Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Muster im User Interface Re-Engineering in ein deklaratives User Interface Framework

Adrian Meister (geb. Zubarev)
Matrikelnummer: 4677686
zubarev@zedat.fu-berlin.de

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

Berlin, 11. September 2022

Zusammenfassung

Hintergrund - Unleashed ist eine mobile Anwendung, die ein gleichnamiges Hardwaremodul drahtlos mit Hilfe der Bluetooth Low Energy steuert. Es bietet dem / der Nutzer*in eine nahezu vollstände Kontrolle über eine digitale Kamera, an der die Hardwarekomponente angeschlossen ist. Der Lebenszyklus der imperativen Views dieser App wird als erreicht betrachtet.

Ziele - Die Suche nach potentiellen Muster in dem Reengineering Prozess braucht eine gewisse Menge an Volumenarbeit, welche durch die Reimplementierung der Views mit einem deklarativen UI-Framework erfolgte. Dabei war es wichtig die Entscheidungen und die Erkenntnisse in diesem Vorgang festzuhalten.

Methoden - Die Implementierung der deklarativen Views verlangt nach einer Funktionsumfangsanalyse, welche je nach Komplexität der Views eine Planung von möglichen Optionen impliziert. Jede der unterschiedlichen Optionen benötigt eine eigene Recherche und viele experimentelle Versuche.

Ergebnisse - Die entwickelten Views und die angewandte Methodik zeigen die Vorteile als auch mögliche Nachteile und die Risiken für den Umstieg auf das deklarative UI-Framework.

Schlussfolgerungen - Das in dieser Arbeit entwickelte Swift Package dient als eine Grundlage für deklarative UI-Basis der Unleashed App, wobei die angewandten Techniken sowie dabei gefundene Muster zur Einschätzung der Komplexität der Weiterentwicklung verwendet werden können.

Abstract

Background - Unleashed is a mobile Application, which was developed to communicate with the hardware component that goes by the same name. The App wirelessly connects to the Unleashed module via Bluetooth Low Energy. The user of the App is able to control a digital camera, to which the Unleashed module is attached. The lifecycle of the imperative UI-components of the app has reached its end.

Objectives - The observation for potential patterns hidden inside the reengineering process requires a certain amount of work, which was done by reimplementing some application views into the declarative paradigm.

Methods - The implementation task requires an analysis of the base functionality for each view and depending on its complexity it also implies an individual consideration of the possible options. Each individual options needs its own research and a set of experimental work.

Results - The procedures during the development of the views and the developed views from this thesis highlight the advantages as well as possible disadvantages and risks for the switch into the declarative UI-paradigm.

Conclusions - The Swift package developed for this thesis represents a base for the UI of the application. The concrete procedures and potential patterns determined during the process could be further used to estimate the complexity and the development of the package.

Inhaltsverzeichnis

1 Einführung

1.1 Unleashed Applikation

Unleashed ist eine, für das gleichnamige Hardwaremodul [77], zur Fernsteuerung von digitalen Kameras entwickelte mobile Anwendung (im folgenden nur noch als Applikation oder App bezeichnet). Die Unleashed App wird von Foolography GmbH [17] entwickelt. Es gibt die App für zwei mobile Betriebssysteme, iOS [78] und Android [79].

Die Bezeichnung "Unleashed", welche vom Englischen übersetzt "Entfesselt" bedeutet, symbolisiert die vielen Möglichkeiten, welche die App und das Hardwaremodul im Einklang für den Endkunden*innen ermöglichen, und dies Drahtlos. Die Bandbreite der App-Features reicht von der nahezu vollständigen Einstellungsmöglichkeit der Kamera bis zur gleichzeitigen Automatisierung von mehreren Kameras. Dabei erfolgt die Kommunikation zwischen der App und der Hardwarekomponente mit Hilfe der Bluetooth Low Energy (kurz BLE) [4] Funktechnik.

Diese Arbeit setzt sich speziell mit dem Reengineering Prozess der Benutzeroberfläche der App für das Betriebssystem iOS auseinander. Im folgenden wird für Benutzeroberfläche nur noch der englisch gleichwertige Begriff "User Interface" oder dessen Abkürzung "UI" verwendet.

1.2 Motivation

Eine "View" ist eine in der Domäne alternative Bezeichnung für eine UI-Komponente. Sie ist ein UI-Datentyp im Programmcode. Das verwendete UI-Framework kann eine Instanz einer Views rendern und anschließend auf den Gerätebildschirm projizieren. Es ist im Sinne der App, den Zustand einer solchen View-Instanz möglichst immer korrekt zu halten. Die Replikation von View-Datentypen erhöht den Aufwand nicht nur im Bereich der Wartung des App-Projekts sondern auch in Behebung von Business-logik-Fehlern. Der / die Entwickler*in ist gezwungen die Änderungen mehrfach vorzunehmen und es besteht die Gefahr, dass einige Views nicht wie vorgesehen aktualisiert werden, da diese in Vergessenheit geraten.

Die Unleashed App baut auf eine Reihe selbstentwickelter UI-Komponenten auf. Eben diese Komponenten befinden sich in einem für Wartung und Weiterentwicklung stark eingeschränktem Zustand. Große Teile der internen UI-Implementierung besitzen wenig Dokumentation, sind abgewandelte Kopien voneinander oder haben andere schwerwiegende Probleme.

Viele der UI-Komponente der Unleashed App sind bereits an mehreren Refactoring-Anläufen gescheitert. UI-Refactoring von visuellen Storyboards bis hin zur Verwendung von Drittanbieter-UI-Frameworks oder dem hauseigenen Versuch das deklarative Paradigma [11] in der imperativen Basis mit Hilfe von Layout-Driven-UI [1] nachzuahmen.

Aus der Sicht der Entwicklung befindet sich der App-Code an einer Schwelle, wo davon ausgegangen wird, dass die imperative UI-Basis Implementierung ihren Lebenszyklus erreicht hat. Dieser Zustand rechtfertigt den Versuch ein vollständiges Reengineering der UI der Unleashed App mit einem deklarativen Framework vorzunehmen, um damit die Stabilität der App auf der UI-Ebene wiederherzustellen und

im Allgemeinen die App mit dem neuen deklarativen Paradigma zukunftssicher zu machen.

1.3 Ziele

1.3.1 User Interface

Das erste große Ziel ist das Reengineering der UI. Der Funktionsumfang soll sich stark an das Original orientieren. Dabei darf die Implementierung auch Abweichungen aufweisen, sofern die gezielte Funktionalität verbessert werden wird. Generisch entwickelte Views haben die Möglichkeit ein größeres Spektrum an Funktionalitäten abdecken zu können. Eine generische Implementierung sollte sich auf das nötigste beschränken.

Idealerweise soll eine große Menge an UI-Komponenten in eine deklarative Struktur überarbeitet werden, während der damit verbundene Aufwand in Grenzen gehalten wird und die Komplexität des Codes deutlich reduziert wird.

1.3.2 Muster

Das zweite große Ziel dieser Bachelorarbeit erschließt sich aus der Volumenarbeit des Reengineering-Prozesses. Dabei wird auf potentielle Muster acht gegeben. Gänzlich neue Muster zu finden ist hierbei keineswegs eine Voraussetzung, aber auch keine Einschränkung dieser Arbeit. Ob es überhaupt irgendwelche Muster geben wird, gilt es mit dieser Bachelorarbeit herauszufinden.

Falls Muster ausfindig gemacht werden, ist es außerdem ein Ziel, diese nach der Bachelorarbeit in der Praxis weiter anwenden zu können.

1.4 Ansprüche an den Quelltext

Der Quellcode wird dokumentiert und zusätzlich ausreichend mit Kommentaren versehen. Ein / eine Entwickler*in mit SwiftUI Vorkenntnissen soll in der Lage sein, das Swift Package relativ schnell aufgreifen zu können und dieses mit wenig Aufwand zu nutzen.

Dabei ist auf den Einsatz von Fremdbibliotheken zu verzichten. Die UI-Komponenten sollen in einem eigenständigen Swift Package gebaut werden. Weitere Implementierungen zur demonstration dieser UI-Komponenten benötigen ein zusätzliches minimales App-Projekt. Sofern es als sinnvoll erscheint, sollen die Views sich nativ und somit wie hauseigene SwiftUI Komponente anfühlen. Dies sorgt dafür, dass die spätere Integration Reibungslos geschehen kann.

1.4.1 Qualitätskontrolle der Implementierung

UI-Tests werden im Rahmen diese Bachelorarbeit etwas gelockert. Bisher gibt es für SwiftUI keine offizielle Testbibliothek, daher werden die UI-Tests in Form von manuellen Testszenarien erzielt. Diese sollen möglichst alle oder die wichtigsten Testfälle zur Laufzeit überprüfen.

1.4.2 Einschränkungen der Implementierungsarbeiten

Im folgenden Abschnitt werden die Abgrenzungen der Implementierungsarbeiten näher erläutert.

Das User Interface einer mobilen App besteht nicht nur aus einer reinen Komposition von Views. Diese benötigt oftmals eine zusätzliche Navigationshierarchie. Während eine solche Hierarchie sehr eng mit den Views verknüpft ist, so liegt diese dennoch nicht im Rahmen dieser Bachelorarbeit. Die Navigationskomponente der Unleashed App ist relativ einfach aufgebaut. Sie beruht auf eine Kombination von Stack-Navigation (vgl. [68]) sowie modalen Präsentationen (vgl. [43]).

Es ist nicht das Ziel die neuen UI-Komponente zum Zeitpunkt der Bachelorarbeit in die App zu integrieren oder einen Zustand zu erreichen, welcher für diesen Zweck ideal wäre. Es reicht vollkommen aus, wenn die neuen Views dem originalen Funktionsumfang decken.

Für die Erkennung und Auswertung potentieller Muster ist es notwendig ein ausreichendes Volumen an Implementierungsarbeit zu leisten. Die Bearbeitung aller UI-Komponenten der Unleashed App würde jedoch den zeitlichen Rahmen der Arbeit sprengen. Daher wird nur eine sorgfältig selektierte Teilmenge von möglichen Views der App bearbeitet.

Die meisten stilistischen Anteile von Views nachzuempfinden, wäre sehr zeitintensiv tragen aber erwartungsgemäß keinen nennenswerten Teil der Musterfindung bei. Daher sind auch diese größtenteils zu vernachlässigen.

1.5 Ansprüche an die Muster

Die Muster im Endergebnis sollen in einer einfachen Textform dargestellt werden. Es ist wünschenswert, dass anhand solcher Muster ein Projektmanagement eine Einschätzung erhält, wie eine mögliche Vorgehensweise für eine Transformation in deklarative SwiftUI-Basis aussieht, welche potentiellen Schwierigkeiten und Zeitaufwand mit sich bringt.

2 Stand der Kunst

2.1 Imperatives Framework

Das User Interface der Unleashed App ist mit dem imperativen UIKit [66] Framework aufgebaut. UIKit ist ein von Apple entwickeltes Framework, dass für den Aufbau und Verwaltung von grafischer, event-basierter UI für iOS und tvOS Apps verwendet werden kann (vgl. Definition [66]). Dieses wurde ursprünglich mit der Objective-C Programmiersprache [35] implementiert.

Die imperative Programmierung ist ein Programmierparadigma welches durch eine Abfolge von Anweisungen die konkrete Änderung am Zustand des Programs beschreibt (vgl. [24]).

2.2 Deklaratives Framework und Swift-DSL

Der Wechsel des Programmierparadigma im Kontext von UI-Entwicklung leitete Apple im Jahre 2019 mit der Vorstellung des neuen SwiftUI Frameworks [88]. Auf den ersten Blick erscheint die deklarative Syntax als magisch, denn sie ist kurz, relativ einfach zu verstehen und sehr clever aufgebaut. Es sind im Grunde nur zwei Stützpfeiler, auf dem das Framework zum großen basiert.

2.2.1 Result Builder

Mit der Veröffentlichung von SwiftUI erhielt Swift die Möglichkeit einzelne Datentypen mit dem @resultBuilder-Attribut [49] zu versehen (im Jahre 2019 noch bekannt als @_functionBuilder) und daraus ein Result Builder [48] zu machen. Dieses Attribut ist das Kernelement für die Erstellung von Swift-DSLs. Swift-DSL steht dabei für Swift Domain Specific Language, es dient hauptsächlich der Definition von deklarativen Syntax (vgl. mit resultBuilder-Attribut Beschreibung [58]).

Die deklarative Syntax von SwiftUI baut auf einigen solcher Result Builder Datentypen auf. Das markanteste Beispiel ist der 'ViewBuilder' [82], der die deklarative Komposition von View-Elementen ermöglicht.

2.2.2 Property Wrapper

Um den Zustand für den deklarativen Ansatz zu ermöglichen, implementiert SwiftUI eine enorm große Anzahl von sogenannten Property Wrapper [45] Datentypen, welche genau wie Result Builder mit einem @propertyWrapper-Attribut [46] gekennzeichnet werden.

Es folgt ein konkretes Beispiel für den Einsatzzweck der Property Wrapper.

```
1 var stringValue: String = "Bachelorarbeit".upperCased() {
2   didSet {
3     stringValue = stringValue.upperCased()
4   }
5 }
```

Listing 1: Einfache String-Manipulation

In dem Listing 1 ist eine Beispiel Variable welche sowohl in der Initialisierungsphase, als auch während der Änderung ihres Zustands die finale Zeichenkette in Großbuchstaben transformiert.

Diese Logik wiederholt zu kopieren ist nicht ideal und es kann durchaus vorkommen, dass durch unsaubere Replikation die initiale Transformation in Vergessenheit gerät.

Das ist eine ideale Aufgabe, welche ein Property Wrapper übernehmen kann.

```
1 @propertyWrapper
2 struct Uppercased {
3   var _value: String
4   init(wrappedValue: String) {
5     self._value = wrappedValue.uppercased()
6   }
```

```
7    var wrappedValue: String {
8      get { _value }
9      set {
10       _value = newValue.uppercased()
11     }
12   }
13 }
14
15 @Uppercased
16 var stringValue: String = "Bachelorarbeit"
17
18 print(stringValue) // "BACHELORARBEIT"
```

Listing 2: Property Wrapper Beispiel

Das Listing 2 zeigt eine mögliche Implementierung eines solchen Property Wrappers und dessen einfache Anwendung an der stringValue-Variable. Wird die Implementierung des Property Wrappers angepasst, so wird sich das geänderte Verhalten automatisch an allen Stellen, wo der dieser zum Einsatz kam, aktualisieren. Fehler durch Replikation von solcher Business-logic gehören der Vergangenheit an.

Die verschiedenen Property Wrapper aus dem SwiftUI Framework ermöglichen das Arbeiten und die Manipulation von Programmzuständen aus dem deklarativen Kontext heraus.

Im Gegensatz zur imperativen Programmierung im UI-Kontext, ist die deklarative Programmierung ein Programmierparadigma, welches direkt den finalen Programmzustand, welcher von dem UI-Framework zu realisieren ist, beschreibt (vgl. [24]).

Es ist eben genau diese Balance aus Result Builder Typen und der Property Wrapper, welche die Schönheit und die Flexibilität des deklarativen SwiftUI Frameworks ausmachen.

2.3 MVVM als populäre Architektur

Eins der populären Architektur-Mustern sowie in UIKit als auch in SwiftUI ist das Model-View-ViewModel (kurz MVVM) [30]. Diese Architektur ermöglicht eine Trennung des Models von der View, indem es zur Verlagerung des Models hinter das ViewModel Bindeglied kommt. Dabei hat die View keinerlei Kenntnis über das Model und umgekehrt.

SwiftUI ist mit drei Jahren immer noch relativ jung. Viele Entwickler fragen sich, welche Architektur für SwiftUI Applikation am geeigneten wäre. Jedoch gab Apple bisher keine klare Antwort auf diese Fragen (vgl. mit Fragen und Antworten zur Architekturen in SwiftUI [16]). Ähnlich ist es auch mit der MVVM Architektur. Zwar ist diese sehr weit verbreitet, so existieren jedoch plausible Argumente diese Architektur in SwiftUI nicht einzusetzen. Es ist nämlich möglich die meisten View-Attribute aus der ViewModel-Brücke direkt in die 'View' zu integrieren und dabei eine deutliche einfachere und vor Allem saubere View-Struktur zu erhalten (vgl. mit Kommentar im Apple Enwickler-Forum [59]).

3 Ansatz

3.1 Visuelle Analyse

Der erste Schritt ist die optische Betrachtung der Unleashed App. Zu Gunsten dieser Arbeit hat die Applikation eine Art Demo Modus. Befindet sich die Unleashed App im Demo Modus, kann der / die Nutzer*in die meisten Funktionen der App testen ohne dabei ein Hardwaremodul zu besitzen. Die App simuliert die Verhaltensweise der Hardwarekomponente.

Die App besitzt vier Hauptmenüs, die eine sehr ähnliche UI-Struktur aufweisen (siehe Abbildung 1).

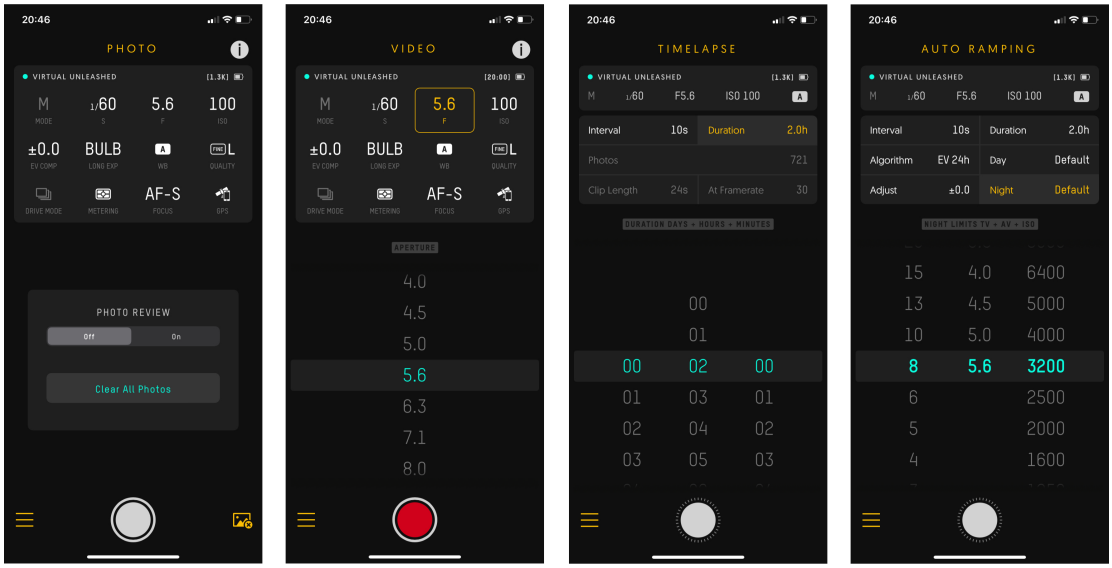

Abbildung 1: Unleashed Modi

Bis auf die Hauptmenüs gibt es außerdem das Onboarding und die Einstellungen. Der Wechsel zwischen Hauptmenüs und den Einstellungen erfolgt über eine Seitenleiste.

Der Fokus der Implementierung liegt auf den Views die in den vier Hauptmodi vorkommen .

Die Tabelle 1 fasst einen zu bearbeitenden Teil der Views zusammen und weist diesen einen Namen sowie einen Grad zu. Die Bezeichnung der Views stammt dabei nicht aus dem originalen Quellcode, sie spiegelt ausschließlich die optisch semantische Funktion der jeweiligen View wieder. Der zugewiesene Grad stellt die vermutete Schwierigkeit der Implementierung dar, und ist in drei Stufen unterteilt: leicht, mittelschwer und schwer.

Die Zuweisung der Schwierigkeitsgrade geschieht nach den folgenden Kriterien:

- Hat eine View die Fähigkeit zu scrollen, und soll diese immer eine bestimmte Endposition erreichen, so bekommt diese View den Grad schwer zugewiesen. Diese Zuweisung geschieht aufgrund meiner Vorkenntnis darüber, dass es keine native SwiftUI Lösung gibt um das Scrollverhalten einer 'ScrollView' zu obser-

vieren. Daher gehe ich davon aus, dass solche Views besonders viel Eigenkreativität in Anspruch nehmen und somit zeitintensiv sind.

- Braucht eine View eine komplexe Animation wie zum Beispiel eine sich ständig wiederholende und synchronisierte Bewegung, so wird diese View als mittelschwer eingestuft.
- Weist eine View eine gesonderte Selektion-Fähigkeit auf, so wird diese ebenfalls als mittelschwer eingestuft.
- Alle weiteren Views erhalten den leichten Grad, da diese trivialer Weise nur einen Zustand mit einer einfachen oder gar keiner Animation darstellen.

Der Abschließende Schritt ist die Sortierung der einzelnen Views nach ihrer Abhängigkeit zueinander.

ComponentTitle	einfach
ConnectionIndicator	einfach
ExposureValueMeter	einfach
Pair	einfach
DeviceStatusBar	einfach
UnleashedStatusBar	einfach
UnleashedPreview	einfach
IntervalometerClock	einfach
IntervalometerTrigger	einfach
PhotoTrigger	einfach
StateButton	einfach
VideoClock	einfach
VideoTrigger	einfach
PhotoRing	mittelschwer
IntervalometerRing	mittelschwer
VPickerGrid	mittelschwer
VPicker	schwer
HPicker	schwer

Tabelle 1: Views mit einem Grad

3.2 Funktionsumfangsanalyse

Die Analyse des Funktionsumfangs einer View startet mit der Suche nach der tatsächlichen Implementierung der imperativen View im originalen Quelltext der Applikation. Für diesen Vorgang ist es hilfreich sich in die Position eines / einer neuen Entwickler*in zu versetzen. Es gibt hierfür die Option mit dem Xcode View-Debugger [13] zu beginnen. Wenn das App Projekt kompiliert und im Simulator läuft, kann der besagte Xcode View-Debugger einen Schnappschuss von der UI der Applikation ihrer aktuellen Hierarchie erstellen (siehe Abbildung 2).

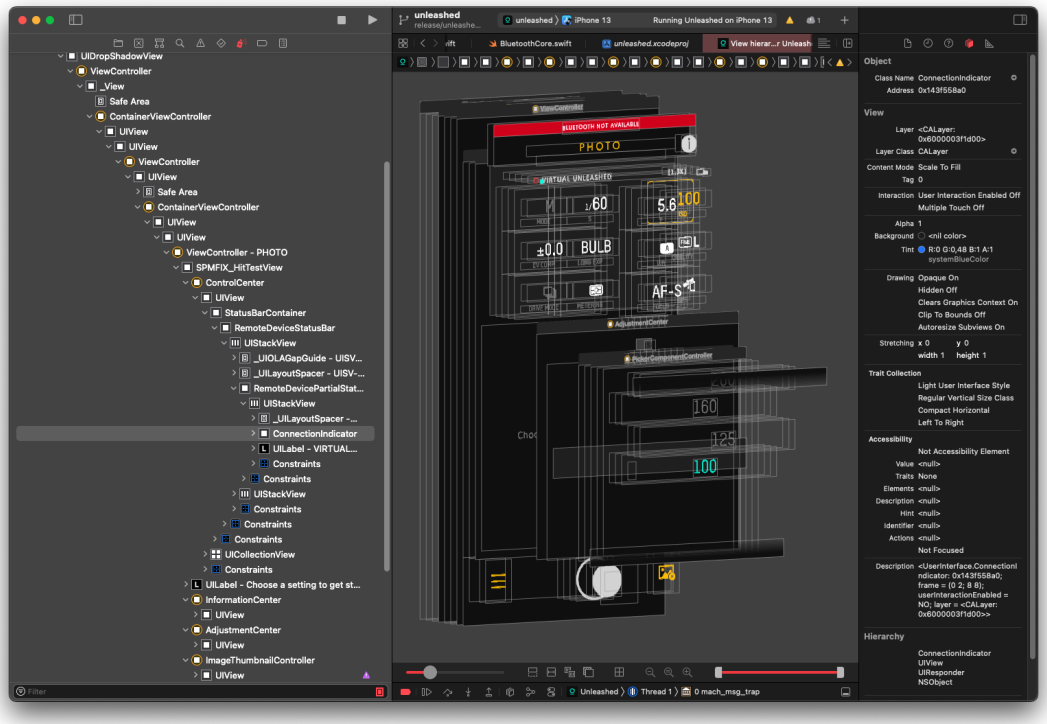

Abbildung 2: Xcode View-Debugger

Der Vorteil des View-Debuggers ist, dass damit Views einzeln betrachtet werden können. Zusätzlich besteht die Möglichkeit die selektierte View in der rechten Inspector Leiste nach ihren Metadaten abzufragen. Darunter befindet sich unter Anderem der Name der Views sowie ein direkter Verweis auf die Datei welche den Quellcode enthält (siehe Abbildung 2).

Alternativ zur Suche mit der View-Debugger besteht, kann auch eine Suche im Quellcode nach passenden Schlüsselwörtern erfolgen. Diese textbasierende Suche ist zwar nicht sehr präzise, sie kann aber dennoch zu weiteren Ergebnissen führen, sofern das Projekt veraltete oder replizierte Implementierungen von Views besitzt.

Es folgt die initiale Konzeptentwicklung der Funktionalität einer zugrunde liegenden View. Die Dokumentation und die Kommentare in den Quellcode-Dateien haben Vorrang, denn diese können nämlich wichtige Hinweise zur Funktionalität einer View enthalten. Daran angeknüpft folgt die intensive Studie der Funktionsweise des gegebenen Codes.

Die Funktionsumfangsanalyse einer View ist beendet, sobald die Erstellung der imperativen Pseudocode-Darstellung, welche sich an die MVVM-Architektur richtet, für die View abgeschlossen ist.

4 Umsetzung

4.1 Implementierung

Die Implementierung beginne ich damit, zwei Projektstrukturen anzulegen. Als erstes erstelle ich ein Swift Package [40]. Dieses dient der eigentlichen Entwicklung der einzelnen Views und verbindet diese in ein Modul [39]. Bei der zweiten Projektstruktur handelt es sich um eine Demo App, welche zum Testen und für Demonstrationszwecke gedacht ist. Die Demo App bindet das Swift Package ein um die Views schon während der Implementierung im iPhone-Simulator [50] testen zu können.

Tatsächlich gibt es keine Vorgabe mit welcher UI-Komponente ich starten muss. Es ist nur wichtig, dass ich damit beginne eine View zu implementieren welche nicht von einer anderen View abhängt. Somit entscheide ich nach dem im Kapitel 3.1 vorgestelltem Schwierigkeitsgrad vorzugehen und mit einer einfachen View zu starten.

Ich weise ausdrücklich darauf hin, dass die folgenden Kapitel nur eine beschränkte Anzahl von Views behandeln. Es geht nicht um die vollständige Implementierung aller Views die für eine App nötig sind, sondern primär um die um Abgrenzung von Fallbeispielen. Diese zeigen die wichtigsten Entscheidungen und Erkenntnisse die sich während der Bearbeitung herauskristalisierten.

4.1.1 Einfache View

Zuallererst nehme ich mir den 'ConnectionIndicator' vor. Dazu ermittle ich, wie schon im Kapitel 3.2 beschrieben, den Funktionsumfang (siehe Listing 3) und schätze dessen Implementierungaufwand ein.

```
1  class VM {
2    var color = CurrentValueSubject<UIColor?, Never>(nil)
3
4    // store a more complex model
5
6    init() {}
7
8    func _update() {
9      // compute and set ‘color‘ for each model change
10   }
11 }
12
13 class ConnectionIndicator: UIView {
14   let circle: UIView = UIView()
15   let vm: VM
16   var subscription: AnyCancellable?
17
18   init(vm: VM) {
19     // initialize stored properties
20
21     // post init: bind vm view properties
22
23     // setup circle view
24
```

```
25      // bind color from vm to the circle view
26      self.subscription = vm
27        .$color
28        .assign(to: \.backgroundColor, on: circle)
29    }
30  }
```

Listing 3: 'ConnectionIndicator' Pseudocode

Da ich bisher nur über geringfügige Erfahrung mit dem SwiftUI Framework verfüge, schätze ich den Zeitaufwand für die Implementierung erst einmal konservativ großzügig ein. Meiner Einschätzung nach, sind 30 Minuten für die Entwicklung der 'ConnectionIndicator'-View ausreichend.

Die initiale Implementierung beginnt mit der Erstellung eines leeren View-Typs und der Konfiguration der zum Testen benötigten 'PreviewProvider'-Datenstruktur (siehe Listing 4).

```
1  public struct ConnectionIndicator: View {
2    public var body: some View {
3      EmptyView()
4    }
5  }
6
7  struct ConnectionIndicator_PreviewProvider: PreviewProvider {
8    struct Test_Color_Change: View {
9      var body: some View {
10       ConnectionIndicator()
11     }
12   }
13
14   static var previews: some View {
15     Test_Color_Change()
16   }
17 }
```

Listing 4: Initiale 'ConnectionIndicator' Implementierung in SwiftUI

Das 'PreviewProvider'-Protokoll, welches aus dem SwiftUI Framework stammt, ist die helfende Hand für die agile Entwicklung von Views mit SwiftUI. Xcode erstellt im sogenannten Canvas [44] für jeden konformen Datentypen eine dem iOS-Simulator ähnliche Vorschau der entsprechenden View. In der Domäne bezeichnet man solch eine Vorschau auch als ein Preview [44]. Die Previews sind aber nicht nur für Vorschauzwecke begrenzt. Es besteht die Möglichkeit mit ihnen interaktiv zu arbeiten, diese für eine visuelle Editierung der Views zu nutzen [84] und unter Anderem damit manuelle Test-Szenarien zu prüfen. Anders ausgedrückt, ein Preview ist einem eigenständigen App Projekt sehr nahe.

Leider ist das Canvas von Xcode 13, sofern es sich um Views im Kontext eines Swift Packages handelt, nicht allzu stabil und es kommt oft zu irreführenden Fehlerdiagnosen. Um diese Problematik zu umgehen, entscheide ich die Previews temporär aus dem Demo App Projekt erreichbar zu machen, um diese im iOS-Simulator laufen zu lassen. Dabei reicht es aus den konkreten 'PreviewProvider'-Datentypen und des-

sen statisch errechnete previews-Eigenschaft (siehe Zeilen 7 und 14 im Listing 4) mit dem public-Schlüsselwort [55] zu versehen. Im übrigen werden weitere errechnete, gespeicherte oder vereinfacht nur Eigenschaften mit deren englischen Terminologie bezeichnet, nämlich Computed-Property, Stored-Property sowie nur Property [57].

Unter der Verwendung der MVVM-Architektur kann an dieser Stelle normaler Weise ein ViewModel-Objekt implement werden, jedoch entscheide ich mich gegen diesen Schritt um der dadurch unnötigen zusätzlich entstehenden Reibung aus dem Weg zu gehen. Das ViewModel würde nichts Anderes tun als nur einen konstanten Wert an die View vermitteln. Daher implementiere ich eine color Stored-Property direkt innerhalb des 'ConnectionIndicator'-Datentyps und gebe den Wert an das View mit der speziellen init-Methode [56], welche für die Initialisierung der View-Instanz zuständig ist, weiter.

Im Vergleich zu dem originalen View-Attribut, verzichte ich auf den konkreten Zustand und entscheide mich stattdessen für einen generischen Farb-Wert. Diese Änderung sorgt für eine erweiterte Flexibilität beim Einsatz dieser neuen View. Erstens ist dieser nicht mehr von einem konkreten Zustand der Unleashed App abhängig und muss somit in der Zukunft nicht mehr für eine Zustandserweiterung gewartet werden. Außerdem kann die View auf diese Weise für weitere Einsatzzwecke verwendet werden.

Die Deklaration des visuellen Anteils der View innerhalb des body-Properties verläuft relativ einfach, da es hier nur um einen farbigen Kreis mit einigen konkreten Dimensionsmerkmalen handelt. Es ist nämlich die body-Property, welche eins der Kern-Elemente des deklarativen SwiftUI Frameworks darstellt. Das 'View'-Protokoll [81] von SwiftUI sorgt dafür, dass jede body-Property implizit von dem '@ViewBuilder'-ResultBuilder umschlossen wird um den deklarativen Syntax zu initiieren.

Die originale in UIKit implementierte View trägt eine Animation-Logik, die während der Zustandsänderung zum Einsatz kommt. Auch diesen Teil habe ich in der Deklaration der View weggelassen. Jegliche Änderung eines Zustands in SwiftUI ist mit einer expliziten oder impliziten Transaktion [62] verbunden. So kann zum Beispiel eine Zustandsänderung in vielen Fällen mit der withAnimation-Funktion [87] umschlossen werden, damit diese vom Framework mit einer Animation erfolgt. Daher kann nun der / die Entwickler*in bei der Verwendung der 'ConnectionIndicator'-View entscheiden, ob der Farbwechsel mit oder ohne Animation geschieht, und mit welcher.

Im vorletzten Schritt geht es um die Testabdeckung für das View. Die Kern-Logik besteht ausschließlich aus einer einzigen Konstante. Damit die View trotzdem getestet bleibt, erstelle ich eine Reihe von Test-Szenarien mit Hilfe des 'PreviewProvider'-Datentyps. Die Testabdeckung ist vollständig wenn es für die View mindestens zwei verschiedene Änderungen des Zustandes mit oder ohne einer Animation gibt.

Natürlich dürfen die Dokumentation und Kommentare nicht aus bleiben. Im letzten Teil der Bearbeitung habe ich alle wichtigen Stellen des Views und der Testabdeckung mit ausreichend Dokumentation und Kommentaren mit Beispielen versehen, welche dem DocC-Standard [14] konform sind.

4.1.2 Mittelschwere Views

Es folgen nun Beispiele von Entwicklungsprozessen für mehrere UI-Komponente, die zuvor als mittelschwer eingestuft wurden. Der Detailgrad hierfür ist deutlich geringer als im vorherigen Abschnitt. Dieser beschränkt sich nur auf die wichtigsten Vorgehensweisen und Erkenntnisse und vernachlässigt die explizite Benennung von wiederkehrenden Schritten wie der anfänglichen Funktionsumfang-Analyse am Anfang oder der Dokumentation und Erstellung von UI-Tests am Ende jeder Implementierung.

'PhotoRing' und 'IntervalometerRing'

Während der Umsetzung des zu animierenden View-Zustands fiel mir auf, das SwiftUI eine große Lücke im Umgang mit Animationen aufweist. Wird eine Animation durch eine Änderung des Zustandes der aktuell animiert wird, unterbrochen, so beginnt die neue Animation ab dem momentan interpolierten Zustand hin zum neuen (vgl. [23]). Dieses Phänomen verhält sich sehr ähnlich wie eine Animation in UIKit, welch mit der beginFromCurrentState-Option [3] bestückt ist. Außerdem besitzt SwiftUI im Vergleich zu den Animation-Schnittstellen von UIKit (vgl. completion-Parameter [2]) keine Callback-Funktionalität um das Ende einer Animation zu signalisieren.

Bei der Suche nach einer Lösung erstellte ich zuerst einen eigenständigen View-Prototypen und experimentierte mit den Modifikationsmöglichkeiten von Animationen in SwiftUI. Tatsächlich gelingt es mir mit dem Wissen, dass eine View immer dann aktualisiert wird wenn ein Zustand sich ändert (vgl. [54]) und der Kombination von 'ViewModifier'-Methoden [83] (onAppear [37] und onChange [38]) aus der View einen Automaten zu bauen, der aus dem aktuellen Zustand finite Update-Zyklen der View verursacht indem zusätzlich ein interner Zustand überwacht und neu berechnet wird. Um eine Zurücksetzung einer potentiell noch laufenden Animation des Views zu erzwingen, startet der erste Update-Zyklus nach Erhalt von einem neuen Zustand eine lineare Animation mit einer Dauer von null Sekunden Der zweite Zyklus sogt dafür, dass nun die erwartete, neue Animation von der zurückgesetzten Position starten kann.

Im weiteren Verlauf stellt sich heraus, dass nur die Technik, die einem Automaten ähnelt, verwendet werden kann. Der Logik für die Animation des Prototypen verwerfe ich, denn diese kann so nicht für die 'PhotoRing' und 'IntervalometerRing' Views verwendet werden. Beide Views besitzen zwar eine Synchronisationskomponente, jedoch benötigen sie eine visuell zyklische und somit dynamische Animation. Solch eine Animation startet am zurückgesetzten Zeitpunkt oder dem aktuellen und läuft unbestimmt weiter, oder endet falls ein bestimmter Zeitpunkt überschritten wurde.

Für die Implementierung der eben beschriebenen Animation kommt die 'TimelineView' [61] zum Einsatz, welches erst seit iOS 15 verfügbar ist. Die 'TimelineView' kann so konfiguriert werden, dass es ihren Inhalt, welcher ebenfalls eine View ist, mit der Bildwiederholrate des Gerätebildschirms invalidiert und den aktuellen Zeitstempel übergibt. Eine Kombination der zuvor genanten 'ViewModifier' mit dem 'TimelineView' ermöglicht die Fertigstellung der Views mit dem gewünschten und akkuraten

Animationseffekt.

4.1.3 Schwere Views

'VPicker' und 'HPicker'

Die Views 'VPicker' und 'HPicker' umfassen die Kernfunktionalität der originalen 'SettingPicker' und 'ThumbnailPicker' Views. Die besagten originalen Views mögen sich optisch stark unterscheiden, jedoch besitzen sie ein und das selbe Scrollverhalten, welches sich nur in der Ausrichtungsachse unterscheidet. Aus diesem Grund sind beide Views zur gleichen Zeit zu bearbeiten.

Abbildung: UIKit-Psedocode von VPicker

Wie bereits im Kapitel 3.1 erwähnt, bietet SwiftUI nicht viel Spielraum wenn es um das Scrollverhalten einer 'ScrollView' [51] geht. So schätze ich den benötigten Zeitaufwand großzügig auf 6 bis 8 Stunden ein.

An dieser Stelle beginne ich mit experimentellen Prototypen für beide UI-Komponente. Der erste Anlauf sind die 'LazyVStack' [26] und 'LazyHStack' [25] Views, denn diese haben ein dynamisches Verhalten im Vergleich zu den regulären 'VStack' [85] und 'HStack' Views (vgl. [10]). Ein Kompromiss für das dynamische Verhalten, das eine höhere Performanz verspricht, ist die reduzierte Layout-Korrektheit (vgl. [10]). Ein weiterer Unterschied von diesen Views zu der herkömmlichen 'UITableView' [71] sowie der 'UICollectionView' aus UIKit ist, dass eine 'LazyV'- bzw. 'LazyHStack'-View innerhalb von einer 'ScrollView' ihre Zell-Views nicht für zusätzlich Performance recycelt (vgl. [86, 15]).

Alternativ für eine 'ScrollView' mit einem 'Lazy*Stack' kommt die 'List' [27] View in Frage. Eine 'List' View hat den Vorteil, dass sie ihre Zell-Views wiederverwendet, da das Framework hierbei unter der Haube eine Subklasse von einer 'UITableView' nutzt (vgl. [28]). Der Nachteil der 'List' View ist, dass es nur für die 'VPicker' View in Frage kommt, da sie ausschließlich vertikal ausgerichtet werden kann. Natürlich besteht auch die Möglichkeit die View um 90° zu rotieren, das erscheint mir aber als nicht notwendig, denn es gibt scheinbar noch eine Alternative mit einer 'LazyHStack' View innerhalb einer 'ScrollView', welche ebenfalls mit einer 'UIKit.UIScrollView' [69] Subklasse implementiert ist (vgl. [28]).

Es ist nun bekannt, dass für beide '*Picker' Views eine Zusammensetzung aus nativen SwiftUI Views existiert, welche jeweils eine 'UIScrollView' Subklasse von SwiftUI unter der Haube verwendet ('UITableView' ist eine Subklasse von 'UIScrollView'). Genau für diese Art von Aufbau entstand auch ein populäres SwiftUI-Introspect Package [21]. Das Package ermöglicht dem / der Nutzer*in den Zugriff auf die von SwiftUI verwendete 'UIScrollView'-Instanz. Im nächsten Schritt habe ich das besagte Package temporär in das Hauptpackage eingebunden um aus dessen Implementierung zu lernen.

Das SwiftUI-Introspect Package erstellt ein unsichtbares Schwester-Element in der SwiftUI 'View'-Hierarchie, welches mit dem 'SwiftUI.UIViewRepresentable'-Protokoll [76] eine eigene 'UIView' [74] Subklasse hostet (vgl. [67]) um von dort aus nach bestimmten Instanzen von 'UIView' Subklassen über die herkömmliche 'UIView'-Hierarchie zu suchen. Mit diesem Wissen gewappnet, verabschiede ich mich von dem

Package und implementiere eine eigene Introspect-Version. Im Gegensatz zur Implementierung im SwiftUI-Introspect Package, verwende ich ein 'SwiftUI.UIViewControllerRepresentable'-Protokoll [75] und eine eigne 'UIHostingController' [65] Subklasse. Der Vorteil ist, dass der 'HostingController' eine native SwiftUI View hosten kann. Somit fange ich die Ziel-View direkt ein und die Implementierung für die Suche nach dem 'UIScrollView' Objekt erfolgt durch eine einfache Verwendung des rekursiven Breiten-Suche-Algorithmus [5].

Die 'UIScrollView' Klasse so wie viele weitere UIKit Komponente verwenden das Delegation Entwurfsmuster um Teile der eigenen Funktionalität an andere Objekte weiterzureichen (vgl. [12]). So besitzt jede 'UIScrollView' Subklasse eine delegate-Property, wobei die 'UITableView' Subklasse noch zusätzlich eine dataSource-Property hat, welche ebenfalls auf dem Delegation Entwurfsmuster aufbaut. Der / die Entwickler*in nutzt diese Properties um das Scrollverhalten dieser Klassen zu steuern sowie auf Events zu reagieren oder die View mit Daten zu versorgen (Siehe den Funktionsumfang der folgenden Protokolle 'UIScrollViewDelegate' [70], 'UITableViewDelegate' [73] und 'UITableViewDataSource' [72]).

Option 1 - Proxy Delegation

Die erste Option, welche mir sofort in den Sinn kommt ist eine eigene Proxy-Delegate Klasse. Diese implementiert die notwendigen Methoden und reicht alle weiteren Signale, welche sie von dem delegierenden Objekt erhält, einfach an die zuvor eingefangene fremde Delegate-Instanz weiter (vgl. mit Beispielimplementierung für eine 'UIScrollView' [19]).

Ich entscheide mich jedoch gegen diese Option aus den folgenden Gründen: Ich habe keine Kenntnisse darüber was genau die SwiftUI Implementierung alles mit dem Delegate-Objekt anstellt. Es besteht immer eine Gefahr in unbekannte Probleme zu laufen. Zum Beispiel kann das originale Delegate-Objekt irgendwo anders zwischengespeichert sein oder die Business-logik der Framework überprüft dynamisch den Datentyp des Objekts. Darüberhinaus konnte ich experimentell ermitteln, dass bei der 'LazyHStack' + 'ScrollView' Komposition die 'UIScrollView'-Instanz ihr eigenes Delegate repräsentiert.

Option 2 - Observation

Jede 'UIScrollView' besitzt eine eine panGestureRecognizer-Property [41]. Dieses Objekt dient der Erkennung der Swipe-Geste und spielt eine fundamentale Rolle für die 'UIScrollView' Klasse (vgl. [29]). Es gibt nun die Chance sich an dieses Objekt zu heften, dessen Events zu überwachen und entsprechend zu agieren. Diese naive Verfahrensweise ist leider nicht sehr intuitiv, denn es gilt vieles herauszufinden. Zum einen ist es wichtig zu wissen, wann die Scrollview von dem / der Anwender*in losgelassen wurde und ob ein Scrollen zu erwarten ist, bzw. wann es zum Stillstand kommt. Die Komputation ist nichts anderes als ein Rohnachbau der Kernlogik der eigentlichen 'UIScrollView' Klasse.

Für die '*Picker' Views ist es wichtig mit der zuvor besprochenen Option festzustellen, wann die 'ScrollView' zum Stillstand kommt um anschließend die 'ScrollView' an die zu diesem Zeitpunkt sich im Zentrum befundene Zell-View zentriert mit einer Animation auszurichten und den Selektionsprozess zu vollenden.

Bevor ich mich an die Implementierung der zweiten Option versuchte, hatte ich einen weiteren Einfall. So teste ich nun zur Laufzeit die jeweiligen Delegate-Objekte, welche Methoden diese überhaupt implantieren. Dies zu bewerkstelligen ist relativ einfach. Objective-C Klassen die dem 'NSObjectProtocol'-Protokoll [31] konform sind, besitzen alle die respondsToSelector-Methode [47]. Als Eingabe bekommt diese Methode einen 'Selektor' [53] übergeben, welcher im Grunde den Namen einer Methode repräsentiert. Da alle Namen der Methoden von dem 'UIScrollViewDelegate'-Protokoll bekannt sind, genügt eine einfache for-Schleife um herauszufinden welche der Methoden implementiert sind.

Option 3 - Laufzeit Implementierung

Die dritte Option basiert auf den Einsatz der Objective-C Runtime Bibliothek [36]. Die Bibliothek macht eine Implementierung von Datenstrukturen während der Laufzeit der Applikation möglich.

Da der Umgang mit dieser Bibliothek für mich neu ist, reicht eine reine Recherche hier nicht aus. Das experimentelle Erlernen und Verstehen der Funktionen mit Hilfe von einem Xcode Playground Projekt [20] ist eine große Hilfe. Ein Playground Projekt eignet sich hervorragend um schnell und viel Wegwerfcode zu bauen.

Ich ermittle experimentell, dass mit den richtigen Parametern die class_addMethod-Funktion [7] die Fähigkeit hat die Implementierung aus einer Spender-Methode als eine Methode an die Ziel-Klasse zur Laufzeit anzubringen.

Der nächste Versuch resultierte in einer erfolgreiche Injizierung der eigenen Methode in die Delegate-Klasse der 'List'-View. Ich weise außerdem darauf hin, dass ich zu diesem Zeitpunkt die ursprünglich geschätzte Zeit für die Implementierung der Views bereits erreicht hatte. Da die dritte Option aber am vielversprechendsten schien, entschied ich, diese weiter zu verfolgen, anstatt mich mit der naiven Lösung der zweiten Option auseinanderzusetzen.

Implementierung

Die für die '*Picker' Views wohl wichtigste Delegate-Methode ist scrollViewWillEndDragging(_:withVelocity:targetContentOffset:) [52]. Das targetContentOffset-Argument beschreibt die finale Position der 'UIScrollView' und kann direkt aus der Methode heraus überschrieben werden. Das Problem hierbei ist die 'LazyHStack' + 'ScrollView' Komposition, denn dessen Delegate-Objekt besitzt bereits eine Implementierung für diese und weitere Methoden. Ich kann hier von der class_replaceMethod-Funktion [8] der Objective-C Runtime Bibliothek gebrauch machen und damit Method-Swizzling [80] betreiben. Wenn ich es nicht korrekt implementiere, so besteht die Gefahr, dass es zum Schaden an der Delegate-Klasse kommt oder schlimmstenfalls die Applikation abstürzt.

Im nächsten Schritt beginne ich mit dem Ausbau des eigenen Introspect-Datentyps, welcher das Einschleusen der fehlenden Delegate Methoden vereinfachen soll und zugleich einer nativen SwiftUI View erlaubt das 'HostingController'-Objekt direkt anzusprechen.

Dieser Teil dauerte deutlich länger als erhofft: Es kostete mich zwei ganze Wochen von Beginn der Implementierung der '*Picker' Views bis hierher. Glücklicherweise hatte ich einige Wochen extra Pufferzeit für die Implementierung eingeplant.

Erst nach all dieser Arbeit fiel mir auf, dass meine Testumgebung zu isoliert war und ich mich auf einen riesigen Fehler zubewegte. Das Einschleusen der fehlenden Methode in die Delegate-Klasse einer SwiftUI 'List' View wirkt sich nicht nur auf die eigenen '*Picker' Views aus sondern auf alle 'List' Instanzen einer Applikation, die das Swift Package einbindet. Diese Erkenntnis erforderte neue Recherchen, denn bei einer fehlenden Lösung bedarf es einem raschen Umschwung auf den ursprünglich verworfenen naiven Lösungsweg.

Die Lösung für das Problem ist ISA-Swizzling [42]. Die object_setClass-Funktion [34] erlaubt bei diesem Verfahren den Austausch der konkreten Klasse eines einzelnen Objekts. ISA stammt von der gleichnamigen Property [22] des 'objc_object'-Datentyps [32] ab. Im Vergleich zum Method-Swizzling ist ISA-Swizzling relativ harmlos. Hierbei kommt nur eine zur Laufzeit dynamisch erstellte Subklasse auf Basis der ermittelten Klasse des Ziel-Objekts zum Einsatz. Die Integrität der zugrunde liegenden Basis-Klasse bleibt dabei unberührt. Es erfolgt nur der Austausch der Klasse des Ziel-Objekts.

Problem 1 - Unerwartete Fehlfunktion

Nachdem ich alle Puzzle-Teile zusammensetzen konnte und das Delegate-Objekt-Klasse erfolgreich mit einer eigenen dynamischen Subklasse ausgetauscht war, stieß ich auf ein unerwartetes Phänomen. Obwohl die 'List' View Delegate-Objekt nun eine Implementierung der notwendigen Methoden besitzt, wurden diese Methoden nicht von der 'UIScrollView'-Instanz aufgerufen.

Eine zusätzliche Recherche ergab, dass es hierbei um eine interne Optimierung der Implementierung handelt, wenn es um das Delegation Entwurfsmuster geht. Dabei merkt sich das delegierende Objekt zum Zeitpunkt der Zuweisung der delegate-Property welche Methoden von dem Delegate-Objekt tatsächlich implementiert sind (vgl. WebKit Implementierung [64]).

Die Lösung hierfür ist nicht sehr intuitiv aber dennoch sehr simpel. Es ist ausreichend zum Zeitpunkt nach dem ISA-Swizzling die Referenz zum Delegate-Objekt zwischen zu speichern, diese dann von der 'UIScrollView'-Instanz zu trennen und anschließend erneut an die View zu koppeln. Das Resultat dieses Vorgehens ist eine erneute Ermittlung der Delegate-Methoden, nun aber unter der Berücksichtigung der zur Laufzeit erstellten Methoden der neuen Subklasse.

Problem 2 - super-Aufruf

Die Delegate-Klasse der 'List' View implementiert keine der Delegate-Methoden. Dieses Phänomen ist keine Garantie, denn es handelt sich nur um die experimentell ermittelte Klasse, welche aus einer bestimmten Version des Frameworks stammt. Es ist anzunehmen, dass das Framework diese Delegate-Klasse frei austauschen kann oder in einer zukünftigen Version des Frameworks die ermittelte Delegate-Klasse die besagten Methoden implementiert.

Um diesem Problem entgegenzuwirken, benötigen alle eingeschleusten Delegate-Methoden einen super-Aufruf. In der Regel ist ein super-Aufruf nur sinnvoll wenn eine Methode eine andere nicht leere Methode überschreibt. Dieser dient dazu, dass zu einem bestimmten Zeitpunkt des Methoden-Aufrufs die Implementierung aus der Oberklasse ausgeführt wird. Hierbei stammt der Begriff super von superclass ab. Der Begriff superclass wiederum ist nicht anderes als die Oberklasse.

Die Objective-C Runtime Bibliothek besitzt einen 'objc_super'-Datentyp [33], dieser ist jedoch in Swift nicht verfügbar. Es besteht die Möglichkeit diesen Datentypen nachträglich dynamisch zu verlinken, jedoch ergab meine Recherche dass der Umgang mit 'objc_super' sehr komplex sein kann und zum Teil das Verständnis von Assembly-Code notwendig ist (vgl. [6]).

Die alternative und finale Lösung für diese Problemstellung baut auf einer Zeiger-Instanz auf, welche auf den Anfang der Implementierung einer Methode verweist [18]. Während der dynamischen Implementierung wird nach dem Zeiger für eine potentielle Implementierung in der Basis-Klasse abgefragt. Falls ein solcher Zeiger existiert, wird dieser zwischengespeichert. Die eigene Implementierung der Methode der Subklasse greift erst zur Laufzeit auf den möglicherweise zwischengespeicherten Zeiger zu, wandelt diesen in eine C-Funktion um und ruft anschließend diese auf.

Somit bleibt auch hier die originale SwiftUI Implementierung unberührt. Sollte diese vorhanden sein, dann wird sie korrekt vor der eigenen Business-logik ausgeführt.

Problem 3 - Namensgebung

Der Name der dynamischen Subklasse ist eine weitere Schwachstelle, die sehr leicht zu übersehen ist. Die 'LazyHStack' + 'ScrollView' Komposition zeigt, dass die Delegate-Klasse durchaus generisch sein kann. Des Weiteren, wie bereits während des super-Aufruf angenommen, kann die Delegate-Klasse von SwiftUI ausgetauscht werden. Deshalb kommt kein konstanter Name für die Subklasse in Frage, denn falls sich, während des dynamischen Vorgangs, die Basis-Klassen von mehreren '*Picker'-Instanzen unterscheiden, kommt es zu einer Namenkollision und das ISA-Swizzling schlägt fehl.

Aus diesem Grund erhält der Name der Subklasse ebenfalls einen dynamischen Charakter.

Problem 4 - Initiale Selektion

Ich habe im Laufe der Implementierung dieser zwei Views und der zahlreichen Recherchen eins der wichtigsten Funktionen eines Pickers vollkommen außer Acht

gelassen. Eine 'List' View und auch die 'LazyHStack' + 'ScrollView' Komposition zeigen beide initial äußerste Position auf ihrer Ausrichtungsachse an. Die '*Picker' Views dagegen richten sich bereits initial an der Zelle aus, welche der gleichen Identität die dem Wert der Selektion entspricht.

Die Implementierung dieser Funktionalität ist alles andere als trivial.

Zum Einen wird die native Ansteuerung der Scroll-Position der 'List' View entweder vom Framework ignoriert oder diese ist ungenau und landet an einer nicht vorhersehbaren Position. Durch experimentelle Versuche gelingt es mir herauszufinden, dass die Delegate-Methoden für die Berechnung der Höhe von Zellen der 'UITableView'-Instanz nicht implementiert sind. Ich erkannte, dass die 'LazyHStack' + 'ScrollView' Komposition gar keine Lösung für dieses Problem hat. Die Implementierung der 'HPicker' View erfolgt somit durch eine um 90° rotierte 'VPicker' View.

Zum Anderen ist es enorm wichtig die 'List' View nach dem ISA-Swizzling Vorgang direkt zum Rendern zu zwingen. Erst danach ist es erlaubt die SwiftUI native Selektion durchzuführen, ohne dass diese vom Framework ignoriert wird. Eine nachträgliche Implementierung der fehlenden Methoden für die Höhe der einzelnen Zellen kombiniert mit dem Erzwingen des Renderns der View löst das Problem nur zum Teil.

Die 'List' View rendert zuerst in der äußersten Position, bevor sie kurz danach zur erwarteten Position springt. Nach vielen Versuchen blieb nichts Anderes übrig als dieses Verhalten zu verstecken. Ähnlich wie schon bei der 'PhotoRing' Implementierung kann hier ein Automat zum Einsatz kommen. Initial stellt die '*Picker' View ihre interne 'List' View transparent dar, führt den ISA-Swizzling Vorgang aus, zwingt die 'List' View zum Rendern und springt anschließend zur erwarteten Position. In einem weiteren Update-Zyklus wird die transparente View erst wieder sichtbar gemacht.

Zusätzliche Qualitätskontrolle

Auf iOS 16 verwendet die 'List' View nicht mehr die 'UITableView' sondern wechselt auf die 'UICollectionView' über. Damit die 'VPicker' View weiterhin wie vorgesehen funktioniert reicht eine kleine Erweiterung der injizierten Delegate-Methoden aus. Ähnlich wie das 'UITableViewDelegate'-Protokoll besitzt das 'UICollectionViewDelegate'-Protokoll [63] Methoden für die Dimensionen ihrer Zellen. Das Einschleusen dieser Methoden löst das Fehlverhalten der initialen Position auf Geräten mit iOS 16 und neuer.

4.2 Muster

In diesem Kapitel führe ich die gesammelten Erkenntnisse zusammen, werte sie aus und präsentiere daraus folgende Entscheidungen.

Die Suche nach potentiellen Mustern innerhalb der Transformation einer gegeben imperativen UI-Basis in ein deklaratives Programmierparadigma führte zu folgenden Ergebnissen:

- Die Möglichkeit der Nutzung eines View-Debuggers sorgte für eine Beschleunigung der Analyse des Funktionsumfangs von Views. Daher war es eine gute

Entscheidung diesen Prozess aus der Position eines / einer neuen Entwicklers / Entwicklerin zu betrachten. Ich konnte für diese Entscheidung kein geeignetes Muster finden.

- Eine Einschätzung des Schwierigkeitsgrades zu Beginn war eine richtige Entscheidung. Die Schwierigkeitsgrade stellten eine gute Übersicht über die Größe des zu bearbeitenden Projekts dar. Vermutetes Muster: "Divide and Conquer" (vgl. Seite 189-191 [9])

- Das Projekt in kleinere Teilaufgaben aufzuspalten und diese nach Ähnlichkeiten und Lösungsoptionen zu untersuchen führte zur Reduzierung der Komplexität mit einem klaren Fokus der zugrunde liegenden Aufgabenstellungen. Vermutetes Muster: "Divide and Conquer" (vgl. Seite 189-191 [9])

- Die Erstellung von Prototypen und experimentellem Wegwerfcode diente dem agilen Voranschreiten und vor Allem der Gewinnung fehlenden Wissens. Dabei fiel mir auf, dass nicht alle Prototypen dem Wegwerfcode gleichzustellen sind. Vermutetes Muster: "Build Prototypes" (vgl. Seite 49-52 [9])

- Entscheidungen zu treffen, zwischen einer naiven, aber als einfach vermuteten und einer vielversprechenden, aber in der Komplexität schwer einzuschätzenden Lösung war nicht trivial. In solch einem Fall konnte ich nicht garantieren ob überhaupt einer der Lösungswege zum Erfolg führen würde. Vermutete Muster: "Build Prototypes" (vgl. Seite 49-52 [9]), "Get On With It" (vgl. Seite 38-41 [9])

- Eine minimale Testabdeckung zu schaffen war besser als gar keine Testabdeckung zu haben. Der 'PreviewProvider' in SwiftUI sind für die Entwicklung der Views konzipiert. Ich konnte diese aber für UI-Tests erweitert nutzen um Fehlverhalten und fehlende Funktion einer View zu ergründen. Vermutetes Muster: "Application Design Is Bounded By Test Design" (vgl. Seite 171-172 [9])

- Automaten eigneten sich gut für den Einsatz in einem deklarativen Kontext. Dabei konnte ich einen gegebenen Zustand in weitere kleinere Zustände aufspalten um damit die gewünschte Funktionalität zu erreichen. Vermutetes Muster: "The Composable Architecture" [60]

- Auf eine Architektur wie MVVM konnte ich während der Implementierung der Basis-Views vollständig verzichten, da diese nur für zusätzliche Reibung gesorgt hätte. Ich konnte dieser Erkenntnis kein meiner Meinung nach geeignetes Muster zuordnen.

- Bei SwiftUI konnten spezifische Problemstellungen, für die keine native Lösung existierte, mit dem Introspecting-Verfahren bereits gelöst werden. Befand sich die gewünschte Änderung außerhalb der Einstellmöglichkeit, so musste ich evaluieren ob sich ein Swizzling-Verfahren dafür eignete. Es galt dabei stets die Korrektheit der Implementierung an vorderster Stelle zu haben um das Programm vor dem Absturz zu bewahren. Vermutete Muster / Verfahrensweisen: "Introspecting", "Swizzling"

- Falls nur noch experimentelles Brute-Force zum Einsatz kommt, habe ich eine erneute Einschätzung der Zeitressourcen getroffen um festzulegen ob es nicht Sinn macht rechtzeitig auf ein alternatives Verfahren umzuschwenken. Vermutetes Muster: "Take No Small Slips" (vgl. Seite 54-55 [9])

5 Fazit und Ausblick

5.1 Fazit

Mit dieser Arbeit habe ich nachweislich gezeigt, dass durch systematisches Vorgehen bei der Transformation einer imperativen UI-Basis in das deklarative Paradigma man durchaus auf wiederholte Muster stoßen kann.

Das Ziel der Bachelorarbeit war nicht die vollständige Abdeckung der gesamten UI-Komponente der gegebenen Applikation. Trotz der breiten Masse an schwierigen bis gar nicht lösbaren Problemen, kann ich dennoch am Ende vom Erfolg sprechen.

Die Komplexität von bestimmten Views wurde tendenziell korrekt eingestuft, wenn aber nicht immer zeitlich richtig. Ich gehe davon aus, dass im praktischen Umfeld die Projektleitung bei ständig anwachsenden Dauer für die Implementierung die Aufgabenstellung erneut evaluieren würde.

Die Ergebnisse der Arbeit können, so wie sie sind, von der Projektleitung oder einem Entwicklerteam übernommen werden. Es lässt sich anhand der erarbeiteten Daten eine bessere Einschätzung über die Komplexität von spezifischen Aufgaben und deren Zeitkomponente treffen.

Das entstandene Swift Package eignet sich aus meiner Sicht sehr gut als ein Startpunkt für die Fertigstellung der UI-Umstrukturierung in der Unleashed App.

5.2 Ausblick

Im nächsten Schritt werden die Ergebnisse in der Firma präsentiert und besprochen. Dabei geht es um die Darstellung der Vorteile, welche das SwiftUI Framework nachweislich bietet, als auch um nicht so Positiven Aspekte der Framework. Die für die schweren Views verwende Methodik sollte eins der letzten Optionen sein bei der Umsetzung von weiteren und zukünftigen UI-Komponenten. Besser wäre es sich strikter an den nativen SwiftUI Designs von Views zu halten. Falls dies nicht umsetzbar ist, sollte zuerst angeschaut werden ob die Implementierung mit der Hilfe von 'UIView' sowie 'UIViewController' Wrapper-Protokollen gelingen kann. Wenn es aber darum geht eine Container-View mit vielen dynamischen Views zu bauen, dann bleiben nicht mehr viele Möglichkeiten übrig und man könnte auf das ISA-Swizzling zurückgreifen. Alternativ gilt es außerdem abzuwägen ob eine Verzögerung der Implementierung einer bestimmten View mehr Sinn machen würde. Damit wäre ein bestimmtes UI-Feature erst mit einer höheren Betriebssystemversion für den / die Nutzer*in verfügbar.

Das Swift Package wird weiter ausgebaut und verbessert, bis es alle wichtigsten Grund-UI-Elemente der App abdeckt. Die '*Picker' Views werden eine zusätzliche Verbesserung erhalten, die es ihnen erlaubt ab iOS 16 auf das komplexe ISA-Swizzling zu verzichten.

Die erkannten Muster und erlernten Techniken werden zukünftig angewandt, um moderneren, verständlicheren und besseren Code für User Interfaces zu implementieren.

Literaturverzeichnis

[1] *Adding Delight to your iOS App - Layout Driven UI*. URL: https://developer.apple.com/videos/play/wwdc2018/233/ (besucht am 08. 09. 2022).

[2] *animate(withDuration:delay:options:animations:completion:)* URL: https://developer.apple.com/documentation/uikit/uiview/1622451-animate (besucht am 10. 09. 2022).

[3] *beginFromCurrentState*. URL: https://developer.apple.com/documentation/uikit/uiview/animationoptions/1622575-beginfromcurrentstate (besucht am 10. 09. 2022).

[4] *Bluetooth Low Energy*. URL: https://en.wikipedia.org/wiki/Bluetooth_Low_Energy (besucht am 08. 09. 2022).

[5] *Breitensuche*. URL: https://de.wikipedia.org/wiki/Breitensuche (besucht am 11. 09. 2022).

[6] *Calling Super at Runtime in Swift*. URL: https://steipete.com/posts/calling-super-at-runtime/ (besucht am 08. 09. 2022).

[7] *class_addMethod(_:_:_:_:)* URL: https://developer.apple.com/documentation/objectivec/1418901-class_addmethod (besucht am 11. 09. 2022).

[8] *class_replaceMethod(_:_:_:_:)* URL: https://developer.apple.com/documentation/objectivec/1418677-class_replacemethod (besucht am 11. 09. 2022).

[9] James O. Complin und Neil B. Harrison. *Organiszational Patterns of Agile Software Development*. Illustrated edition (15 July 2004). Pearson Education, 2004. ISBN: 978-0131467408.

[10] *Creating performant scrollable stacks*. URL: https://developer.apple.com/documentation/swiftui/creating-performant-scrollable-stacks (besucht am 10. 09. 2022).

[11] *Deklarative Programmierung*. URL: https://de.wikipedia.org/wiki/Deklarative_Programmierung (besucht am 08. 09. 2022).

[12] *Delegation*. URL: https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID276 (besucht am 11. 09. 2022).

[13] *Diagnosing and Resolving Bugs in Your Running App - Inspect and Resolve View Layout Issues*. URL: https://developer.apple.com/documentation/xcode/diagnosing-and-resolving-bugs-in-your-running-app (besucht am 09. 09. 2022).

[14] *DocC*. URL: https://developer.apple.com/documentation/docc (besucht am 10. 09. 2022).

[15] Doyeon, Hrsg. *Few things that you must know about UITableView in Swift*. 25. Jan. 2021. URL: https://medium.com/doyeona/things-that-you-must-know-about-uitableview-in-swift-fa2f6330a337 (besucht am 11. 09. 2022).

[16] *Every question and answer from WWDC 22's SwiftUI digital lounge! - Architecture*. URL: https://midnight-beanie-ccb.notion.site/swiftui-lounge-wwdc22-e20094b91f074398ba395c3fa245e63d (besucht am 09. 09. 2022).

[17] *Foology GmbH*. URL: https://www.foology.com/ (besucht am 08. 09. 2022).

[18] *IMP*. URL: https://developer.apple.com/documentation/objectivec/objective-c_runtime/imp (besucht am 08. 09. 2022).

[19] *In Swift, how do I have a UIScrollView subclass that has an internal and external delegate?* URL: https://stackoverflow.com/a/26965576 (besucht am 11. 09. 2022).

[20] *Interactive Playgrounds*. URL: https://developer.apple.com/swift/blog/?id=35 (besucht am 11. 09. 2022).

[21] *Introspect for SwiftUI*. URL: https://github.com/siteline/SwiftUI-Introspect (besucht am 11. 09. 2022).

[22] *isa*. URL: https://developer.apple.com/documentation/objectivec/objc_object/1418809-isa (besucht am 11. 09. 2022).

[23] Javier, Hrsg. *Advanced SwiftUI Animations – Part 1: Paths*. 26. Aug. 2019. URL: https://swiftui-lab.com/swiftui-animations-part1/ (besucht am 10. 09. 2022).

[24] Kaan Enes KAPICI, Hrsg. *Procedural(Imperative) UI vs Declerative UI and Approach on the Android Side*. 10. Apr. 2021. URL: https://kaaneneskpc.medium.com/procedural-imperative-ui-vs-declerative-ui-and-approach-on-the-android-side-1725275c53f4 (besucht am 09. 09. 2022).

[25] *LazyHStack*. URL: https://developer.apple.com/documentation/swiftui/lazyhstack (besucht am 10. 09. 2022).

[26] *LazyVStack*. URL: https://developer.apple.com/documentation/swiftui/lazyvstack (besucht am 10. 09. 2022).

[27] *List*. URL: https://developer.apple.com/documentation/SwiftUI/List (besucht am 11. 09. 2022).

[28] *List view a UITableView equivalent in SwiftUI*. URL: https://sarunw.com/posts/list-view-uitableview-equivalent-in-swiftui/#static-content (besucht am 11. 09. 2022).

[29] Ilya Lobanov, Hrsg. *How UIScrollView works*. 22. Juni 2020. URL: https://medium.com/@esskeetit/how-uiscrollview-works-e418adc47060 (besucht am 11. 09. 2022).

[30] *Model View ViewModel*. URL: https://de.wikipedia.org/wiki/Model_View_ViewModel (besucht am 09. 09. 2022).

[31] *NSObjectProtocol*. URL: https://developer.apple.com/documentation/objectivec/nsobjectprotocol (besucht am 11. 09. 2022).

[32] *objc_object*. URL: https://developer.apple.com/documentation/objectivec/objc_object (besucht am 11. 09. 2022).

[33] *objc_super*. URL: https://developer.apple.com/documentation/objectivec/objc_super (besucht am 08. 09. 2022).

[34] *object_setClass(_:_:)* URL: https://developer.apple.com/documentation/objectivec/1418905-object_setclass (besucht am 11. 09. 2022).

[35] *Objective-C*. URL: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html (besucht am 09. 09. 2022).

[36] *Objective-C Runtime*. URL: https://developer.apple.com/documentation/objectivec/objective-c_runtime (besucht am 11. 09. 2022).

[37] *onAppear(perform:)* URL: https://developer.apple.com/documentation/swiftui/anyview/onappear(perform:) (besucht am 10. 09. 2022).

[38] *onChange(of:perform:)* URL: https://developer.apple.com/documentation/swiftui/view/onchange(of:perform:) (besucht am 10. 09. 2022).

[39] *Package Manager - Conceptual Overview - Modules*. URL: https://www.swift.org/package-manager/ (besucht am 10. 09. 2022).

[40] *Package Manager - Conceptual Overview - Packages*. URL: https://www.swift.org/package-manager/ (besucht am 10. 09. 2022).

[41] *panGestureRecognizer*. URL: https://developer.apple.com/documentation/uikit/uiscrollview/1619425-pangesturerecognizer (besucht am 11. 09. 2022).

[42] HELDER PINHAL, Hrsg. *Swizzling with Swift*. 24. Juli 2020. URL: https://notificare.com/blog/2020/07/24/Swizzling-with-Swift/ (besucht am 11. 09. 2022).

[43] *Presenting View Controllers Modally*. URL: https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/PresentingaViewController html (besucht am 09. 09. 2022).

[44] *Previews in Xcode*. URL: https://developer.apple.com/documentation/swiftui/previews-in-xcode (besucht am 10. 09. 2022).

[45] *Property Wrappers*. URL: https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID617 (besucht am 09. 09. 2022).

[46] *propertyWrapper-Attribut*. URL: https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID621 (besucht am 09. 09. 2022).

[47] *respondsToSelector:* URL: https://developer.apple.com/documentation/objectivec/1418956-nsobject/1418583-respondstoselector?language=objc (besucht am 11. 09. 2022).

[48] *Result Builders*. URL: https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID630 (besucht am 09. 09. 2022).

[49] *resultBuilder-Attribut*. URL: https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID633 (besucht am 09. 09. 2022).

[50] *Running Your App in the Simulator or on a Device*. URL: https://developer.apple.com/documentation/xcode/running-your-app-in-the-simulator-or-on-a-device (besucht am 10. 09. 2022).

[51] *ScrollView*. URL: https://developer.apple.com/documentation/swiftui/scrollview (besucht am 10. 09. 2022).

[52] *scrollViewWillEndDragging(_:withVelocity:targetContentOffset:)* URL: https://developer.apple.com/documentation/uikit/uiscrollviewdelegate/1619385-scrollviewwillenddragging (besucht am 11. 09. 2022).

[53] *Selector*. URL: https://developer.apple.com/documentation/objectivec/selector (besucht am 11. 09. 2022).

[54] *State and data flow*. URL: https://developer.apple.com/documentation/swiftui/state-and-data-flow (besucht am 10. 09. 2022).

[55] *Swift - Access Control*. URL: https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html (besucht am 10. 09. 2022).

[56] *Swift - Initialization*. URL: https://docs.swift.org/swift-book/LanguageGuide/Initialization.html (besucht am 10. 09. 2022).

[57] *Swift - Properties*. URL: https://docs.swift.org/swift-book/LanguageGuide/Properties.html (besucht am 10. 09. 2022).

[58] *Swift-DSL*. URL: https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID633 (besucht am 09. 09. 2022).

[59] *SwiftUI View is a ViewModel*. URL: https://developer.apple.com/forums/thread/699003?answerId=712349022#712349022 (besucht am 09. 09. 2022).

[60] *The Composable Architecture*. URL: https://github.com/pointfreeco/swift-composable-architecture (besucht am 08. 09. 2022).

[61] *TimelineView*. URL: https://developer.apple.com/documentation/swiftui/timelineview (besucht am 10. 09. 2022).

[62] *Transaction*. URL: https://developer.apple.com/documentation/swiftui/transaction (besucht am 10. 09. 2022).

[63] *UICollectionViewDelegate*. URL: https://developer.apple.com/documentation/uikit/uicollectionviewdelegate (besucht am 09. 09. 2022).

[64] *UIDelegate.mm - setDelegate*. URL: https://github.com/WebKit/WebKit/blob/906621c54f03203bc747337e9af1b08f17f96e05/Source/WebKit/UIProcess/Cocoa/UIDelegate.mm#L105-L209 (besucht am 11. 09. 2022).

[65] "UIHostingController". In: (). URL: https://developer.apple.com/documentation/swiftui/uihostingcontroller (besucht am 11. 09. 2022).

[66] *UIKit*. URL: https://developer.apple.com/documentation/uikit (besucht am 09. 09. 2022).

[67] *UIKitIntrospectionView.swift*. URL: https://github.com/siteline/SwiftUI-Introspect/blob/master/Introspect/UIKitIntrospectionView.swift (besucht am 11. 09. 2022).

[68] *UINavigationController*. URL: https://developer.apple.com/documentation/uikit/uinavigationcontroller (besucht am 09. 09. 2022).

[69] *UIScrollView*. URL: https://developer.apple.com/documentation/uikit/uiscrollview (besucht am 11. 09. 2022).

[70] *UIScrollViewDelegate*. URL: https://developer.apple.com/documentation/uikit/uiscrollviewdelegate (besucht am 11. 09. 2022).

[71] *UITableView*. URL: https://developer.apple.com/documentation/uikit/uitableview (besucht am 11. 09. 2022).

[72] "UITableViewDataSource". In: (). URL: https://developer.apple.com/documentation/uikit/uitableviewdatasource (besucht am 11. 09. 2022).

[73] *UITableViewDelegate*. URL: https://developer.apple.com/documentation/uikit/uitableviewdelegate (besucht am 11. 09. 2022).

[74] *UIView*. URL: https://developer.apple.com/documentation/uikit/uiview (besucht am 11. 09. 2022).

[75] *UIViewControllerRepresentable*. URL: https://developer.apple.com/documentation/swiftui/uiviewcontrollerrepresentable (besucht am 11. 09. 2022).

[76] *UIViewRepresentable*. URL: https://developer.apple.com/documentation/swiftui/uiviewrepresentable (besucht am 11. 09. 2022).

[77] *Unleashed*. URL: https://www.foolography.com/products/unleashed-18/ (besucht am 08. 09. 2022).

[78] *Unleashed - Camera Remote (AppStore)*. URL: https://apps.apple.com/de/app/unleashed-camera-remote/id1361146202 (besucht am 08. 09. 2022).

[79] *Unleashed (Google Play Store)*. URL: https://play.google.com/store/apps/details?id=com.foolography.unleashed&hl=en&gl=US (besucht am 08. 09. 2022).

[80] Matheus de Vasconcelos, Hrsg. *Method Swizzling | Swift*. 25. Juni 2020. URL: https://medium.com/a-swift-journey/method-swizzling-swift-2f281aae189b (besucht am 11. 09. 2022).

[81] *View*. URL: https://developer.apple.com/documentation/swiftui/view (besucht am 10. 09. 2022).

[82] *ViewBuilder*. URL: https://developer.apple.com/documentation/swiftui/viewbuilder (besucht am 09. 09. 2022).

[83] *ViewModifier*. URL: https://developer.apple.com/documentation/swiftui/viewmodifier (besucht am 10. 09. 2022).

[84] *Visually edit SwiftUI views*. URL: https://developer.apple.com/videos/play/wwdc2020/10185/ (besucht am 10. 09. 2022).

[85] *VStack*. URL: https://developer.apple.com/documentation/swiftui/vstack (besucht am 10. 09. 2022).

[86] *What are the downsides to using lazy stacks?* URL: https://developer.apple.com/forums/thread/651593?answerId=616783022#616783022 (besucht am 11. 09. 2022).

[87] *withAnimation(_:_:)* URL: https://developer.apple.com/documentation/swiftui/withanimation(_:_:) (besucht am 10. 09. 2022).

[88] *WWDC 2019 Keynote*. URL: https://developer.apple.com/videos/play/wwdc2019/101/ (besucht am 09. 09. 2022).

Quellcode Verzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

A Anhang

Die Implementierung, welche zu einem großen Teil undokumentiert, noch nicht vollständig aufgeräumt und Teilweise nur in einem proof-of-concept Zustand ist, kann unter dieser Adresse erreicht werden: https://git.imp.fu-berlin.de/zubarev/thesis/