komponenten nutzen, Einschränkungen können jedoch durch die Bibliotheken und die Menge der vom Framework unterstützten Funktionen auftreten. Die Usability der Anwendung ist abhängig davon, ob das Framework native Komponenten nutzt oder lediglich ein natives Verhalten emuliert. Die Performance der Anwendung ist abhängig davon, ob nativer Binärcode kompiliert wird. Ist die Anwendung durch die Verwendung nichtnativen Codes auf weitere Abstraktionsschichten angewiesen, so kommt es zu Performance-Verlusten. Dieser Ansatz kann also Vorteile nativer Umsetzung, wie hohe Performance und Usability, mit Vorteilen hybrider Umsetzung, wie Cross-Plattform Development, kombinieren, ist jedoch weiterhin von den Einschränkungen und der Qualität des jeweiligen Frameworks abhängig. Außerdem entsteht auch hier für dir Umsetzung von plattformabhängigen Normen und Konventionen Mehraufwand für den Entwickler oder die Entwicklerin, da er oder sie sich mit diesen auseinandersetzen muss [11]. Beispiele für solche Frameworks sind Unity3D[4] und Corona SDK[5].

Weiters gibt es auch Frameworks die auf einer bestimmten Architektur basieren, wie etwa auf OSGi[6]. Die meisten dieser Frameworks sind standardmäßig nicht für Android ausgelegt, es gibt jedoch spezielle Abwandlungen wie Apache Felix[7]. Diese erfordern jedoch auch eine spezielle Laufzeitumgebung weswegen sie eher für größere Projekte von Vorteil sind. Eine weitere Möglichkeit wäre die Nutzung eines Frameworks basierend auf dem System der Service Oriented Architecture (SOA). Prinzipiell sind diese jedoch meist auf Webservices zugeschnitten. Außerdem kann das Verwenden diverser Middleware Schichten laut [18] zu Performance Einbußen führen. Projekte in einem solchen Framework umzusetzen kann zu unnötigem Overhead führen und zwingt dem Projekt eine modulare Architektur auf. Dadurch, dass die gesamte Anwendung in einem einzigen Prozess läuft, ist keine Interprozesskommunikation nötig. Wenn jedoch eine Komponente zum Absturz des Prozesses führt, ist das ganze System betroffen. In nativem Android sind auf andere Prozesse ausgelagerte Komponenten nicht betroffen. Native Android Programmierung bietet außerdem mehr Kontrolle und Anpassungsfähigkeit an die Anforderungen der Architektur. Bei manchen größeren Projekten oder wenn bereits OSGi Komponenten existieren, die auf die neue Anwendung übertragen werden können, kann die Verwendung eines auf OSGi basierenden Frameworks für Android und damit die Umsetzung einer

[4] https://unity3d.com/
[5] https://coronalabs.com/products/corona-sdk/
[6] http://www.osgi.org/Main/HomePage
[7] http://felix.apache.org/

modularen Architektur natürlich von Vorteil sein [20, 28, 18].

Framework:

Die Vorteile sind:

- Schnell und einfach – niedrigerer Entwicklungsaufwand [33, 41, 11, 56],
- Kosteneffizienz [41],
- nutzt weit verbreitete Technologien und Programmiersprachen zur Entwicklung [27, 33, 11],
- Wartbarkeit – Änderungen an der Kernfunktionalität sind schnell umgesetzt [33, 41, 11, 56],
- Cross-Platform Development [27, 33, 41, 11, 56],
- Interprozesskommunikation ist nicht immer nötig [28].

Die Nachteile sind:

- Eingeschränkte Kontrolle – abhängig von Framework Bibliotheken [27, 41, 11],
- bei Frameworks die keine nativen Komponenten nutzen, kommt es zu Einschränkungen in der Usability [33, 41, 11],
- eine Cross-Plattform Umsetzung kann erhöhten Aufwand bezüglich Usability und Design Guidelines für verschiedene Plattformen mit sich ziehen [33, 41, 11],
- Performanceverlust bei mehreren Abstraktionsschichten – Gefahr von erhöhtem Overhead abhängig von der Entwicklungsmethode [27, 33, 41, 11, 56, 18],
- bei komplexeren und größeren Anwendungen kann es zu erhöhtem Wartungsaufwand kommen [33],
- Security Features sind nicht immer ausreichend unterstützt [41, 11],
- die Kommunikation mit anderen Anwendungen ist nicht immer möglich [11],
- oft gibt es mangelnde Debugging-Möglichkeiten und schlechtere Entwicklungswerkzeuge [41, 11].

Native Android:

Die Vorteile sind:

- Volle Kontrolle [41, 11],
- Performance – Geschwindigkeit durch schnelle Grafik API und Multithreading [27, 33, 41, 11, 56],

- Zugriff auf alle vom Betriebssystem bereitgestellten Schnittstellen und Hardware-Funktionen sowie die aktuellste API [27, 33, 41, 11, 56],
- Usability und native User Experience wird unterstützt [41, 11],
- bei komplexeren und größeren Anwendungen sind native Anwendungen meist besser zu skalieren und zu warten [33],
- sehr gute Werkzeuge, Support und Best Practices des Herstellers [41, 11],
- Security Umgebung wird durch API Zugriff unterstützt [41],
- Robuste Programmiersprachen [11],
- die Kommunikation mit anderen Anwendungen ist möglich [11],
- einfaches Debugging [41, 11].

Die Nachteile sind:

- Mehr Aufwand und komplexe Umsetzung [27, 33, 11, 56],
- höhere Kosten bei der Entwicklung und Wartung [27, 33, 11],
- kein Cross-Plattform Development [27, 33, 41, 11, 56],
- Interprozesskommunikation ist nötig [28].

Frameworks dienen hauptsächlich dazu dem Entwickler oder der Entwicklerin Arbeit abzunehmen, da sie Aufgaben, die von bestimmten Arten von Anwendungen oft verwendet werden, übernehmen oder erleichtern. So gibt es zum Beispiel eigene Frameworks die auf die Umsetzung von Spielen spezialisiert sind und es ermöglichen schneller neue Spieleanwendungen umzusetzen. Außerdem bieten die meisten Frameworks Cross-Plattform Development an. Firmen, die ihre Anwendung auf mehreren Plattformen anbieten wollen und nicht mehrere native Projekte umsetzen können oder wollen, die für die jeweilige Plattform zugeschnitten sind, können sich eines solchen Frameworks bedienen. Dadurch werden sie jedoch auch mit den Grenzen des Frameworks konfrontiert, das verwendet wird. Native Umsetzung bietet im Gegensatz zu Frameworks volle Kontrolle über jeden Aspekt der Anwendung. Die Architektur kann auf die Anforderungen der Anwendung zugeschnitten werden und jeder Aspekt der Plattform kann genutzt werden [27, 28, 33, 41, 11, 56]. In Bezug auf die Usability eignet sich eine native Umsetzung besonders für nicht immersive Apps, während für immersive Apps oft nicht-native Entwicklungsmethoden geeignet sind. Dies sollte bei der Wahl der Architektur beachtet werden. Native Entwicklungsumgebungen bieten Standard-Interfacekomponenten und ermöglichen eine Umsetzung nahe an Kultur, Konventionen und Norm der Plattform, die Nutzern bekannt ist [11].

Für die Entwicklungsumgebung fiel die Entscheidung auf eine native Umsetzung, da die konkreten Anforderungen der Anwendungen flexibel anpassbar sein sollten, ohne durch ein eventuell zu Beginn falsch gewähltes

Framework eingeschränkt zu werden. Nicht-immersive Anwendungen sollen genauso umsetzbar sein und die Architektur sollte auch gute User Experience und Performance ermöglichen, falls dies nötig ist. Die Architektur in nativem Code umzusetzen bietet volle Kontrolle über die Anwendung und mehr Freiheiten bei der Struktur. Alle vom Betriebssystem bereitgestellten Schnittstellen und Hardware-Funktionen sowie die aktuellste API sollen für die Anwendung verwendet werden können. Außerdem ist kein Cross-Plattform Development nötig – die Architektur soll auf Android zugeschnitten sein. Frameworks bei der Umsetzung der Plugins sind jedoch je nach Projektanforderung möglich. Lediglich die Kommunikation mit dem Client muss eventuell nativ gelöst werden wenn Inter-Process-Communication vom Framework nicht oder nicht ausreichend unterstützt wird. Dies ist von der Wahl des IPC und der Schnittstelle abhängig. Dieser Ansatz lässt es den Plugins frei Frameworks zu verwenden, wenn es Sinn macht, ermöglicht jedoch der Clientanwendung Zugriff auf alle nativen Möglichkeiten [27, 28, 33, 41, 11, 56].

3.1.2 Architektur

Neben der Entwicklungsumgebung ist auch die Architektur der Anwendung von großer Bedeutung. Da die Wahl auf eine native Umsetzung gefallen ist, kann die Architektur beliebig aufgebaut sein, muss jedoch auf native Android Komponenten angepasst werden. Dem generellen Aufbau der Anwendung können verschiedene Muster als Grundlage dienen. Eine Architektur die möglichst flexibel und leicht erweiterbar ist sollte gewählt werden.

Grundarchitektur: Architekturmuster wie Kanäle und Filter, das Blackboard Muster oder das Microkernel Muster sind sehr spezifisch. Nur bestimmte Anwendungen würden von einer solchen Architektur profitieren. Ziel ist es jedoch eine Möglichkeit zu finden eine allgemein anwendbare Architektur vorzustellen. Daher wurden solche Ansätze in diesem Fall außer Acht gelassen [1, 4].

Für den Grundaufbau wäre eine Schichtenarchitektur eine Möglichkeit. Der Ansatz ist stabil und flexibel. Schichten können durch die Verwendung von Schnittstellen leicht ausgetauscht werden und Code Abhängigkeiten werden so minimiert. Dieser Ansatz hat jedoch auch Schwächen. Bei schlechter Strukturierung können sich Änderungen über die Schichten hinaus auswirken. Außerdem können Abhängigkeiten innerhalb einer Schicht problematisch werden und die richtige Anzahl an Schichten ist von Bedeutung. Lediglich wenn die Schichten klar getrennt und austauschbar sind kann eine solche Umsetzung für gewisse Projekte eine Möglichkeit sein, schnell und einfach die Flexibilität und Stabilität der Anwendung zu verbessern. Der Ansatz ist nicht perfekt, jedoch bei kleineren Projekten gut umsetzbar und ausreichend anpassbar [1, 4, 12].

Eine weitere Möglichkeit wäre eine Komponentenarchitektur. Durch die modularen Umsetzungen werden Abhängigkeiten stark reduziert und die Anwendung ist extrem flexibel. Komponenten können leicht hinzugefügt, entfernt oder ausgetauscht werden. Abhängig von der Struktur kann dies jedoch zu einem erhöhten Aufwand in der Entwicklung sorgen. Ein solcher Ansatz kann die Erweiterbarkeit und Wartbarkeit einer Softwareanwendung enorm erleichtern, da er extrem geringe Kopplung mit sich zieht. Vor allem für kleine Projekte zahlt sich eine solche Architektur jedoch oft aufgrund des Mehraufwands in der Entwicklung nicht aus [12, 17].

Ein ähnlicher Ansatz wäre die Service Orientierte Architektur. Diese ist hauptsächlich auf eine Umsetzung für Webservices spezialisiert. Durch die Kapselung der Services werden Abhängigkeiten reduziert. Die Anwendung kann jederzeit durch neue Services erweitert werden, alte Services können entfernt oder ausgetauscht werden. Vor allem für Anwendungen, die bereits existierende Services von anderen Plattformen auch mobil nutzen wollen ist eine solche Umsetzung von großem Vorteil [47, 18].

Um eine flexible Architektur umzusetzen wurde die Komponentenarchitektur gewählt. Aufgrund des modularen Aufbaus ist sie besonders geeignet. Zwar ist die Umsetzung komplexer als, beispielsweise, die einer Schichtenarchitektur, jedoch sind so einzelne Teile der Anwendung erweiterbar und austauschbar, während die Schichtenarchitektur anfällig für Abhängigkeiten zwischen und innerhalb der einzelnen Schichten ist. Die Erweiterung einer Schichtenarchitektur kann dadurch problematisch werden, da neue Schichten schwer nachträglich eingebaut werden können und die Änderung der Schnittstelle einer Schicht sich direkt auf angrenzende Schichten auswirkt. Eine Komponentenarchitektur hingegen kann leicht durch das hinzufügen neuer Komponenten erweitert werden und eignet sich so ausgezeichnet für die Umsetzung einer Plugin Architektur. Dem ähnlichen Ansatz der Service Orientierten Architektur wurde aufgrund seiner Spezialisierung auf Webservices die Komponentenbasierte Architektur vorgezogen. Da die Service Orientierte Architektur eher für komplexe und verteilte Systeme, so wie für Server-Client Systeme gedacht ist und auf Webservices spezialisiert ist, wurde eine generelle komponentenbasierte Architektur gewählt [1, 4, 9, 12, 49, 16, 54, 17].

Interaktionsarchitektur: Neben einer flexiblen Grundstruktur, wie in diesem Fall der Aufteilung einzelner Software Features auf unabhängige Komponenten, sollte auch die Trennung von Darstellung, Logik und Daten berücksichtigt werden. In modernen Anwendungen ist es oft nötig das Layout, Grafiken und Benutzer-Interfaces zu ändern, ohne dass die Funktionsweise davon betroffen wird. Dies ist essenziell für Änderungen von Trends bezüglich Design und Usability vor allem für länger laufende Anwendungen. Es gibt mehrere Architekturmuster zur Gestaltung interaktiver Systeme, die

dabei helfen eine Anwendung flexibel zu strukturieren. Nicht alle sind für eine Implementation in Android ausgelegt. Die Projektstruktur der Plattform, ebenso wie die von ihr bereitgestellten Komponenten, müssen dabei jedoch beachtet werden [45, 14].

Ein möglicher Ansatz zur Strukturierung von interaktiven Systemen wird von PAC geliefert. Neben einem Ansatz zur Trennung von Darstellung, Logik und Daten wird hier jedoch auch ein System vorgestellt, wie die einzelnen Agenten kommunizieren sollen. Eine hierarchische Anordnung einzelner Agenten führt jedoch zu einer Art Schichtenarchitektur. Für den generellen Aufbau der Anwendung wird jedoch eine Komponentenarchitektur aufgrund der leichten Erweiterbarkeit und Änderungsfreundlichkeit bevorzugt. Außerdem kann durch den hierarchischen Aufbau die Komplexität der Kontrollelemente problematisch werden. Neben der Logik des einzelnen Agenten muss auch der Zugriff auf Agenten niedrigerer Ebenen berücksichtigt und verwaltet werden. Aus diesen Gründen wird dieses Muster in der vorliegenden Architektur nicht berücksichtigt [1]. Model-View-ViewModel ist ein weiteres Model zur Trennung von Darstellung, Logik und Daten. Es weist Ähnlichkeiten zu MVC und MVP auf, wird jedoch durch das Binden zwischen Komponenten realisiert. Dieses wird standardmäßig nicht von Android unterstützt, zu einer Umsetzung wäre ein zusätzliches Framework nötig. Da für das Basismodel entschieden wurde eine native Lösung zu nutzen, und die Anwendung nicht von Frameworks abhängig zu machen, um jede Art von Anwendung bestmöglich zu unterstützen, wurde dieses Muster in der vorliegenden Architektur ebenfalls nicht berücksichtigt [14].

Die wohl bekanntesten Ansätze zur Strukturierung von interaktiven Systeme sind MVC und MVP. Diese beiden Ansätze sind sich ähnlich, der Unterschied liegt hier in der Aufgabe des Controllers beziehungsweise des Presenters. In MVC ist der Controller verantwortlich für die Verwaltung von Maus- und Tastatur-Events. In MVP behandeln GUI Komponenten den Userinput selbst und delegieren dann die Interpretation an den Presenter. Da in Android Activities und Fragments sowie die einzelnen UI Elemente auf Userinput reagieren, ist ein Ansatz hier am ehesten dem des MVP Musters entsprechend. View Elemente reagieren zwar mittels Event Listenern auf Userinput, delegieren diesen jedoch an die zugehörige Logikkomponente um den Input zu interpretieren. Da die Logik und die Darstellung jedoch nach dem Seperation of Concerns Prinzip getrennt sein sollten, und Views austauschbar sein sollen, macht eine Passive View Implementation Sinn. Eine Umsetzung von MVC würde bedeuten, dass direkt in der Controllerkomponente auf Userinput reagiert wird. Eine solche Implementation ist nicht intuitiv mit der Projektstruktur von Android und den Aufgaben, die Activities übernehmen, vereinbar, da Activities häufig verwendet werden, um sowohl die Logik als auch Aufgaben der Darstellung zu übernehmen. Zwar übernehmen XML Layouts einige Teile, jedoch werden Listen und ähnliche Elemente oft von Code direkt in der Activity oder dem Fragment betroffen.

Bei Änderungen im Layout File kann es zu Auswirkungen auf die zugehörige Activity oder das zugehörige Fragment kommen, obwohl die Logik der Anwendung unverändert bleibt. So betrifft das Verwenden einer anderen Liste oder eines anderen View Elements im Layout File auch die Activity oder das Fragment, in dem auf die Liste oder das View Element zugegriffen wird. Eine Möglichkeit ist es, sämtlichen Code der die Darstellung betrifft in der Activity oder dem Fragment zu behalten und die Logik auszulagern. Eine andere Möglichkeit ist es, die Darstellung auf Fragments, oder Layout-Implementationen zu übertragen und die Logik in der Activity zu behalten. Eine Kombination ist ebenso eine Option. Damit ist die Umsetzung eines MVC Ansatzes in Android möglich. Die Implementation entsprechend des Passive MVC Musters klingt vielversprechend und bietet eine Umsetzung die mit der gewählten Komponentenarchitektur gut zusammenpasst, eine Umsetzung entsprechend dem MVP Muster ist jedoch besser auf die GUI Elemente der Android Plattform zugeschnitten. Durch das Verwalten von Interface-Komponenten und Event Listenern in der View, ist die Implementation der Darstellung völlig von der Logik entkoppelt und leicht austauschbar, solange die Schnittstelle bezüglich des Presenters eingehalten wird. Über diese Schnittstelle kann der Presenter auch über Interface Events benachrichtigt werden um diese dann zu verarbeiten. Durch die Umsetzung des Passive View Musters kann die View auch vom Model entkoppelt werden, was zu einer übersichtlicheren Verwaltung der Abhängigkeiten zwischen den Komponenten sorgt [34, 36, 45, 14, 55].

Erweiterbarkeit (und Personalisierung): Eine gute Architektur ist wichtig, um eine Anwendung flexibel für Änderungen und Erweiterungen zu machen. Neben der Aktualisierung der Anwendung durch Änderungen auf die konventionelle Weise des Einspielens von Updates gibt es auch Methoden eine Anwendung mit optionalen Erweiterungen anzubieten. Nicht immer sind Updates für die Funktionsweise der Anwendung essentiell, manche Erweiterungen haben rein ästhetische Zwecke oder bieten zusätzliche Features an, die nur für einen Teil der Benutzer und Benutzerinnen nötig oder wünschenswert sind. Eventuell will eine Anwendung einem Nutzer oder einer Nutzerin die Möglichkeiten bieten diese durch Erweiterungen zu personalisieren. Benutzer oder Benutzerinnen, die bestimmte Features nicht nutzen, sollten nicht ihre Anwendung aktualisieren müssen, wenn es nicht für die Funktionsweise der Grundanwendung nötig ist.

DLC: Wenn eine bereits lauffähige Anwendung erweitert werden soll, so kann ein Großteil über simple Updates gehandhabt werden. Oft betrifft dies hauptsächlich das Herunterladen von neuem Content in der Form von Assets. Um dem Nutzer oder der Nutzerin das Herunterladen solcher Assets möglichst angenehm zu gestalten ist eine gute Downloadable Content (DLC)

Pipeline essentiell. Ein Problem bei DLC Updates ist, dass diese oft nur einen Teil der Assets betreffen. Auf dieses Problem kann mit Delta-basierten Packs reagiert werden. Dabei enthält ein neues Assetpack nur Assets die geändert wurden, was zu kleineren Updates führt. Beim Herunterladen mehrerer Updates muss so jedoch ein User die Basis und sämtliche Deltas herunterladen, auch wenn manche der Basis- und Delta-Assets eventuelle in der nächsten Delta überschrieben werden. Eine andere Möglichkeit dieses Problem anzusprechen ist, dem Client lediglich jene Assets, die für eine erste Nutzung minimal nötig sind, bereitzustellen. Zusätzlich erhält der Client einen Index, über den alle Assets heruntergeladen werden können. Wenn ein Asset nötig ist, wird überprüft, ob er im Speicher existiert, wenn nicht wird er heruntergeladen. So werden nur Assets heruntergeladen, die nötig sind. Soll ein Asset aktualisiert werden, so wird einfach der neue Asset hochgeladen und der Index aktualisiert, der diesen referenziert. Das Textdokument, das den Index enthält, ist einfach zu ändern und nach jedem Update werden nur die veränderten Assets heruntergeladen, wenn diese benötigt werden. Auch die Methode des Hosten von Assets ist von Bedeutung. Statische Assets aus Anwendungs-Servern oder sogar eigenen Web-Servern zu hosten kann stark an den Server I/O Ressourcen zehren. Dies kann verhindert werden indem Assets über Amazon S3, Google Cloud Store oder jeden anderen Cloud Storage Provider dem Client zum Download zugänglich gemacht werden. So können Server für andere Aufgaben genutzt werden. Cloud Storage skaliert außerdem ausgezeichnet, auch bei großen Mengen ohne Aufwand für die Entwickler oder Entwicklerinnen. Außerdem verringert dies die Risiken von DOS Angriffen. Neben dem Hosten von Assets über einen spezialisierten Service kann auch ein Content Distribution Network (CDN) wie Cloud Front oder Akami genutzt werden. Dies verteilt Kopien der Assets an diverse globale Server und übermittelt dem Client Assets stets von einer nahegelegenen Kopie. So können bei Problemen auch andere Anbieter für die Assets einspringen und bei globaler Nutzung der Anwendung kann das Herunterladen mittels einem nahegelegenen Server deutlich schneller sein [38].

Das Nutzen einer guten DLC Pipeline ist für die meisten Anwendungen ausreichend um neuen Content anzubieten, vor allem wenn es sich größtenteils um Ressourcen handelt. Der Nutzer oder die Nutzerin kann bis zu einem gewissen Grad mitbestimmen welche Ressourcen er nutzen möchte und diese können auf Bedarf heruntergeladen werden, um die Anwendung zu personalisieren. Wenn jedoch größere Features angeboten werden sollen, die zusätzlich zur Client-Anwendung genutzt werden können, so kann die Umsetzung rein mittels DLC-Updates unzureichend sein. Hier kann ein Plugin System Abhilfe schaffen.

Plugins: Eine andere Möglichkeit eine Anwendung zu erweitern, die bereits lauffähig ist, ohne Änderungen direkt an dieser vorzunehmen ist die

Nutzung von Plugins. Diese können zur Personalisierung durch den Benutzer oder die Benutzerin dienen, oder um zusätzliche Features, gegen Gebühr oder gratis, anzubieten. Die Client Anwendung und die Plugins können separat aktualisiert werden. Diverse mobile Anwendungen am Markt Nutzen das Plugin System, wie beispielsweise Dolphin, Line, Dashclock und CM Launcher [44]. Der Plugin Ansatz ermöglicht eine Anwendung beliebig durch eigenständige Anwendungen zu erweitern. Dabei werden die diversen Plugins, von einem Plugin Manager seitens des Clients verwaltet, um neue Plugins flexibel herunterzuladen und in die Anwendung zu integrieren. Der Client stellt dazu eine Schnittstelle bereit, die der Erweiterung zur Kommunikation mit dem Client dient. Die Schnittstelle sollte möglichst stabil bleiben um die Funktionalität von älteren Plugins bei Updates zu gewährleisten. Neue Kommunikations-Schnittstellen können jedoch jederzeit hinzugefügt werden, solange die Version des Clients und der Plugins beachtet wird. Da Plugins und Client unabhängig voneinander aktualisiert werden können sollte darauf geachtet werden, Synchronisationsprobleme mit inkompatiblen Versionen von Client und Plugin zu vermeiden [4–6].

Die Nutzung von Plugins kann für die Erweiterbarkeit und Personalisierung einer Anwendung von Vorteil sein, ist jedoch nicht immer nötig. Je nach Anwendung, kann die Erweiterbarkeit mittels neuen Contents über eine DLC Pipeline völlig ausreichend sein und sogar bis zu einem gewissen Grad eine dynamische Personalisierung durch den Nutzer oder die Nutzerin zulassen. Dies beschränkt sich jedoch grundsätzlich auf die Verwaltung von Erweiterungen durch Asset-Ressourcen. Eine Plugin Architektur hingegen bietet zusätzlich die Möglichkeit eine Anwendung durch beliebige Funktionen modular zu erweitern und zu personalisieren. Daher sollte eine Architektur, um jede Art von Erweiterung zu unterstützen, auch die Möglichkeit anbieten Plugins für eine Anwendung zu entwickeln.

3.2 Lösungsansatz

Um die Architektur einfach und flexibel zu halten, wird auf die Verwendung eines Frameworks in diesem Fall verzichtet und eine native Umsetzung vorgeschlagen. Als Grundlage für die gewählte Architektur dient die generelle Struktur von nativen Android Anwendungen und die von der Plattform zur Verfügung gestellten Komponenten, unter anderem Activities, Fragments und Services [28]. Die generelle Struktur beruht auf dem Ansatz des Android Passive MVC, wie es in [14] vorgestellt wird (siehe Abb. 3.1). Es wurde jedoch statt des MVC Musters das MVP Muster genutzt (siehe Abschnitt 3.1.2) und dementsprechend die vorgeschlagene Architektur angepasst. Eine Komponentenarchitektur bildet die Basis der Applikation. Die Applikation beinhaltet sowohl Logik-Komponenten als auch Komponenten, die zur Darstellung von Informationen dienen. Neue Darstellungen können unabhängig

voneinander hinzugefügt werden, indem sie an eine Activity weitergegeben werden, die diese anzeigt. Diese Komponenten bedienen sich des MVP Prinzips. Jede Komponente beinhaltet getrennte Darstellung, die von einer Presenter Klasse mit allen von ihr benötigten Informationen gefüttert wird. Über direkte Methoden wird die Darstellung der Komponente vom Presenter initialisiert, die erhaltenen Daten werden in Form von Android Interface Komponenten dargestellt und Event Listener können auf visuelle Interfaceelemente platziert werden. Über ein vom Presenter zur Verfügung gestelltes Interface kann die View diesen über User Interaktionen informieren. Lediglich das Presenter Element hat Zugang zu dem Model der Applikation, in dem die Daten gespeichert werden. Über das Observer Muster werden Presenter über Änderungen im Model informiert und über Methoden erhalten sie Zugang zu den Daten. Logik Komponenten werden hingegen an der Application registriert. Auch diverse Services sollen von der Anwendung verwendet werden können. Ein Service Manager verwaltet diese und kann jederzeit um neue Services erweitert werden. Zur Verwaltung der Daten gibt es neben dem Model noch ein DAO Layer, das sich um die Speicherung der Daten und die Synchronisation mit einer möglichen Datenbank beschäftigt. Der Layer bietet ein Interface zur Speicherung und zum Zugriff auf die Daten, welches den Austausch der unterliegenden Implementierung ermöglicht [14]. Für Content-betonte Anwendungen sollte eine entsprechende DLC Pipeline umgesetzt werden um ein flexibles Hinzufügen, Austauschen und Entfernen von Ressourcen zu ermöglichen. Mittels des Laden von Ressourcen „on demand“ und via Cloud-Hosting kann sowohl Downloadzeit verringert als auch Speicherplatz gespart werden [38]. Weiters wird ein Plugin Ansatz genutzt, der die Erweiterbarkeit und Personalisierung der Anwendung ermöglicht. Entsprechend dem Plugin Muster kann eine Grundanwendung ein Interface zur Erweiterung zur Verfügung stellen, mit dem andere Anwendungen kommunizieren können. Eine eigene Komponente sollte die Plugins als Art Pluginmanager verwalten [5, 6].

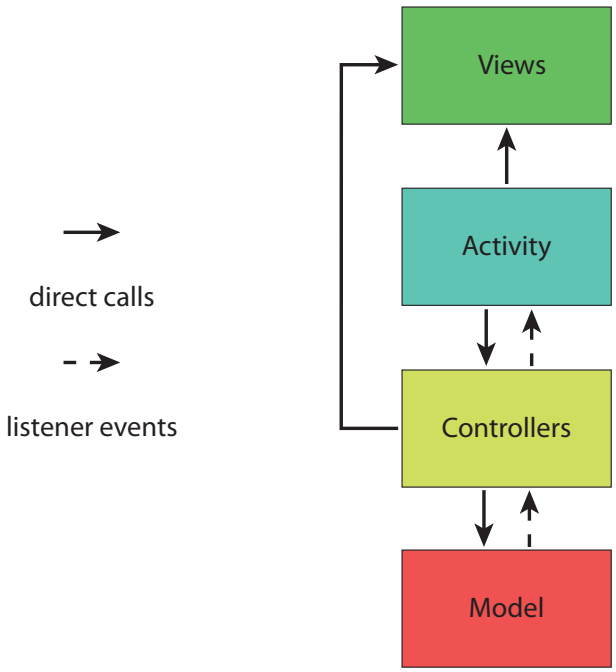


Abbildung 3.1: Illustration des Android Passive MVC Ansatzes.

Kapitel 4

Implementation

Dieses Kapitel beschäftigt sich mit einer konkreten Implementierung der im vorherigen Kapitel vorgeschlagenen Architektur. Das GEMPLAY Projekt dient als Grundlage für eine Anwendung, bei deren Umsetzung besonders auf die Wartbarkeit und Erweiterbarkeit geachtet werden soll. Von den zuvor erörterten Möglichkeiten werden jene, die für die Anwendung sinnvoll sind, entsprechend umgesetzt und deren Implementation in Android wird dabei näher betrachtet.

4.1 Das GEMPLAY Projekt

Das GEMPLAY Projekt[1] beschäftigt sich mit personalisierten, genderkonformen Spielen zur Bewegungsmotivation unter Verwendung von Persuasionsprinzipien. Im Zuge dieses Projekts sollen mobile Spieleprototypen, die für spezielle Spielertypen (beruhend auf dem BrainHex Model [8]) personalisiert sind, entwickelt und getestet werden. Zum Beginn der Studie waren die Game Designs für die unterschiedlichen Prototypen noch nicht vorhanden, jedoch war bereits bekannt, dass es sich um mobile Spiele handeln soll, die mittels GPS oder ähnlichen Funktionen den Spieler oder die Spielerin zur Bewegung motivieren und seinen oder ihren Fortschritt erkennen sollen. Eine Standardanwendung soll die allgemeinen Aufgaben zur Bewegungserkennung durchführen und zum Beispiel die Position oder die Orientierung des Spielers oder der Spielerin feststellen. Diese sollte erweiterbar sein durch beispielsweise ein Profil oder Statistiken zur Bewegung eines Benutzers oder einer Benutzerin um dessen oder deren Fortschritt zu speichern. Außerdem sollten im Laufe des Projekts die einzelnen Spiele als Plugins für diese Client Anwendung erstellt werden und dann von dieser gestartet werden können. Die Spiele erhalten somit Zugriff auf die Funktionen des Clients und können diesem auch mitteilen, wenn sie bestimmte Informationen benötigen.

[1] www.gemplay.at

Der Client sollte dabei organisatorische Aufgaben für die Spiele übernehmen [32].

Die Anforderungen des Projekts an die Anwendung haben sich öfters geändert, da die Anwendung stark von den Anforderungen der Spiele und den Anforderungen der Studie abhängig war. Neue Funktionen mussten dynamisch im Client hinzugefügt werden können und das Interface zur Kommunikation mit den Spielen wurde durch neue Funktionen öfters erweitert. Auch wurden einige der für den Client ursprünglich geplanten Funktionen für die Studie gestrichen. Da die Studie nur kurz lief und keine Anmeldung der Testpersonen erfolgte, wurden Profile und Statistiken, wie sie ursprünglich geplant waren, aus der Anwendung gestrichen. Ursprünglicher Plan war, das Angebot an Spielen und Aufgaben zu personalisieren. Abhängig vom Profil sollten Spiele die dem Spielertyp des Nutzers oder der Nutzerin entsprechen heruntergeladen werden. Aufgaben, die im Laufe des Spiels anfallen sollten außerdem an den Bewegungs- und Motivationstyp des Nutzers oder der Nutzerin sowie an seine oder ihre körperliche Verfassung angepasst sein. Für die Studie wurden jedoch schließlich nur Prototypen für zwei Spielertypen gestaltet und getestet, die nicht bezüglich anderer Merkmale personalisiert wurden, um die Studie eindeutig zu halten. Profilinformationen waren nicht mehr nötig, da beide Spiele, in unterschiedlicher Reihenfolge, von unterschiedlichen Spielertypen getestet werden sollten. Ein Optionsmenü, in dem installierte Spiele ein- und ausgeblendet werden können, sowie deren Reihenfolge eingestellt werden kann, wurde stattdessen hinzugefügt. Die Architektur musste also sehr flexibel sein und Komponenten hinzufügen, entfernen und austauschen können, wenn sich an den Projektanforderungen etwas änderte. Dies betrifft sowohl einzelne Bildschirmdarstellungen als auch Backgroundservices.

Die Umsetzung der Spiele Plugins selbst war nicht Teil dieser Arbeit. Für die Studie wurde eine entsprechend angepasste Version des Clients gemeinsam mit diversen eigens für die Studie entwickelten Plugins genutzt. Ein Spiel namens Gemini wurde im Laufe der Studie von Mitarbeitern der FH OÖ Forschungs- und Entwicklungs GmbH entworfen. Dessen einzelne, auf die Spielertypen Seeker und Mastermind zugeschnittenen Teilaufgaben sind in der Form von Plugins mit Hilfe des Clients ausführbar (siehe Abb. 4.1). Die Spiele können dynamisch hinzugefügt und entfernt werden, indem sie auf einem mobilen Android-Gerät, auf dem auch der Client installiert ist, installiert oder deinstalliert werden. Der Client erkennt selbstständig, wann ein Spiel hinzugefügt oder entfernt wird. Beim Start wird vom Client eine Liste der installierten Spiele angezeigt (siehe Abb. 4.2 (a)). Diese können über einen Tap gestartet werden. Außerdem kann mittels Long-Klick auf das GEMPLAY Logo im Client ein Menü geöffnet werden. Durch Auswahl des einzig verfügbaren Eintrages, „Studien-Einstellungen“, kann der Benutzer auf ein Optionsmenü zugreifen um Sichtbarkeit und Anordnung der Spiele zu ändern (siehe Abb. 4.2 (b)). Über das Plugin Interface kann zwischen

Spiel und Client kommuniziert werden. Das Spiel kann vom Client anfordern, dass es Updates über die Bewegung des Geräts erhalten möchte, indem es dem Client mitteilt, dass eine Aufgabe gestartet werden soll. Wenn Updates angefordert wurden, schickt der Client regelmäßig Informationen zur GPS-Position, zur Orientierung, und zur seit Start der Anforderung vergangenen Zeit und zurückgelegten Strecke an das Spiel. Das Update-Intervall kann vom Spiel eingestellt werden, ebenso wie die Genauigkeit der GPS-Position. Das Spiel kann dem Client mitteilen, wenn keine Updates mehr erforderlich sind, weil die Aufgabe abgeschlossen ist. Interne Statistiken am Client über die Zeit und die zurückgelegte Strecke werden aktualisiert und das Spiel erhält keine weiteren Updates mehr. Wenn das Spiel beendet wird kann dies ebenfalls dem Client mitgeteilt werden. Sollte zu diesem Moment noch eine Aufgabe laufen, so wird diese beendet. Da ein Service Management System am Client genutzt wird um Services wie GPS Tracking, Orientation Tracking oder Time Tracking anzubieten, können weitere Services leicht hinzugefügt werden, wenn dies nötig ist. Jedoch muss, um die Kommunikation mit dem Spiel zu erweitern, das Plugin-Interface aktualisiert werden. Eine neue Interfacekomponente kann ebenfalls leicht hinzugefügt werden. Es kann eine neue Activity erstellt werden, als generelles Layout für einen neuen Bildschirm. Außerdem können in jeder Activity, je nach Layout eine oder mehrere Interface Komponenten hinzugefügt werden. Diese können ausgetauscht und auch wieder entfernt werden. Jede dieser Komponenten gibt der Activity ein Fragment, welches diese in ihr Layout einbetten kann. Logische Komponenten können direkt zur Applikation hinzugefügt werden, wann immer ein neues Feature eingebaut werden soll, das nicht mit einem Interface verbunden ist. Wenn eine Komponente nicht mehr gebraucht wird, kann sie jederzeit wieder entfernt werden.

Der GEMPLAY Client wurde dem zuvor vorgestellten Lösungsansatz für eine flexible Architektur entsprechend implementiert. Anhand der Implementation wurde die Änderungsfreundlichkeit und Erweiterbarkeit der vorgeschlagenen Architektur getestet um deren Nutzen zu evaluieren und eventuelle Probleme festzustellen.

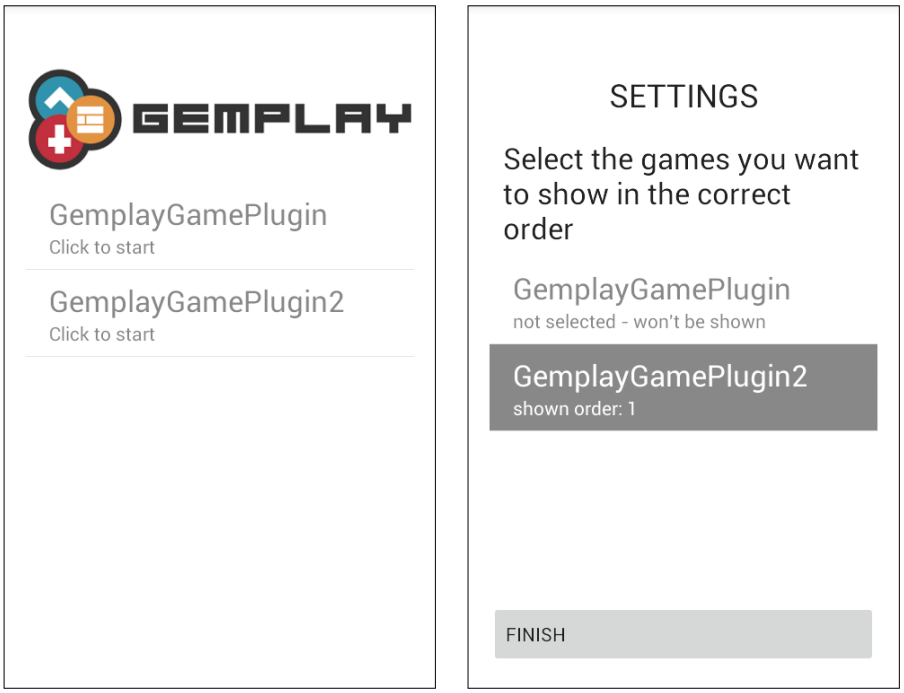
4.2 Architektur

Entsprechend dem vorgeschlagenen Lösungsansatz, wird die Android Anwendung großteils mittels modularer Komponenten realisiert, die in die Applikation nach Bedarf eingebaut werden können. Das in [14] vorgeschlagene Muster für ein passives MVC System wurde in abgewandelter Form entsprechend dem Passive View MVP Muster zur Umsetzung der Interface-Komponenten verwendet. Der Vorschlag einer DAO Schicht wurde ebenfalls aus [14] übernommen (siehe Abb. 4.3).

(a) (b)

Abbildung 4.1: Screenshots des Gemini-Spiels für den Mastermind (a) und den Seeker (b) Spielertyp.

Interface Komponenten: Einzelne Activities in der Anwendung dienen zur Repräsentation von Bildschirmen. Diese bieten ein grundlegendes Layout für Interface Komponenten. Interface Komponenten bestehen aus einer Presenter Klasse, die Logik für die Verarbeitung von Daten enthält und einer View Komponente, die durch eine Ableitung der Fragment Klasse implementiert wird. Der Presenter erzeugt eine Instanz eines View-Fragments bei der Erzeugung in seinem Konstruktor und hat so Zugriff auf public Methoden der View um ihr Daten zur Darstellung zu überreichen. Neben einer Fragment Klasse wird jedem View Element auch ein Layout File in XML zugeordnet. Dies wird in der Fragment Klasse verwaltet, die auch auf Interface-Komponenten des Layouts Zugriff hat und so die Darstellung manipulieren kann. Das View-Fragment kann über ein Interface seiner zugehörigen Logik Komponente mitteilen, wenn eine Interaktion durch den Nutzer oder die Nutzerin stattfindet. Außerdem hat der Presenter Zugriff auf das Model der Anwendung. Er kann über eine vom Model bereitgestellte Schnittstelle dessen Methoden aufrufen und wird bei Änderungen am Model



(a) (b)

Abbildung 4.2: Screenshots des Clients – Im Mainscreen (a) wird eine Liste der verfügbaren Spiele angezeigt und im Optionsmenü (b) kann festgelegt werden welche Spiele angezeigt werden und in welcher Reihenfolge.

von diesem entsprechend dem Observer-Observable Muster benachrichtigt. Er kann dann zugehörige Views entsprechend aktualisieren. Einer Activity können beliebig Interface-Komponenten in Form von Presentern hinzugefügt werden, um Zugriff auf deren Fragments zu erhalten und sie in ihr Layout einzufügen (siehe Abb. 4.4). Auf diese Weise wurden ursprünglich Activities zur Darstellung und Verwaltung von Profil, Statistiken, Aufgaben und Spielen erstellt. Interface Komponenten sind in sich geschlossen und haben keinerlei Zugriff auf andere Komponenten. Sie können jedoch über ein Interface mit der Activity, in der sie eingebettet sind, kommunizieren. Diese kann Nachrichten an andere Komponenten übermitteln. Die Activity kann neben Interface Komponenten auch mit anderen Komponenten der Applikation kommunizieren. Außerdem kann sie, wenn sie von einer Komponente eine entsprechende Nachricht erhält, eine neue Activity starten.

Model – Datenverarbeitung: Das Model der Anwendung wird von der Application-Klasse verwaltet. Bei Start der Application wird das Model instanziiert. Einzelne Sektionen stellen die verschiedenen Teile des Models dar, die die Daten verwalten. Diese stellen Interfaces zur Manipulation zur

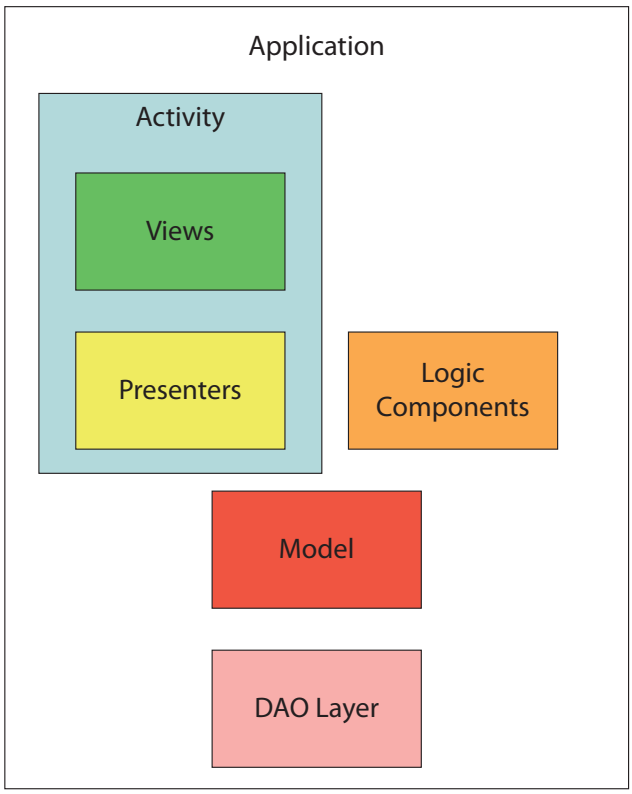


Abbildung 4.3: Illustration des groben Aufbaus der Gemplay Architektur.

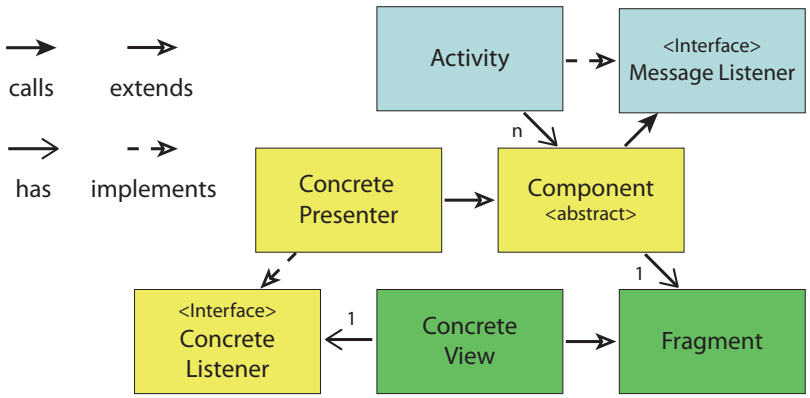


Abbildung 4.4: Illustration des groben Aufbaus der Gemplay Interface Komponenten.

Verfügung. Model Komponenten sind hierarchisch angeordnet. Dabei dienen Sektionen zur Verwaltung von Datenkomponenten. Nur höhere Verwaltungsschichten haben Zugriff auf unterliegende Komponenten. Komponenten derselben Abstraktionsschicht sind voneinander gekapselt. Außerdem kann das Model über das Observer-Observable Muster andere Komponenten über Änderungen informieren (siehe Abb. 4.5). Jede Sektion implementiert dazu die Observable Schnittstelle. Wann immer in dem Model Daten geändert werden, so kann es mittels der geerbten Methoden Observer Klassen informieren, indem es `setChanged()` und `notifyObservers()` aufruft. Presenter von Interface-Komponenten und Logik Komponenten können die Observer Schnittstelle implementieren und so über den Aufruf der Methode `ObservableSection.addObserver(this)` festlegen von welchen Sektionen sie über Änderungen informiert werden wollen. Auf Änderungen reagieren können Observer in der geerbten Methode `update(Observable observable , Object data)`.

Das Model besteht aus dem Profil, den Statistiken, den Aufgaben und den Spielen. Im Profil werden die Userdaten gespeichert. Aufgaben können von Spielen gestartet werden und dienen dazu, zu verfolgen, ob ein Nutzer oder eine Nutzerin gerade eine Tätigkeit durchführt. Außerdem dienen sie dazu, die Bewegung des Nutzers oder der Nutzerin während einer Tätigkeit für Statistiken aufzubereiten. Statistiken dienen dazu die Bewegung des Nutzers oder der Nutzerin und seinen oder ihren Erfolg beim Durchführen von Aufgaben zu dokumentieren. Zum Speichern der Statistiken und Aufgaben werden Basisklassen genutzt. Diese können erweitert werden, um verschiedene Arten von Aufgaben und Statistiken zu speichern (siehe Abb. 4.6). Die Spiele Sektion speichert Informationen, die zum Anzeigen der Spiele und zum Start der Spiele benötigt werden. Ein Spiel kann optional eine Activity und einen Service enthalten.

Das Model ist aufgrund der aktuellen Datenspeicherung komplett serialisierbar umgesetzt. Jede Komponente implementiert dazu die Serializable Schnittstelle. Sämtliche Objekte innerhalb von Komponenten, die nicht serialisierbar sind, wurden als transient gekennzeichnet um die Datenspeicherung nicht zu behindern. Um transient Variablen nicht einfach zu übergehen, wurden diese zusätzlich über Variablen gespeichert, die von Serializable unterstützt werden.

Feature Komponenten – Datenverarbeitung: Teile der Logik der Anwendung, die nicht direkt von einem bestimmten Interface abhängig sind, sondern eigenständige Features darstellen, werden ebenfalls als Komponenten implementiert. Diese werden jedoch nicht von einer Activity, sondern von der Application selbst als Teil des Models verwaltet. Diese Komponenten, die die Logik der Anwendung beinhalten, werden von der Application beim Start instanziiert. Receiver, die von diesen Komponenten benötigt werden,

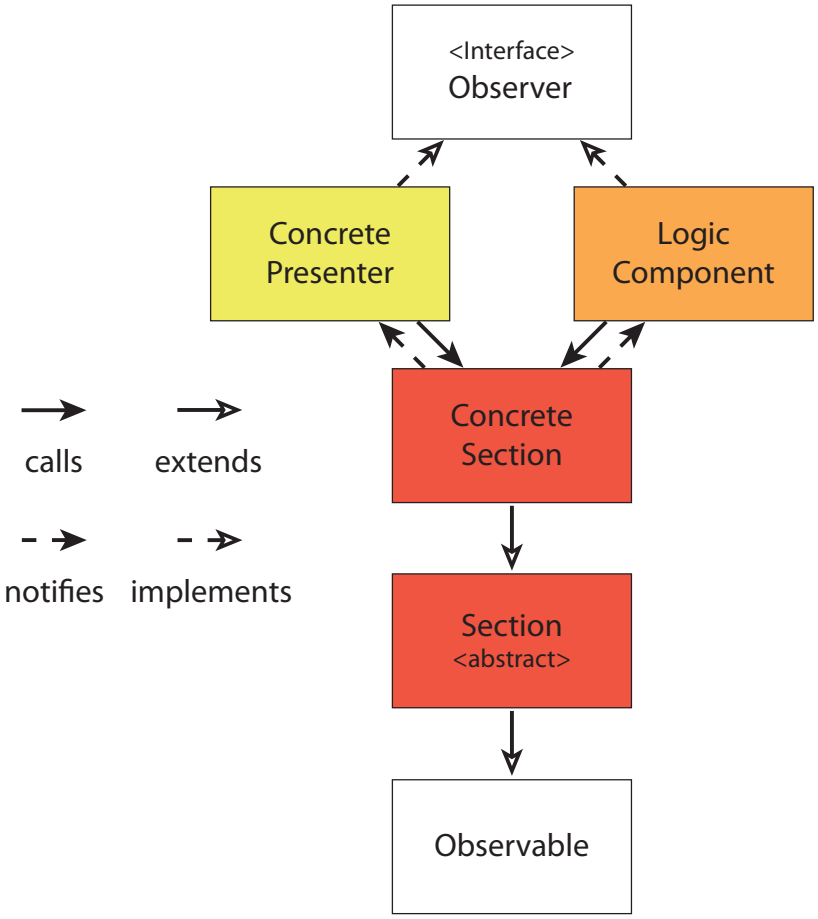


Abbildung 4.5: Illustration des Datenaustausches zwischen Model und den verschiedenen Komponenten.

werden ebenfalls von der Application registriert.

Zur Verwaltung des Models bestehend aus Daten und Logik zur Datenverarbeitung wird die Klasse `GemplayApplication`, welche die Application Klasse erweitert, erstellt. Komponenten können hier hinzugefügt werden. Die Application bietet außerdem Getter-Methoden an, um auf die von ihr verwalteten Komponenten zuzugreifen. Komponenten bieten über public Methoden Schnittstellen an, damit andere Komponenten auf diese mittels der in der Application gespeicherten Instanzen auf diese zugreifen können. Model-Komponenten sollten zwar Zugriff auf die Daten des Models haben, jedoch nicht auf Interface-Komponenten oder andere unabhängige Model-Komponenten. Dies wurde in dieser Implementation jedoch nicht immer berücksichtigt. Um Zugriffe auf Model-Komponenten völlig zu ent-

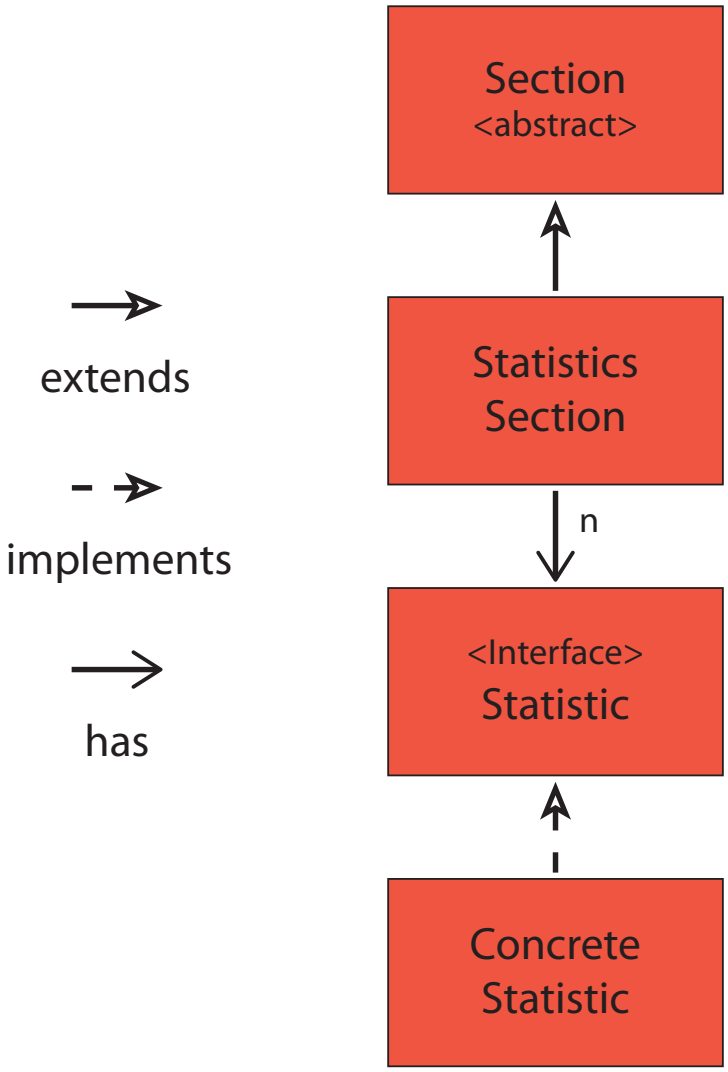


Abbildung 4.6: Illustration des Aufbaus der Daten im Model mittels Basisklassen am Beispiel der Statistik-Sektion.

koppeln sollten Informationen nur von der Application selbst übermittelt werden. Der Zugriff auf die Komponenten mittels Gettern ist fehleranfällig. Ein Interface für Model-Komponenten könnte genutzt werden, um diese zu verwalten und Nachrichten an diese weiterzureichen, ohne direkt auf eine Komponente oder deren Methoden zuzugreifen. Dadurch wären Komponenten leichter austauschbar und könnten geändert, hinzugefügt und entfernt

werden, ohne andere Teile der Anwendung direkt zu beeinflussen. Logik-Komponenten wurden für einzelne Funktionen der Gemplay Anwendung genutzt. Die Grundfunktionen der Anwendung werden von folgenden Komponenten umgesetzt:

- Der `StatisticLogicController` aktualisiert alle vorhandenen Statistiken, wenn ein Task abgeschlossen wurde. Wenn der Status eines Tasks auf `FINISHED` gesetzt wird, überprüft er, um welchen Task es sich handelt anhand dessen `TYPE` und vermittelt ihn dann an eine Logik Komponente, die darauf spezialisiert ist die Statistiken für diesen Task zu aktualisieren. Logik-Komponenten zum Aktualisieren der Statistiken existieren für jeden Task Typ und werden nicht wie andere Logik-Komponenten in der Application registriert, sondern nach Bedarf vom `StatisticLogicController` aufgerufen.
- Der `TaskCreationController` wurde nicht fertiggestellt. Er sollte ursprünglich dazu dienen, Aufgaben zu erzeugen, die dem Profil und den Statistiken des Nutzers oder der Nutzerin entsprechend personalisiert wurden.
- Der `TaskController` behandelt das Starten und Stoppen der Tasks, wenn dies vom Spiel gefordert wird. Er verwaltet außerdem die vom Client angebotenen Services, die zur Durchführung der Tasks nötig sind und startet und stoppt diese mit den Tasks. Services können direkt im `TaskController` hinzugefügt und entfernt werden, je nachdem ob gewisse Features von der Anwendung benötigt werden. Der `TaskController` ist außerdem dafür zuständig Tasks nach deren Abschluss zu aktualisieren. Dazu kann er, ebenso wie jede andere Komponente, Daten aus lokalen Broadcasts der Services nutzen indem er einen lokalen Broadcast Receiver registriert (siehe Abb. 4.7). Nachrichten, die von einem der Services des `TaskControllers` ausgesendet werden, werden durch einen Intent Filter mit der String Konstante `TASK_BROADCAST` markiert. Die verschiedenen Services, die vom `TaskController` verwaltet werden, sind:
  - `TimeService` – Er dient dem Tracking der vergangenen Zeit in Sekunden.
  - `LocationService` – Er dient dem Tracking der GPS Position und der zurückgelegten Strecke.
  - `OrientationService` – Er dient dem Tracking der Orientierung des Geräts.
  - `ActivityIntentService` – Er dient dem Tracking des Aktivitätstyps des Nutzers oder der Nutzerin (still, walking, ...). Der Service wurde aufgrund von Verzögerungen bei dem Erkennen des Aktivitätstyps nicht genutzt. Ursprünglich war er zum Verbessern des Location Tracking gedacht.

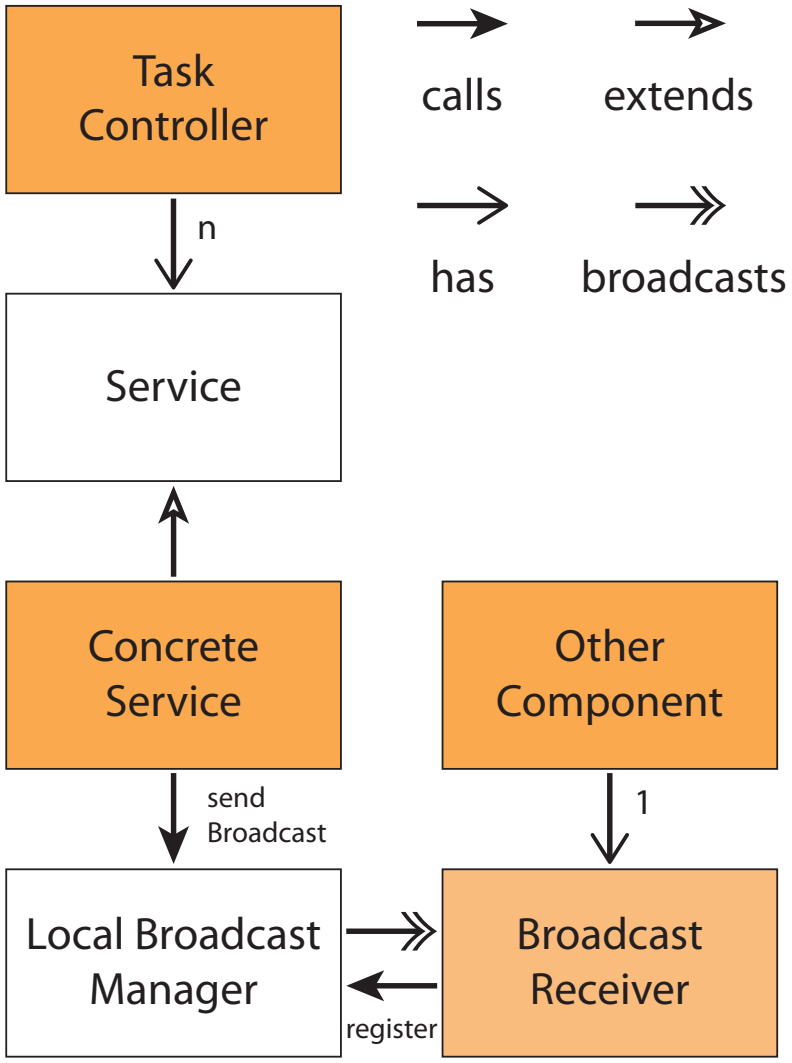


Abbildung 4.7: Illustration der Verwaltung der Services, sowie deren Kommunikation mit anderen Komponenten.

DAO – Datenspeicherung: Bei Start der Application wird das Model instanziiert. Eventuell gespeicherte Daten, werden dabei von der DAO Implementation in das Model gespeichert. Bei Beenden der Anwendung werden die aktuellen Daten des Models von der DAO Implementation gespeichert. Der DAO Layer bietet ein Interface an, über das er vom Model Daten erhalten, beziehungsweise diese an das Model zurückgeben kann. Da das Model

nicht selbst für die Datenspeicherung zuständig ist sondern diese der DAO Implementation überlässt, kann die konkrete Implementation der Datenverwaltung jederzeit ausgetauscht werden, solange sie dem Interface entspricht. Aktuell wird die Datenspeicherung mittels Serializables gelöst. Dies bricht mit der Trennung von Model und Datenspeicherung, da sämtliche Klassen des Models serialisierbar erstellt werden müssen. Um eine Anwendung tatsächlich so flexibel wie möglich zu halten, sollte eine Speicherung rein auf einem separaten Layer erfolgen und sich nicht auf das Model auswirken. Eine solche Umsetzung war geplant, wurde jedoch in diesem Prototypen nicht umgesetzt.

Plugins: Die Spiele selbst werden in separaten Anwendungen in der Form von Plugins realisiert. Sie können dynamisch hinzugefügt und entfernt werden, die Client-Anwendung erkennt dies automatisch und speichert verfügbare Spiele in einer Liste. Da dies ein simpler Prototyp ist, müssen die Spiele manuell heruntergeladen und installiert werden, um erkannt zu werden. Um eine benutzerfreundliche Plugin-Anwendung umzusetzen, sollte der Client jedoch über vorhandene Plugins informiert sein und dem Benutzer oder der Benutzerin diese anzeigen. Der Client kann entweder selbst das Herunterladen für den Benutzer oder die Benutzerin übernehmen, oder auf den Playstore verweisen, sodass dieser oder diese dort Plugins herunterladen kann. Zum verwalten der Plugins dienen folgende Komponenten:

- Der `PluginController` sucht nach Plugins über den Package Manager mittels eines Intent Filters und hört auf Broadcasts die vom System als Reaktion auf Package Events ausgesendet werden. Es findet Activities und Services einer Plugin Anwendung und speichert sie in einem `ExtensibleGame` Objekt. Die gefundenen Spiele werden an das Model übergeben und aktualisiert, wann immer ein neues Spiel installiert oder deinstalliert wird.
- Der `PluginGamesController` selektiert das aktiv laufende Spiel und startet dessen Activity mittels Intents.
- Der `GameCommunicationController` verwaltet die Inter Process Communication (IPC) zwischen Client und Spiele Plugins, indem er auf Broadcasts der Spiele reagiert und im Gegenzug relevante Informationen an die Spiele mittels Broadcasts sendet. Um Daten von Plugins zu erhalten wird von der `BroadcastReceiver` Klasse abgeleitet und mittels eines Intent Filters werden Broadcasts der Plugins herausgefiltert. Services können Informationen mittels lokaler Broadcasts aussenden, die wiederum hier empfangen werden. Globale Broadcasts gehen ausschließlich von dieser einen Klasse aus und werden mittels einer entsprechenden `action` für Plugins erkenntlich gemacht. Dabei werden nur jene Daten gesendet, die für das Plugin relevant sind. Die IPC rein dieser Komponente zu überlassen bringt den Vorteil mit sich, dass de-

ren konkrete Implementierung leicht geändert werden kann. So könnte statt globaler Broadcasts auch ein Messenger implementiert werden. Auch Permissions für die Plugins können hier verwaltet werden.

4.3 Plugin Schnittstelle

In Android wird jedes Programm in einer separaten Applikation in einem eigenen Prozess ausgeführt. Standardmäßig sind diese nicht exportiert und nicht von außen zugänglich. Plugins werden als separate Applikationen umgesetzt, die jedoch exportiert und somit einer Client Anwendung zugänglich gemacht werden. Zur Kommunikation zwischen Client und Plugin ist aufgrund der Struktur von Android Anwendungen die Implementation von Inter Process Communication (IPC) nötig [23].

Plugins können dynamisch hinzugefügt werden und werden vom Client automatisch erkannt. Im Zuge der Verwendung dieses Prototyps, müssen die Plugins manuell vom Benutzer oder der Benutzerin installiert werden. Der Client bietet keine Services an, um verfügbare Plugins zu verlinken oder selbst herunterzuladen. Das Erkennen von installierten Plugin Anwendungen erfolgt mittels des Package Managers und mittels Package bezogenen System-Broadcasts, die über das Installieren und Deinstallieren von Anwendungen informieren. Intents und Intent Filter werden für die Erkennung der Anwendungen und deren Komponenten sowie dem Start der Komponenten genutzt. IPC wird mittels Broadcasts und Broadcast Receivern auf beiden Seiten umgesetzt (siehe Abb. 4.8). Eine Signatur kann absichern, dass tatsächlich nur mit Anwendungen, die vom Entwickler oder der Entwicklerin bestimmt wurden, kommuniziert wird [23, 44, 46]. Die Plugin-Erkennung erfolgt im `PluginController` und wurde entsprechend nach [23, 44, 46] umgesetzt:

- Bei Erstellung der Komponente wird erstmals im Konstruktor eine Liste für die Plugins erstellt.
  - Dazu werden zuerst Plugin Services und Plugin Activties in Listen gespeichert.
  - Zum Erkennen der Plugin Activities wird der Package Manager genutzt. Plugins müssen eine bestimmte `action` unterstützen, um mittels Intent Filters vom Package Manager gesucht werden zu können. Der Befehl `packageManager.queryIntentActivities (baseIntent, PackageManager.GET_RESOLVED_FILTER)` liefert eine Liste von Activities mit der entsprechenden Aktion. Der Package Manager hat Zugriff auf diverse Informationen der Activity, wie den Package Pfad, den Namen, Aktionen und Kategorien. Diese speichert er in ein `GameActivity` Objekt.
  - Dasselbe gilt auch für die Services. Der Package Manager lie-

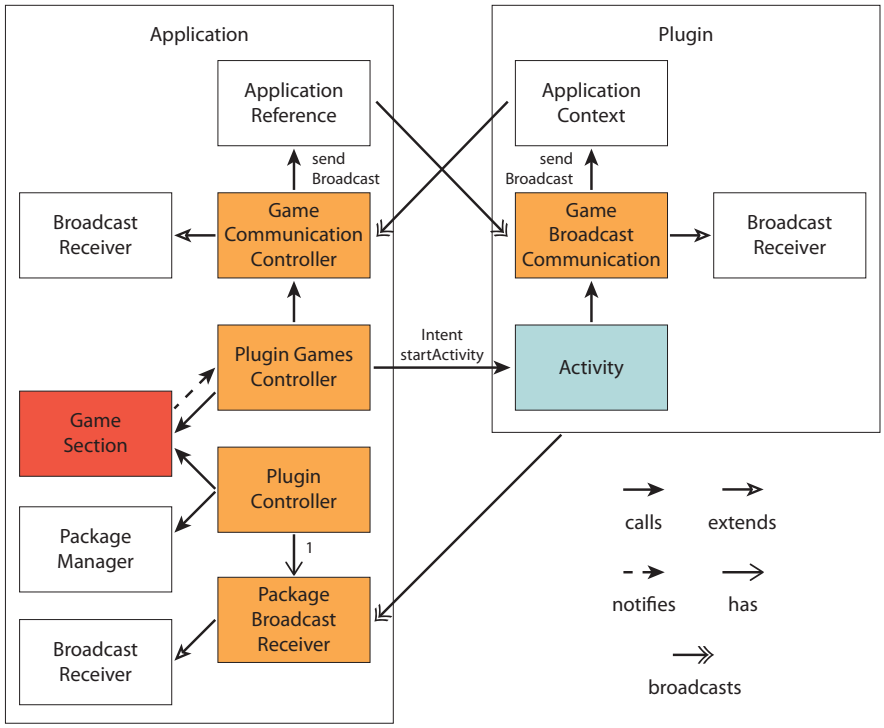


Abbildung 4.8: Illustration der Erkennung der Plugins durch den Plugin Controller, deren Verwaltung im Model, deren Ausführung durch den Plugin Game Controller und der Umsetzung der Kommunikation zwischen Client und Plugin mittels Broadcasts.

fert über den Befehl `packageManager.queryIntentServices( baseIntent, PackageManager.GET_RESOLVED_FILTER)` die Services mit der entsprechenden Aktion und speichert deren Informationen in ein `GameService` Objekt.

– Die gefundenen Activities und Services werden entsprechend ihres Package Pfades in `ExtensibleGame` Objekte gespeichert. Haben eine Activity und ein Service denselben Pfad, so werden sie als zusammengehörig interpretiert und in dasselbe Objekt gespeichert. Ein Plugin sollte nur einen Service und eine Activity exportieren.

- Die gefundenen Spiele werden im Model gespeichert.
- Anschließend wird ein Broadcast Receiver implementiert, der über einen Intent Filter von den Package-Aktionen `ACTION_PACKAGE_ADDED` , `ACTION_PACKAGE_REPLACED` und `ACTION_PACKAGE_REMOVED` benachrichtigt wird.
- Wird `onReceive(Context context, Intent intent)` des Broadcast Receivers aufgerufen, so wird eine neue Liste mit Plugins, wie zuvor

im Konstruktor, erstellt und das Model entsprechend aktualisiert.

Bei der Umsetzung von Plugins können jedoch auch leicht Security Probleme auftreten. Die Umsetzung mit Broadcasts ist anfällig, da diese über das gesamte System ausgestrahlt werden. Auch andere Umsetzungen sind jedoch darauf angewiesen, dass sie tatsächlich mit einem Plugin kommunizieren und nicht mit einer schädlichen Anwendung, die vorgibt ein Plugin zu sein. Generell sollten Plugins für Anwendungen, die sicherheitskritische Daten wie etwa Accounts oder Bankinformationen verwenden, nur mit äußerster Vorsicht genutzt werden. Solche Daten sollten nicht ohne entsprechende Maßnahmen an andere Anwendungen weitergegeben werden [2, 3, 44]. Folgende Schritte können laut [2, 3, 44] Abhilfe schaffen:

- Um Risiken zu verringern, sollten nur jene Komponenten einer Anwendung exportiert werden, die für die IPC zuständig sind.
- Der Client sollte stets die Identität von aufgerufenen Komponenten überprüfen.
- Explizite Intents sollten impliziten Intents vorgezogen werden.
- Normal Permissions sollten von verwendet werden, um Plugins von Drittparteien zu nutzen.
- Signature Permissions sollten genutzt werden, die eigenen Plugins intern entwickelt werden. Diese werden einer Anwendung nur dann gestattet, wenn sie vom selben Entwickler oder derselben Entwicklerin gezeichnet wurde.
- Clients sollten keine Permissions delegieren, und Informationen nur an Anwendungen übergeben, die dazu berechtigt sind diese zu nutzen.
- Bei Aufrufen sollten Plugins ebenfalls die Identität der aufrufenden Partei überprüfen, bevor sie Ergebnisse an diese liefern.

Der Prototyp der Gemplay Anwendung wurde zuerst ohne Permissions und später mit entsprechenden Normal Permissions umgesetzt. Da es sich rein um intern entwickelte Anwendungen zu Studienzwecken handelt, ist Sicherheit für diesen Prototyp nicht die Priorität gewesen. Die Permissions werden im Manifest wie folgt bei Client und Plugin definiert:

```
1 <permission
2   android:name="gemplay.permission.IPC"
3   android:protectionLevel="normal" />
4
5 <uses-permission android:name="gemplay.permission.IPC" />
```

Die Broadcast Receiver, die für die IPC verantwortlich sind, müssen nun diese Permission einfordern. Dies passiert im Client direkt bei der Registrierung des Receivers in der `GemplayApplication`:

```
1 registerReceiver(myGameCommunicationController, myFilter,
      GameCommunicationController.PERMISSION, null);
```

Im Falle des Plugins muss die Permission im Manifest zum Eintrag des Receivers hinzugefügt werden:

```
<receiver
  android:name=".GameBroadcastCommunication"
  android:permission="gemplay.permission.IPC"
  android:enabled="true"
  android:exported="true" >
  <intent-filter>
    <action android:name="gemplay.intent.BroadcastToUnity" />
    <action android:name="gemplay.intent.GPSUpdate" />
    <action android:name="gemplay.intent.ProviderAvailability" />
    <action android:name="gemplay.intent.OrientationUpdate" />
    <action android:name="gemplay.intent.DistanceUpdate" />
    <action android:name="gemplay.intent.TimeUpdate" />
  </intent-filter>
</receiver>
```

Zur Umsetzung der Plugins wurde Unity genutzt. Dort kann qualitativ hohes Game Play mit ansprechenden Grafiken einfach und schnell umgesetzt werden. Außerdem kann so die Entwicklung der Spielelogik sowie die Erstellung von Assets, von der Client Anwendung unabhängig, erledigt werden. Die Entwickler oder Entwicklerinnen der Spiele müssen sich so nicht mit Android Native Programmierung auseinandersetzen. Zur Kommunikation mit dem Client war jedoch eine native Schnittstelle nötig. Diese wird vom Entwickler oder der Entwicklerin des Clients umgesetzt und in Form eines JAR-Pakets bereitgestellt. Dieses kann als Plugin in Unity geladen werden und entsprechend der Dokumentation zu dessen Verwendung in die Unity Anwendung integriert werden. So können Methoden der Schnittstelle aus Unity aufgerufen werden um Zugriff auf Daten zu erhalten, die der Client mittels IPC übermittelt. Die Implementierung der Schnittstelle erfolgt mittels eines Broadcast Receivers. Dieser kann sowohl Broadcasts empfangen, die einen bestimmten Intent Filter erfüllen, als auch selbst Broadcasts an den Client senden. Ein Game Plugin kann außerdem sowohl eine Activity als auch einen Service dem Client zugänglich machen. Vom aktuellen Prototyp wird beides erkannt, jedoch nur die Activity genutzt um das Spiel zu starten. Zusammengehörige Activities und Services werden über den Package Pfad zugeordnet, weswegen dieser derselbe sein sollte [23, 44, 46]. Um eine Anwendung in Unity als Plugin umzusetzen müssen folgende Schritte erfolgen:

- Die Main-Activity der Anwendung muss für den Client zugänglich gemacht werden, sodass dieser das Spiel starten kann. Dazu muss im

Manifest der Anwendung die Activity bei den Attributen `android:exported="true"` gesetzt werden. Außerdem sollte der Activity im Manifest ein Intent Filter hinzugefügt werden, damit das Plugin gefunden wird. Dazu muss ein Intent Filter für die Aktion `android:name="gemplay.intent.action.PICK_PLUGIN"'` definiert werden.

- Zur Kommunikation muss das `GemplayGamePlugin.jar` als Plugin in das Unity Projekt importiert werden. Dieses enthält einen Broadcast Receiver, der ein Interface zur Kommunikation mit dem Client anbietet. Im Broadcast Receiver ist die IPC mit dem Client bereits implementiert. In Unity muss zur Nutzung des Broadcast Receivers der Package Pfad des Projekts dem des Receivers entsprechen. Der Receiver sollte außerdem im Manifest hinzugefügt werden und einen Intent Filter enthalten, der definiert, welche Nachrichten empfangen werden sollen [43]. Dazu muss folgender Eintrag unter den Eintrag der Activity kopiert werden:

```
1 <receiver
2   android:name=".GameBroadcastCommunication"
3   android:enabled="true"
4   android:exported="true" >
5   <intent-filter>
6     <action android:name="gemplay.intent.
      BroadcastToUnity" />
7     <action android:name="gemplay.intent.GPSUpdate" />
8     <action android:name="gemplay.intent.
      ProviderAvailability" />
9     <action android:name="gemplay.intent.
      OrientationUpdate" />
10    <action android:name="gemplay.intent.
      DistanceUpdate" />
11    <action android:name="gemplay.intent.TimeUpdate"
      />
12  </intent-filter>
13 </receiver>
```

Es ist hierbei wichtig den korrekten Pfad innerhalb des Plugin Packages anzugeben wenn der Receiver im Manifest eingetragen wird [43].

- Um den Receiver zu nutzen können verschiedene statische Methoden aufgerufen werden. Das Spiel sollte zuerst überprüfen ob der Client verfügbar ist. Um regelmäßige Benachrichtigungen von den verschiedenen Services des Clients anzufordern muss ein Task gestartet werden. Danach kann mittels Methoden auf die zuletzt erhaltenen Werte zugegriffen werden. Es kann die Verfügbarkeit von Providern, die GPS Position, die zurückgelegte Strecke, die Orientation des Geräts

und der Timer eines laufenden Tasks abgefragt werden. Wenn keine Benachrichtigungen mehr nötig sind, kann der Task beendet werden. Wenn das Spiel beendet wird, kann der Client ebenfalls benachrichtigt werden. Das Interface des Broadcast Receivers ist entsprechend dokumentiert:

```
1 public class GameBroadcastCommunication extends
      BroadcastReceiver
2 {
3
4   //for unity instance creation
5   public static void createInstance()
6
7   //returns the start message - for testing (its "
      Hello Game")
8   public static String getMessageText()
9
10  //not implemented
11  public static Serializable getGameProgress()
12
13  //not implemented
14  public static void setGameProgress(Serializable
      newGameProgress)
15
16  //to check whether a connection with the client is
      available and intents can be sent to it
17  //true after the first intent from the client was
      received
18  //should be checked before other methods are called
19  public static boolean isClientAvailable()
20
21  //starts all client services needed for game tasks
22  //needed to receive updates on gps position and
      orientation
23  public static void startTask()
24
25  //stops all client services needed for game tasks
26  public static void stopTask()
27
28  //tells the client, that the game was closed
29  public static void exit()
30
31  //set the interval for receiving location updates -
      needs to be called before start task to take effect
```

```
32   public static void setLocationUpdateInterval(int
       intervalInMilliseconds)
33
34   //set the accuracy for receiving location updates -
       needs to be called before start task to take effect
35   //accuracy: 0 = no updates, 1 = low battery(approx.
       city), 2 = balanced(approx. block), 3 = high
       accuracy(standard client preset)
36   public static void setLocationAccuracy(int accuracy)
37
38   //true if a gps provider is available and the client
       's location service is running (only after
       starttask)
39   public static boolean hasGPS()
40
41   //true if a network provider is available and the
       client's location service is running (only after
       starttask)
42   public static boolean hasNetwork()
43
44   //java location object!!
45   public static Location getGPSPosition()
46
47   public static double getCurrentLatitude()
48
49   public static double getCurrentLongitude()
50
51   public static double getCurrentOrientationInRadians
       ()
52
53   public static int getDistanceTravelledInMeter()
54
55   public static int getTimer()
56
57 }
```

Ein komplettes Manifest für ein Plugin mit einer Activity und einem Service könnte wie folgt aussehen:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res
      /android"
3   package="com.gemplay.sarah.gemplaygameplugin" >
4
5   <application
```

```
6     android:allowBackup="true"
7     android:icon="@drawable/game"
8     android:label="@string/app_name"
9     android:theme="@style/AppTheme" >
10    <activity
11      android:name=".MainActivity"
12      android:exported="true"
13      android:label="@string/app_name" >
14      <intent-filter>
15        <action android:name="gemplay.intent.action.
      PICK_PLUGIN" />
16        <category android:name="gemplay.intent.category.
      GAME_PLUGIN_DEFAULT" />
17      </intent-filter>
18    </activity>
19    <service
20      android:name=".GameService"
21      android:exported="true"
22      android:label="@string/app_name">
23      <intent-filter>
24        <action android:name="gemplay.intent.action.
      PICK_PLUGIN" />
25        <category android:name="gemplay.intent.category.
      GAME_PLUGIN_DEFAULT" />
26      </intent-filter>
27    </service>
28    <receiver android:name=".GameBroadcastCommunication" >
29      <intent-filter>
30        <action android:name="gemplay.intent.
      BroadcastToUnity" />
31        <action android:name="gemplay.intent.GPSUpdate" />
32        <action android:name="gemplay.intent.
      ProviderAvailability" />
33        <action android:name="gemplay.intent.
      OrientationUpdate" />
34        <action android:name="gemplay.intent.DistanceUpdate
      " />
35        <action android:name="gemplay.intent.TimeUpdate" />
36      </intent-filter>
37    </receiver>
38  </application>
39
40 </manifest>
```

Der Service würde von dem Client erkannt und gespeichert, wird jedoch im aktuellen Prototyp nicht genutzt.

Kapitel 5

Diskussion

Der GEMPLAY Client wurde im Laufe einer Studie in Verbindung mit zwei Spielen, die separat entwickelt wurden, getestet. Während der Implementation und Nutzung wurden Vor- und Nachteile des Prototyps notiert und mögliche Verbesserungspunkte der Anwendung ausfindig gemacht. Anhand dessen werden die einzelnen Teile der Implementation in Bezug auf deren Auswirkung auf die Änderungsfreundlichkeit und Erweiterbarkeit der Anwendung untersucht. Außerdem wird die umgesetzte Anwendung hinsichtlich deren Nutzungsfreundlichkeit und Bedienbarkeit betrachtet.

5.1 Implementation

Die Implementation im Zuge des Projekts zeigt einige der Vor- und Nachteile der vorgestellten Architektur auf. Während der Implementation musste flexibel auf sich ändernde Ansprüche an die Anwendung seitens des Projekts reagiert werden. Dabei wurde schnell klar welche Teile der Architektur besonders einfach anpassbar und erweiterbar sind und welche Teile problematisch oder verbesserungsfähig sind.

5.1.1 Generelle Architektur

Die gewählte Architektur funktioniert grundsätzlich gut und die Umsetzung mittels Komponenten ermöglicht es den Anforderungen entsprechend, auf Änderungen während des Projekts einzugehen, ohne die bereits existierenden Komponenten zu beeinträchtigen. So muss bereits existierender funktionstüchtiger Code beim Hinzufügen neuer Features nicht oder kaum geändert werden. Einzelne Funktionen können den Ansprüchen entsprechend zum Client hinzugefügt oder wieder entfernt werden. Entfernte Komponenten können aufgehoben und wenn nötig wiederverwendet werden.

Auch das Umsetzen mancher Komponenten mittels Services und die Nutzung eines Service Managers erwiesen sich als praktisch. Diese von Android

angebotene Komponente bietet sich für das Abarbeiten von Background Tasks an. Mittels des Service Managers können diese Aufgaben einfach verwaltet und nach Belieben gestartet und gestoppt werden. Neue Services können aufgrund der gemeinsamen Schnittstelle der Service Basisklasse leicht hinzugefügt werden. Das Hinzufügen der Services direkt in der Implementierung des Service Managers ist jedoch verbesserungswürdig.

Während Service Komponenten völlig modular umgesetzt wurden und ausschließlich mittels lokaler Broadcasts zu anderen Komponenten kommunizieren können, erfolgt die Kommunikation mit Komponenten die zur Datenverarbeitung des Models gehören über deren öffentliche Methoden. Diese Methoden sind jedoch über das Model öffentlich zugänglich und werden beispielsweise von Activities der Anwendung aufgerufen. Bei Änderungen an den Komponenten des Models würde dies Auswirkungen außerhalb der Komponenten selbst mit sich ziehen. In einer modularen Architektur sollte die Übermittlung von Daten an die Komponenten einheitlich an einer Stelle erfolgen. Ein weiteres Problem, das die Kapselung der Komponenten beeinträchtigt, ist die Registrierung von Receivern seitens der Komponenten. Da dies in Android nur für bestimmte Komponenten wie Activities oder die Application möglich ist, können Komponenten nicht selbst ihre Receiver registrieren. Receiver müssen also explizit für solche Komponenten hinzugefügt werden und bei Änderungen an den Komponenten oder beim Entfernen der Komponenten berücksichtigt werden.

Die Umsetzung der Datenverwaltung seitens des Models erwies sich ebenfalls als verbesserungsfähig. Der Zugriff auf Daten erfolgt über eine Hierarchie von Klassen, die mittels öffentlicher Methoden direkten Zugriff auf Daten ermöglicht. Dem Modell fehlt dabei ein einheitliches und übersichtliches Interface, das den Zugriff auf einzelne Daten abstrahiert und den Austausch der Implementierung ermöglicht. Zwar wurden Teile des Models mittels Schnittstellen umgesetzt, was die Änderung an einzelnen Modelkomponenten leicht durch deren Ableitung ermöglicht, dennoch ist die Verwaltung starr und koppelt das Modell direkt an alle Komponenten die darauf Zugriff haben. Obwohl eine Komponentenarchitektur als Grundlage dienen sollte entspricht das Model eher einer Schichtenarchitektur. Dabei kann aus der obersten Schicht, die aus den dem ursprünglichen Konzept entsprechenden Komponenten für Interface und Logik besteht, direkt auf alle unterliegenden Modelschichten zugegriffen werden.

Die Speicherung der Daten im Zuge des Prototyps wurde nur schematisch umgesetzt. Die Nutzung von Serializable beeinflusst hier direkt das Model, da jede einzelne Komponente des Models serialisierbar umgesetzt werden muss. Änderungen am Model beeinflussen so direkt den DAO Layer.

Die DLC Pipeline wurde im Zuge des Prototyps nicht umgesetzt, da die Testanwendung nicht viele Ressourcen benötigte und die Personalisierung der Anwendung rein über Plugins umgesetzt wurde.

5.1.2 MVP

Die Umsetzung der Interface Komponenten mittels MVP hat eindeutig Vorteile mit sich gebracht. Da diese im Laufe der Entwicklung besonders oft ausgetauscht und verändert werden und ständig neue Interface Elemente hinzugefügt werden, hat es sich als sinnvoll erwiesen diese entsprechend möglichst schlank zu halten. Datenverarbeitungsaufgaben wurden soweit es möglich war in das Model und dessen Logik-Komponenten ausgelagert. So bleiben diese Verarbeitungsschritte erhalten und können im Hintergrund passieren, auch wenn das Interface für Teile der Anwendung reduziert, verändert oder entfernt wird. Presenter Komponenten enthalten minimale Logik und dienen lediglich dazu, die Benutzerinteraktion des zugehörigen View Elements zu interpretieren und an das Model zur Verarbeitung weiterzureichen. So wurde in der Gemplay Anwendung die Logik zum Starten der Tasks an eine eigene reine Logik-Komponente weitergegeben. Tasks können nicht nur über den Benutzer oder die Benutzerin mittels des ursprünglich geplanten Userinterfaces gestartet werden, sondern auch im Hintergrund an anderen Stellen der Anwendung, wie etwa über die Spiele. Dies entkoppelt die Funktion des Startens von Tasks, von der Interface Komponente zum Anzeigen und Starten der Tasks und erleichtert so Änderungen an der Logik ebenso wie an der Darstellung der Anwendung. Die Umsetzung des Einbaus von Interface Komponenten mittels Activities funktioniert ebenfalls sehr gut und ermöglicht es Teilkomponenten zu kombinieren. Die Kommunikation der Komponenten mittels Activities ist jedoch eher unpraktisch. Activities müssen so Befehle der Komponenten interpretieren und mittels Methodenaufrufe an andere Komponenten weitergeben. Dazu nutzen die Activities direkt die Methoden einzelner Komponenten innerhalb der Activity selbst sowie der Komponenten in der Application. Dies würde durch Änderungen an dem Aufbau der Anwendung sofort beeinflusst werden. Activities, welche für die Verwaltung und das Layout des Benutzerinterfaces zuständig sind, erhalten so Aufgaben, die die Logik der Anwendung betreffen. Dies entspricht nicht dem Single Responsibility Prinzip, das von Komponenten eingehalten werden sollte [45].

5.1.3 Plugins

Die Nutzung eines Plugin Systems wurde hauptsächlich als positiv empfunden und brachte einige Vorteile bei der Entwicklung der GEMPLAY Anwendung mit sich. Feedback seitens des Entwicklers der Spiele [1] verweist folgende Vorteile der Implementierung:

- Ein Objekt stellt die gesamte Schnittstelle zwischen Spiel-Prototyp und Client dar.

[1] Daniel Rammer, FH OÖ Forschungs- und Entwicklungs GmbH

- Die klare Trennung schafft einen größeren Abstand zwischen Hardware und Spiel - der Fokus während der Entwicklung konnte somit stärker auf die eigentlichen Spielmechaniken gerichtet werden.
- Große Flexibilität. Diese Schnittstelle kann nach Belieben erweitert werden - ist während der Entwicklung auch mehrmals geschehen.
- Modularität: Der modulare Aufbau hat sich als äußerst praktisch erwiesen. Wenn in Zukunft noch weitere Prototypen entwickelt werden, so ist die Weiterverwendung des Client zu empfehlen.

Die Implementierung der Plugins funktioniert seitens des Clients ausgezeichnet. Eine eigene Komponente dient dazu Plugins zu erkennen, sodass diese jederzeit dynamisch erkannt werden. Über den Plugin Manager können entsprechende Plugins gestartet werden und eine eigene Komponente ist für die Kommunikation mit den Plugins zuständig. Die Umsetzung mit nativem Code funktioniert ohne Probleme, die Schnittstelle erfolgt mittels festgelegter Nachrichten, die über Broadcasts gesendet werden. Neue Broadcasts können jederzeit hinzugefügt werden, wenn die Schnittstelle erweitert werden soll.

Die Umsetzung der Schnittstelle in Verbindung mit Unity erwies sich jedoch als problematisch. Die Schnittstelle kann zwar, als Jar-Paket exportiert, in Unity importiert und verwendet werden, jedoch muss der Package Pfad der Schnittstelle mit dem des Unity Projekts übereinstimmen. Es kann jedoch nur eine Anwendung mit diesem Pfad installiert werden, da eine andere Anwendung mit demselben Pfad bei der Installation von Android als dieselbe Anwendung identifiziert wird und somit die alte Anwendung überschrieben wird. Dies macht es unmöglich mehrere Unity Anwendungen mit demselben Jar-Paket der Schnittstelle umzusetzen. Für jedes Plugin muss explizit ein Jar-Paket mit dem entsprechenden Package Pad des Plugins erstellt und exportiert werden. Dies führt zu Mehraufwand und macht den Entwickler oder die Entwicklerin der Plugins abhängig davon, mittels nativem Android direkt die Schnittstelle zu bearbeiten. Die Schnittstelle sollte universell für beliebig viele Plugins nutzbar sein, eine solche entsprechende Lösung wurde jedoch im Laufe des Projekts für eine Umsetzung mit Unity nicht gefunden.

5.2 Nutzungsfreundlichkeit und Usability

Der GEMPLAY Client wurde im Laufe der Studie in Verbindung mit mehreren Spielen getestet. Bei mehreren Testläufen wurden Client und Spiele entsprechend dem Feedback der Testpersonen angepasst. Für die Studie

selbst wurden neben dem Client eine Einleitung in die Spielewelt und verschiedene Versionen der Spiele installiert. Von unterschiedlichen Testpersonen wurden neben der Einleitung die Spiele in verschiedenen Versionen und unterschiedlicher Reihenfolge durchgespielt. Einstellungen zur Aktivierung, Deaktivierung und Reihung der Spiele wurden nicht genutzt, da man die Spiele ohne ständig umstellen zu müssen schneller starten konnte. Studienteilnehmer und Studienteilnehmerinnen kamen nie mit dem Client selbst in Kontakt, so konnten nicht versehentlich Spiele gestartet werden die nicht genutzt wurden. Da immer alle Spiele installiert und sichtbar waren wurde die Personalisierung also nicht genutzt oder getestet. Spiele wurden entsprechend einer Liste vom Studienleiter gestartet. Die Reihenfolge und welche der Spiele gestartet wurden war nicht von der Anwendung sondern von dem Studienleiter ausgehend.

Während der Studie gab es mit der Anwendung die Architektur betreffend keine Probleme. Die Umsetzung der Plugins war zufriedenstellend. Probleme betrafen hauptsächlich die Spiele selbst, sowie das vom Client gelieferte GPS, welches als zu ungenau empfunden wurde. Es war jedoch beim Testen der Spiele teilweise schwer nachzuvollziehen, wenn Bugs auftraten, ob der Client oder das Spiel für ein Problem verantwortlich war.

Generell hat sich die Anwendung im Laufe der Studie als praktisch erwiesen, der modulare Aufbau ermöglicht das Hinzufügen beliebiger Plugins die nach Bedarf eingeblendet, ausgeblendet und verschieden angeordnet werden können. Es bestand seitens der Studienleitung wenig Erfahrung mit ähnlichen Systemen und die Erwartungen an die Anwendung waren sehr gering. Der Prototyp wurde ohne Probleme entsprechend der mitgelieferten Anleitung verwendet. Da bei Aktualisierungen der einzelnen Spiele nur einzelne Plugins aktualisiert und nicht die ganze Anwendung neu installiert werden musste, konnte für kleine Änderungen Zeit gespart werden. Einzelne Spiele waren die Dateigröße betreffend eher klein, Downloadzeiten kurz und die Anwendung schnell aktualisiert. Aus diesem Grund wurde die Architektur als für die Anforderungen der Studie sehr passend empfunden. Wenn neue Spiele hinzugefügt wurden, konnte man diese einfach installieren und sofort mit dem bereits installierten Client nutzen. Das System des Plug-and-play ist im Fall des GEMPLAY Projekts ideal, da dem Client jederzeit neue für die Studie nötige Spiele hinzugefügt werden können. Ob die Umsetzung mittels Plugins für die Personalisierung einer tatsächlichen Anwendung geeignet ist kann jedoch nicht bestimmt werden, da dies im Laufe der Studie nicht getestet wurde [2].

[2]Daniel Rammer, FH OÖ Forschungs- und Entwicklungs GmbH; Jakob Rehm, AIT Austrian Institute of Technology GmbH

5.3 Ausblick und Erweiterungsmöglichkeiten

Die aktuelle Architektur hat einige Vorteile mit sich gebracht und ist generell sehr geeignet um einfache Änderungen und Erweiterungen umzusetzen. Sie ist jedoch durchaus noch verbesserungsfähig und es gibt noch einige Schwachstellen die beachtet werden sollten.

Folgende Änderungen sollten für eine Verbesserung der Architektur in Betracht gezogen werden:

- Die Umsetzung der Kommunikation zwischen Komponenten sollte einheitlich mittels Messages oder Broadcasts erfolgen. Dies betrifft sowohl Interface Komponenten als auch Logik Komponenten. Die Logik der Komponenten selbst sollte die Nachrichten oder Events empfangen und interpretieren können. Nachrichten die das Interface betreffen können von der Activity an andere vorhandene Interface Komponenten weitergereicht werden, unabhängig davon welche Komponenten vorhanden sind und ob sie die Nachricht interpretieren können. Dies sollte über eine einheitliche Schnittstelle seitens dieser Komponenten erfolgen.
- Activities sollten weniger logische Aufgaben übernehmen und lediglich als Layout für Interface Komponenten dienen. Auch sollte sie keinen Zugriff mehr auf externe Komponenten haben.
- Neue Services sollten nicht im Service Manager implementiert werden, sondern über dessen Schnittstelle. Mittels einer Methode soll über die Application das Hinzufügen neuer Services erfolgen. Es sollte außerdem möglich sein über die Schnittstelle des Service Managers verschiedene Services einzeln zu starten. Dies würde die Erweiterung durch neue Services und deren Nutzung durch die Anwendung erleichtern.
- Die Application kann Nachrichten seitens der Activity an ihre Komponenten weitervermitteln, sollte diese jedoch nicht nach außen freigeben.
- Komponenten, die Receiver nutzen, sollten einheitlich verwaltet werden. Über eine klare Schnittstelle sollten die nötigen Informationen für die Registrierung des Receivers nach außen an die Activity oder Application weitergegeben werden. Diese wiederum sollte solche Komponenten automatisch registrieren, indem sie diese erkennt und über deren Schnittstelle nötige Informationen aufruft oder indem die Activity oder die Aplication selbst eine Schnittstelle anbietet über die Komponenten mittels Methodenaufrufs mit den nötigen Informationen die Registrierung anfordern können. So erfolgt die Registrierung automatisch, nicht mehr manuell in der Application oder Activity, was eine einfachere Austauschbarkeit der Komponenten ermöglicht.
- Abhängigkeiten zwischen dem Model und den Komponenten sollten genauer betrachtet werden. Das Close-Coupling zwischen dem Model und anderen Komponenten ist ein Grund für erhöhte Fehleranfälligkeit bei Änderungen. Bei Änderung des Models würde es zu Problemen kom-

men, daher sollte das System überarbeitet werden, sodass Änderungen und Erweiterungen auch hier leichter umgesetzt werden können. Das Model sollte entsprechend des Open-Closed-Prinzips überarbeitet werden. Dieses ist im Moment nur bei einzelnen Teilen des Models umgesetzt. Schnittstellen der Model Komponenten sollten nicht mehr geändert werden um Probleme mit Abhängigkeiten zu verhindern, da andere Komponenten auf das Model direkt Zugriff haben. Ableitungen von Basiskomponenten können jedoch genutzt werden, wenn neue Funktionen zu einer Modelkomponente hinzugefügt werden sollen. Es sollte eine einheitliche und übersichtliche Schnittstelle bereitgestellt werden. So ist die zu Grunde liegende Implementierung jederzeit austauschbar. Mehrere Schnittstellen für unabhängige Teile des Models, auf die mittels einer Komponente zu deren Verwaltung zugegriffen werden kann, würden die Erweiterung des Models erleichtern.

- Die Schnittstelle für die Kommunikation mit Unity kann verbessert werden. Offensichtlich ist die Umsetzung in Abhängigkeit des Package Pfades sehr unpraktisch, daher sollte entsprechend recherchiert werden, ob dies anders umgesetzt werden kann. Entsprechend dem Interface Segregations-Prinzip wäre es möglich statt dem Ändern der existierenden Schnittstelle neue hinzuzufügen, wenn Plugins unterschiedlichere Anforderungen haben sollten. Dies war im Fall des GEMPLAY Projekts nicht nötig. Außerdem kann statt der IPC mittels Broadcasts ein Messenger eine elegante Lösung sein, um die Kommunikation zwischen zwei Komponenten zu handhaben. Dabei können mittels Nachrichten einfach verschiedene Daten übermittelt werden. Die Kommunikation kann leicht erweitert werden, indem neue Nachrichten definiert werden. Permissions, wenn möglich „signed“, sollten ebenfalls in Betracht gezogen werden, um die Kommunikation sicher zu gestalten.
- Eine passenden DLC Pipeline für Anwendungen deren Schwerpunkt auf Ressourcen liegt sollte ebenfalls implementiert werden.

Generell sollte die Nutzungsfreundlichkeit der Anwendung in Bezug auf die Erweiterung und Aktualisierung noch im Laufe einer Studie ausführlicher überprüft werden. Sowohl die Personalisierung durch Plugins als auch durch DLC sollte noch mit tatsächlichen Nutzern und Nutzerinnen getestet werden.

Kapitel 6

Schlussfolgerungen

Architekturmuster sind sehr hilfreich bei der Umsetzung von Applikationen und ermöglichen auch für die Android Plattform eine bessere Strukturierung von Anwendungen. Durch die Kombination der korrekten Muster entsprechend den Anforderungen der Anwendung kann die Flexibilität und Erweiterbarkeit verbessert werden. Dies erleichtert Entwicklern und Entwicklerinnen die Umsetzung von Änderungen und das Hinzufügen neuer Features und bietet außerdem Möglichkeiten Anwendungen von Benutzern und Benutzerinnen selbst personalisieren zu lassen. Bei einer solchen Umsetzung müssen jedoch Eigenheiten der Plattform beachtet werden. Dies betrifft vor allem Interface Komponenten, die von Android vorgegeben werden. Ansonsten funktioniert die native Umsetzung generell wie in der aus Eclipse bekannten Java Umgebung. Nicht alle Muster sind für die Plattform gleich gut geeignet. Die meisten Muster zur generellen Strukturierung der Architektur sind gut auf Android umsetzbar. MVC hat sich als schwierig und wenig intuitiv herausgestellt, während MVP gut funktioniert hat und nicht sehr schwer umzusetzen war. Die Umsetzung von MVVM wäre eine Möglichkeit, wurde jedoch wegen Bindings nicht berücksichtigt, da dies von Android standardmäßig nicht unterstützt wird. Durch die Nutzung von Frameworks können solche Einschränkungen umgangen werden, man wird dadurch jedoch davon abhängig, dass nötige Features von besagtem Framework unterstützt werden.

Die Architektur, die entsprechend der Vorlage verschiedener Software Muster entworfen wurde, bietet generell eine gute Grundlage zur Entwicklung von Anwendungen die möglichst flexibel anpassbar und leicht zu erweitern sein sollen. Die Implementation funktionierte sehr gut und erwies sich generell als flexibel, wenn auch einige Punkte klar verbesserungsfähig sind. Vor allem die Erweiterung einer Anwendung ist mit den passenden Ansätzen leicht umsetzbar. Neben der Standardlösung der Nutzung einer entsprechenden DLC Pipeline hat sich auch das Plugin Muster als praktisch erwiesen. Es wurden nicht alle Muster im Zuge der Test-Implementation umgesetzt,

da der Aufwand für einen Prototypen dadurch sehr angestiegen wäre. Außerdem sind nicht alle Muster für jede Anwendung gleich sinnvoll, es sollte daher nicht eine konkrete Architektur umgesetzt werden, sondern stets eine auf die Anwendung zugeschnittene Abwandlung. Für Anwendungen, die viele Ressourcen enthalten, macht eine DLC Pipeline Sinn, während für eine personalisierbare Anwendung ohne viele Ressourcen, wie etwa den Client des GEMPLAY Projekts, der Plugin Ansatz allein ausreichend ist. Die modulare Entwicklung ist extrem hilfreich bei der Anpassung und Änderung der Anwendung im Laufe der Entwicklung und beim Hinzufügen neuer Features zu der Grundanwendung. Vor allem der Austausch von Interface Komponenten unabhängig von der Logik der Anwendung ist von großem Vorteil. Die Architektur ist nach wie vor verbesserungsfähig. Außerdem sollte die modulare Umsetzung überarbeitet werden. Diese wurde für den Prototyp nicht komplett eingehalten. Es sollte überdacht werden, wann die komplett modulare Umsetzung für eine Anwendung und der damit verbundene Entwicklungsaufwand Sinn machen. Ein Framework, eine Bibliothek oder ein Plugin, könnten Grundfunktionen zur Entwicklung nativer Anwendungen mittels Komponenten zur Verfügung stellen und so Abhilfe schaffen. Generell hat sich herausgestellt dass die Nutzung von Software Mustern zwar enorm zur Flexibilität einer Android-Anwendung beitragen kann, dennoch sind Muster auch hier keine Allzwecklösung. Während bestimmte Muster besonders gut für Flexibilität und Erweiterbarkeit einsetzbar sind, so sind doch nicht alle für jede Anwendung gleich ideal. Es sollte aus den vorhandenen Mustern für jede Anwendung die passende Architektur zusammengestellt werden, um ideale Ergebnisse zu erzielen.

Quellenverzeichnis

Literatur

[1] Frank Buschmann u. a. *Pattern-oriented Software Architecture: A System of Patterns*. New York: John Wiley & Sons, Inc., 1996 (siehe S. 6–9, 13, 14, 18, 28–30).

[2] Erika Chin u. a. „Analyzing inter-application communication in Android“. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. (Bethesda, MD, USA). New York, NY, USA: ACM, 2011, S. 239–252 (siehe S. 20, 50).

[3] Adrienne Porter Felt u. a. „Permission re-delegation: Attacks and defenses“. In: *Proceedings of the 20th Usenix Security Symposium*. (San Francisco, CA, USA). Berkeley, CA, USA: USENIX Association, 2011, S. 331–346 (siehe S. 50).

[4] Joachim Goll und Manfred Dausmann. *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java*. Wiesbaden: Springer-Verlag, 2013 (siehe S. 5–14, 28, 29, 33).

[5] Clemens Kern. „Media Entwicklung eines Realtime Game-Editors unter besonderer Berücksichtigung der Stabilität und Erweiterbarkeit“. Diplomarbeit. Stuttgart, Deutschland: Hochschule der Medien, Computer Science und Media, Sep. 2010 (siehe S. 33, 34).

[6] Dia Kharrat und Syed Salman Quadri. „Self-registering plug-ins: an architecture for extensible software“. In: *Proceedings of the Canadian Conference on Electrical and Computer Engineering*. (Saskatoon, Sask, Canada). Washington, DC, USA: IEEE, 2005, S. 1324–1327 (siehe S. 33, 34).

[7] Jianye Liu und Jiankun Yu. „Research on Development of Android Applications“. In: *Proceedings of the 4th International Conference on Intelligent Networks and Intelligent Systems*. (Kunming, China). Washington, DC, USA: IEEE, 2011, S. 69–72 (siehe S. 20–22).

[8] Lennart E. Nacke, Chris Bateman und Regan L. Mandryk. „BrainHex: A neurobiological gamer typology survey“. In: *Entertainment Computing* 5.1 (Jan. 2014), S. 55–62 (siehe S. 36).

[9] Yuri Natchetoi, Viktor Kaufman und Albina Shapiro. „Service-oriented architecture for mobile applications“. In: *Proceedings of the 1st International Workshop on Software Architectures and Mobility - SAM '08*. (Leipzig, Germany). New York, NY, USA: ACM Press, 2008, S. 27–32 (siehe S. 10, 11, 29).

[10] Godfrey Nolan, Onur Cinar und David Truxall. *Android Best Practices*. New York: Apress, 2014 (siehe S. 17).

[11] Karsten Nolte. „Native App Design Entwicklungsmethoden und deren Einfluss auf die Usability“. Diplomarbeit. Gelsenkirchen, Deutschland: Westfälische Hochschule, Fachbereich Informatik und Kommunikation, Feb. 2013 (siehe S. 24–28).

[12] Jeff Plummer. „A flexible and expandable architecture for computer games“. Diplomarbeit. Tempe, Arizona, United States: Arizona State University, Dez. 2004 (siehe S. 7, 9, 10, 28, 29).

[13] Muktamyee Sarker. „An overview of Object Oriented Design Metrics“. Diplomarbeit. Umeå, Sweden: Umeå University, Department of Computer Science, Juni 2005 (siehe S. 6).

[14] Karina Sokolova, Marc Lemercier und Ludovic Garcia. „Android Passive MVC: a Novel Architecture Model for the Android Application Development“. In: *Proceedings of the Fifth International Conference on Pervasive Patterns and Applications (PATTERNS'13)*. (Valencia, Spain). IARIA, 2013, S. 7–12 (siehe S. 13, 14, 16, 17, 30, 31, 33, 34, 38).

[15] Clemens Szyperski, Dominik Gruntz und Stephan Murer. *Component Software: Beyond Object-oriented Programming*. 2. Aufl. New York: ACM Press/Addison Wesley Publishing Co., 2002 (siehe S. 9).

[16] Do Van Thanh und Ivar Jørstad. „A service-oriented architecture framework for mobile services“. In: *Proceedings of the Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/E-Learning on Telecommunications Workshop (AICT/SAPIR/ELETE'05)*. (Lisbon, Portugal). Washington, DC, USA: IEEE, 2005, S. 65–70 (siehe S. 10, 11, 29).

[17] Volker Wulf, Volkmar Pipek und Markus Won. „Component-based tailorability: Enabling highly flexible software applications“. In: *International Journal of Human Computer Studies* 66.1 (2008), S. 1–22 (siehe S. 9, 10, 29).

[18] Mussab Zneika u. a. „Towards a Modular and Lightweight Model for Android Development Platforms“. In: *Proceedings of the 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social*

Computing, GreenCom-iThings-CPSCom 2013. (Beijing, China). IEEE. Washington, DC, USA, 2013, S. 2129–2132 (siehe S. 25, 26, 29).

Online-Quellen

[19] URL: http://www.usability.gov (siehe S. 1).

[20] URL: http://stackoverflow.com/questions/6241757/is-it-appropriate-to-use-osgi-framework-on-small-java-app (siehe S. 26).

[21] *Android*. URL: www.android.com (siehe S. 19).

[22] *Android - Messenger vs Binder IPC*. URL: http://debuggingisfun.blogspot.co.at/2014/03/android-messenger-vs-binder-ipc.html (siehe S. 22).

[23] *Android Developers – API Guides*. URL: http://developer.android.com/guide/index.html (siehe S. 20–22, 48, 51).

[24] *Design | Android Developers*. URL: http://developer.android.com/design/index.html (siehe S. 1).

[25] *Designing a User Interface(Windows)*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff728820%28v=vs.85%29.aspx (siehe S. 1).

[26] *Developing for Android: PhoneGap versus Native*. 2015. URL: https://www.lennu.net/developing-for-android-phonegap-versus-native/ (siehe S. 24).

[27] Sapan Diwakar. *Titanium vs Phonegap vs Native application development*. 2012. URL: http://sapandiwakar.in/api-research-study-iphone-and-android-applications/ (siehe S. 26–28).

[28] Michael Eckel. *Component Technologies on Google Android*. URL: https://homepages.thm.de/~hg51/Veranstaltungen/MasterSeminar1011/MichaelEckel.pdf (siehe S. 26–28, 33).

[29] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: http://www.martinfowler.com/articles/injection.html (siehe S. 6).

[30] Aleksandar Gargenta. *Deep Dive into Android IPC/Binder Framework – Android Builders Summit 2013*. 2013. URL: https://youtu.be/Jgampt1DOak (siehe S. 22).

[31] Marko Gargenta. *Architecting Android Apps – AnDevCon IV*. 2012. URL: https://youtu.be/x7gUSunWBSM (siehe S. 20–22).

[32] *GEMPLAY*. URL: www.gemplay.at (siehe S. 37).

[33] Rob Gravelle. *Comparing Mobile Development Framework Types*. URL: http://www.htmlgoodies.com/html5/mobile/comparing-mobile-development-framework-types.html (siehe S. 26–28).

[34] Phil Haack. *Everything You Wanted To Know About MVC and MVP But Were Afraid To Ask*. 2008. URL: http://haacked.com/archive/2008/06/16/everything-you-wanted-to-know-about-mvc-and-mvp-but.aspx/ (siehe S. 31).

[35] Hao He. *What Is Service-Oriented Architecture*. 2003. URL: http://www.xml.com/pub/a/ws/2003/09/30/soa.html (siehe S. 6).

[36] *Interactive Application Architecture Patterns*. 2007. URL: http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/ (siehe S. 16, 31).

[37] *iOS Human Interface Guidelines: Designing for iOS*. URL: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/ (siehe S. 1).

[38] Usman Ismail. *How to build great downloadable content pipelines*. 2013. URL: http://techtraits.com/dlchowto/ (siehe S. 32, 34).

[39] *ISO 14915-1:2002 – Software ergonomics for multimedia user interfaces – Part 1: Design principles and framework*. 2002. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=25578 (siehe S. 1).

[40] *ISO 9241-11:1998 – Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability*. 1998. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=16883 (siehe S. 1).

[41] Bernard Kohan. *Native vs Hybrid / PhoneGap App Development Comparison*. 2015. URL: http://www.comentum.com/phonegap-vs-native-app-development.html (siehe S. 26–28).

[42] Robert C. Martin. *Design Principles and Design Patterns*. 2000. URL: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf (siehe S. 6).

[43] Jean Meyblum. *Communication between an Android App and Unity*. 2014. URL: http://jeanmeyblum.weebly.com/scripts--tutorials/communication-between-an-android-app-and-unity (siehe S. 52).

[44] Mark Murphy. *Plugin Architectures for Android – XDA:DevCon 2013*. 2013. URL: https://youtu.be/Xu8Z_3TaWuE (siehe S. 21, 33, 48, 50, 51).

[45] Joshua Musselwhite. *Android Architecture: Part 1, Intro – Android Developer*. 2011. URL: http://www.therealjoshua.com/2011/11/android-architecture-part-1-intro/ (siehe S. 30, 31, 59).

[46] Gábor Paller. *Plugins*. 2010. URL: http://mylifewithandroid.blogspot.co.at/2010/06/plugins.html (siehe S. 22, 48, 51).

[47] Helmut Petritsch. *Service-Oriented Architecture (SOA) vs. Component Based Architecture*. 2006. URL: http://petritsch.co.at/download/SOA_vs_component_based.pdf (siehe S. 29).

[48] Nico Prins. *The Evolution Of Mobile App Design*. 2015. URL: http://usabilitygeek.com/the-evolution-of-mobile-app-design/ (siehe S. 2).

[49] Ralf Schneeweiß. *Komponentenbasierte Entwicklung von Embedded Software*. 2012. URL: http://www.oop-trainer.de/Themen/EmbeddedKomponenten.html (siehe S. 9, 29).

[50] Greg Shackles. *Building Android Apps with MVVM and Data Binding*. 2013. URL: https://visualstudiomagazine.com/articles/2013/02/01/building-android-apps.aspx (siehe S. 17).

[51] *Software Architecture and Design, Chapter 3: Architectural Patterns and Styles*. 2009. URL: https://msdn.microsoft.com/en-us/library/ee658117.aspx (siehe S. 10).

[52] *UsabilityNet: Tools and Methods*. URL: http://www.usabilitynet.org/tools.htm (siehe S. 1).

[53] *User Interface | Android Developers*. URL: https://developer.android.com/guide/topics/ui/index.html (siehe S. 1).

[54] *Vorgehensmodell für komponentenbasierte Softwareentwicklung*. 2009. URL: https://www.swt.tu-berlin.de/fileadmin/fg130/lehre/SeminarSS09/Senyuava_Cennet_CBSE.pdf (siehe S. 9, 10, 29).

[55] Joel Wenzel. *MVVM vs MVP vs MVC: The differences explained*. 2011. URL: http://joel.inpointform.net/software-development/mvvm-vs-mvp-vs-mvc-the-differences-explained/ (siehe S. 16, 17, 31).

[56] Aldo Ziflaj. *Native vs Hybrid App Development*. 2014. URL: http://www.sitepoint.com/native-vs-hybrid-app-development/ (siehe S. 26–28).