Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Entwicklung zweier Apps für digitale Spenden an bettelnde Personen

Paul Weber
Matrikelnummer: 4389664
p.weber@fu-berlin.de

Betreuer/in: Victoria Brekenfeld
Eingereicht bei: Prof. Prof. Dr. Lutz Prechelt
Zweitgutachter/in: Prof. Dr. Claudia Müller-Birn

Berlin, 18. Februar 2022

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung zweier Smartphone-Apps, die es ermöglichen sollen, z.B. per Kreditkarte, digitale, spontane Kleinspenden an bedürftige Personen zu entrichten. Die Verwendung moderner Technologien, wie der deklarativen UI-Frameworks SwiftUI und Jetpack Compose und der Cross-Platform-Lösung Kotlin Multiplatform Mobile ermöglicht es, mit Hilfe einer eigens entworfenen Architektur, eine zukunftssichere Codebasis zu schaffen. Es werden einige Limitationen der verwendeten Technologien aufgezeigt, sowie Ideen formuliert, wie diese Limitationen überwunden werden können.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit beschreibt den Prozess der Entwicklung zweier funktionsgleicher Anwendungen für die Smartphone-Betriebssysteme iOS und Android. Dabei wird, ausgehend von der ursprünglichen Motivation für das Projekt, der gesamte Entwicklungsprozess dokumentiert.

Es geht explizit nicht darum, Applikationen in den jeweiligen App-Stores der mobilen Betriebssysteme zu veröffentlichen. Eine Bachelorarbeit bietet, als Projekt an dem lediglich eine Person über eine sehr begrenzte Zeit arbeitet, weder die juristischen noch personellen Vorraussetzungen, um dies adäquat zu realisieren. Daher wird klar abgegrenzt zwischen dem minimal lauffähigen Produkt, dem eigentlichen Ziel dieser Arbeit, und einer, später außerhalb des Rahmens dieser Arbeit weiter zu entwickelnden, produktionsreifen Version des Produkts. Diese Abgrenzungen werden begründet. Daraus resultierend wird der anvisierte Funktionsumfang entwickelt.

Ausgehend von diesem Funktionsumfang werden technologische Überlegungen angestellt: Es wird beschrieben, warum welche Technologien benutzt wurden, um einzelne Programmteile umzusetzen. Des Weiteren beschäftigt sich ein großer Teil der Arbeit damit, den resultierenden Programmcode so zu strukturieren, dass er einfach zu warten und erweitern ist.

Diese beiden, das Produkt sowie die Technologie betreffenden, Sammlungen begleiten den Rest der Arbeit. Sie thematisiert dabei den gesamten Entwicklungsprozess vom Entwurf über die Implementierung und die Auswertung mit Benutzertests hin zum Endprodukt. Im Rahmen der Implementierung werden die aufgetretenen Schwierigkeiten beschrieben. Die Auswertung der Benutzertests beinhaltet die darauf folgende Iteration. Das abschließende Fazit bewertet die erarbeitenden Resultate sowie einzelne Entscheidungen während der Entwicklung.

1.1 Motivation

1.1.1 Vorstellung des Problems

Als Anfang 2020 das Coronavirus Einzug nahm, wurde es selbst in der, im internationalen Vergleich bargeldaffinen, deutschen Gesellschaft nahezu überall möglich, mancherorts sogar gefördert, bargeldlos zu bezahlen [1]. Ohnehin geht, auch ohne das Coronavirus, der Trend in Richtung bargeldlose Gesellschaft [1].

Diese Entwicklung schadet allerdings Menschen, die auf das Betteln auf der Straße angewiesen sind. Wenn ein*e Passant*in kein Bargeld in der Tasche hat, kann er / sie auch keine spontane Spende an eine bettelnde Person tätigen.

Es geht also darum eine Lösung zu finden, die es potentiellen Spender*innen ermöglicht einem bettelnden Gegenüber auf der Straße spontan und bargeldlos Geld zu spenden.

1.1.2 Lösungsansätze

In anderen Ländern gibt es dafür bestehende Lösungsansätze, von denen einige hier aufgegriffen werden.

Da meist jedoch ein Bankkonto die Vorraussetzung für einen dieser Ansätze ist, erst ein paar Worte zum Konto in Deutschland. Dies ist seit 2016 per Gesetz auch wohnungslosen Personen von allen Banken als sogenanntes Basiskonto zur Verfügung zu stellen [2]. Im Jahr 2020 urteilte der Bundesgerichtshof zudem, dass für dieses Basiskonto von

den Banken zuweilen zu hohe Gebühren verlangt wurden und diese Praxis abzustellen ist [3]. Leider gibt es keine belastbaren Zahlen dazu, wie verbreitet Basiskonten sind - weder allgemein noch im Kontext der Wohnungslosigkeit. Meine, von Studien zu belegende, These ist, dass die Verbreitung gering ist. Insbesondere gilt dies für wohnungslose Menschen in Deutschland, die aus dem Ausland kommen.

- NFC-Kartenlesegeräte: In Schweden ist die Möglichkeit verbreitet, dass sich bettelnde Personen mit einem Kartenlesegerät ausstatten. Wie an der Supermarktkasse können Personen bei diesen Geräten mit ihrer Kreditkarte oder dem Smartphone bezahlen und so spenden. Der erste offensichtliche Nachteil ist, dass das Kartenlesegerät als extra Hardware mit sich zu führen ist. Des Weiteren muss die bettelnde Person über ein Bankkonto verfügen. Neben einem Konto benötigt man zudem noch einen Zahlungsdienstleister, der das Kartenlesegerät betreibt und die Spenden an das Konto des / der Empfänger*in überweist. Ich konnte keinen Zahlungsdienstleister ausfindig machen, der dies in Deutschland für wohnungslose Personen anbietet.

- Kryptowährungen: Was in Filmen und Fernsehserien wie Mr. Robot schon als Möglichkeit dargestellt wird, könnte in Zukunft Realität werden: Bettelnde Personen mit QR-Codes zu ihrem Bitcoin-Wallet auf Pappschildern. In Kalifornien soll dies laut diversen Zeitungsartikeln schon recht verbreitet sein. Neben der Tatsache, dass Kryptowährungen bis dato eher spekulativen als Währungscharakter haben, benötigt das Aufsetzen und sichere Besitzen eines Krypto-Wallets eine gewisse technische Vorbildung, sowie anspruchsvolle Sicherheitsvorkehrungen. Auch sind Transaktionen, ebenso wie die Umwandlung der Kryptowährung in Bargeld an Automaten, mit hohen Gebühren verbunden [4]. Eine praktikable Lösung stellt dieses Modell deshalb nicht dar.

- Digitale Zahlungsdienstleister: Ein weiteres Phänomen aus den USA sind mobile Zahlungsdienstleister wie Venmo oder CashApp mit denen man seinem Gegenüber per Smartphone digital Geld zukommen lassen kann. Hier ist der offensichtlichste Makel, dass diese Firmen profitorientiert arbeiten. Das heißt in der Folge, dass auf dem Rücken von ökonomisch schwächer Gestellten Geld verdient wird. Auch wird hierfür wieder ein Bankkonto benötigt.

1.1.3 Lösungsvorschlag

Mein Lösungsvorschlag ist deshalb die Entwicklung einer mobilen Anwendung (im folgenden auch: App) zum spontanen, digitalen Spenden von Kleinbeträgen, die es den Spendenempfänger*innen ermöglicht, dieses digitale Geld in Bargeld umzuwandeln.

Dabei würde die bettelnde Person einen persönlichen QR-Code besitzen und bei sich führen. Dieser Code könnte auf Pappe oder Plastikbecher gedruckt sein aber auch digital auf einem Handy bereitgestellt werden. Der / die Spender*in würde diesen QR-Code mit der Handykamera scannen, den gewünschten Spendenbetrag wählen und mit einem digitalen Zahlungsmittel wie Kreditkarte, Apple Pay, Google Pay oder Paypal bezahlen.

Um das Spendenguthaben in Bargeld umzuwandeln gäbe es zuerst den Ansatz einen Dienstleister wie barzahlen.de [5] zu benutzen. Dieser Dienstleister ermöglicht die Generierung eines Barcodes, welcher dann zum Beispiel im Supermarkt oder in der Drogerie an der Kasse gescannt wird. Der Supermarkt zahlt dann den, vorher von der bettelnden

Person festgelegten, Betrag in bar aus. Sollte ein derartiger Dienstleister nicht in Frage kommen, wäre es ebenfalls möglich mit solidarischen Unternehmen in den Kommunen, wie zum Beispiel Cafés, Stadtteilläden oder Spätis, zusammenzuarbeiten und eine ähnliche Lösung selbst zu implementieren. Die Generierung dieses Codes würde entweder in der App selbst, oder auf einer Webseite - die auch aus einem Internet-Café oder einer Bibliothek ansteuerbar wäre - stattfinden.

Einer App, die aus dem offiziellen App-Store des Betriebssystems heruntergeladen wurde, vertrauen potentielle Spender*innen eher [6], was die Eingabe ihrer Zahlungsdaten angeht, als einer beliebigen Webseite, die aufgerufen wurde weil sie einen QR-Code gescannt haben [7]. Auch ist diese Art der Spende deutlich weniger anfällig für Phishing, weil nicht geprüft werden muss, ob die Webseite, auf der gerade gespendet wird, die echte ist.

Moderne Betriebssystemfunktionen wie App Clips auf iOS [8] oder Instant Apps [9] auf Android verringern zudem die Reibung, die durch die vorher nötige Installation der App entstehen würde. Diese Funktionen laden, zum Beispiel bei Aufruf einer Webseite oder Scannen eines QR-Codes, eine maximal zehn Megabyte große Mini-App, die eine Teilfunktionalität der eigentlichen App enthält, und starten diese auf Wunsch des / der Nutzer*in.

Im ersten Schritt soll also eine App gebaut werden in der das Spenden, das Empfangen von Spenden sowie das Abheben der empfangenen Spenden implementiert ist.

2 Entwurf

In diesem Abschnitt wird der Weg von der Idee der App mit den genannten drei Kernfunktionalitäten hin zu einer klaren Definition des Funktionsumfangs beschrieben. Dabei werden die verwendeten Technologien eingeführt und Ansprüche an den entstehenden Quelltext formuliert.

2.1 Zielsetzungen

2.1.1 Produkt

Um die Thematik der spontanen Bargeldspende an Menschen auf der Straße möglichst deckungsgleich in eine App zu übertragen, sind meiner Auffassung nach folgende Kriterien relevant.

Zum einen ist dies die möglichst hohe Anonymität. Spender*innen sollten komplett anonym spenden können. Bei den Empfänger*innen ist dies natürlich nicht möglich. Für diese müssen Benutzerkonten erstellt werden, zu denen die Spenden zugeordnet werden können. Um dabei Problemen wie verlorenen Passwörtern oder ähnlichem vorzugreifen, wäre es wünschenswert die Benutzerkonten lediglich über den Authentifizierungsdienst der jeweiligen Plattform - Sign in with Apple [10] und Sign In With Google [11] - anzulegen, statt die Authentifizierungsdaten selbst zu erheben. Da ein Google-Konto eine Voraussetzung dafür ist, im Play Store Apps herunterzuladen, ebenso wie ein Apple-Konto für den App Store, verfügen alle potentiellen Nutzer*innen über eines dieser Konten.

Ein weiteres Kriterium ist, dass die Person, die auf der Straße eine Spende empfängt auch Inhaberin des Benutzerkontos ist, das hinter dem QR-Code steht, der gerade gescannt wurde. Ein denkbares Missbrauchsszenario wäre, dass mehrere Personen dazu genötigt werden, den gleichen QR-Code mit sich zu führen und so um Spenden für ein

einziges Benutzerkonto zu bitten. Um dieses Szenario abzuwenden, wäre es möglich, Empfänger*innen beim Einrichten des Kontos nach einem möglichst identifizierenden Alias oder dem Hochladen eines Bildes zu fragen. Die Spender*innen müssten dann darauf hingewiesen werden, dass dieses Bild und der Alias dazu da sind, die Person die ihnen gegenübersteht zuzuordnen.

Das letzte wichtige Kriterium ist, dass die spendenempfangende Person frei über die Verwendung des erhaltenen Geldes entscheiden kann. Das heißt, das Geld ist in Bargeld umwandelbar, nicht etwa nur als Guthaben bei bestimmten Einrichtungen oder Geschäften verwendbar.

Aus diesen Kriterien leiten sich drei Anwendungsfälle ab:

- Das Einrichten eines Benutzerkontos, damit Spenden empfangen werden können: Um dies zu realisieren wird eine Serverapplikation (im folgenden: Backend) benötigt, bei der Benutzerkonten erstellt werden können. Es muss zudem möglich sein, den Konten identifizierende Daten wie ein Alias oder ein Bild zuzuweisen. Nachdem ein identifizierendes Datum vorliegt, generiert das Backend einen, dem Benutzerkonto zugewiesenen, QR-Code, den die App darstellen kann.

- Die Spende an eine*n andere Benutzer*in: Um Kreditkarten- oder andere Zahlungen entgegenzunehmen wird die Bibliothek eines Zahlungsdienstleisters eingebunden. Die App ermöglicht das Scannen von QR-Codes. Nachdem ein dem Dienst zugehöriger QR-Code gescannt wurde, stellt das Backend der App die identifizierenden Daten der empfangenden Person zur Verfügung. Anhand der identifizierenden Daten entscheidet der / die Spender*in über den Spendenbetrag, gibt die Zahlungsdaten ein und veranlasst die Spende. Das Backend erhält vom Zahlungsdienstleister Bescheid darüber, ob die Spende erfolgreich bezahlt wurde. Im Erfolgsfall bucht das Backend den Spendenbetrag auf das Guthabenkonto des / der Empfänger*in. Es wäre außerdem wünschenswert wenn das Scannen eines QR-Codes in der Betriebssytemkamera, die App direkt in der Spendenansicht öffnen würde - genauso als wäre der QR-Code in der App selbst gescannt worden.

- Das Umwandeln von Spendenguthaben in Bargeld: Sobald ein Konto über ausreichend Guthaben verfügt, ist es in der App möglich einen Betrag zu wählen, für den dann backendseitig ein QR-Code generiert wird. Sobald dieser QR-Code gescannt wird, muss dem / der Nutzer*in der gewählte Betrag in Bar ausgezahlt werden. Der Betrag wird daraufhin im Backend vom Guthaben abgezogen.

2.1.2 Abgrenzungen

Um den Umfang der Arbeit überschaubar zu halten, liegt der Fokus auf der Implementierung der App. Das für die Funktionalität erforderliche Backend wird zwar implementiert, aber so einfach wie möglich gehalten. Es wird beispielsweise auf eine Datenbank verzichtet. Die gesammelten Datensätze und erstellten Nutzerkonten werden lediglich im Speicher gehalten, das heißt sie gehen bei Neustart des Backends verloren. Das würde für den Produktionsbetrieb selbstverständlich nicht ausreichen, für eine Demonstration der Appfunktionalitäten genügt es jedoch.

Ohne Backenddatenbank fällt leider auch die Verbindung mit den Authentifizierungsdiensten der Plattformen weg. Die Implementierung der Authentifizierungsdienste ist eine aufwendige Angelegenheit, die aber hauptsächlich im Backendcode stattfinden würde und eine persistente Datenbank erfordert. Ob die Nutzerkonten am Ende mit

Drittpartei-Konten oder mit einer eigenen E-Mail / Passwort Kombination angelegt werden, macht für den Rest der Anwendungsfälle keinerlei Unterschied. Um Entwicklerressourcen zu schonen wurde dieser Makel also bewusst in Kauf genommen.

Eine DSGVO-konforme Benutzerverwaltung, samt Datenschutzerklärung und allgemein gültigen Geschäftsbedingungen wurde ebenfalls ausgespart. Ohne juristischen Beistand wären die genannten Dokumente nicht vernünftig zu realisieren. Die Benutzerverwaltung würde sich auch hauptsächlich im Backendcode befinden, daher wurde auf diese verzichtet. Da diese Funktionalitäten und Dokumente für eine Veröffentlichung in den App Stores der Betriebssysteme erforderlich wären [12, 13], ist diese damit im Rahmen dieser Arbeit ausgeschlossen.

Da sich nach mehrtägiger Evaluation herausgestellt hat, dass die bereits genannten App Clip, beziehungsweise Instant App Funktionalitäten der Betriebssysteme nur korrekt funktionieren, wenn eine App im jeweiligen Store veröffentlicht ist, fiel diese ursprünglich gewünschte Facette damit bedauerlicherweise weg.

Zuletzt ist das tatsächliche Auszahlen in Bargeld ebenfalls nicht im Rahmen einer Bachelorarbeit zu realisieren. Während es der, für die Spenden gewählte, Zahlungsdienstleister ermöglicht, in einem Testmodus mit Spielgeld, das durch spezifische Testkreditkartennummern generiert wird [14], zu arbeiten, ist dies bei der Auszahlung nicht möglich. Ohne echte Spende gibt es natürlich auch kein echtes Geld, das ausgezahlt werden kann. Daher wird der Auszahlungsvorgang nur simuliert: Das Ziel ist es einen QR-Code zu generieren, der die URL einer Webseite enkodiert. Sobald diese Webseite einmal aufgerufen wird, wird dann der gewünschte Betrag vom Guthaben des / der Nutzer*in abgezogen.

2.2 Technologische Überlegungen

Als Nutzer eines iPhones war es für mich wichtig, die resultierende App für iOS zu entwickeln. Da jedoch davon auszugehen ist, dass Android sowohl bei der allgemeinen Bevölkerung [15, 16] als auch, durch die begrenzten finanziellen Mittel, bei der Zielgruppe, den bettelnden Menschen, weiter verbreitet ist, war es ebenso wichtig, auch eine Version für Android zu erstellen.

Das Ziel, zwei funktionsgleiche Apps zu implementieren, legt nahe, sich mit sogenannten Cross-Plattform-Technologien auseinanderzusetzen. Cross-Plattform-Technologien ermöglichen es, aus einer Codebasis zwei Apps für, in diesem Fall, beide mobilen Betriebssysteme zu veröffentlichen. Ohne diese Technologien ist es notwendig, zwei separate Codebasen zu pflegen, in denen in der jeweiligen nativen, das heißt von den Betriebssystemen offiziell unterstützten, Programmiersprache geschrieben und mit den offiziellen Schnittstellen der Plattformbetreiber interagiert wird.

2.2.1 Cross-Plattform-Technologien

Es gibt mehrere Ansätze und Technologien, die als Cross-Plattform-Lösung dienen können. Zum einen gibt es Frameworks wie Flutter [17] oder React Native [18]. Diese ermöglichen es, in einer Programmiersprache, mit der auf der eigentlichen Plattform ansonsten nicht gearbeitet wird, Apps zu entwickeln, die sich dennoch anfühlen sollen wie native Programme [17, 18]. Diese Frameworks sind in der Industrie besonders beliebt, um möglichst schnell einen Prototypen auf den Markt zu bringen [19]. Da lediglich ein Entwicklerteam aufgebaut werden muss, um beide Plattformen zu versorgen, versprechen derartige Technologien auch häufig Kostensenkungen. Die Technologien bergen

aber auch einige Nachteile. Hat man beispielsweise erst einmal eine Codebasis für Flutter geschrieben, ist man darauf angewiesen, dass der Frameworkentwickler, in diesem Fall Google, das Framework in Zukunft weiter pflegt. In den letzten zehn Jahren gab es zahlreiche Cross-Plattform-Frameworks, die mal mehr, mal weniger beliebt waren, aber mittlerweile nicht mehr von deren Entwicklern unterstützt werden. Auch muss bei neuen Betriebssystemfunktionen darauf gewartet werden, dass das Framework diese unterstützt. Zudem ist, durch die zusätzliche Programmschicht die zwischen dem Betriebssystem und dem eigenen Code vermittelt, mit Laufzeiteinbußen zu rechnen. [20]

Eine weitere Möglichkeit ist es, nur einen Teil des Codes zu teilen. Dies ist besonders verbreitet für den Teil des Programmcodes, der sich um die Geschäftslogik dreht, also das Bereitstellen und Verarbeiten von Daten. Diese Geschäftslogik wird dann als Bibliothek in den nativen Programmcode eingebunden. Dadurch ist es, wie bei einer komplett nativen App, möglich, mit den offiziellen Schnittstellen des Systems zu arbeiten. Alle Funktionalitäten und Eigenheiten der Plattform können genutzt und implementiert werden, aber ein Teil des Programmcodes muss trotzdem nur einmal geschrieben werden. In der Vergangenheit wurde für diese Herangehensweise oft C++ genutzt, um den geteilten Codeteil zu programmieren (Siehe zum Beispiel Facebook in 2015: [21]). Der größte Nachteil war daher die Programmiersprache. Denn die Apps selbst müssen in Java oder Kotlin für Android und Objective-C oder Swift für iOS implementiert werden. Daher musste ein separates Entwicklerteam für den C++ Teil eingestellt werden. Dieser Ansatz wurde deshalb hauptsächlich von großen Teams genutzt, für kleinere Entwicklerteams war dies aufgrund der höheren Kosten unüblich.

2.2.2 Kotlin Multiplatform Mobile

Kotlin ist eine Programmiersprache, die von Jetbrains entwickelt wurde. Jetbrains ist ein Unternehmen, das 2000 gegründet wurde und mit den von ihm veröffentlichten integrierten Entwicklungsumgebungen wie IntelliJ IDEA bekannt wurde. Android Studio, die offizielle von Google veröffentlichte Entwicklungsumgebung für Android Applikationen, basiert auf den Produkten von Jetbrains, speziell auf IntelliJ IDEA [22]. Kotlin als Programmiersprache wurde vorerst als Alternative zu Java eingeführt und konnte zu Java Bytecode kompiliert werden. Die moderne Syntax und zeitgemäßen Sprachfeatures, die es einfach machen, sicheren und korrekten Code zu programmieren [23], verhalfen Kotlin schnell zu hoher Beliebtheit. Dies führte dazu, dass Google ab 2019 Java als bevorzugte Programmiersprache für die Entwicklung von Android Applikationen durch Kotlin ersetzte [24].

Parallel zum Siegeszug auf der Java Virtual Machine (im folgenden: JVM) arbeitete Jetbrains an zwei weiteren Applikationen für Kotlin. Zum einen ein Transpilierer von Kotlin zu Javascript, der es ermöglicht im Browser lauffähigen Code in Kotlin zu schreiben. Zum anderen entwickelten sie, für dieses Projekt relevanter, Kotlin Native. Kotlin Native ist in der Lage, Kotlin Code in Maschinencode umzuwandeln. Das bedeutet, es ist keine JVM mehr nötig, um Programme, die in Kotlin geschrieben wurden, auszuführen. Alle drei Laufzeitumgebungen zusammen werden Kotlin Multiplatform genannt. Kotlin Multiplatform ermöglicht es also, Programmcode einmal zu schreiben und sowohl auf der JVM, als auch nativ als Maschinencode und sogar im Browser als Javascript auszuführen. Ein Kernprodukt von Kotlin Multiplatform ist dabei Kotlin Multiplatform Mobile (im Folgenden: KMM) was speziell darauf ausgerichtet ist, Apps für die beiden mobilen Plattformen zu entwickeln. Jetbrains selbst veröffentlicht KMM seit 2021 mit dem Label “Alpha”. Nach deren Auffassung ist KMM damit bereit für den

Produktivbetrieb [25].

Da Kotlin, als offizielle Sprache für die Entwicklung von Android Apps, unter Entwicklern für mobile Plattformen weit verbreitet ist, ist KMM ein geeigneterer Kandidat für die im vorigen Abschnitt genannte Möglichkeit der Cross-Plattform-Entwicklung als C++. Die moderne Syntax und viele Sprachfeatures ähneln zudem Swift, der am weitesten verbreiteten Sprache für iOS Entwicklung, sehr.

2.2.3 Deklarative UI-Frameworks

Der bislang größte Paradigmenwechsel im Bereich der Entwicklung von Smartphone-Apps ist der Einzug der deklarativen UI-Frameworks. Nachdem in der Webentwicklung, seit der Veröffentlichung von React im Jahr 2013, schnell auf die deklarativen Frameworks umgestiegen wurde, dauerte es verhältnismäßig lange bis diese im mobilen Sektor ankamen. Erst im Jahr 2019 veröffentlichte Apple SwiftUI als eigenen deklarativen Baukasten für Benutzeroberflächen. Im gleichen Jahr kündigte Google Jetpack Compose an. Ebenfalls ein deklaratives UI-Framework, wurde Jetpack Compose im Juli 2021 als Version 1.0 und damit offiziell bereit für den Produktivbetrieb veröffentlicht.

Im Folgenden werden, weil dies in der Domäne üblich ist, die englischen Begriffe "View" und "State" verwendet. View ist dabei ein Objekt, definiert im Programmcode, welches vom UI-Framework gerendert wird und somit vom System auf dem Bildschirm des Handys dargestellt werden kann. State sind die Daten die den aktuellen Status des Programms beschreiben. Im Fall der Benutzeroberflächenprogrammierung sind mit State meist spezifischer nur genau die Daten gemeint, die notwendig sind um eine View zu versorgen. Eine View die einen Text darstellt benötigt beispielsweise einen String als State.

Deklarative UI-Frameworks stehen in der Tradition der Funktionalen Programmierung. Sie sehen eine View dabei als eine Funktion, die als Argument den nötigen State bekommt und daraus Ansichten rendert. Das bedeutet, dass Views keine Objekte im Sinne der objektorientierten Programmierung mehr sind. Sie werden nicht mehr im Speicher gehalten und je nach State verändert. Stattdessen wird bei jeder Änderung des States eine neue View generiert und die alte weggeworfen.

Die vorherigen, imperativen UI-Frameworks auf beiden mobilen Plattformen nutzten XML-Dateien, in denen die Layouts der Ansichten definiert wurden. Diese Layouts wurden zur Laufzeit als Objekte einer Klasse instanziert und es konnte mit Funktionen und Methoden in der jeweiligen Programmiersprache auf die Felder, die in den XML-Dateien definiert wurden, zugegriffen werden. Diese XML-Dateien sind mit den neuen, deklarativen UI-Frameworks nicht mehr notwendig. Mit diesen wird der komplette Benutzeroberflächencode in der Sprache verfasst in der auch der Rest des Programms geschrieben wird. Dieser Code ist sehr schlank und deskriptiv im Gegensatz zu den sehr verbosen XML-Definitionen der Vorgängerframeworks. Dies ist ein großer Vorteil wenn es um Code Reviews geht.

Was die beiden Frameworks für dieses Projekt besonders interessant macht, ist dass sich die Art, wie man auf beiden Plattformen Benutzeroberflächen implementiert, damit sehr angeglichen hat. Das zeigt sich am Besten anhand eines Beispiels.

```
// Jetpack Compose
@Composable
fun ExampleView(icon: Icon, title: String, subtitle: String) {
  Row(horizontalArrangement = spacedBy(8.dp)) {
      Icon(icon, contentDescription = "")

      Column(verticalArrangement = spacedBy(2.dp)) {
        Text(
          title,
          style = MaterialTheme.typography.body1,
        )
        Text(
          subtitle,
          style = MaterialTheme.typography.caption,
          color = MaterialTheme.colors.secondary
        )
      }
  }
}
```

```
// SwiftUI
struct ExampleView: View {
  let title: String
  let subtitle: String

  var body: some View {
    HStack(spacing: 8) {
      Image(systemName: "icon")

      VStack(spacing: 2) {
        Text("title")
          .font(.body)
        Text("subtitle")
          .font(.caption)
          .foregroundColor(.secondary)
      }
    }
  }
}
```

Beide Codeblöcke stellen dabei das gleiche dar: Ein kleines Bild und rechts daneben einen Titel mit Untertitel, die übereinander stehen. `Row`, `HStack`, `Column` und `VStack` sind dabei Layoutcontainer. Die beiden erstgenannten richten die enthaltenen Elemente horizontal aus, die letztgenannten vertikal. In beiden Beispielen kann ein `spacing` angegeben werden, was die Abstände zwischen den enthaltenen Elementen bestimmt. Textelemente, sowie Bilder sind nur Funktionsaufrufe, die ähnlich konfigurierbar sind. Da sich die Layoutmöglichkeiten faktisch nicht unterscheiden, ist es teilweise möglich, schon bestehenden SwiftUI Code in eine Jetpack Compose Struktur zu kopieren und lediglich die Framework-Funktionsaufrufe auszutauschen. So kann eine zweite Plattform

sehr schnell an die bestehende angeglichen werden.

2.3 Ansprüche an den Quelltext

2.3.1 Zugänglichkeit

Der Quelltext sollte so zugänglich wie möglich sein. Neben einer verständlichen Dokumentation, die neuen Entwicklern den Einstieg in das Projekt erleichtert, sollte die Benutzung von externen Softwarebibliotheken, speziell Frameworks, vermieden werden. Die mobile App-Entwicklung hängt bereits stark von den Frameworks der Betriebssysteme ab. Es ist also davon auszugehen, dass Entwickler, die Erfahrung in der Entwicklung von mobilen Apps haben, auch im Umgang mit diesen Systemframeworks erfahren sind. Wird nun jedoch ein weiteres Framework einer Drittpartei eingegliedert, muss erst der Umgang mit diesem erlernt werden, bevor die eigentliche Codebasis korrekt verstanden und erweitert werden kann. Dies erhöht offensichtlich die Reibung bei der Einarbeitung, beziehungsweise engt den Kreis der Entwickler die schnell am Projekt mitarbeiten können ein auf diese, die schon im jeweiligen Framework erfahren sind.

Weiterhin sollten die jeweiligen nativen Teile des Codes so gestaltet sein, dass sich Plattformerfahrene Entwickler zu Hause fühlen. Das bedeutet weiterhin, dass auch der geteilte Kotlin Code Schnittstellen für Swift bereitstellen sollte, die sich anfühlen wie Schnittstellen, die in Swift geschrieben wurden.

2.3.2 Teilbarkeit

Das Versprechen von KMM, dass Code zwischen den Plattformen geteilt werden kann, die UI-Frameworks die sich sowohl das Paradigma teilen als auch mit ähnlicher Syntax angesprochen werden und die gewünschte deckungsgleiche Funktionalität der beiden Apps verleiten zu folgendem Wunsch: So viel Code wie möglich in das geteilte, in Kotlin implementierte Modul zu stecken und lediglich die Benutzeroberflächen in den nativen Codebasen zu definieren. Geteilter Code muss nur einmal geschrieben und getestet werden.

2.3.3 Korrektheit

Der entstehende Code sollte möglichst einfach auf seine Korrektheit testbar sein. Die zu definierende Architektur der Software sollte also darauf ausgerichtet sein, dass möglichst isolierte Komponenten mit wenigen, im besten Fall austauschbaren, Abhängigkeiten und minimalen Seiteneffekten entstehen. Die wenigen Abhängigkeiten können dann im Fall eines Modultests einfach durch sogenannte Mock-Objekte ausgetauscht werden. Mock-Objekte sind Objekte die als Platzhalter für die echten Objekte stehen. In diesen werden die, für die Ausführung der zu testenden Komponente notwendigen, Funktionalitäten der Abhängigkeit mit Testdaten simuliert, um so zu determinierten Testergebnissen zu kommen [26].

3 Ansatz

Als Basis der Architektur wird das, in der mobilen Entwicklergemeinde weit verbreitete, MVVM-Schema verwendet. MVVM steht hierbei für “Model View ViewModel”, nennt damit die Namen der das Schema dominierenden Komponenten [27]. Model steht für die

Geschäftslogik und all ihrer Komponenten, beziehungsweise die aus dem Geschäftslogikteil bereitgestellten Daten [27]. Die View ist der Teil des Codes, der sich, wie im obigen Abschnitt schon angerissen, darum kümmert, mit Funktionen des UI-Frameworks eine darstellbare Ansicht zu rendern [27]. Das ViewModel ist das Bindeglied zwischen Model und View. Eine ViewModel-Instanz kommuniziert mit verschiedenen Komponenten aus der Model-Schicht und erhält von diesen Daten [27]. Aufgrund dieser erhaltenen Daten errechnet das ViewModel den darzustellenden State. Dieser State enthält alle Informationen, die für die View notwendig sind, um ihre Ansicht zu rendern. In einem folgenden Abschnitt wird noch einmal genauer darauf eingegangen wie sich das genau gestaltet.

Relevant für eine echte MVVM-Implementierung sind zwei Kriterien [28]:

Das erste ist die korrekte Entkoppelung der einzelnen Schichten. Eine Model-Komponente darf keine Informationen darüber haben von welchem ViewModel sie benutzt wird. Eine Referenz zu einer ViewModel-Klasse in Model-Code ist daher nicht erlaubt. Ebenso verhält es sich mit Referenzen von ViewModels zu Views: Ein ViewModel stellt lediglich State, sowie Funktionen, die es dazu veranlassen diesen State zu verändern, zur Verfügung, weiß aber niemals von welcher View dieser State konsumiert, beziehungsweise diese Funktionen benutzt werden. Das heißt ein ViewModel hält keinerlei Referenzen zu einer View. Das Model und die View kommunizieren nie direkt miteinander. Es ist in jedem Fall ein ViewModel dazwischengeschaltet. In der Praxis bedeutet das also, dass ein ViewModel Referenzen zu bestimmten Model-Komponenten hält und eine View eine Referenz auf ein ViewModel. [28]

Das zweite Kriterium ist der sogenannte unidirektionale Datenfluss. Daten die in der Modelschicht bereitgestellt werden, werden im ViewModel zu State verarbeitet und durch die View dargestellt. Durch Benutzerinteraktionen, wie zum Beispiel das Drücken eines Knopfes im UI oder die Eingabe von Text werden, aus der View heraus, Funktionen des ViewModels aufgerufen [28]. Diese erzeugen wiederrum neuen State - entweder direkt oder durch das Aufrufen von Funktionen im Model. Der View ist es dabei explizit nicht gestattet, selbst State zu generieren oder diesen zu verändern [28].

3.1 Das Model

Innerhalb der Modelschicht wird sich lose an das, von Google für Android Entwickler vorgeschlagene, Schema zur Definition des “Data Layers” [29] gehalten.

Für jede Domäne an verschiedenen Daten gibt es ein sogenanntes Repository. Es gibt zum Beispiel ein UserRepository das für die Bereitstellung von User-Objekten, sowie aller Funktionalitäten die um das User-Objekt angelegt sind, wie das Einloggen, das Ausloggen oder das Registrieren, verantwortlich ist. Ein Repository verfügt über verschiedene Datenquellen wie zum Beispiel einer Komponente, deren Aufgabe es ist mit dem Backend zu kommunizieren. Oder zum Beispiel eine Komponente, die Authentifizierungsdaten im persistenten Speicher des Smartphones vorhält.

Wichtig hierbei ist, dass sowohl das Repository, als auch die Datenquellen selbst, erst einmal nur in Interfaces definiert werden. Ein Interface, als Sprachfeature aus Java, Kotlin oder anderen objektorientierten Programmiersprachen bekannt, enthält Funktions- sowie Variabeldefinitionen die von allen Klassen, die dieses Interface implementieren, bereitgestellt werden müssen. Ohne diese Vorgabe müssten, jedes mal wenn eine Komponente modulgetestet wird, all ihre Abhängigkeiten korrekt initialisiert werden. Bei netzwerkabhänigen Komponenten wie den Datenquellen müsste eine Internetverbindung bestehen, et cetera. Wenn eine Komponente jedoch als Abhänigkeit lediglich ein

Interface definiert, kann dieses Interface auch von einem Mock-Objekt implementiert werden, welches determinierte Antworten auf Funktionsaufrufe gibt. So werden einzelne Komponenten also isoliert testbar gemacht.

3.2 Das ViewModel

Die primäre Aufgabe des ViewModels ist es, wie schon erwähnt, den für das Rendern der Benutzeroberfläche notwendigen, State zur Verfügung zu stellen. Das heißt, jedes ViewModel wird von einer Definition des bereitgestellten States begleitet. Zudem bietet das ViewModel ein beobachtbares Feld vom Typ dieses States an. Außerdem wird für jede mögliche Benutzerinteraktion eine Aktion festgelegt, welche von der View an das ViewModel gereicht werden kann. Das folgende, einfache Beispiel versorgt eine fiktive View, die das Ziel hat einen String darzustellen.

```
1  sealed interface FooState {
2      object Loading : FooState
3      object Error : FooState
4      data class Content(val text: String) : FooState
5  }
6
7  sealed interface FooAction {
8      object Reload: FooAction
9  }
10
11 interface FooViewModel {
12     // Flow ist eine beobachtbare Struktur, die in diesem Fall
13     // immer einen Wert vom Typen FooState annimmt.
14     val state: Flow<FooState>
15     fun perform(action: FooAction)
16 }
```

Die zugehörige View kann nun das `state` Feld des `FooViewModels` beobachten und anhand des Wertes verschiedene Ansichten rendern. Bei einem Wert von `FooState.Loading` könnte ein Ladebalken oder ein drehendes Rad angezeigt werden. Nach dem Laden und Empfang des `FooState.Content` Wertes kann der enthaltene String angezeigt werden. Im Ladefehlerfall kann eine Fehlermeldung angezeigt, sowie ein Knopf, um den Ladevorgang erneut zu starten. Dieser Knopf ruft die `perform` Methode mit dem Argument `FooAction.Reload` auf, was gegebenenfalls den Ladevorgang anstößt. Die explizite Modellierung jedes möglichen States, sowie der möglichen Aktionen, zwingt die entwickelnde Person dazu, sich genau damit auseinanderzusetzen, was in der ensprechenden View, unter Berücksichtigung der Möglichkeiten die die darunterliegende Modelschicht bereitstellt, überhaupt dargestellt werden kann. So werden später bei der Implementierung der View auftretende, unerwartete Stati vermieden. Außerdem hilft die Durchsetzung dieses expliziten Schemas dabei, auf einen Blick zu verstehen, worum es in der von diesem ViewModel versorgten Ansicht geht.

3.3 Navigation

Ein nicht zu vernachlässigender Aspekt bei der mobilen App-Entwicklung ist die Navigation zwischen den verschiedenen Ansichten. Anders als zum Beispiel bei Webanwen-

dungen, bei denen die Navigation gewöhnlich vom Browser gehandhabt wird, müssen mobile Anwendungen diese selber implementieren. Dabei gibt es zwei Klassen an Navigationsebenen. Einmal die horizontale Navigation, bei der die Ansichten, bildlich gesehen, nebeneinander liegen und man, zum Beispiel mit Hilfe einer Tableiste, zwischen den Ansichten hin- und herschaltet. Zum anderen gibt es die vertikale Navigation. Dabei kann eine Ansicht zu einer weiteren Ansicht navigieren, welche die bisherige ersetzt. Jede annavigierte Ansicht ist dabei mit einem "Zurück"-Knopf versehen, auf dessen Betätigung wieder zurück zur ursprünglichen Ansicht navigiert wird. Diese Art der Navigation ist zu vergleichen mit einem Kartenstapel, bei dem immer nur die oberste Karte zu sehen ist. Meistens wird in mobilen Anwendungen eine Kombination der beiden Navigationsklassen umgesetzt. Dies resultiert dann, bildlich, in mehreren Kartenstapeln, die nebeneinander liegen.

Beide Formen der Navigation können durch Datensätze, also State, dargestellt werden. Da eines der Ziele der Architektur war, so viel Code wie möglich zwischen den Plattformen zu teilen, soll auch der Navigationsstate im geteilten Kotlinmodul implementiert werden.

Im Falle der horizontalen Navigation ist dies denkbar simpel. Angenommen es soll zwischen zwei Ansichten (Foo & Bar) hin und hergeschaltet werden, wird ein ViewModel definiert, welches sowohl ein `FooViewModel` als auch ein `BarViewModel` hält. Der State dieses ViewModels würde dann lediglich abbilden, welches der beiden gerade aktiv ist. Zudem müsste eine Aktion bereitgestellt werden mit der man zwischen den beiden ViewModels hin und her schalten kann. Dies könnte so aussehen:

```
1  enum class Tab { FOO, BAR }
2
3  data class TabState(val selectedTab: Tab)
4
5  sealed interface TabAction {
6      data class SelectTab(val newTab: Tab): TabAction
7  }
8
9  class TabViewModel(
10     state: TabState = TabState(Tab.FOO),
11     val fooViewModel: FooViewModel,
12     val barViewModel: BarViewModel
13 ) {
14     val state = StateFlow(state)
15
16     fun perform(action: TabAction) {
17         when (action) {
18             is TabAction.SelectTab -> state.value = TabState(action.newTab)
19         }
20     }
21 }
```

Die vertikale Navigation ist etwas komplizierter. Jedes ViewModel kann potentiell neben seinem eigenen State noch über zusätzlichen Navigationsstate verfügen. Nehmen wir in der Folge an, dass das `FooViewModel` zu einem `BarViewModel` navigieren und zusätzlich noch diverse Fehlerdialoge anzeigen kann. Damit die zugehörige FooView

weiß, dass sie navigieren soll, muss sie dies über eine Änderung des States mitgeteilt bekommen. Zudem benötigt sie das ViewModel der View, zu der sie navigieren soll. Das alles hat semantisch nichts damit zu tun, wie die FooView auszusehen hat und welche Daten sie dafür benötigt, sondern nur mit der Navigation an sich. Daher wird der vertikale Navigationsstate für das `FooViewModel` in seiner eigenen Definition, der `FooRoute`, gekapselt. Analog zum State wird auch für die Route eine beobachtbare Struktur auf dem ViewModel eingerichtet, auf die die jeweilige View hören kann. Es folgt ein Beispiel einer statischen View, die zu einer weiteren View navigieren, sowie einen Dialog präsentieren kann. Sie ist in diesem Fall der Kürze willen statisch, denn so muss kein State definiert werden. In der Realität werden die allermeisten ViewModels sowohl State als auch Routen haben.

```
1  sealed interface FooRoute {
2    data class Bar(val viewModel: BarViewModel): FooRoute
3    data class Dialog(
4      val message: String,
5      val onDismiss: () -> Unit
6    ): FooRoute
7  }
8  enum class FooAction { BAR, DIALOG }
9
10 class FooViewModel(
11   route: FooRoute? = null,
12   val onBack: (() -> Unit)? = null
13 ) {
14   val route = StateFlow(route)
15
16   fun perform(action: FooAction) {
17     when (action) {
18       FooAction.BAR -> {
19         val viewModel = BarViewModel(onBack: this::routeToNull)
20         route.value = FooRoute.Bar(viewModel)
21       }
22       FooAction.DIALOG -> {
23         val message = "Lorem ipsum doloret"
24         val onDismiss = { routeToNull() }
25         route.value = FooRoute.Alert(message, onDismiss)
26       }
27     }
28   }
29   fun routeToNull() {
30     route.value = null
31   }
32   fun onBackButtonPressed() {
33     onBack?()
34   }
35 }
```

Sobald die `route` des `FooViewModels` nicht `null` ist, wird von der beobachtenden

FooView zu dem in der entsprechenden `FooRoute` spezifizierten Ziel navigiert. Dazu werden die in der `FooRoute` enthaltenen Daten verwendet, im Beispiel das `BarViewModel` zum Initialisieren einer BarView oder der String für die Nachricht des zu präsentierenden Dialogs. Zudem bietet das `FooViewModel` beiden Routen in Form seiner `routeToNull()` Methode die Möglichkeit, wieder zurück zur FooView zu navigieren. Dies wird explizit nicht als `FooAction` modelliert, da die Aktionen nur für die FooView, nicht für andere Views oder ViewModels, intendiert sind. Die FooView könnte wiederrum von einer anderen View annavigiert worden sein. In diesem Fall hätte das navigierende ViewModel dem `FooViewModel` seine eigene `routeToNull()` Methode als `onBack` im Konstruktor mitgegeben. Das systemeigene Navigationselement mit dem "Zurück"-Knopf, in dem die FooView präsentiert wird, würde nun `onBackButtonPressed()` auf dem `FooViewModel` ausführen, woraufhin die `routeToNull()` Funktion des navigierenden ViewModels ausgeführt werden würde. Damit würde also zurücknavigiert. Der "Zurück"-Knopf ist dabei auch explizit nicht als Aktion definiert. Das hat den Grund, dass das `FooViewModel` nicht davon abhängig ist ob es an der Wurzel der Navigationskette steht oder nicht und gleichermaßen der "Zurück"-Knopf auch kein Teil der FooView ist.

Da, wie bereits erwähnt, jedes ViewModel potentiell teil einer vertikalen Navigation ist, sollten diese Standartfunktionalitäten wie das Zurücknavigieren in einer Basisklasse implementiert werden von der dann alle ViewModels erben können.

4 Umsetzung

Nachdem im vorigen Abschnitt die Kriterien und der gewählte technologische, sowie architektonische Ansatz für die Implementierung des Projekts besprochen wurde, geht es nun um die eigentliche Umsetzung.

Als erfahrener iOS-Entwickler waren für mich sowohl Jetpack Compose als Android UI-Framework als auch Kotlin Multiplatform als Technologie völlig neu. Auch war im Vorfeld unklar, ob der architektonische Wunsch, sowohl das Model als auch die ViewModel-Schicht komplett im geteilten Kotlin Code zu realisieren, ohne weiteres umsetzbar ist. Da die größte zu erwartende Reibung zwischen dem geteilten Kotlinmodul und der iOS-App lag, entschied ich mich dazu, zuerst die iOS-App zu implementieren. Die Hoffnung war, dass, wenn die Implementierung für iOS mit den in Kotlin geschriebenen ViewModels, States und Routen funktioniert, die Android-App unkompliziert nachzuziehen ist. Die Android-App würde unabhängig von der iOS Umsetzung in Kotlin geschrieben werden, also würde das jetzt geteilte Modul vermutlich nicht großartig anders aussehen, wenn es eine reine Android-App gewesen wäre. Jetpack Compose zu erlernen hielt ich ebenfalls für keine große Hürde; Da sich die deklarativen UI-Frameworks, die ich bisher kennengelernt hatte, alle stark ähneln, ging ich davon aus dass dem auch in diesem Fall so sein würde. Die Hoffnung war, dass, wenn die Implementierung für iOS mit den in Kotlin geschriebenen ViewModels, States und Routen funktioniert, die Android-App unkompliziert nachzuziehen ist.

Geplant hatte ich, etwa ein Sechstel der Gesamtzeit damit zu verbringen, herauszufinden, inwiefern die gewünschte Architektur mit Koltin Multiplatform im Zusammenspiel mit SwiftUI zu realisieren ist. Drei Sechstel sollten für die Implementierung der App in SwiftUI, in Kombination mit dem geteilten Kotlinmodul, aufgewandt werden. Ein Sechstel plante ich für die Benutzeroberfläche in Android und ein weiteres Sechstel für ein nachträgliches Glattziehen, Refaktorisieren und Qualitätssichern des Quellcodes. Danach war eine Evaluierung der Apps durch echte Benutzer*innen und eine darauffol-

gende Iteration geplant.

Abgesehen davon, dass das anfängliche Forschen hinsichtlich KMM + iOS / SwiftUI deutlich länger dauerte als angedacht, ging dieser Plan auf. Die Zeit, die am Anfang für die Forschung gebraucht wurde, konnte, als die großen Fragen geklärt waren und sich Schemata, wie einzelne Probleme allgemein zu Lösen sind, aufzeigten, bei der iOS-Implementierungsphase wett gemacht werden.

4.1 Probleme

Im Folgenden wird auf Probleme eingegangen, die bei der Implementierung des Projekts auftraten. Durch die für mich neuen Technologien gab es einige zu erwartende Hürden, aber auch durchaus einige Überraschungen. Ob der Fülle an Problemen wurde sich in dieser Ausarbeitung auf die drei prominentesten beschränkt.

4.1.1 Debugging

Sie stellte zwar keine große Hürde dar, aber war doch ein zeitraubender Aspekt beim Debugging: Die Kompilierungszeit des geteilten Kotlinmoduls für die iOS-Plattform. Wurde auch nur eine Zeile im geteilten Code verändert, musste die komplette Bibliothek erneut kompiliert werden. Die offiziellen Tipps, um die Kompilierungszeit zu verkürzen [30] wurden zwar alle befolgt, aber dennoch waren die Kompilierungszeiten jenseits der 4 Minuten. Für die moderne App-Entwicklung ist dies bei einem Projekt von dieser, eher überschaubaren, Größe überraschend lang gewesen. Speziell in der anfänglichen Phase des Projekts, in der schnell iteriert wurde um eine funktionierende Basis zu schaffen, war dies äußerst hinderlich. Als diese Basis dann funktional und erprobt war und mehrere Klassen geschrieben werden konnten, ohne zwischendurch ständig kontrollieren zu müssen ob Fehler auftreten, rückte dieses Problem dann etwas in den Hintergrund. Auch führte es dazu, dass ich bei der Entwicklung der Benutzeroberflächen erstmals die Vorschaufunktionen der Entwicklungsumgebungen nutzen lernte. Diese erlauben es einzelne Komponenten, während man sie entwickelt, in der Entwicklungsumgebung isoliert kompiliert zu betrachten, ohne die App auf einen Emulator oder Simulator zu installieren.

Keine große Überraschung waren für mich die aussagearmen Fehlerberichte bei Abstürzen der App. Speziell auf iOS war das Auslesen des Speichers und die Zurückverfolgung der ausgeführten Prozeduren völlig unbrauchbar. Statt Prozeduren mit den Symbolen die ich in Kotlin definiert hatte, wurde ich ausnahmslos mit Assemblerkommandos und Speicheraddressen, garniert mit der abschließenden zum Absturz führenden Fehlermeldung `EXC_BAD_ACCESS`, konfrontiert. Da diese Art der Fehlerdiagnose als iOS-Entwickler leider zu meinem Alltag gehört, ging ich nicht weiter auf die Suche nach einer möglichen Abhilfe, sondern versuchte zu erraten welche meiner Änderungen zu diesem Fehler geführt haben könnte. Gepaart mit der gerade erwähnten, langen Kompilierungszeit war diese Phase der Umsetzung eine besonders unschöne Angelegenheit. Viel zu spät bin ich in der offiziellen Dokumentation über die Möglichkeit gestolpert, die Debug-Symbole aus dem Kotlinmodul in das iOS-Projekt zu integrieren [31]. Vermutlich hätte sich diese Thematik darauf in Wohlgefallen aufgelöst. Bisher habe ich das allerdings nicht ausprobiert, da ich, ab einem gewissen Punkt der Erfahrung mit der Technologie, derartige Abstürze praktisch nicht mehr verursacht habe.

4.1.2 Drei-Sprachen-Problem

Swift und Kotlin ähneln sich von ihren syntaktischen Möglichkeiten sehr. Beides sind moderne Programmiersprachen, die ähnliche Foki auf Lesbarkeit, Sicherheit und funktionale Programmierung setzen. Das Wechseln zwischen den beiden Sprachen fällt dadurch sehr leicht. Die größte Hürde ist lediglich, dass sich manche Schlüsselwörter unterscheiden. Mit diesem Vorwissen und einiger Erfahrung in beiden Sprachen, ging ich davon aus, dass es ein Leichtes sein würde, in Kotlin definierte Datenstrukturen in Swift zu verwenden. Leider stellte sich, nach einigen Fehlschlägen dahingehend, heraus, dass Kotlin Native nicht direkt mit Swift interagiert. Das Kotlin Modul wird nach Kompilierung mit einer Objective-C Headerdatei versehen [32]. Diese Objective-C Headerdatei kann über die Objective-C-Swift-Brücke in Swift verwendet werden [33]. Als, relativ gesehen, alte Programmiersprache verfügt Objective-C nicht über viele der modernen Funktionalitäten, die Kotlin und Swift ausmachen.

Während sich also Kotlin und Swift sehr viele Sprachfunktionalitäten teilen, geht viel Information auf dem Weg über Objective-C verloren. Einige der Auswirkungen dieser Tatsache, von mir "Drei-Sprachen-Problem" getauft, und wie ich damit umgegangen bin, will ich hier aufzeigen.

```
1  enum State {
2    case loading
3    case error(errorMessage: String)
4    case content(isLoggedIn: Bool)
5  }
6  let state: State
7
8  switch state {
9    case .content(true): print("User is logged in")
10   default: break
11 }
12 switch state {
13   case .loading: print("Loading")
14   case .error(let message): print("Error: \(message)")
15   case .content(let isLoggedIn): print("User log in state: \(isLoggedIn)")
16 }
```

Beim Enumerationstypen in Swift können für jeden Fall weitere Informationen in Form von Datenstrukturen definiert werden. Diese sogenannten "assoziierten Datentypen" können dabei nicht nur Primitive, sondern jede Art von Typ, zum Beispiel auch Klassen und weitere Enumerationen, sein. Die Kontrollstruktur `switch` kann neben einem Musterabgleich auch die assozierten Daten auslesen und verwenden. Dabei bleibt die Abgeschlossenheit der Enumeration im Vordergrund. In einer `switch`-Definition müssen alle möglichen Fälle abgebildet werden. Sollte dies nicht gewünscht sein, kann auch ein Standardfall (siehe Zeile 9) definiert werden, der ausgeführt wird wenn die vorher aufgelisteten Fälle nicht zutreffen. Der Übersetzer stellt sicher, dass alle möglichen Fälle genannt werden. Ist dem nicht so, tritt ein Kompilierungsfehler auf. Diese beiden Eigenschaften machen die Enumeration in Swift zum perfekten Kandidaten um State zu modellieren. Es können alle möglichen Daten abgebildet werden und die Abgeschlossenheit führt dazu, dass auch wirklich alle möglichen States berücksichtigt werden.

Die Enumerationen in Kotlin teilen diese Eigenschaften zwar nicht, aber es gibt ein funktional deckungsgleiches Pendant: Die geschlossenen Klassen

```
sealed class State {
    object Loading: State()
    data class Error(val errorMessage: String): State()
    data class Content(val isLoggedIn: Boolean): State()
}
val state: State

when (state) {
    State.Content(true) -> println("User is logged in")
    else -> Unit
}
when (state) {
    State.Loading -> println("Loading")
    is State.Error -> println("Error: ${state.errorMessage}")
    is State.Content -> println("User log in state: ${state.isLoggedIn}")
}
```

Geschlossene Klassen sind ebenfalls abgeschlossen und können mit jeglichen Datentypen assoziiert werden. Die Syntax und Semantik bei der Verwendung ist offensichtlich sehr ähnlich, lediglich die Schlüsselwörter und Schlüsselzeichen unterscheiden sich.

Naiv würde man also meinen, dass geschlossene Klassen in Swift als Enumerationen abgebildet werden könnten. In der Realität gibt es in Objective-C kein Sprachkonstrukt, das diese beiden Eigenschaften vereint. Den generierten Objective-C Code verbanne ich aus Platz- und Verständnisgründen in den Anhang. Der durch die Objective-C-Swift-Brücke generierte, nutzbare Code für die geschlossene Klasse aus dem Beispiel sieht so aus:

```
public protocol State { }
public class StateLoading : KotlinBase, State { }
public class StateError : KotlinBase, State {
        open var errorMessage: NSString { get }
}
public class StateContent : KotlinBase, State {
  open var isLoggedIn: Bool { get }
}
```

Während hierbei die assoziierten Daten abgebildet sind, geht die Abgeschlossenheit offensichtlich komplett verloren. In jeder `switch`-Anweisung, die diese Struktur verwendet, müsste nun ein zusätzlicher allgemeiner Fall, welcher in der Realität niemals vorkommt, angegeben werden. Dadurch verliert der entstehende Code an Klarheit. In diesem speziellen Fall der geschlossenen Klassen, gibt es durch ein Kotlin Übersetzerplugin eines Drittanbieters Abhilfe [34]. Dieses Plugin generiert für jede geschlossene Klasse automatisch eine Swift Enumeration. In vielen weiteren Fällen gibt es aber leider keine Lösung für derartige Reibung.

Die größten Auswirkungen auf die Implementierung des Projekts hatte das Drei-Sprachen-Problem als es darum ging, die in Kotlin definierten ViewModels in SwiftUI

zu verwenden. Um für eine in SwiftUI definierte View als State-Anbieter fungieren zu können, muss eine Klasse das Swift-Protokoll `ObservableObject` implementieren. Dieses Protokoll ist Teil des SwiftUI Frameworks. Jetbrains stellt für Kotlin Multiplatform Mobile, in Kotlin sichtbare, Mäntel für iOS-Plattformframeworks zur Verfügung, was deren Nutzung im geteilten Code ermöglicht. Die Hoffnung war also, dass eine Basisklasse, von der alle ViewModels erben können, das `ObservableObject`-Protokoll implementieren kann und damit alle ViewModels in SwiftUI verwendbar gemacht werden. SwiftUI ist allerdings komplett in Swift geschrieben und, da Kotlin nur direkt mit Objective-C interagieren kann, damit nicht in KMM sichtbar. Der zweite Ansatz war, in Swift eine Mantelklasse für das Basis-ViewModel zu definieren, welche das Protokoll implementiert. Das Basis-ViewModel in Kotlin ist generisch über State, Route und Aktionen. Diese generische Information geht auf den Brücken in Richtung Swift leider ebenfalls verloren, so dass eine generische Lösung auf diesem Weg nicht möglich war.

Aus Platz- und Fokusgründen wird in dieser Arbeit auf die Ausführung der Lösung dieses Problem verzichtet. Sie erforderte ein tiefes Verständnis der Typsysteme aller drei Sprachen, sowie ein Arsenal an generischen Tricksereien, Mäntel um Kotlin Datentypen und SwiftUI-Definitionen. Ich verweise an dieser Stelle auf die Dokumentation des Projekts, in der ihr ein ganzes Kapitel gewidmet wird.

4.1.3 Einbindung von Bibliotheken

Kotlin Multiplatform bietet die Möglichkeit in Objective-C verfasste Bibliotheken einzubinden. Diese Bibliotheken können dann im geteilten Code verwendet werden. Soweit zumindest die Theorie. In der Praxis wollte ich die Bibliothek des gewählten Zahlungsdienstleisters Stripe integrieren. Daran bin ich bislang gescheitert. Das Artefakt des geteilten Kotlinmoduls ist auf zwei Wegen in eine iOS-App einzubinden. Zum einen kann ein Objective-C Framework kompiliert werden welches dann direkt im iOS Projekt eingebunden werden kann. Der andere Weg geht über, das in der iOS Entwicklung weit verbreitete Abhängigkeitsmanagementwerkzeug, CocoaPods [35]. Nach zwei Arbeitstagen, an denen ich mehrfach zwischen den beiden Wegen hin- und herwechselte weil ich von einer Sackgasse bei der Einbindung der Stripe-Bibliothek in die nächste stolperte, bin ich schlussendlich auf einen Vorgang auf der Webseite von Jetbrains gestoßen. In diesem Vorgang [36] wurde erwähnt dass es zum jetzigen Zeitpunkt nicht möglich ist, Bibliotheken einzubinden, in deren Objective-C Headerdateien sogenannte Paketimporte verwendet werden. Mittels Paketimport wird, statt mehrere Klassen aus einem Modul einzeln zu importieren, gleich das ganze Modul importiert. Stripe verwendet diese Art von Importsyntax um Module zum Zwecke der Darstellung von Benutzeroberflächenelementen aus dem Apple-Framework einzubinden. Die einzige Möglichkeit die mir zur Verfügung stand dieses Problem zu lösen, wäre das Forken der Stripe-Bibliothek gewesen. In diesem Fork hätte ich die Paketimporte durch entsprechende Klassenimporte austauschen können. Das resultierende Artefakt wäre dann vermutlich in KMM verwendbar gewesen. Ich habe mich schlussendlich gegen diese Lösung entschieden, da ich das Thema Zahlungsverkehr für zu heikel halte, um einen Fork zu erstellen, egal aus welchem Grund.

Die pragmatische Lösung war es dann, die ViewModel-Funktionalitäten, die die Stripe-Bibliotheken benötigen, im jeweiligen App-Modul zu implementieren. So konnte in der iOS-App die Stripe-Bibliothek für iOS eingebunden und benutzt werden, wie vom Zahlungsdienstleister vorgesehen. Genauso ging ich, der Konsistenz wegen, auf Android vor, auch wenn dies nicht zwingend nötig gewesen wäre.

Es bleibt aber ein Problem, dem ich mich in der Zukunft nochmals widmen will. ViewModel-Funktionalität sollte, um den formulierten architektonischen Wünschen und Vorgaben zu entsprechen, einzig und allein im geteilten Code implementiert werden.

4.2 Ergebnisse

4.2.1 Backend

Es wurde, plangemäß, lediglich so viel Backend implementiert, wie für die Demonstration der Apps nötig ist. Im Falle einer Weiterführung dieses Projektes würde die jetzige Backendumsetzung vermutlich weggeworfen werden. Der Grund ist schlicht, dass ich aktuell über zu wenig Domänenwissen verfüge, um ein Backend zu implementieren, welches den Ansprüchen an einen Produktivbetrieb genügt. Dieses Wissen anzueignen würde den Rahmen einer Bachelorarbeit sprengen.

Da nicht geplant ist, den Code weiter zu pflegen, wurden auch keine großen Ansprüche an die Qualitätssicherung gestellt. Es gibt weder Tests, noch viel Dokumentation. Aufgrund der Simplizität der Lösung und des verwendeten Frameworks ist der Code allerdings leicht zu lesen und zu verstehen.

Als Serverframework wurde Ktor verwendet. Ktor ist sehr leicht zu benutzen und hat zwei weitere große Vorteile für dieses Projekt gehabt. Zum einen ist es, genau wie die Apps auch, in Kotlin geschrieben was es ermöglichte, Schnittstellenobjektdefinitionen hin und her zu kopieren und damit Zeit zu sparen. Zum anderen wird in den Apps für die HTTP-Kommunikation Ktor als Client-Bibliothek eingebunden. Das Framework und die Bibliothek teilen sich einige Konzepte, was sich ebenfalls als nützlich herausstellte.
Zu finden ist der Backendcode auf GitHub unter der folgenden URL:

https://github.com/pumapaul/spenderino-backend

Commit-Hash: 2aa21af57a95cd3d77e53b5eb6d05f498571dd8d

Gehostet wird das Backend aktuell auf Heroku. Heroku ist ein Dienst des Unternehmens Salesforce, der verwendet wurde weil er kostenlos ist und die Einrichtung überraschenderweise in fünf Minuten möglich war. Leider fährt Heroku das Backend nach 30 Minuten Inaktivität in eine Art Schlafzustand herunter. Nach dem Aufwecken, was automatisch bei einem eintreffenden HTTP-Aufruf geschieht, ist der Arbeitsspeicher wieder leer, was die unschöne Nebenwirkung hat, dass damit auch alle Benutzerdaten wieder gelöscht sind. Das ist ein zweischneidiges Schwert, da sich so wenigstens nicht um das Löschen von Benutzerdaten gekümmert werden musste.

4.2.2 Apps

Die zentralen Ergebnisse dieser Bachelorarbeit sind die beiden mobilen Anwendungen. Diese können zur Ansicht wie folgt auf einem Smartphone installiert werden:

iOS: Zur Teilnahme am öffentlichen Beta-Test auf dem iPhone die TestFlight-App herunterladen und danach diese URL aufrufen: https://testflight.apple.com/join/nz6rnk9m

Android: Da die App zum Zeitpunkt der Fertigstellung dieser Arbeit noch nicht von Google für einen öffentlichen Test freigegeben wurde, muss sie von der folgenden URL heruntergeladen und installiert werden: https://paul-weber.de/apps/spenderino_6.apk

Der zugehörige Quelltext ist auf GitHub zu finden:

https://github.com/pumapaul/Spenderino

Commit-Hash: d26b84cb73999bb0e9f8d1c7cb0040b890486e24

Dort ist auch eine Dokumentation zu finden, die den Einstieg in den Code vereinfacht und die zentralen Konzepte erklärt.

4.2.3 Qualitätssicherung

Teil des Projekts auf GitHub ist eine sogenannte kontinuierliche Integration. Jeder Pull Request auf den Hauptbranch des Projekts stößt eine Reihe an automatisierten Abläufen an. Nur wenn jeder dieser angestoßenen Abläufe erfolgreich endet, kann der Pull Request als akzeptiert gelten. Schlägt nur ein Ablauf fehl, müssen weitere Commits folgen, bis keine weiteren Fehler mehr auftreten.

Zusammen mit der Regel, dass alle Commits eines akzeptierten Pull Requests beim Zusammenführen mit dem Hauptbranch zu einem einzigen Commit zusammengefasst werden, ist so sichergestellt, dass jeder einzelne Commit auf dem Hauptbranch zu jeder Zeit den Qualitätsansprüchen genügt, die durch die kontinuierliche Integration überwacht werden. Das sollte jede*n Entwickler*in freuen, die an dem Projekt arbeitet, denn so ist es ausgeschlossen dass man auf der Reise durch die Vergangenheit des Projekts auf Commits stößt, bei denen der Code nicht einmal kompiliert.

Die erste Art der Qualitätssicherung, die so automatisiert wurde ist das Linten. Dieser Begriff beschreibt den Prozess des Durchführens einer statischen Analyse des Quelltextes. Diese Analyse bewertet den Quelltext anhand von vielerlei Kriterien wie Komplexität von Funktionen, Länge von Klassen, Duplizierung von Codebausteinen und vielen weiteren sogenannten “Code Smells”. Eine weitere nützliche Funktion der verwendeten Linter ist die Einhaltung von, in Konfigurationsdateien festgelegten, Stilrichtlinien an den Code. Dies betrifft zum Beispiel den weißen Raum zwischen Codezeilen oder dass Leerzeichen zwischen Operatoren gesetzt sind und vieles mehr. Im Allgemeinen hilft dieser Schritt der kontinuierlichen Integration also dabei, den Quelltext lesbar und zugänglich zu machen. Ist er dies nicht, schlägt der Schritt fehl.

Die restlichen Integrationsabläufe befassen sich mit dem Ausführen von automatisierten Tests. Neben dem erfolgreichen Ausführen der Tests an sich, wird somit sicher gestellt, dass die beiden Apps auch tatsächlich fehlerfrei kompilieren. Das ist besonders dann interessant wenn eine Änderung nur in der Android-App vorgenommen wurde, aber durch das geteilte Codemodul auch die iOS Codebasis betrifft. Wird zum Beispiel bei einer Android-Refaktorisierungsaktion eine Klasse im geteilten Code umbenannt, muss die entsprechende Benutzung dieser Klasse im Swift Code angepasst werden. Würde dies nicht automatisch getestet, könnte das leicht vergessen werden. Um nicht bei jeder geänderten Zeile alle Testsuiten laufen zu lassen, werden diese nur angestoßen wenn auch wirklich etwas zu Grunde liegendes verändert wurde. Wird zum Beispiel nur Swift Code verändert, muss lediglich die iOS-Testsuite laufen. Wird jedoch etwas im

geteilten Modul geändert, müssen alle Suiten angestoßen werden, da dies auf alle drei Bereiche Einfluss haben kann.

Die Testsuite des geteilten Moduls schlägt ebenfalls fehl, wenn die Testabdeckung unter eine gewisse Grenze fällt. Während Testabdeckung von Zeilen nicht generell als Metrik für eine gelungene Testsuite herhalten sollte [37], zwingt ein Minimalwert hier wenigstens zum Nachdenken, wie hinzugefügter Code getestet werden kann. Die wirkliche Qualität von geschriebenen Tests kann nach meiner Einschätzung nur ein Code-Review wirklich beurteilen.

5 Evaluation

Ein Wunsch zu Beginn der Arbeit an dieser Bachelorarbeit war, die entstandenen Apps von echten Benutzer*innen testen zu lassen. Auf Basis der daraus entstandenen Evaluation, sollte gegebenenfalls die Benutzererfahrung angepasst werden. Die pandemische Situation mit einer zeitweiligen Inzidenz von über drei Prozent machte dies, meiner Einschätzung nach, jedoch unmöglich. Die Teilzielgruppe, bestehend aus wohnungslosen Menschen, ist aufgrund des Mangels an Möglichkeiten zur Quarantäne und vielerlei anderen, den Umständen geschuldeten Gründen, besonders schutzbedürftig was das Coronavirus angeht.

Als Alternative habe ich Menschen in meinem Umfeld gebeten, die App auszuprobieren. Zehn iOS-Nutzer*innen und vier Androidnutzer reichen zwar nicht für eine wirkliche quantitative Untersuchung der Benutzerfreundlichkeit der Apps aus, aber konnten doch einige Mängel im Design aufzeigen.

Die Tester*innen erhielten Zugang zu den Apps über die jeweilige Distributionsplatform für Beta-Versionen. Bei manchen Testläufen war ich persönlich dabei, bei anderen wiederum nur über Telefon oder Videochat. Als Vorabinformationen gab es nur eine Beschreibung der Idee des Produkts mit der Zusatzinformation dass QR-Codes Teil des Konzepts sind. Alle Tester*innen haben dabei nacheinander beide Rollen eingenommen. Jeweils zur Hälfte wurde als Spender*in angefangen. Die andere Hälfte fing als Empfänger*in an.

Neben Korrekturen der erklärenden Texte und Beschwerden über fehlende erklärende Grafiken, wurde hauptsächlich bemängelt, dass der Prozess des Auszahlens von eingegangen Spenden unnötig kompliziert versteckt ist. Dies empfanden ganze 14/14 der Tester*innen.

Das Spenden und die Einrichtung eines Benutzerkontos, so dass Spenden empfangen werden können, hingegen wurde von allen als benutzbar bewertet. Bei den Zahlungsmitteln wurde sich noch die Möglichkeit, Google Pay benutzen zu können, gewünscht aber das war es auch schon.

5.1 Auswirkungen

Die lokalisierten Texte wurden angepasst und sollten nun besser verständlich sein.

Die Auszahlungsfunktionalität war vor der Evaluation hinter zwei vertikalen Navigationsebenen im Einstellungs-Tab versteckt. Nach dem einhelligen Feedback wurde sich dazu entschieden, diese zentrale Funktionalität direkt auf dem, bei Start der App angewählten, Home-Tab zu realisieren. So haben Nutzer*innen, die Spenden empfangen wollen, einen zentralen Anlaufpunkt bei dem sie sowohl ihren QR-Code, als auch die Möglichkeit zum Auszahlen des angesammelten Guthabens vorfinden.

10 der 14 Tester*innen haben sich die neue Version angeschaut und die Rückmeldungen zur neuen Erfahrung fielen durch die Bank weg positiv aus.

6 Fazit & Ausblick

6.1 Fazit

Es soll nun rückblickend betrachtet werden, ob die anfangs formulierten Ziele erreicht wurden. Was den Funktionsumfang des Produkts angeht, kann man sagen, dass dies der Fall ist. Die drei zentralen Anwendungsfälle sind, soweit es möglich war, abgebildet. Die Benutzertests zeigten, dass die endgültige Benutzererfahrung zufriedenstellend war. Etwas enttäuschend war für mich persönlich dass die eingangs erwähnten Betriebssytemfunktionen App Clip und Instant App nicht zu realisieren waren. Die Versuche, die ich in die Richtung unternahm, kosteten zudem wertvolle Zeit, die ohnehin recht knapp war.

Was den Zustand der Codebasis angeht, bin ich persönlich sehr zufrieden. Nach anfänglichen Schwierigkeiten in der praktischen Umsetzung, konnte der architektonische Ansatz, als er schematisch wurde, sein Potential voll entfalten. Die Simplizität dessen, implementiert in einem von den beiden Plattformen komplett geteilten Modul, hat es einfach gemacht, neue Funktionalitäten und Ansichten zu implementieren. Selbst als es am Ende darum ging, die Auszahlungsfunktionalität von einer Ecke der App in eine ganz andere zu verlagern, war es einfach und schnell möglich beide Apps zeitgleich in einem Rutsch abzuändern.

Wie oben bereits erwähnt, stellte mich die Einbindung von iOS-Bibliotheken nicht zufrieden. Dort werde ich auf jeden Fall noch weitere Nachforschungen anstellen. Die Implementierung der Navigation auf Android funktioniert zwar tadellos, entspricht allerdings nicht ganz meiner Vorstellung. Die von Jetpack Compose angebotene Art der Navigation ist sehr verbos, imperativ und passt nicht zum restlichen, deklarativen Ansatz des Frameworks. Dies hat mich im Zusammenspiel mit meiner Architektur so frustriert, dass ich plane, nach dem Abschluss der Bachelorarbeit, eine Bibliothek für echt deklarative Navigation in Jetpack Compose zu entwerfen.

Sollte ich in Zukunft mit Projekten für beide mobilen Plattformen zu tun haben, bei denen es darum geht, auf der grünen Wiese etwas neues zu entwerfen, würde ich jederzeit wieder zu Kotlin Multiplatform und einer Architektur, die der dieses Projektes ähnelt, greifen. Vermutlich würde ich nur den Android-Teil zuerst machen damit ich nicht ständig auf den Übersetzer warten muss.

6.2 Ausblick

Die beiden Apps sind als Prototypen eine gute Basis für die Zukunft des Projekts. Die nächsten Schritte werden die Schaffung eines juristischen Unterbaus, sowie die Abstimmung mit befreundeten UI-Designern sein. Ebenfalls sehr interessiert bin ich daran, ein produktionsreifes Backend zu schaffen. Dies würde in einem ersten Schritt den Anschluss der plattformeigenen Authentifizierungsdiensten ermöglichen, welcher nochmal einiges an Reibung in der Nutzerakquise verringern würde. Perspektivisch plane ich, gegen Ende des Jahres ein veröffentlichungsfähiges Produkt geschaffen zu haben.

Technologisch kann ich mir vorstellen, Teile der Codebasis auszulösen, um ein Beispielprojekt für die Architektur von, in Kotlin Multiplatform geschriebenen, Apps zu

erstellen. Soweit ich das bisher überblicken konnte, gibt es zu wenig Grundlagenarbeit in diese Richtung und ich bin der Auffassung dass der im Rahmen dieser Arbeit eingeschlagene Weg zumindest kein schlechter ist. Besonders freue ich mich in diesem Zusammenhang auf Rückmeldungen von anderen, in Kotlin Multiplatform erfahreneren Entwicklern.

Literatur

[1] Robert Peters Simone Ehrenberg-Silies und Christian Wehrmann. "Welt ohne Bargeld – Veränderungen der klassischen Banken- und Bezahlsysteme". Thesenpapier. Büro für Technikfolgenabschätzung, Deutscher Bundestag, 2020, S. 3–10. URL: https://www.bundestag.de/resource/blob/699638/6dde07478be95dce09864eab16ba67cc/Thesenpapier-data.pdf.

[2] Hugo Grote. "Der Rechtsanspruch auf ein Basiskonto nach dem Zahlungskontengesetz". In: *Zeitschrift für das gesamte Insolvenzrecht* 19.25 (2016), S. 1239–1243. DOI: doi:10.1515/zinso-2016-2507. URL: https://doi.org/10.1515/zinso-2016-2507.

[3] *Unwirksame Entgeltklausel für Basiskonto*. XI. Zivilsenat des Bundesgerichtshofs. URL: https://www.bundesgerichtshof.de/SharedDocs/Pressemitteilungen/DE/2020/2020084.html.

[4] *Bitcoin ATM*. Wikipedia. URL: https://en.wikipedia.org/wiki/Bitcoin_ATM.

[5] *Barzahlen.de*. viafintech GmbH. URL: https://www.barzahlen.de/.

[6] Xiong Li u. a. "Measuring ease of use of mobile applications in e-commerce retailing from the perspective of consumer online shopping behaviour patterns". In: *Journal of Retailing and Consumer Services* 55 (2020), S. 102093. ISSN: 0969-6989. DOI: https://doi.org/10.1016/j.jretconser.2020.102093. URL: https://www.sciencedirect.com/science/article/pii/S0969698919304485.

[7] Saeid Saeida Ardekani Fereshte Rasty Seyyed Habibollah Mirghafoori und Peyman Ajdari. "Trust barriers to online shopping: Investigating and prioritizing trust barriers in an intuitionistic fuzzy environment". Magisterarb. International Journal of Consumer Studies, 2020, S. 1031–1033. URL: https://onlinelibrary.wiley.com/doi/10.1111/ijcs.12629.

[8] *App Clips*. Apple Inc. URL: https://developer.apple.com/app-clips/.

[9] *Google Play Instant - Native Android apps, without the installation*. Google LLC. URL: https://developer.android.com/topic/google-play-instant/.

[10] *Sign in with Apple*. Apple Inc. URL: https://developer.apple.com/sign-in-with-apple/.

[11] *Sign In With Google*. Google LLC. URL: https://developers.google.com/identity/gsi/web.

[12] *App Store Review Guidelines*. Apple Inc. URL: https://developer.apple.com/app-store/review/guidelines/%5C#legal.

[13] *Richtlinien zu Nutzerdaten*. Google LLC. URL: https://support.google.com/googleplay/android-developer/answer/10144311?hl=de&ref_topic=9877467%5C#.

[14] *Test your integration*. Stripe, Inc. URL: https://stripe.com/docs/testing%5C#cards.

[15] *Mobile Operating System Market Share Germany*. Statcounter. URL: https://gs.statcounter.com/os-market-share/mobile/germany.

[16] *Market share of mobile operating systems in Germany from 2010 to 2021*. Statista Inc. URL: https://www.statista.com/statistics/693829/market-share-mobile-operating-systems-germany/.

[17] Lukas Dagne. "Flutter for cross-platform App and SDK development". Bachelorarbeit. Metropolia University of Applied Sciences, 2019, S. 6–7. URL: https://www.theseus.fi/bitstream/handle/10024/172866/Lukas%20Dagne%20Thesis.pdf?.

[18] Oliver Gill. "Using React Native for mobile software development". Bachelorarbeit. Metropolia University of Applied Sciences, 2018, S. 1–4. URL: https://www.theseus.fi/bitstream/handle/10024/143282/Gill_Oliver.pdf?sequence=1&isAllowed=y.

[19] Awel Eshetu Fentaw. "Cross platform mobile application development: a comparison study of React Native Vs Flutter". Masterarbeit. Faculty of Information Technology, University of Jyväskylä, 2020. URL: https://jyx.jyu.fi/handle/123456789/70969.

[20] Rasmus Eskola. "React Native Performance Evaluation". Masterarbeit. School of Science, Aalto University, 2018, S. 40–49. URL: https://aaltodoc.aalto.fi/handle/123456789/32475.

[21] *Under the hood: Building Moments*. Meta Platforms, Inc. URL: https://engineering.fb.com/2015/06/15/android/under-the-hood-building-moments/.

[22] Wallace Jackson. *Android Apps for Absolute Beginners*. Apress, 2017, S. 33. ISBN: 978-1-4842-2268-3.

[23] Bruno Gois Mateus und Matias Martinez. *An empirical study on quality of Android applications written in Kotlin language*. 2019. URL: https://link.springer.com/article/10.1007/s10664-019-09727-4.

[24] *Android's Kotlin-first approach*. Google LLC. URL: https://developer.android.com/kotlin/first.

[25] *Stability of Kotlin components*. JetBrains s.r.o. URL: https://kotlinlang.org/docs/components-stability.html.

[26] Matthew A. Brown und Eli Tapolcsanyi. "Mock Object Patterns". Paper. 2003. URL: https://hillside.net/plop/plop/plop2003/Papers/Brown-mock-objects.pdf.

[27] T. Lou. "A comparison of Android Native App Architecture MVC, MVP and MVVM". Masterarbeit. Eindhoven University of Technology / Aalto University, 2016, S. 18–19. URL: https://pure.tue.nl/ws/portalfiles/portal/48628529/Lou_2016.pdf.

[28] Wat Wongtanuwat und Twittie Senivongse. "A comparison of Android Native App Architecture MVC, MVP and MVVM". Paper. Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, 2020, S. 55–56. URL: https://dl.acm.org/doi/10.1145/3411174.3411193.

[29] *Data layer*. Google LLC. URL: https://developer.android.com/jetpack/guide/data-layer/.

[30] JetBrains s.r.o. *Tips for improving Kotlin/Native compilation times*. Kotlin Foundation. URL: https://kotlinlang.org/docs/native-improving-compilation-time.html.

[31] JetBrains s.r.o. *Symbolicating iOS crash reports*. Kotlin Foundation. URL: https://kotlinlang.org/docs/native-ios-symbolication.html.

[32] JetBrains s.r.o. *Interoperability with Swift/Objective-C*. Kotlin Foundation. URL: https://kotlinlang.org/docs/native-objc-interop.html.

[33] *Importing Objective-C into Swift*. Apple Inc. URL: https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis/importing_objective-c_into_swift.

[34] Aleksey Mikhailov. *MOKO KSwift - Kotlin sealed interface/class to Swift enum*. IceRock Development. URL: https://github.com/icerockdev/moko-kswift.

[35] *CocoaPods Webseite*. CocoaPods. URL: https://cocoapods.org.

[36] *Ilya Matveev*. JetBrains s.r.o. URL: https://youtrack.jetbrains.com/issue/KT-39120.

[37] Laura Inozemtseva und Reid Holmes. "Coverage Is Not Strongly Correlated with Test Suite Effectiveness". Paper. School of Computer Science, University of Waterloo, 2014, S. 439–442. URL: https://dl.acm.org/doi/10.1145/2568225.2568271.

A Anhang

Beispiel für den generierten Objective-C Quellcode einer geschlossenen Klasse aus Kotlin.

A.1 Kotlin

```kotlin
    sealed class State {
        object Loading: State()
        data class Error(val errorMessage: String): State()
        data class Content(val isLoggedIn: Boolean): State()
    }
```

A.2 Objective-C

```objc-c
__attribute__((swift_name("State")))
@interface SharedState : SharedBase
- (instancetype)init __attribute__((swift_name("init()")))
↪  __attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable,
↪  message="use object initializers instead")));
@end;

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("State.Content")))
@interface SharedStateContent : SharedState
- (instancetype)initWithIsLoggedIn:(BOOL)isLoggedIn
↪  __attribute__((swift_name("init(isLoggedIn:)")))
↪  __attribute__((objc_designated_initializer));
- (instancetype)init __attribute__((swift_name("init()")))
↪  __attribute__((objc_designated_initializer))
↪  __attribute__((unavailable));
+ (instancetype)new __attribute__((unavailable));
- (BOOL)component1 __attribute__((swift_name("component1()")));
- (SharedStateContent *)doCopyIsLoggedIn:(BOOL)isLoggedIn
↪  __attribute__((swift_name("doCopy(isLoggedIn:)")));
- (BOOL)isEqual:(id _Nullable)other
↪  __attribute__((swift_name("isEqual(_:)")));
- (NSUInteger)hash __attribute__((swift_name("hash()")));
- (NSString *)description __attribute__((swift_name("description()")));
@property (readonly) BOOL isLoggedIn
↪  __attribute__((swift_name("isLoggedIn")));
@end;

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("State.Error")))
@interface SharedStateError : SharedState
```

```
24 - (instancetype)initWithErrorMessage:(NSString *)errorMessage
   ↪  __attribute__((swift_name("init(errorMessage:)")))
   ↪  __attribute__((objc_designated_initializer));
25 - (instancetype)init __attribute__((swift_name("init()")))
   ↪  __attribute__((objc_designated_initializer))
   ↪  __attribute__((unavailable));
26 + (instancetype)new __attribute__((unavailable));
27 - (NSString *)component1 __attribute__((swift_name("component1()")));
28 - (SharedStateError *)doCopyErrorMessage:(NSString *)errorMessage
   ↪  __attribute__((swift_name("doCopy(errorMessage:)")));
29 - (BOOL)isEqual:(id _Nullable)other
   ↪  __attribute__((swift_name("isEqual(_:)")));
30 - (NSUInteger)hash __attribute__((swift_name("hash()")));
31 - (NSString *)description __attribute__((swift_name("description()")));
32 @property (readonly) NSString *errorMessage
   ↪  __attribute__((swift_name("errorMessage")));
33 @end;
34
35 __attribute__((objc_subclassing_restricted))
36 __attribute__((swift_name("State.Loading")))
37 @interface SharedStateLoading : SharedState
38 + (instancetype)alloc __attribute__((unavailable));
39 + (instancetype)allocWithZone:(struct _NSZone *)zone
   ↪  __attribute__((unavailable));
40 - (instancetype)init __attribute__((swift_name("init()")))
   ↪  __attribute__((objc_designated_initializer))
   ↪  __attribute__((unavailable));
41 + (instancetype)new __attribute__((unavailable));
42 + (instancetype)loading __attribute__((swift_name("init()")));
43 @property (class, readonly, getter=shared) SharedStateLoading *shared
   ↪  __attribute__((swift_name("shared")));
44 @end;
```