Technology
Arts Sciences
TH Köln

Vergleich der Performanz und Funktionen von Apples UIKit und SwiftUI zur Entwicklung nativer IOS-Applikationen.

Bachelorarbeit zur Erlangung des Bachelor-Grades
Bachelor of Science im Studiengang Medieninformatik
an der Fakultät für Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt:	Bastian Schmalbach
Matrikel-Nr.:	11120399
Adresse:	Neckarstr. 17
	40219 Düsseldorf
	bastian.schmalbach@web.de
eingereicht bei:	Prof. Dr. Matthias Böhmer
Zweitgutachter:	Prof. Dr. Christian Kohls

Gummersbach, 11. März 2021

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 11. März 2021

Bastian Schmalbach

Kurzfassung

Mit der Veröffentlichung von *SwiftUI* brachte Apple 2019 nach 11 Jahren ein weiteres Framework neben *UIKit* für das Erstellen von Benutzeroberflächen für *IOS* heraus. Damit wird den Entwicklern erstmals die Wahl eines Frameworks für die native *IOS-Entwicklung* ermöglicht. Sowohl der Arbeitsaufwand als auch die Entwicklungszeit werden von der Wahl eines Frameworks beeinflusst, weswegen diese so überlegt wie möglich getroffen werden sollte.
In dieser Arbeit wird ein detaillierter Vergleich dieser beiden gezogen. Um die Forschungsfrage, inwiefern sich UIKit und SwiftUI unterscheiden, zu beantworten, wurden einleitend zwei identische Beispielapplikationen mit den Frameworks entwickelt. Darüber hinaus wurden ebenfalls Applikationen von anderen Entwicklern betrachtet. Die meisten Unterschiede lassen sich aus den verschiedenen Grundparadigmen ableiten. Während bei UIKit Referenzen der Views in Variablen gespeichert und direkt verändert werden, werden die Views in SwiftUI als Structs erstellt und verändern sich abhängig von Variablen automatisch. UIKit verfolgt einen imperativen Ansatz, während SwiftUI viele Funktionen automatisch abwickelt und einen reaktiven deklarativen Ansatz verfolgt. Dadurch werden Datenflüsse und Datenabhängigkeiten zwischen Modellen und Views in SwiftUI vom Framework verwaltet und dem Programmierer wird diese Arbeit abgenommen. Die Messergebnisse der Performanz und Dateigröße hingegen ergaben keine nennenswerten Differenzen.
Schlussendlich führen die genannten Abweichungen jedoch zu einem bedeutenden Unterschied der Codegröße beider Frameworks. Der Umfang der entwickelten SwiftUI Applikation mit den identischen Funktionen beträgt somit nur ein Drittel (35 %) der UIKit Version. Es kann erwartet werden, dass sich SwiftUI bei stetiger Erweiterung der Funktionalität als neuer Standard der IOS-Entwicklung etabliert. Weitere Forschungen in diesem Bereich könnten Apples selbst entwickelte Desktop Prozessoren mit in den Vergleich für die Entwicklung einbeziehen.

Inhaltsverzeichnis

Glossar

Begriff	Beschreibung
Child-View	Ein View, der in der View-Hierarchie unter anderen Views liegt.
Container	Ein Behälter, welche weitere Elemente beinhalten kann, wird in dieser Arbeit als *Container* bezeichnet.
Framework	Ein *Framework* ist wie von Ralph E. Johnson definiert eine semi-vollständige Applikation, die weitere Funktionen zur Verfügung stellt.
IOS	*IOS*, auch *Iphone-OS*, ist Apples Betriebssystem für Handys und *Tablets.*
Lifecycle	Die Phasen eines Views, welche bei der Erstellung, Nutzung und Zerstörung von diesem durchlaufen werden.
Lines of Code (LOC)	Lines of Code sind die Anzahl der Zeilen eines Programms.
Modifier	In dieser Arbeit bezeichnet *Modifier* vorgegebene Methoden, die auf Elemente des *SwiftUI Frameworks* angewendet werden können.
Nativ	Wenn eine Applikation nur für ein Betriebssystem entwickelt wurde und nur auf diesem läuft, wird diese als *nativ* bezeichnet.
Paging	Das *Paging* beschreibt in dieser Arbeit die Funktion in einer Liste von Elementen genau ein Element weiter oder zurück zu springen.
Parent-View	Ein View, der in der View-Hierarchie über anderen Views steht und *Child-Views* besitzt.
Scrollen	Das Wischen über Elemente einer Benutzerschnittstelle wird in dieser Arbeit auch als *scrollen* bezeichnet.
Stack	Ein *Stack* ist ein Stapel, in welchen Views eingefügt und angeordnet werden können.
State Management	Als *State Management* wird in dieser Arbeit das Verwalten von Daten, welche die Benutzeroberfläche beeinflussen, beschrieben.
SubView	Ein View, der in der View-Hierarchie unter anderen Views liegt.

Views	Als *Views* werden in dieser Arbeit SwiftUI Elemente des Typs View bezeichnet, welche die Darstellung der Benutzerschnittstelle beschreiben.

Abbildungsverzeichnis

Tabellenverzeichnis

Code-Ausschnittverzeichnis

1 Einleitung

Apple veröffentlichte am 11. Juli 2008 mit der zweiten Version des Betriebssystems *iPhoneOS* auch den *Appstore*. Entwickler konnten nun mithilfe von Apples Framework UIKit und der Sprache Objective-C eigene Applikationen erstellen. Da Objective-C im Jahre 2014 schon 30 Jahre alt und immer weniger aktuell geworden war (vgl. González García u. a., 2015), gab Apple bekannt, dass eine hauseigene Programmiersprache namens Swift entwickelt wurde, welche nun für die Entwicklung von IOS Apps eingesetzt werden solle (vgl. Apple, 2014c). Fünf Jahre später, im Jahr 2019, veröffentlichte Apple nun ein neues Framework namens SwiftUI, was die Konzepte von Swift in vollem Maße ausnutzen sollte (vgl. Apple, 2019f). Somit brachte Apple seit der Veröffentlichung von UIKit in 2008 erstmals Konkurrenz für dieses heraus. Ebenso wie Objective-C von Swift abgelöst wurde (siehe Kapitel 2.3), könnte SwiftUI der neue Standard für die Entwicklung von Benutzeroberflächen von IOS-Applikationen werden. Das Interesse an SwiftUI ist groß, weswegen sich ein Vergleich der beiden Frameworks anbietet (vgl. Google, 2021). Die Relevanz und Wichtigkeit dieser Arbeit ergibt sich ebenfalls aus den noch nicht vorhandenen Forschungen zu SwiftUI und fehlenden Vergleichen der beiden Frameworks.

1.1 Ziele

> "SwiftUI is a modern way to declare user interfaces for any Apple platform. Create beautiful, dynamic apps faster than ever before." - (Apple, 2019i)

Ob SwiftUI dieser Beschreibung von Apple im Vergleich zu UIKit gerecht werden kann, gilt als zentraler Inhalt dieser Arbeit. Anhand folgender Forschungsfrage soll dies untersucht werden: "Inwiefern unterscheiden sich Funktionen und Performanz von Apples UIKit und SwiftUI?". Weitere Fragen, welche sich aus diesem Forschungsziel bilden, beziehen sich auf den Entwicklungsaufwand einer Applikation mit beiden Frameworks sowie mögliche Gemeinsamkeiten oder Unterschiede der Grundkonzepte von UIKit und SwiftUI.

1.2 Methodisches Vorgehen

Für einen umfassenden Vergleich bietet sich das Erstellen der gleichen Applikationen mit beiden Frameworks an. Ebenfalls sollen Apps von anderen Entwicklern zum Unterstützen der gewonnenen Erkenntnisse herangezogen und evaluiert werden. Die Simulation einer umfangreichen IOS-Anwendung soll mit Funktionen wie API-Abfragen, Animationen und komplexen Anordnungen von Views erreicht werden. Dabei werden sowohl die Grundkonzepte als auch

Faktoren der Apps wie die Performanz, Prozessorauslastung, App Größe, Lines of Code, Code Komplexität und weitere betrachtet.

1.3 Struktur der Arbeit

Die vorliegende Arbeit gliedert sich in fünf Abschnitte. Infolge der Einleitung wird im zweiten Kapitel ein Überblick über die Grundlagen der nativen App Entwicklung für IOS gegeben. Der dritte Abschnitt behandelt die Gegenstände dieser Arbeit, mithilfe derer die Unterschiede der Frameworks analysiert werden. Anhand dieser und weiteren Forschungen wird im vierten Teil ein Vergleich zwischen UIKit und SwiftUI gezogen. Abschließend fasst das fünfte und letzte Kapitel die gewonnenen Erkenntnisse der vorangehenden Abschnitte zusammen, ordnet diese in den Gesamtkontext ein und gibt einen Ausblick.

2 Grundlagen der IOS-Entwicklung

Zu Beginn werden die Grundlagen der IOS-Entwicklung erläutert, um den Kontext herzustellen. Der Programmcode für IOS Apps kann in UIKit sowohl mit Objective-C als auch mit Apples eigener Programmiersprache Swift geschrieben werden, welche entwickelt wurde, um Objective-C abzulösen. Aus diesem Grund und weil SwiftUI lediglich mit Swift als Sprache genutzt werden kann, wird in dieser Arbeit ausschließlich die Programmierung mit Swift betrachtet wird.

2.1 Programmiersprache Swift

Wie bereits erwähnt, veröffentlichte Apple 2014 seine hauseigene Programmiersprache Swift, welche Objective-C als Standard für die Programmierung von IOS und MacOS Apps ablösen sollte (vgl. Apple, 2014c). Mit Version 2.0 kündigte Apple bei ihrer weltweiten Entwicklerkonferenz (WWDC) an, Swift Open-Source werden zu lassen (vgl. Apple, 2015d). Es wurden seither jährlich Updates publiziert, in welchen Funktionalitäten hinzugefügt, aber auch einige Grundkonzepte verändert wurden, wie in 2016 mit Version 3.0. Mittlerweile hat sich Apple jedoch von einem knappen Release-Zyklus entfernt und veröffentlicht neue Versionen, wenn sich ausreichend Änderungen ergeben haben. In Folge dessen entwickelt sich Swift zu einer immer reiferen Programmiersprache. Als Nachfolger von Objective-C hat Swift viele der Funktionalitäten von dieser übernommen, verbessert und erweitert. Parallelen zwischen den zwei Programmiersprachen lassen sich laut Apple beispielsweise im selben LLVM Compiler, Optimizer und Autoverctirizer sowie im ARC Memory Management erkennen (vgl. Apple, 2014g). Objective-C ist 1984 erschienen und nach heutigem Stand in vielerlei Hinsicht verbesserungswürdig, was die Einführung einer eigenen und neueren Sprache begründet und dessen Relevanz verdeutlicht (vgl. González García u. a., 2015). Swift übernimmt viele der Konzepte und Funktionalitäten, sodass sie mit Objective-C Code in Symbiose verwendet werden kann. Apple selbst stellt keine Statistiken über die Adoption von Swift bereit. Die Plattform “Stack Overflow” führt aber seit 2010 jährlich eine Umfrage über die Präferenzen von Programmierern durch. Es handelt sich um keine wissenschaftliche Studie/Umfrage, jedoch wird durch die Vielzahl an Teilnehmern, 65.000 Entwicklern im Jahr 2020, ein Einblick über die Popularität der zwei Sprachen gewonnen. Stack Overflow selbst sagte hierzu, dass die Ergebnisse von den Antwortmöglichkeiten, Vorurteilen gegenüber der Technologien und weiteren Faktoren verzerrt werden (vgl. Overflow, 2015). Es werden Fragen zu den Präferenzen, der Nutzung der Programmiersprachen und vielen weiteren Aspekten gestellt. In der folgenden Tabelle wurden die Daten über die meist genutzten Technologien ab dem Jahr 2013 gefiltert und zusammengeführt:

Jahr	Objective-C	Swift	Gesamt
2013	11.0 %	N/A	11.0 %
2014	11.9 %	N/A	11.9 %
2015	7.8 %	N/A	7.8 %
2016	6.5 %	N/A	6.5 %
2017	6.4 %	6.5 %	12.9 %
2018	7.0 %	8.1 %	15.1 %
2019	4.8 %	6.6 %	11.4 %
2020	4.1 %	5.9 %	10.0 %

Tabelle 2.1: Meist Genutzte Technologien der Befragten (nur Objective-C und Swift angezeigt)
(vgl. Overflow, 2015; Overflow, 2016; Overflow, 2017; Overflow, 2018; Overflow, 2019; Overflow, 2020)

Bis 2016 wurden in den Ergebnissen der beliebtesten Technologien lediglich die ersten zwölf Plätze aufgeführt. Swift hatte bis zum Jahr 2016 keinen dieser Ränge eingenommen. Eine mögliche Annahme ist ein Zusammenhang zwischen dem Abfall des Interesses an Objective-C und der Veröffentlichung von Swift im Jahre 2014. Während 2014 noch 11.9 % der Befragten Objective-C als eine beliebte Sprache auswählten, waren es 2015 nur mehr 7.8 %. Durch die fehlenden Daten über die Beliebtheit von Swift für den Zeitraum von 2014-2016 lässt sich jedoch schwer ein eindeutiger Bezug zwischen der Veröffentlichung von Swift und dem Abfall der Nutzung von Objective-C beweisen. Allerdings ist ein Zusammenhang möglich, da die Popularität der beiden Sprachen über die restliche Periode konstant über 10 % lag. Eine weitere Erkenntnis aus dieser Umfrage ist, dass Swift seit 2015 jedes Jahr unter den Top 9 der beliebtesten Programmiersprachen liegt.

2.1.1 Typsicherheit

Einer der wichtigsten Punkte, mit welchem sich Swift von Objective-C abhebt, ist die Typsicherheit von Variablen. Es ist demnach nicht möglich, Variablen Werte von unpassenden Variablentypen zuzuweisen. Wenn einer Variable des Typs `String` eine Zahl zugeschrieben werden soll, wirft Swift nicht erst bei der *Runtime*, sondern schon beim Kompilieren/Schreiben des Codes einen (*Typecast*) *Error* und verbietet dies:

```
var str = "Hello World"
str = 1
//error: Cannot assign value of type 'Int' to type 'String'
```

Code-Ausschnitt 2.1: Typecast Error in Swift

Eine weitere Funktion, über welche Swift verfügt, die im vorherigen Code Ausschnitt zu erkennen ist, ist die automatische Typbestimmung von Variablen. Der Typ der Variable `str` wird bei der Initialisierung anhand des zugewiesenen Wertes vom Compiler unaufgefordert festgelegt. Wenn Variablen ohne einen Initialwert erstellt werden sollen, muss der Typ dieser jedoch festgelegt werden, da der Compiler den Typ wegen des fehlenden Wertes nicht automatisch bestimmen kann. Diese Variablen müssen mit einem Fragezeichen als optional festgelegt werden:

```
var str: String?
```

Code-Ausschnitt 2.2: Optionale Variable in Swift

Wenn optionale Variablen nun im Code verwendet werden, muss ein Standardwert angegeben werden, der verwendet wird, falls diese keinen Wert enthält (vgl. Apple, 2016b). Dadurch können Objekte keinen Nullwert enthalten und Fehler dieser Art werden ebenfalls verhindert. Beispielsweise können Funktionen mit einem `String` als Parameter ausschließlich diesen Typ akzeptieren und lösen schon bei einem optionalen `String` einen Error aus. Der Compiler zeigt für fehlerhafte Typecasts vor dem Ausführen des Codes Fehlermeldungen an und verringert dadurch das Abstürzen der App beim Ausführen bedeutend. Hier werden die Vorteile der automatischen Typbestimmung und Type Safety über Objective-C deutlich.
Eine weitere Funktion von Swift, welche diese sicherer macht, stellt die Tatsache dar, dass Variablen, die mit dem Keyword `let` gekennzeichnet sind, nicht mehr veränderbar sind. Apples Compiler schlägt dies automatisch für alle Variablen, die im Code nicht mutiert werden, vor. Nur Variablen, die mit `var` gekennt sind, lassen sich somit mutieren.

2.1.2 Klassen und Structs

Der Unterschied zwischen Klassen und Structs ist für den Vergleich der zwei Frameworks bedeutsam, da sie von diesen unterschiedlich verwendet werden. Dies wird in Kapitel 4.2 ausführlich thematisiert, während in diesem Abschnitt die Unterschiede zwischen Klassen und Structs vorerst erklärt werden. Klassen und Structs können jeweils Variablen und Funktionen implementieren und diese verwalten. Sie können Subscripts (vgl. Apple, 2014d) für den Zugriff der Variablen und eigenen Methoden für die Initialisierungen besitzen.
Der größte Unterschied ist, dass Klassen *Reference Types* und Structs *Value Types* sind (vgl. Apple, 2018b). Das bedeutet, dass Klassen beim Zuweisen zu anderen Variablen nicht kopiert werden, sondern eine Referenz des Objektes weitergegeben wird. Wenn Structs hingegen mehrfach verwendet werden, werden die Daten dieser kopiert und eine neue Instanz erstellt. Klassen besitzen zudem weitere Funktionalitäten. Sie verfügen über die Funktion der *Inheritance*; auf Deutsch Vererbung. Dies bedeutet, dass eine Klasse die Funktionen und Methode an ihre Unterklassen vererbt. Dadurch werden zwar viele Funktionen übernommen und müssen nicht selbst implementiert werden, doch ebenso werden eventuell ungenutzte Funktionen weitergegeben. Dadurch können die Klassen größer und weniger performant werden als sie sein müssten (vgl. Apple, 2014a). Ein weiterer Vorteil von Swift über Objective-C wird

hier deutlich, da eigens erstellte Klassen und Structs automatisch für das gesamte Projekt verfügbar gemacht werden und nicht von Datei zu Datei herumgereicht und importiert werden müssen.
Der Start der IOS Entwicklung geschah jedoch nicht mit Swift, sondern mit Objective-C und dem Framework UIKit.

2.2 UIKit Framework

Für die Entwicklung von nativen IOS-Applikationen veröffentlichte Apple 2008 eine eigene Entwicklungsumgebung mit verschiedenen Frameworks zur Unterstützung namens *Cocoa Touch*. Für Cocoa Touch wurde die Entwicklungsumgebung für MacOS-Programmierung als Orientierung genutzt und um die Funktionalität für mobile Geräte erweitert (vgl. Apple, 2013). UIKit ist ein Teil des Cocoa Touch und wird für die Erstellung von Benutzeroberflächen verwendet (vgl. Apple, 2009a). Das Erstellen des UIs ist mit UIKit sowohl über den *Interface Builder* als auch programmatisch möglich. Mit dem Interface Builder lassen sich die Views auf den *Canvas* (zu Deutsch Leinwand) ziehen und wie in einem Design Programm anordnen (vgl. Apple, 2014b). Dabei ist das Schreiben von Code dennoch nicht zu umgehen, da einige Funktionen, wie die Aktionen eines Buttons, programmatisch umgesetzt werden müssen. Die zweite Möglichkeit ist es, die Benutzeroberfläche ganz programmatisch zu erstellen.
Die Entwicklung folgt dem imperativen Programmierparadigma, welches eine sehr große Kontrolle über die Datenflüsse und den Ablauf der App bietet (siehe Kapitel 4.1). Dadurch werden ebenfalls mehr Aufgaben an die Programmierer übertragen. Die ersten sechs Jahre wurde Objective-C als einzige Programmiersprache für native IOS-Entwicklung verwendet, bis im Jahr 2014 Swift veröffentlicht wurde.

2.3 SwiftUI Framework

2019 publizierte Apple ein neues Framework für das Erstellen der Benutzeroberfläche namens SwiftUI, welches für die Sprache Swift entwickelt wurde (vgl. Apple, 2019f). In SwiftUI werden Benutzeroberflächen nicht grafisch mit dem Interface Builder, sondern ausschließlich programmatisch erstellt. *Live Previews* ermöglichen es dabei, die Veränderungen des Codes während der Entwicklung als eine Vorschau der App zu sehen.
Im Gegenteil zu UIKit ist SwiftUI ein deklaratives Framework, welches das Paradigma der reaktiven Programmierung umsetzt (vgl. Apple, 2019f). Aufgaben, die in UIKit explizit verwaltet werden mussten, werden in SwiftUI automatisch abgewickelt. SwiftUI wurde nicht für Objektive-C, sondern mit Swift als einzige Programmiersprache entwickelt und soll alle Vorteile von dieser ausgenutzt können (vgl. Apple, 2019f). Die Unterschiede ziehen sich jedoch weiter und ein detaillierter Vergleich der beiden Frameworks ist notwendig, um diese aufzudecken.

3 Vergleichsgegenstände

Um Unterschiede und Gemeinsamkeiten der beiden Frameworks und deren interne Funktionen zu ermitteln, bietet es sich an, identische Applikationen entwickelt mit jeweils einem der beiden Frameworks zu vergleichen. Hierfür wurden zwei Apps entwickelt und evaluiert. Des Weiteren wurden Ergebnisse eines zusätzlichen Entwicklers, Paul Hudson, zusammengetragen und bewertet, um die Resultate gegenüberzustellen. Ein weiteres Projekt mit zwei Applikation war die *LogIn App* von OkCupid (vgl. OkCupid, 2020), bei welcher ein LogIn mit beiden Frameworks implementiert wurde. Bei diesem Projekt lag der Fokus allerdings nicht auf dem Vergleich der zwei Frameworks, sondern auf dem Erstellen eines Code-Konstrukts, mit welchem beide Frameworks für ein Projekt eingesetzt werden können. Dies ist ebenfalls ein aussagekräftiges Projekt, jedoch nicht relevant für diese Arbeit.

3.1 Abwandlung der *Sustainable App*

Im Rahmen des Moduls Praxisprojekt der Technischen Hochschule Köln wurde eine umfangreiche IOS-Applikation namens *Sustainable App* entwickelt (vgl. Schmalbach, 2020), welche einen nachhaltigen Lebensstil unterstützt. Für die Entwicklung von dieser wurde zu Beginn ausschließlich SwiftUI genutzt. Für benötigte Funktionen, über die SwiftUI nicht verfügt, wurde UIKit verwendet. Hieraus ergab sich das Interesse an einem Vergleich der beiden Frameworks. Für diese Arbeit wurde eine abgewandelte Version dieser App sowohl in SwiftUI als auch in UIKit entwickelt. Diese umfasst nicht alle Funktionen, soll jedoch durch asynchrone API-Abfragen, Animationen und komplexe Anordnungen von Views eine umfangreiche Applikation simulieren. Die abgewandelten Apps zeigen einen der vier Hauptscreens der Sustainable App (Abbildung 3.1), welche folgende Funktionen umfasst:

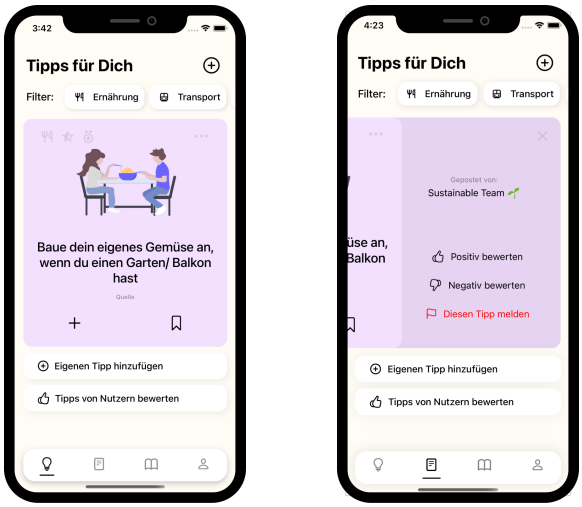

Abbildung 3.1: Aussehen der abgewandelten *Sustainable App*

Funktion	Beschreibung
Tipps ansehen	Zeigt einen horizontalen Scrollview mit allen Tipps an, die den ausgewählten Filtern entsprechen.
Detailansicht der Tipps	Genauere Informationen und Interaktionsoptionen werden angezeigt.
Tipps filtern	Tipps können manuell nach verschiedenen Kriterien gefiltert angezeigt werden.
Button Animationen	Interaktionen mit Buttons sind möglich und geben dem Nutzer optisches und haptisches Feedback.
Tabbar Animationen	Das Auswählen der Menüpunkte löst Animationen der Tabbar aus.

Tabelle 3.1: Implementierte Funktionen der abgewandelten *Sustainable App*

Der Code für beide Applikationen ist auf *GitHub* veröffentlicht und einsehbar (vgl. Schmalbach, 2021).

3.2 List App von Paul Hudson

Um die Auswertungen der abgewandelten *Sustainable App* zu validieren, wurde nach Applikationen gesucht, die in beiden Frameworks umgesetzt wurden. Ein für diese Arbeit passendes Projekt wurde von Paul Hudson umgesetzt und von ihm in reduziertem Maße analysiert (vgl.

Husdon, 2019). Die Apps umfassen eine Liste von Objekten, deren Daten aus einer Datei geladen werden. Jedes Element der Liste zeigt ein Bild und zwei Texte an und soll beim Auswählen von diesem zu einem DetailView navigieren, welcher aus dem selbigen Bild in anderer Größe, einem Text und einem Button besteht. Der Button soll einen *Alert* (vgl. Apple, 2016a) mit einem Text und zwei Buttons anzeigen. Die Benutzeroberfläche der Apps ist in den in Abbildung 3.2 Screenshots dargestellt:

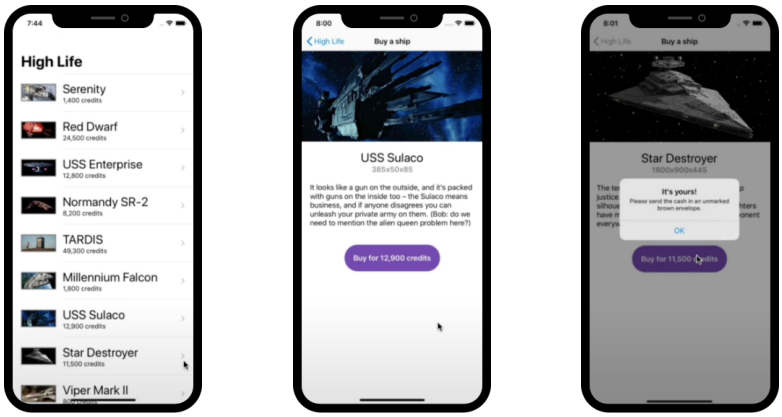

Abbildung 3.2: Views der von Hudson erstellten App

Hudson's Fokus lag bei diesem Projekt sowohl auf der Entwicklungszeit als auch auf den Lines of Code, die für diese Apps benötigt werden. Da Hudson gute Vorkenntnisse beider Frameworks hatte, bietet sich der Vergleich der Zeiten für das Erstellen der beiden Apps an. Für die Entwicklung der Abwandlung der Sustainable App wäre ein solcher Vergleich aufgrund des größeren Projektumfangs ebenfalls aufschlussreich. Von diesem wurde jedoch abgesehen, da die Vorkenntnisse in SwiftUI denen in UIKit deutlich überwogen, wodurch das Ergebnis dieser Messung verfälscht und nicht gehaltvoll wäre.

4 Vergleich der Frameworks

Für den Vergleich der Frameworks wurden verschiedene Themenfelder und Metriken ermittelt, in welche das folgende Kapitel unterteilt ist.

4.1 Imperativ vs. Deklarativ

Ein großer Unterschied der beiden Frameworks ist das Paradigma der Programmierung. Hierbei wird zwischen einem imperativen und deklarativen Ansatz unterschieden. Beim imperativen Programmieren (lateinisch imperare = befehlen) verwaltet der Entwickler die Verhaltensweisen des Frameworks selbst und befiehlt, wie sich das Programm zu welchen Momenten zu verhalten hat. Dies gibt dem Entwickler mehr Bestimmungsmöglichkeiten, aber auch einen größeren Raum für mögliche Fehler wie in Code-Ausschnitt 4.1 deutlich wird. Um dies zu verdeutlichen, sind im Folgenden alle Methoden eines Views aufgelistet, die deren Lebenszyklus verwalten:

```
UIViewController.init
UIViewController.prepare(for:sender:)
UIViewController.loadView()
UIView.init
UIView.awakeFromNib()
UIViewController.viewDidLoad(_:)
UIViewController.viewWillAppear(_:)
UIView.didAddSubview()
UIView.willMove(toSuperview:)
UIView.willMove(toWindow:)
UIView.didMoveToWindow()
UIView.didMoveToSuperview()
UIView.updateConstraints()
UIViewController.updateViewConstraints()
UIViewController.viewWillLayoutSubviews()
UIView.layoutSubviews()
UIViewController.viewSafeAreaInsetsDidChange()
UIViewController.viewLayoutMarginsDidChange()
```

```
UIViewController.viewDidLayoutSubviews()
UIViewController.viewDidAppear(_:)
```

Code-Ausschnitt 4.1: UIKit Methoden für das Verwalten des Lifecycles eines Views (vgl. Apple, 2009b)

Durch die Auflistung dieser Methoden wird deutlich, wie viele Möglichkeiten, aber auch Aufgaben dem Entwickler für die Verwaltung eines einzelnen Views übertragen werden. Dahingegen bedeutet deklarativ (lateinisch declarativus = erläuternd) für SwiftUI, dass der Entwickler erklärt, welches Ergebnis erzielt werden soll und das Framework übernimmt die nötigen Schritte, um dieses zu erreichen. Bei der deklarativen Programmierung in SwiftUI werden diese Methoden, wie aus den Analysen in Kapitel 4.8 hinausging, ebenfalls aufgerufen, jedoch im Hintergrund von dem Framework verwaltet. Somit muss sich der Entwickler einer SwiftUI App über die einzelnen Zeitpunkte der Initialisierung und Erstellung der Views und Variablen keine Gedanken machen. Die Unterschiede des State Management, der Erstellung und Veränderung der Views, welche aus den abweichenden Ansätzen der Frameworks resultieren, werden in den folgenden Kapiteln 4.3 und 4.4 im Detail behandelt.

4.2 Klassen und Structs

Einer der grundlegenden Aspekte, in welchen sich UIKit und SwiftUI unterscheiden, ist die abweichende Verwendung von Klassen und Structs. Wie in Kapitel 2.1.2 beschrieben, ähneln sich Klassen und Structs in einigen Punkten, besitzen jedoch unterschiedliche Vorteile. In UIKit sind alle Views Klassen und leiten sich von UIKits Standard View UIView ab. Dies hat den Vorteil, dass Funktionen und Methoden des UIViews an alle Views weitergegeben werden, da sie diesem abstammen (vgl. Apple, 2009b). Dies stellt gleichzeitig einen Nachteil dar, da hierdurch die Views auch teilweise überflüssige Funktionen und Methoden der Oberklasse erben, die für diesen View keine Anwendung finden. Dadurch werden die Views wesentlich größer als sie sein müssten. Allein der UIView vererbt 173 Funktionen und Methoden an alle weiteren Unterklassen (vgl. Apple, 2009b). SwiftUI hingegen verwendet für Views ausschließlich Structs. Dadurch, dass diese Views keine Subklassen von anderen Views sein können, nehmen sie sehr wenig Speicher in Anspruch. SwiftUI Views benötigen lediglich den Speicherplatz, den die Variablen einnehmen. Dies kann mithilfe der Methode `MemoryLayout.size(ofValue: MyView())` ermittelt werden. Somit ist auch ein komplexer View mit mehreren Funktionen und einer Variable des Typs `Int8` lediglich 1 `Byte` groß. Da Structs nicht als Referenzen weitergegeben werden können, werden die Daten von SwiftUI-Views automatisch abgekapselt und potentiell unvorhersehbares Verhalten reduziert, da auf die Variablen nur von innerhalb des Views zugegriffen werden kann. Das Erstellen der Views in SwiftUI benötigt aufgrund der geringen Größe wenig Ressourcen, weswegen diese bei Veränderungen neu erstellt werden anstatt den bestehen View zu verändern. Dies wäre für UIKit sehr ineffizient. Stattdessen werden die bestehenden Views dort direkt manipuliert und die Eigenschaften dieser verändert. Die unterschiedliche Nutzung der Klassen und Structs spielt auch bei der Verwaltung von Daten eine Rolle, was im folgenden Kapitel behandelt wird.

4.3 State Management

Die Analyse der beiden Frameworks lässt einen großen Unterschied des State Managements deutlich werden. State Management bedeutet in diesem Kontext das synchron Halten der Views und Daten. Dies erfolgt meist nach einem Muster, nach welchem die App in abgegrenzte Bereiche unterteilt wird.
Für UIKit-Apps sieht Apple das *Model-View-Controller Design* vor, bei welchem die Objekte nach ihrem Zweck aufgeteilt werden (vgl. Apple, 2017a). Eine UIKit App nach dem MVC-Muster besteht somit aus den folgenden drei Komponenten: Modelle, welche die Datenstrukturen vorgeben/enthalten, den Views, welche die Benutzeroberfläche beschreiben und einem Controller (auch Viewcontroller genannt), welcher die Views verwaltet, aktualisiert und mit den Modellen verbindet (vgl. Apple, 2017a). Dadurch sollen die verschiedenen Komponenten logisch und räumlich separiert werden, wie die Abbildung 4.1 visualisiert (vgl. Apple, 2018d).

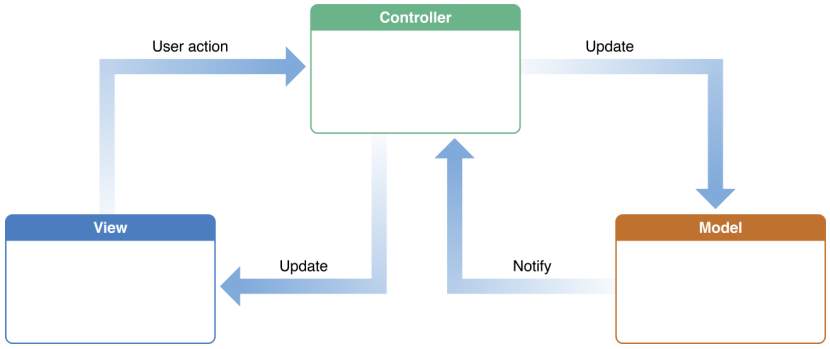

Abbildung 4.1: Umsetzung des MVC-Musters in UIKit (Apple, 2018d)

Wie in Kapitel 4.2 beschrieben, arbeitet UIKit mit Klassen als Views und speichert Referenzen dieser in Variablen ab. Anhand dieser verändert UIKit die Objekte, wenn ein Nutzer mit der App interagiert, direkt und imperativ. Wenn nach einer Veränderung einer Variable mehrere Views verändert werden müssen, muss der Entwickler die Referenzen jedes Views verwalten und diese einzeln bearbeiten, um die Veränderung für den Nutzer widerspiegeln zu lassen. Ein üblicher Ablauf des Programms könnte folgender sein:
Der Nutzer interagiert mit einem View, welcher die Interaktion an den Viewcontroller kommuniziert. Dieser verarbeitet diese Informationen und ermittelt, welches Modell verändert werden muss. Das Modell übermittelt den Status der Anweisungen nach der Durchführung. Der Viewcontroller verarbeitet die Ergebnisse des Models und befiehlt dem View, welche Veränderungen dieser vornehmen soll.
Da der Viewcontroller nach Apple viele der beschriebenen Aufgaben übernehmen soll, kann dieser sehr komplex und unübersichtlich werden. Apple empfiehlt das MVC-Muster, schreibt dieses jedoch nicht vor und eine UIKit-App kann ebenso mit dem *Model-View-ViewModel-Muster* (MVVM) umgesetzt werden (vgl. Apple, 2017a). Da Apple das MVC-Muster empfiehlt, wird in dieser Arbeit das MVC-Muster für die UIKit-App verwendet und betrachtet.
Für SwiftUI wird von Apple kein Muster für die Unterteilung der Komponenten empfohlen. In SwiftUI wird kein Viewcontroller verwendet, weswegen sich annehmen lässt, dass ein MVVM-

Muster implementiert wird. Um das verwendete Muster zu ermitteln, muss analysiert werden, wie die Daten verwaltet werden. SwiftUI implementiert das Konzept der deklarativen Programmierung wie in Kapitel 4.1 beschrieben nicht nur für das Verwalten von Views, sondern ebenfalls für das State Management (vgl. Apple, 2020e). Hierfür nutzt es das Programmierparadigma des reactive programming. Ein reactive programming Framework hört auf Veränderungen der Daten und setzt das Aktualisieren der Benutzeroberfläche automatisch um (vgl. Apple, 2020e). Es verwaltet asynchrone Datenflüsse zwischen verschiedenen Modellen und Views, die auf diese reagieren. Da SwiftUI View-Structs verwendet und keine Referenzen dieser existieren können, um diese zu manipulieren, werden die Views so erstellt, dass sie sich abhängig von Variablen automatisch anpassen. Dies wird mithilfe von Property Wrappern umgesetzt, welche Variablen mit weiterer Funktionalität ausstattet (vgl. Apple, 2014f). Diese Variablen werden *DynamicProperties* genannt und können nicht nur einfache Datentypen wie `Strings` und `Ints`, sondern ebenfalls selbst erstellte komplexe Datentypen umfassen (vgl. Apple, 2019b). SwiftUI verwaltet alle Views, die von einer DynamicProperty abhängig sind und aktualisiert diese automatisch, wenn sich diese Variablen oder Objekte verändern. Es muss zwischen lokalen und globalen Daten unterschieden werden, von welchen die Veränderungen der Views ausgehen.
Lokale Variablen werden meist für den Status der Benutzeroberfläche verwendet und von dem View verwaltet, in welchem diese erstellt wurden. `@State` ist der in der SwiftUI-App am häufigsten genutzte Property Wrapper, welcher ausschließlich in Views für lokale Variablen verwendet werden kann (vgl. Apple, 2020i). Der View, welcher `@State` Variablen verwaltet, erstellt die Verbindung dieser zu den abhängigen Views selbst. Wenn die Variable verändert wird, lädt sich der View neu und die Aktualisierungen sind sichtbar. Wenn eine `@State` Variable an einen Subview weitergegeben wird, wird dieser ebenfalls automatisch neu gerendert, wenn die Variable im Parent-View verändert wird. Der Child-View beeinflusst durch das Verändern dieser jedoch nicht die `@State` Variable im Parent-View, da nur eine Kopie dieser verändert wird. Damit eine bidirektionale Verbindung erstellt werden kann, wird `@Binding` in den Subviews verwendet, um eine DynamicProperty anzunehmen (vgl. Apple, 2020h). Nun kann auch der Subview die selbige Variable direkt verändern und alle Veränderungen dieser werden automatisch in allen Views, die diese verwenden, angewendet. Bei einem Binding wird ein $ vor die Variable eingefügt, wenn sie an einen View weitergegeben wird. Wie unterschiedlich die Ansätze von UIKit und SwiftUI für das Aktualisieren von Views sind, kann mit einem Beispiel verdeutlicht werden. Hierfür soll lediglich ein Button, dessen Bild sich nach dem Anklicken verändert, angezeigt werden.

Abbildung 4.2: Zwei Zustände des Buttons

```
class MyViewController: UIViewController {

    var isSelected = false

    let myButton = UIButton(frame: CGRect(x: 165, y: 370, width: 50, height: 50))
    myButton.setImage(UIImage(systemName: "xmark"), for: .normal)
    myButton.addTarget(self, action: #selector(buttonAction), for: .touchUpInside)

    override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(myButton)
    }
    @objc func buttonAction(sender: UIButton!) {
        isSelected = !isSelected
        sender.setImage(UIImage(systemName: isSelected ? "checkmark" : "xmark"), for:
        ↪  .normal)
    }
}
```

Code-Ausschnitt 4.2: Umsetzung eines klickbaren Buttons mit UIKit

Zuerst wird die Variable `isSelected` angelegt, um den Status des Buttons zu verwalten. Der Button wird erstellt und beim Klicken von diesem wird auf eine externe Funktion verwiesen, die das Bild verändert und den Status des Buttons aktualisiert. Die Funktion nimmt den zu modifizierenden Button als Parameter und verändert das Bild des Buttons direkt. In SwiftUI wird der gleiche Button wie folgt erstellt.

```
struct MyView: View {
    @State var isSelected = false

    var body: some View {
        Button(action: {
            isSelected.toggle()
        }) {
            Image(systemName: isSelected ? "checkmark" : "xmark")
        }
    }
}
```

Code-Ausschnitt 4.3: Umsetzung eines klickbaren Buttons mit SwiftUI

In Code Ausschnitt 4.3 wird der Property Wrapper `@State` genutzt, um die Variable `isSelected` mit weiteren Funktiontionalitäten zu versehen. Wenn `isSelected` nun durch ein Klicken des Buttons verändert wird, rendert das Framework den View automatisch mit der veränderten

Variable neu. Eine wichtiger Bestandteil von Swift, um diese Funktionalität mit SwiftUI umzusetzen, ist der *Ternary Operator*:

```
question ? answer1 : answer2;
```

Dieser evaluiert eine Aussage und gibt anhand des booleanischen Ergebnisses eins von zwei Elementen wieder (vgl. Apple, 2014e). In diesem Fall wird anhand der Variable `isSelected` ein unterschiedlicher String wiedergegeben und somit ein anderes Bild gerendert. Der Ternary Operator ist für SwiftUI besonders wichtig, da die View-Structs nach dem Erstellen nicht mehr verändert werden können. Die Variablen, von welchen abhängt, welche Views angezeigt werden, sind jedoch weiterhin zu mutieren.
Globale Daten werden meist von mehreren Views genutzt und manipuliert. Diese werden von beiden Frameworks in Klassen gespeichert, verwaltet und müssen über die verschiedenen Views synchron gehalten werden. Hierfür werden in UIKit und SwiftUI Singletons, auch globale Variablen genannt, verwendet (vgl. Apple, 2018c; Apple, 2019d). In Apples *Cocoa Design Patterns* wird diese Umsetzung explizit beschrieben und als Lösung für UIKit vorgeschlagen. Apple sieht die Gestaltung eines Singletons wie folgt vor (vgl. Apple, 2018c):

```
class Singleton {
   static let sharedInstance: Singleton = {
       let instance = Singleton()
       var value = 0
       return instance
   }()
}
```

Code-Ausschnitt 4.4: Erstellen eines Singletons für UIKit

Mit dem `static let` wird sichergestellt, dass bei jeder Instanziierung dieser Klasse keine neue Kopie erstellt, sondern auf eine Instanz dieser zugegriffen wird. Da UIKit-Views von Viewcontrollern verwaltet werden, wird in diesen auf die globalen Variablen zugegriffen. Wenn in UIKit nun mehrere Viewcontroller eine Instanz dieser Klasse erstellen, greifen alle auf dasselbe Objekt zu und Veränderungen dieser werden automatisch an alle Viewcontroller mitgeteilt. Jedoch hängt der Inhalt der Views nicht direkt von diesen Variablen ab. Sie werden somit nicht automatisch verändert, wenn der Singleton verändert wird. Der Viewcontroller muss auf Veränderungen von diesem hören, darauf hin Funktionen ausführen und die Eigenschaften der Views direkt verändern. Hierfür wird ein *Property Observer* verwendet (vgl. Apple, 2018e).

```
class MyObserver: NSObject {
   @objc var singleton: Singleton
   var observation: NSKeyValueObservation?

   init(object: Singleton) {
       singleton = object
       super.init()
       observation = observe(
           \.singleton
       ) { object, change in
           // functions and ui changes
       }
   }
}
```

Code-Ausschnitt 4.5: Erstellen eines Observers für UIKit

Der Observer und das zu beobachtende Objekt, der Singleton, müssen nun im Viewcontroller verbunden werden, da dieser den Singleton verwendet (vgl. Apple, 2018e).

```
let observed = MyObjectToObserve()
let observer = MyObserver(object: observed)
```

Code-Ausschnitt 4.6: Verbinden von UIKits Singleton und Observer

Nun kann der Viewcontroller auf die Veränderungen des Singletons reagieren und abhängig von diesen die nötigen Views anpassen. Da die Views nicht direkt von den globalen Variablen abhängig sind, können Inkonsistenzen der Elemente entstehen, wenn vergessen wird einen View zu verändern. In SwiftUI gibt es für die Nutzung von globalen Variablen ein eigenes Protokoll und einen Property Wrapper. Das *ObservableObject* Protokoll nimmt dem Entwickler das Implementieren dieser Funktionen ab (vgl. Apple, 2019d). Wie in UIKit wird in SwiftUI eine Klasse erstellt, welche den Singleton enthält.

```
class Singleton: ObservableObject {
    @Published var value = 0
}
```

Code-Ausschnitt 4.7: Erstellen eines Singletons für SwiftUI

In der Klasse wird die Variable, welche zur Verfügung gestellt werden soll, durch den Property Wrapper `@Published` gekennzeichnet. Ein View, der diesen Singleton verwendet, erstellt diesen mit dem Property Wrapper `@ObservedObject`.

```
@ObservedObject var singleton = Singleton()
```

Code-Ausschnitt 4.8: Instanziierung eines Singletons in SwiftUI

Wenn der Singleton nun von einem View verändert wird, teilt dieser die Veränderung an alle Views mit, die diesen mit `@ObservedObject` verwenden (vgl. Apple, 2020d). Die Views verwalten diese Informationen und rendern die Komponenten neu, die sich dadurch verändern. Die Umsetzung für SwiftUI ist mit diesen Zeilen Code abgeschlossen, da die Singletons direkt von den Views verwendet und nicht über einen Viewcontroller an diese vermittelt werden müssen. Der Singleton sollte bei dieser Implementation einmalig erstellt und an die Views die diesen ebenfalls verwenden weitergegeben werden, da bei jeder weiteren Erstellung des Singletons eine neue Instanz erstellt wird. Da SwiftUI den Lifecycle der Views im Hintergrund verwaltet, kann es dazu kommen, dass Views zerstört werden, in welchen ein `@ObservedObject` erstellt wurde. Wenn dieser View wieder benötigt wird, wird das Objekt ebenfalls neu erstellt und deren Inhalt zurückgesetzt. Sollte das Objekt dabei nicht neu erstellt werden, kann der Property Wrapper `@StateObject` verwendet werden (vgl. Apple, 2020f). Dieser sorgt dafür, dass die DynamicProperty des Views nicht mit gelöscht, sondern davon ausgenommen werden. Dies ist nur sinnvoll, wenn der View das Objekt selbst erstellt. Falls der View das Objekt von dem Parent-View bekommt, ist das Verwenden des ObservedObject ausreichend, da der Parent-View die Instanz verwaltet. Das Verwenden von StateObject ist ebenfalls nützlich, wenn es sich um größere komplexe Objekte handelt, da das Neuerstellen von diesen möglicherweise mehr Zeit und Leistung in Anspruch nimmt als es gespeichert zu lassen. Durch die Umsetzung wie in Code Ausschnitt 4.8 muss das ObservedObject als Parameter an die Views übergeben werden, was bei komplexen und tiefen View-Hierarchien aufwendig und unübersichtlich ist. Um dieses Problem zu umgehen, gibt es den `@EnvironmentObject` Property Wrapper. Dieser ermöglicht es, Variablen aus der Laufzeitumgebung (dem Environment) zu laden (vgl. Apple, 2020c). Dadurch sind die Objekte ohne das Weitergeben dieser von View zu View in der gesamten App verfügbar. Hierfür müssen die Singletons einmalig in der App, der Struct der die App erstellt und alle Views enthält, mit `@ObservableObject` erstellt und mit dem View-Modifier `.environmentObject()` an den obersten View weitergegeben werden.

```
@main
struct MyApp: App {
    @ObservedObject var singleton = Singleton()

    var body: some Scene {
        WindowGroup {
            MyView()
                .environmentObject(singleton)
        }
    }
}
```

Code-Ausschnitt 4.9: Laden eines SwiftUI Singletons in das Environment (Laufzeitumgebung)

Somit sind die Objekte für alle Child-Views, also alle Views der App, automatisch mit dem `@EnvironmentObject` Property Wrapper verfügbar. Die `@ObservedObject` Variablen müssen anstatt diesem dafür nur noch mit dem `@EnvironmentObject` Wrapper versehen werden und nun nicht mehr von View zu View weitergegeben werden:

```
@EnvironmentObject var singleton: Singleton
```

Code-Ausschnitt 4.10: Benutzung eines SwiftUI Singletons aus dem Environment

Durch das `@EnvironmentObject` sucht das Framework automatisch nach dem passenden Typen der Klasse in der Laufzeitumgebung, wodurch das Weitergeben dieser durch die Initialisierer nicht mehr nötig ist (vgl. Apple, 2020c). Weitere Property Wrapper sind `@AppStorage`, welcher das persistente Speichern von Daten und `@SceneStorage`, welcher das Speichern von Daten der Szenen ermöglicht (vgl. Apple, 2020e). Hudson hat in einem seiner weiteren Projekte SwiftUIs Konzept des State Managements in UIKit implementiert, was jedoch sehr aufwendig ist (vgl Hudson, 2020). Zusammenfassend lässt sich sagen, dass eine SwiftUI-App eher nach dem MVVM und nicht wie eine UIKit-App nach dem MVC-Muster aufgeteilt ist, da die beschrieben Property Wrapper die Rolle eines Viewmodels übernehmen, indem sie die Views und Daten automatisch synchron halten.

4.4 Views

Weitere Unterschiede der beiden Frameworks werden in der Art und Weise deutlich, wie Views erstellt, angepasst und angeordnet werden.

4.4.1 View Anpassung

Um dies zu verdeutlichen wird ein Text-View mit gelbem Hintergrund und Einrückung auf allen Seiten erstellt. Anhand der Code-Ausschnitten wird die unterschiedliche Herangehensweise deutlich:

```
var textView = UITextView()
textView.text = "Hello World"
textView.contentInset = UIEdgeInsets(top: 10, left: 10, bottom: 10, right: 10)
textView.backgroundColor = .yellow
```

Code-Ausschnitt 4.11: Erstellen und anpassen eines Views in UIKit

Da UIKit-Views in Variablen gespeichert werden, werden die Eigenschaft dieser direkt angesprochen und bearbeitet. In SwiftUI hingegen werden keine Referenzen der Views erstellt und in Variablen speichert, also müssen die Eigenschaften dieser auf einem anderen Weg verändert werden. Hierfür implementiert SwiftUI das Protokoll der View-Modifier. Diese werden auf Views angewendet und produzieren ein verändertes Ergebnis. Sie verändern die Views nicht direkt, sondern wrappen den View in einen neuen veränderten View (vgl. Apple, 2019k). Das ist nötig, um mehrere View-Modifier und somit Veränderungen auf einen View anwenden zu können. Dies vereinfacht das Anwenden von mehreren Veränderungen an einem View:

```
Text("Hello World")
        .padding()
        .background(Color.yellow)
```

Code-Ausschnitt 4.12: Erstellen und anpassen eines Views in SwiftUI

Die Reihenfolge der Modifier ist jedoch im Gegenteil zu UIKit relevant. Jeder Modifier wrapped den darunterliegenden View erneut in einen eigenen ModifiedContent-View, auf welchem der nächste Modifier angewendet werden kann (vgl. Apple, 2019k). Aus diesem Grund erzielen die beiden Code-Ausschnitte aus 4.3 folgende unterschiedliche Ergebnisse:

```
Text("Hello World")
        .background(Color.yellow)
        .padding()
```

```
Text("Hello World")
        .padding()
        .background(Color.yellow)
```

Code-Ausschnitt 4.13: Unterschiedliche Anordnungen der View-Modifier

Abbildung 4.3: Ergebins von unterschiedlicher Anordnungen der View-Modifier aus Code-Ausschnitt 4.3

Eine weitere nützliche Funktion der View-Modifier ist, dass sie in das Environment geladen und an potenzielle Child-Views weitergegeben werden. Diese werden darauf hin bei allen Child-Views eingesetzt, falls diese anwendbar sind (vgl. Apple, 2019k). Somit kann der View-Modifier `.foregroundColor`, der bei zwei Text Views angewendet werden soll, in den Parent-View hochgezogen werden und wird somit dennoch auf beide Child-Views angewendet:

```
VStack {
        Text("Hello World")
        Text("Hello World")
}.foregroundColor(.white)
```

Code-Ausschnitt 4.14: View-Modifier in der View-Hierarchie anheben

Dadurch kann der Code reduziert und die Lesbarkeit erhöht werden.

4.4.2 View Anordnung

Auch die Anordnung der Views unterscheidet sich bei den Frameworks. In UIKit kann jeder View mehrere SubViews enthalten, während in SwiftUI nur bestimmte Views diese Funktion umsetzen. Diese sind: `VStack`, `HStack`, `ZStack` und `Group`. Die Ausrichtung dieser Subviews erfolgt in UIKit mithilfe des Autolayouts, bei welchem die Views in Relation zu anderen Views angeordnet werden (vgl. Apple, 2016c). Es wird beispielsweise der Abstand oder der Mittelpunkt des Views in Abhängigkeit eines anderen Views beschrieben. Diese werden in UIKit Constraints (deutsch = Beschränkung) genannt (vgl. Apple, 2015c).

```
NSLayoutConstraint.activate([
            textView.topAnchor.constraint(equalTo: viewA.topAnchor, constant, 10),
            textView.bottomAnchor.constraint(equalTo: viewA.bottomAnchor),
            textView.centerXAnchor.constraint(equalTo: viewA.centerXAnchor),
        ])
```

Code-Ausschnitt 4.15: Positionierung eines Views in UIKit

Dabei ist zu beachten, dass sich die angegebenen Constraints nicht widersprechen dürfen, da die App sonst abstürzt. Es darf den Views beispielsweise keine explizite Größe zugewiesen

werden, wenn die Constraints ein Anpassen der Größe vorsehen. Dahingegen werden die Views in SwiftUI durch Alignments angeordnet (vgl. Apple, 2019g).

```
VStack (alignment: .leading){
    Text("Hello World")
}
```

Code-Ausschnitt 4.16: Positionierung eines Views in SwiftUI

Im Gegensatz zu den Constraints sind Alignments in SwiftUI optional. Wenn kein Alignment angegeben ist, ordnet SwiftUI die Views automatisch an. Der Unterschied der beiden Frameworks wird hier deutlich, da in der UIKit App insgesamt 54 Anpassungen der Anordnung vorgenommen wurden, während es in SwiftUI lediglich neun waren. Ebenfalls werden in beiden Frameworks Views verwendet, um leere Flächen einzunehmen und andere Views zur Seite zu drängen. In UIKit werden `UIView()`s hierfür verwendet, während SwiftUI hierfür mit dem `Spacer()` einen eigenen View für diese Funktion besitzt. Die UIKit-App verwendet mit 10 dieser UIViews im Vergleich zu SwiftUIs 24 Spacern weniger Views dieser Art. Dennoch sind die benötigten Anpassungen der Ausrichtung in SwiftUI deutlich geringer, da das Framework aufgrund des deklarativen Syntax viele von diesen für den Entwickler übernimmt.
Eine weitere Funktion, die SwiftUI vereinfacht, ist das Anordnen von mehreren Views. Wie beschrieben kann ein View in UIKit immer mehrere SubViews besitzen, während in SwiftUI spezielle Views dafür genutzt werden müssen. Eine Funktion, welche das deklarative Konzept von SwiftUI für diese ermöglicht, ist der function builder. Ein function builder ermöglicht es, mehrere Elemente einer Funktion zu einem Wert zusammenzufügen (vgl. Apple, 2019j). In SwiftUI ist dies bereits für Views umgesetzt und wird als View Builder bezeichnet. Um die Wichtigkeit und den Nutzen von diesem zu erläutern, hilft ein Code-Ausschnitt. Mehrere Views in vertikaler Anordnung zusammenzufügen wird ohne einen Viewbuilder, wie in UIKit, imperativ umgesetzt.

```
var stack = VStack()
    stack.add(Text("..."))
    stack.add(Image(uiImage: image))
    stack.add(Text("..."))
    return stack()
}
```

Code-Ausschnitt 4.17: Anordnen von Views ohne View-Builder

Dieser Code-Ausschnitt ähnelt der Umsetzung eines UIStackViews in UIKit, mit welchem mehrere SubViews in vertikaler oder horizontaler Ausrichtung angeordnet werden können. Das Erstellen des selbigen Stacks ist mit SwiftUI in folgender verkürzter Form möglich.

```
VStack {
    Text("...")
    Image(uiImage: image)
    Text("...")
}
```

Code-Ausschnitt 4.18: Anordnen von View in SwiftUI mithilfe des View-Builders

Die Frage, wie der function builder diese Verkürzung in SwiftUI ermöglicht, wird nachstehend erläutert. Da dies eine noch neue Funktion ist, gibt es zum Zeitpunkt des Schreibens dieser Arbeit noch keine Dokumentation für diese von Apple. Es existiert lediglich eine kurze Dokumentation des ViewBuilders (vgl. Apple, 2019j) und da Swift Open-Source ist ein Vorschlag der Implementation eines function builders auf GitHub (vgl. McCall, 2019). Durch function builder lassen sich mehrere Views zu einem einzelnen zusammenführen, ohne es wie ein UIStackView explizit programmieren zu müssen. In Apples Dokumentation des ViewBuilders geben sie keine Implementierung von diesem. Es lässt sich jedoch anhand der Funktionen von diesem eine Implementierung vermuten.

```
@functionBuilder
struct VStack {

    static func buildBlock() -> EmptyView {
        return EmptyView()
    }

    static func buildBlock<A: View>(_ A: View) -> some View {
        return A
    }

    static func buildBlock<A: View, B: View>(_ A: View, _ B: View) ->
    ↪  TupleView<(A, B)> {
        return TupleView<(A, B)>
    }

    ...
}
```

Code-Ausschnitt 4.19: Potentielle Umsetzung eines VStacks und Nutzung des function-builders

Die Methoden des ViewBuilders wurden hier in einem Struct zusammengeführt, um eine mögliche Umsetzung dessen zu entwerfen. Bei mehreren Views werden diese in einen eige-

nen TupleView zusammengefügt und als ein View wiedergegeben. Dies ist durch `type(of: )` Befehl in SwiftUI zu prüfen.

```
print(type(of: stack))
// VStack<TupleView<(Text, Image, Text)>>
```

Code-Ausschnitt 4.20: Viewhierarchie des SwiftUI-Views aus Code-Ausschnitt 4.18

In einem View Block in SwiftUI dürfen aufgrund des ViewBuilders nur ausgewählte Objekt und Ausdrücke verwendet werden. Zum Zeitpunkt des Schreibens dieser Arbeit dürfen Views, if-, if-let- und switch-Abfragen im ViewBuilder verwendet werden. Mehrere Views werden, wie gezeigt, in einem TupleView zusammengefügt, während if- und switch-Abfragen in ConditionalContent-Views und If-let-Abfragen in Optionale Container zusammengeführt werden. Diese Methoden, mithilfe des View Builders für das Erstellen von Views verwenden zu können, ermöglicht es Apps mit einem deutlich geringeren Aufwand und weniger Code zu erstellen.

4.5 Länge des Codes

4.5.1 Lines of Code

Um den Umfang der zwei Applikationen zu messen und zu vergleichen bietet sich das Zählen der Lines of Code an. Die Effizienz eines Programms anhand der Zeilenanzahl zu messen ist jedoch nur bedingt sinnvoll. Die Zeilen eines Programms lassen sich auf ein Minimum reduzieren, was sich negativ auf die Lesbarkeit des Codes auswirken kann. Für das Zählen der LOC sind sowohl leere Zeilen als auch Kommentare zu berücksichtigen, da diese zu der Struktur und Leserlichkeit des Codes beitragen, jedoch nicht benötigt werden, um Funktionen mit möglichst wenigen Zeilen umzusetzen. Vor der Zählung wurden beide Codebases erneut betrachtet und auf Redundanz und fehlende Modularität untersucht und in einem sinnvollen Maß gekürzt, ohne dass die Lesbarkeit darunter leidet. Es wurden drei Methoden für die Zählungen verwendet. Dabei wurden die Ordner UITests und Tests, die von Apple automatisch erstellt und nicht verwendet wurden, nicht berücksichtigt. Diese sind lediglich für das Testen von Code Ausschnitten nötig und somit für diese Auszählung irrelevant. Die Messung erfolgt somit in den Ordnern UIKitvsSwiftUI/UIKitTest/UIKitTest und UIKitvsSwiftUI/SwiftUITest/SwiftUITest. Mit folgendem Befehl wurden alle Zeilen der .swift-Dateien des Anwendungsordners gezählt.

```
find . \( -iname \*.swift \) -exec wc -l '{}' \+
```

Code-Ausschnitt 4.21: Script zum Zählen aller Zeilen Code

Nach dieser Messung benötigt die UIKit App mit 1153 mehr als doppelt so viele Zeilen wie die SwiftUI App (569). Da die Lines of Code auf diese Weise zu zählen jedoch nicht sehr nützlich ist, wurden weitere Methoden betrachtet, um ein aussagekräftiges Ergebnis zu erhalten. Die zweite Messung erfolgte mit Hilfe eines von AlDanial entwickelten Packages *cloc* (vgl. AlDanial, 2015), welches sowohl leere Zeilen als auch Kommentare berücksichtigt und das Ergebnis detailliert wiedergibt. Durch das Ergebnis dieser Zählung wird deutlich, dass die Anzahl der Zeilen, die Programmcode enthalten, bei UIKit mit nun 840 Zeilen deutlich geringer sind, als es die erste Messung zu erkennen gab. Dies war zu erwarten, da in UIKit aufgrund der fehlenden Einrückungen für die Lesbarkeit des Codes öfter Absätze genutzt wurden. Auch bei SwiftUI verringerten sich die LOC auf 484. Die tatsächlichen LOC für beide Apps unterscheiden sich jedoch weiterhin deutlich. Die UIKit App umfasst über 74 % Zeilen mehr als die SwiftUI App. Eine weitere Bibliothek, die es ermöglicht die LOC nach den gleichen Kriterien berechnen zu lassen, ist *scc*. Die Ergebnisse dieser entsprechen exakt der *cloc*-Bibliothek. Zusätzlich liefert *scc* einen Wert, welcher die Komplexität des Codes widerspiegelt, in dem es die Zweige, wie if-Statements und Loops, im Code zählt und den Wert für jedes Vorkommen dieser erhöht (vgl. Boyter, 2020). Für den Vergleich der beiden Frameworks ist dieser Wert jedoch nicht sinnvoll. Die zwei entwickelten Apps unterscheiden sich in diesem Bereich mit einer Differenz von 23 % nicht bedeutend, da es sich hauptsächlich um UI-Frameworks, mit wenigen Abzweigungen handelt. Die Messungen der Lines of Code mithilfe der Bibliotheken wurden dazu mit eigenen Messungen abgeglichen. Dafür wurde folgendes Script entwickelt:

```
find . \( -iname \*.swift \) -exec awk ' { {  //runs script for all .swift files
        gsub(/^ +/,"")                        //deletes spaces at beginning
    } {
        if (length($0) > 0 && /^[^\/]/) {     // digards blank & commented lines
            totlin+=1
        }
    }
}
END {
    printf("loc: %d\n", totlin)
} ' '{}' \+
```

Code-Ausschnitt 4.22: Script für das gefilterte Zählen der Zeilen

Das Script ignoriert alle Leerzeichen der Zeilen, die für das Einrücken der Zeilen genutzt werden und zählt die Zeilen nur, wenn:

- die Länge der Zeile, abzüglich der Leerzeichen für die Einrückung, größer als null ist und
- es sich um keine Kommentarzeile handelt.

Die Ergebnisse dieses Script stimmten exakt mit den Ergebnissen der beiden Bibliotheken überein und bestätigten diese damit. Wie in Kapitel 3.2 erwähnt, entwickelte Hudson ebenfalls

eine App, welche für die Auswertung der Lines of Code relevante Ergebnisse erzielte. Hudson hat für die Zählung der Zeilen das Modell der Datenstruktur und die Decodierung dieser vernachlässigt, da diese in beiden Anwendungen identisch sind. Hudson veröffentlichte seine Methode zur Zählung der Zeilen leider nicht, führte diese allerding nach denselben Kriterien durch. Leer- und Kommentarzeilen wurden nicht mitgezählt. Während der Code für SwiftUI 52 Zeilen einnahm, wurden für UIKit 92 Zeilen benötigt (vgl. Husdon, 2019). Für die Entwicklung mit UIKit sind somit fast 77 % mehr Zeilen Code benötigt. Zu berücksichtigen ist, dass Hudson einige Funktionen des Anlegens der Liste in UIKit mit dem Interface Builder gelöst hat. Hätte Hudson dies ebenfalls programmatisch umgesetzt, würden sich die Lines of Code der UIKit App und die zugehörige Prozentzahl erhöhen. Die Ergebnisse von Hudson sind denen der Abwandlung der Sustainable App mit einer Abweichung von lediglich 2,9 % sehr ähnlich und bestärken die Ergebnisse. Hudson hatte neben der Lines of Code zusätzlich die benötigte Entwicklungszeit seiner beiden Applikationen gemessen. Dies wurde für die Entwicklung der gekürzten Version der Sustainable App nicht gemessen, da verschieden ausgeprägte Vorkenntnisse vorlagen, die das Ergebnis verfälscht hätten. Da Hudson sowohl in UIKit als auch in SwiftUI ausreichende Vorkenntnisse hatte, bot sich solch ein Vergleich für ihn an. Für die Entwicklung mit UIKit benötigte er 16 Minuten und 46 Sekunden, während die Entwicklung der SwiftUI Version 9 Minuten und 25 Sekunden dauerte (vgl. Husdon, 2019). Die Entwicklung der UIKit App hat daher ungefähr 78 % mehr Zeit benötigt. Auch wenn Hudson kein wissenschaftliches Experiment durchgeführt hat, wird deutlich, dass die Entwicklung einer simplen Applikation mit weniger Zeit und Zeilen Code in SwiftUI umsetzbar ist. In den nachfolgenden Tabellen ist der Unterschied der beiden Frameworks prozentual von UIKit über SwiftUI angegeben:

Methode (writer)	**UIKit**	**SwiftUI**	**Unterschied***
Lines of Code inkl. Kommentar-, Leerzeilen	1153	569	102 %
Lines of Code (mit Bibliotheken *cloc* & *scc*)	840	484	74 %
Lines of Code (mit Script)	840	484	74 %

Tabelle 4.1: Ergebnisse der Berechnungen der Zeilenanzahl

* UIKit über SwiftUI in Prozent

4.5.2 Länge der Zeilen

Eine weitere Messung, die ein aufschlussreiches Ergebnis produzieren könnte, ist zuzüglich die Länge der Zeilen zu messen. Die Programmierung zeigte bereits, dass sich nicht nur die Anzahl der Zeilen, sondern auch die Länge dieser unterscheiden. Die Messung der Zeilenlänge ist ebenfalls ein ausschlaggebender Faktor der den Entwicklungsaufwand direkt beeinflusst. Ein Framework, welches die Hälfte der Zeilen benötigt, die jedoch die doppelte Länge

umfassen, würde bei einer Messung der Lines of Code fälschlicherweise besser abschneiden. Aus diesem Grund wurde das selbst erstellte Script angepasst, um die Zeilenlänge bestimmen zu können. Hierfür wurden die zuvor erstellten Filter genutzt. Die Zeilenlängen wurden addiert und abschließend durch die Anzahl der gezählten Zeilen geteilt, um die durchschnittliche Länge einer Zeile für das Framework zu ermitteln.

```
find . \( -iname \*.swift \) -exec awk ' { {   //runs script for all .swift files
        gsub(/^ +/,"")                         //disgards spaces at beginning of line
    } {
        if (length($0) > 0 && /^[^\/]/) {      // digards commented lines
            totlen+=length($0)
            totlin+=1
        }
    }
}
END {
    printf("average line length: %d\n", totlen/totlin)
} ' '{}' \+
```

Code-Ausschnitt 4.23: Script für die Berechnung der Zeilenlänge

Eine Zeile Code der UIKit-App umfasst im Schnitt ungefähr 38 Zeichen, während eine Zeile der SwiftUI-App durchschnittlich 23 Zeichen lang ist. Die Zeilen der UIKit-App sind somit im Durchschnitt circa 65 % länger.

Methode	UIKit	SwiftUI	Unterschied*
Durchschnittliche Zeilenlänge	38	23	65 %

Tabelle 4.2: Durchschnittliche Zeilenlänge der beiden Apps

Um das gesamte Programmvolumen zu ermitteln, werden die Ergebnisse der durchschnittlichen Zeilenlänge mit der Zeilenanzahl zusammengeführt. Hierfür werden die prozentualen Unterschiede der beiden Ergebnisse multipliziert. 1,75 * 1,65 = 2,89. Um dieses Ergebnis zu validieren, wurde das Script angepasst, um alle Zeichen der beiden Apps nach den obigen Filtern zu zählen.

Methode	UIKit	SwiftUI	Unterschied*
Anzahl aller Zeichen	32033	11360	182 %

Tabelle 4.3: Anzahl aller Zeichen der beiden Apps

Diese beiden Ergebnisse stimmen mit einer Abweichung von 7 % aufgrund des Abrundens der Prozentzahlen überein. Somit umfasst die UIKit App insgesamt mit ca. 282 % fast dreimal mehr Zeichen als die SwiftUI App, was mit einem höheren Arbeitsaufwand verbunden werden kann.

4.6 Code Komplexität

In diesem Kapitel werden die beiden abgewandelten Versionen der Sustainable App über die Lines of Code und Zeilenlänge hinaus auf die Komplexität des Codes untersucht. Hierfür wurden verschiedene Techniken für die Bewertung von Texten betrachtet. Methoden wie der Flesch Index (vgl. Flesch u. Gould, 1949), Coleman-Liau Index (vgl. Coleman u. Liau, 1975) und Gunnings FOG Index (vgl. Gunning u. a., 1952) sind für die Messung der Lesbarkeit von Texten nützlich. Hierbei werden Faktoren wie die Satzlänge, Wortlänge und Silbenanzahl des Textes betrachtet, weswegen diese Methoden für Programmcode jedoch nicht sinnvoll anwendbar sind. Satzzeichen, Klammern und andere Sonderzeichen würden in diesen Berechnungen nicht mit einbezogen und das Ergebnis weniger aussagekräftig werden. Anstatt der Lesbarkeit wurde die Komplexität des Codes gemessen. Wie in Kapitel 4.5 genannt, gab die Bibliothek scc für die Messung der Lines of Code ebenfalls einen Wert für die Komplexität des Codes wieder. Die Berechnung von diesem erfolgte dabei lediglich durch das Zählen der Abzweigungen im Code (vgl. Boyter, 2020). Eine solche Messung ist für Algorithmen, jedoch nicht für das Evaluieren von UI-Frameworks sinnvoll, da die Nutzung von Abzweigungen wie Loops und konditionale Abfragen bei diesen keine zentrale Rolle spielen. Die McCabe-Metrik (vgl. McCabe, 1976) berechnet die Komplexität des Codes ebenfalls über diesen Weg, weshalb sie bei dieser Arbeit keine Verwendung findet. Eine Metrik, dessen Berechnungen sich für den Vergleich der Frameworks UIKit und SwiftUI eignet, ist die Halstead Metrik. Maurice Howard Halstead (vgl. Halstead, 1977) erstellte diese umfangreiche Metrik, welche mehrere Berechnungen enthält, im Jahre 1977. Die gezählten Elemente des Code teilt Halstead in Operanden und Operatoren auf. Operanden sind die Objekte, auf welchen Funktionen ausgeführt werden. Diese Funktionen werden von Halstead Operatoren genannt. Das Zählen aller Operanden und Operatoren für beide Apps überschreitet, da diese insgesamt 1324 Zeilen umfassen, den Umfang dieser Arbeit. Aus diesem Grund wurde Halsteads Metrik bei einem einzelnen optisch identischem UI Element der beiden Apps angewendet. Der betrachtete View ist ein Button, welcher ein Bild, einen Text und Hintergrundfarbe enthält und einen Schatten wirft.

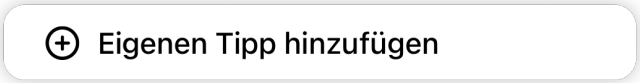

Abbildung 4.4: Analysierter View für die Code Komplexität

Die Abbildungen 4.5 und 4.6 zeigen den benötigten Code, um dies in UIKit und SwiftUI umzusetzen.

```
class LabelButton: UIButton {
    override init(frame: CGRect) {
        super.init(frame: frame)
    }

    init(icon: String, title: String) {
        super.init(frame: .zero)
        setImage(UIImage(systemName: icon, withConfiguration:
UIImage.SymbolConfiguration(pointSize: 18, weight: .medium, scale: .default)),
for: .normal)
        tintColor = .black
        setTitle(title, for: .normal)
        titleLabel?.font = UIFont.systemFont(ofSize: 16, weight: .medium)
        setTitleColor(.black, for: .normal)
        titleEdgeInsets = UIEdgeInsets(top: 0, left: 33, bottom: 0, right: 0);
        imageEdgeInsets = UIEdgeInsets(top: 0, left: 22, bottom: 0, right: -10);
        contentHorizontalAlignment = .left
        backgroundColor = .white
        layer.cornerRadius = 15
        layer.shadowColor = UIColor.black.cgColor
        layer.shadowOffset = CGSize(width: 0, height: 0)
        layer.shadowRadius = 10
        layer.shadowOpacity = 0.05
        heightAnchor.constraint(equalToConstant: 45).isActive = true
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Abbildung 4.5: Code-Ausschnitt des Buttons in UIKit (Operanden blau und Operatoren rot eingefärbt)

```
struct ButtonLabel: View {

    var icon: String
    var text: String

    var body: some View {
        Button(action: {}) {
            HStack {
                Image (systemName: icon)
                    .font(.system(size: 18, weight: medium))
                Text(text)
                    .font(.system(size: 16))
                    .fontWeight(.medium)
            }
            .padding(13)
            .padding(.leading, 10)
            Spacer()
        }
        .frame(width: UIScreen.main.bounds.width - 30, height: 45)
        .background(Color(.white))
        .cornerRadius(15)
        .shadow(color: Color(.black).opacity(0.05), radius: 5, x: 4, y: 4)
    }
}
```

Abbildung 4.6: Code-Ausschnitt des Buttons in SwiftUI (Operanden blau und Operatoren rot eingefärbt)

In den Code-Ausschnitten sind die Operanden blau und die Operatoren rot eingefärbt, um die Zählung dieser zu verdeutlichen. Die Elemente des Codes einer der beiden Kategorien zuzuordnen ist nicht immer eindeutig, weswegen es kurz anhand eines Beispiels (siehe Abbildung 4.7) erklärt wird:

```
                setTitle(title, for: .normal)
button.setTitle(title, for: .normal)
```

Abbildung 4.7: Code-Ausschnitt des Buttons in UIKit (Operanden blau und Operatoren rot eingefärbt)

Die Funktionsnamen der Funktionen, die den View verändern, sind blau markiert, da diese auf den View angewendet, jedoch nicht explizit erwähnt werden muss. Es handelt sich dennoch um eine Zuweisung eines Wertes, weswegen die Klammern dieser rot markiert sind. Somit ist die Zeile in Code-Ausschnitt 4.7 eine Zuweisung des Titels für den Status *.normal* des Buttons, weswegen sowohl der Button, *title*, *for* und *.normal* Operanden sind. Ein Operator ist der Doppelpunkt, welcher der Variable *for* den Wert *.normal* zuweist und die Klammern, welche den zusammengesetzten Titel dem Titel des Buttons zuweisen. Die Klammern sind beide rot markiert, werden jedoch als eine Operation gezählt, da die schließende Klammer die Operation lediglich beendet. Halstead (vgl. Halstead, 1977) nimmt die Operanden und Operatoren als Basis für alle weiteren Berechnungen. Hierfür werden sowohl unterschiedliche Operanden und Operatoren als auch die Anzahl der Vorkommnisse aller gezählt. Die folgenden Aufzählungen und Berechnungen erfolgen nach Halsteads Mertik (vgl. Halstead, 1977):

	UIKit	**SwiftUI**
n1 = Anzahl einzigartiger Operatoren	4	3
N1 = Anzahl aller Operatoren	56	32
n2 = Anzahl einzigartiger Operanden	71	46
N2 = Anzahl aller Operanden	88	55

Tabelle 4.4: Anzahl der Operatoren und Operanden nach Halstead

Anhand dieser Daten lassen sich weitere Werte berechnen. Nach Halstead ist die Vokabelgröße des Programms die Summe aller einzigartigen Operanden und Operatoren.

$$n = n1 + n2$$

	UIKit	SwiftUI
Vokabelgröße n	75	49

Tabelle 4.5: Vokabelgröße der Button-Views nach Halstead

Die Programmlänge ist demnach die Summe aller verwendeter Operanden und Operatoren.

$$N = N1 + N2$$

	UIKit	SwiftUI
Programmlänge N	144	87

Tabelle 4.6: Programmlänge der Button-Views nach Halstead

Das Programmvolumen gibt den Informationsgehalt von diesem in Bits an.

$$V = N * log2(n)$$

	UIKit	SwiftUI
Programmvolumen V	896.95 Bits	488.48 Bits

Tabelle 4.7: Programmvolumen der Button-Views nach Halstead

Dieses Ergebnis mit den Berechnung des Programmvolumens in Kapitel 4.5 zu vergleichen, gibt Aufschluss über die Aussagekräftigkeit dieser Resultate. Hierfür wurde das Programmvolumen der beiden Views mit dem Script aus Kapitel 4.5.1 ermittelt.

	UIKit	SwiftUI
Anzahl aller Zeichen	931	438

Tabelle 4.8: Anzahl aller Zeichen der Button-Views

Nach Halsteads Berechnung des Programmvolumens beträgt das des Views in UIKit ungefähr 83 % mehr, während das Volumen nach der Berechnung mithilfe des Scripts ca. 112 % größer

ist. Die Abweichung von 29 % macht eine Schwäche der Halstead Metrik deutlich. Diese betrachtet bei der Berechnung der Werte lediglich die Wortanzahl und ignoriert die Länge der dieser. Um die Wortlänge in Halsteads Berechnung des Programmvolumens zu integrieren, muss die Gesamtzahl aller Zeichen durch die Anzahl der Worte geteilt werden:

	UIKit	**SwiftUI**
Anzahl aller Zeichen	931	438
Anzahl der Wörter	117	65
Durchschnittliche Wortlänge	7.95	6.74

Tabelle 4.9: Berechnung der Wortlängen der Button Views

Dies ergibt die durchschnittliche Wortlänge der Views. In UIKit beträgt die Wortlänge ca. 7.95 und in SwiftUI ca. 6.74 Zeichen. Die Wörter in der UIKit Umsetzung sind somit ca 17 % länger. Nun muss die Anzahl der Operanden in UIKit mit diesem Wert multipliziert werden. Das Volumen des UIKit Views beträgt nun mit 1024 Bits ca. 110 % des Volumens des Views in SwiftUI (488 Bits). Die Abweichung beträgt nach dem Einbeziehen der Wortlänge lediglich 2 % und entspricht somit fast den Berechnungen des gesamten Codes aus Kapitel 4.5. Die Halstead Metrik auf den gesamten Code anzuwenden anstatt den Vergleich von einem einzelnen View auf das gesamte Projekt zu übertragen wäre in weiteren Auswertungen möglich. Andere Berechnungen nach Halstead sind die des Levels und des Schwierigkeitsgrad der Programmiersprache (vgl. Halstead, 1977). Ein höheres Level bedeutet ein höheres Abstraktionslevel der Sprache und somit ein geringeres Fehlerpotenzial (vgl. Halstead, 1977).

$$L = 2/n1 + n2/N2$$

	UIKit	**SwiftUI**
Programmlevel L	0.40	0.56

Tabelle 4.10: Programmlevel der Button-Views nach Halstead

Es spiegelt sich passend zu den gesammelten Erkenntnisse vorheriger Kapitel wieder, dass SwiftUI den Entwicklern durch den deklarativen Syntax Aufgaben abnimmt und das Abstraktionslevel somit hebt.
Der Schwierigkeitsgrad wird mit einer leicht veränderten Formel berechnet und beschreibt die Anfälligkeit für Fehler.

$$D = n1/2 + n2/N2$$

	UIKit	SwiftUI
Programm Difficulty D	2.48	1.79

Tabelle 4.11: Programm Difficulty der Button-Views nach Halstead

Dieser wird durch weniger unterschiedliche Operanden und Operatoren und die kleinere Vokabelgröße bei SwiftUI gesenkt.
Die letzte relevante Berechnung ist die der Entwicklungszeit. Diese berechnet sich aus dem Quotienten des Programmvolumens V, des Programmlevels L und dem Produkt der Stroud Nummer S sowie des Umrechnungsfaktors f. Die Stroud Nummer gibt den Wert an, wie viele Entscheidungen ein Mensch im Durchschnitt pro Sekunde tätigen kann (vgl. Halstead, 1977). Der Wert ist für die Berechnung nach Halstead mit 18 festgelegt und wird mit dem Umrechnungsfaktor f, des Wertes 60, von Sekunden in Minuten gewandelt (Halstead, 1977).

$$T = V/L/(f * S)$$

	UIKit	SwiftUI
Development Time T	123.52	48.67

Tabelle 4.12: Development Time der Button-Views nach Halstead

Das Ergebnis dieser Rechnung ist für UIKit 123.52 und SwiftUI 48.67 Minuten. Hudson hat wie in Kapitel 3.2 dargelegt die Entwicklungszeit der beiden Apps gemessen, welche für UIKit ca 78 % länger war. Der Unterschied der Entwicklungszeit für den Button-View nach der Halstead Metrik beträgt jedoch 153.79 %, was den von Hudson gemessenen Wert deutlich überschreitet. Das Ergebnis dieses Wertes sollte somit kritisch betrachtet werden, ist dennoch eine interessante Erkenntnis. Abschließend lässt sich sagen, dass Halsteads Metrik für die Berechnung der Komplexität viele aufschlussreiche Ergebnisse erzielte. Durch die Berechnungen des Programmvolumens wurde eine Schwäche, das fehlende Einbeziehen der Wortlänge, deutlich. Dieser Aspekt konnte durch das in Kapitel 4.5.2 erstellte Script jedoch korrigiert werden. Die Berechnungen des Levels und der Schwierigkeit der Frameworks bestätigte die Analysen der obigen Kapitel, dass SwiftUIs Abstraktion den Schwierigkeitsgrad senkt.

4.7 App Größe

Ein weiterer Vergleichsaspekt zwischen den Apps ist die Größe dieser. Im Vergleich wird zwischen zwei Werten unterschieden. Die komprimierte App, welche aus dem AppStore heruntergeladen wird und die Größe der App, welche diese auf dem Gerät installiert einnimmt. Um

die Größe dieser Dateien zu minimieren, hat Apple die Funktion des App Slicing implementiert (vgl. Apple, 2015a). Dies ermöglicht, die Dateigröße der Apps für die verschiedenen Iphones zu verbessern. Dafür werden je nach Gerät unterschiedliche Dateien zum Download angeboten, die auf dieses zugeschnitten sind (vgl. Apple, 2015a). Es werden für dieses Gerät ungenutzte Dateien nicht mit heruntergeladen und somit weniger Speicherplatz eingenommen (vgl. Apple, 2015a). Diese Funktion wird für Apple Geräte, welche die IOS Version 9.0 oder eine neuere installiert haben, genutzt. Damit können alle iPhones, die seit 2011 veröffentlicht wurden, diese Funktionen nutzen (vgl. Apple, 2015b). Laut Apple haben 90 % der Nutzer Version 13.0 des IOS oder neuer auf ihren Endgeräten installiert, weswegen im Folgenden die Dateigrößen der Apps mit App Slicing betrachtet (vgl. Apple, 2020a). Apple stellt Informationen zu der geschätzten Größe je nach Gerät zur Verfügung. Diese wurden zusammengetragen und der Mittelwert als Vergleichswert verwendet. Diese Daten stellt Apple für Entwickler auf ihrer Plattform AppStore Connect, über welche die Apps für die Veröffentlichung vorbereitet und weitere Informationen für den Kontext und die Beschreibung dieser von Apple verlangt werden, zur Verfügung (vgl. Apple, 2019a).

Kategorie	**UIKit**	**SwiftUI**
Größe beim Download[1]	$\approx$ 297,5 KB	$\approx$ 272 KB
Größe auf dem Gerät[2]	$\approx$ 599 KB	$\approx$ 556 KB

Tabelle 4.13: Geschätzte Größen der Sustainable App nach Apple (siehe Anhang 5.1)

[1] "This is the compressed size of the app downloaded over the air."(Apple, 2019a)

[2] "This is the amount of disk space the app will take up on the customer's device."(Apple, 2019a)

Aus Apples Daten geht hervor, dass die UIKit App sowohl für den Download als auch installiert im Durchschnitt ca. 1 % größer als die SwiftUI App ist. Bei größeren Applikationen werden die Unterschiede wahrscheinlich bedeutender, dennoch ist die App Größe in diesem Projekt kein ausschlaggebender Vergleichspunkt.

4.8 Performanz

Wenn benötigte Funktionen in einem Framework nicht performant umsetzbar sind, kann dies die Wahl eines Frameworks beeinflussen. Um dies zu analysieren, werden in mehreren Tests die Ausführungszeiten und Prozessorleistungen gemessen und verglichen.

4.8.1 Technische Voraussetzungen

Alle folgenden Versuche wurden auf einem 15 Zoll Macbook Pro aus dem Jahre 2019 mit dem installierten Betriebssystem macOS Catalina der Version 10.15.7 durchgeführt. Dieses besitzt

einen 2,2 GHz 6-Core Intel Core i7 Prozessor, 16 GB Arbeitsspeicher und einen Speicher von 250 GB. Die beiden Apps wurden mit Apples Entwicklungsumgebung Xcode, Version 12.4, erstellt und evaluiert. Die Startzeiten der Apps wurden sowohl mit dem Simulator als auch auf einem Hardwaregerät gemessen. Als simuliertes Gerät wurde ein iPhone 11 Pro mit 4 GB Ram und Apples A13 Bionic Prozessor und als Hardware Gerät ein iPhone X mit 3 GB Ram und Apples A11 Bionic verwendet. Die Version 14.4 von Apples Betriebssystem war auf beiden installiert. Die Ergebnisse mit dem Simulator und dem Hardwaregerät waren identisch, weswegen diese zusammengeführt werden.
Vor der Versuchsdurchführung wurden alle Geräte auf 100 % Akkustand geladen, der Flugmodus aktiviert und lediglich das WLAN für den Datenempfang eingeschaltet. Alle Applikationen des Iphones und Macbooks, abgesehen von Xcode, wurden geschlossen und die Geräte neu gestartet.

4.8.2 Build Times

Die Zeiten, die für das Kompilieren des Codes benötigt werden, werden Build Times genannt. Hierbei wird zwischen einem Clean Build und einem Incremental Build unterschieden. Ein Incremental Build nutzt die vorherigen Builds zu seinem Vorteil und verwendet unveränderte Code-Abschnitte, ohne diese neu zu bauen. Je weniger Code verändert wird, desto schneller sind die Incremental Builds (vgl. Apple, 2018a). Von einem Clean Build wird gesprochen, wenn alle Informationen der vorherigen Builds gelöscht werden und dieser somit komplett neu erstellt wird (vgl. Apple, 2018a). Für dieses Experiment wurde der Programmcode zwischen den Incremental Builds nicht verändert, um ein konstantes und wiederholbares Ergebnis zu erzielen. Die Zeiten der Builds werden in Xcode angezeigt und ausgelesen.

Messung	**UIKit**	**SwiftUI**
1	$\approx$ 2.092s	$\approx$ 2.214s
2	$\approx$ 2.090s	$\approx$ 2.212s
3	$\approx$ 2.094s	$\approx$ 2.219s
4	$\approx$ 2.090s	$\approx$ 2.216s
Mittelwert	$\approx$ 2.092s	$\approx$ 2.215s

Tabelle 4.14: Build Zeiten für Clean Builds

Messung	UIKit	SwiftUI
1	$\approx$ 0.037s	$\approx$ 0.027s
2	$\approx$ 0.039s	$\approx$ 0.032s
3	$\approx$ 0.034s	$\approx$ 0.033s
4	$\approx$ 0.032s	$\approx$ 0.036s
Mittelwert	$\approx$ 0.036s	$\approx$ 0.032s

Tabelle 4.15: Build Zeiten für Incremental Builds

Während die Clean Builds in der UIKit App ca 6 % schneller sind, sind die Incremental Builds in SwiftUI ca. 12 % schneller. Bezüglich der Build Zeiten lässt sich somit kein bedeutender Unterschied zwischen den beiden Frameworks feststellen.

4.8.3 Launch Times

Eine weitere Messung, um die Apps miteinander zu vergleichen, ist die der Startzeiten. Dies bezeichnet die Zeit, welche die App benötigt, die internen Prozesse für den Start der App abzuwickeln und weitere Views zu erstellen (vgl. Apple, 2019e). Die Messungen der Prozessorauslastung und der Startzeit erfolgen mit Xcodes Applikation *Instruments* für *MacOS* (vgl. Apple, 2019c).

Messung	UIKit	SwiftUI
1	$\approx$ 0.44s	$\approx$ 0.58s
2	$\approx$ 0.47s	$\approx$ 0.55s
3	$\approx$ 0.44s	$\approx$ 0.57s
4	$\approx$ 0.40s	$\approx$ 0.56s
Mittelwert	$\approx$ 0.4375s	$\approx$ 0.565s

Tabelle 4.16: Messungen der Startzeiten

Die UIKit-App startet nach diesen Ergebnissen fast 29 % schneller als die SwiftUI-App. Aus welchen Gründen sich dieser Unterschied ergibt, konnte auch nach erheblichen Ermittlungen noch nicht erschlossen worden. Dies könnte in weiteren Untersuchungen genauer betrachtet werden.

4.8.4 Prozessorauslastung

Für das Starten der App wurde ebenfalls die Prozessorauslastung gemessen.

Messung	UIKit	SwiftUI
Höchstwert	260%	190%
Niedrig	10%	40%
Mittelwert	134%	104%
Abweichung	25.4104	28.0129

Tabelle 4.17: Prozessorauslastung während des Startens der Apps

Die Werte der sechs Kerne werden zusammengerechnet und können somit maximal 600 % erreichen. Die Prozessorauslastung der UIKit-App ist mit ca. 29 % höher. Dies ist jedoch kein aussagekräftiges Ergebnis, da der Start der App den Prozessor lediglich über eine halbe Sekunde auslastet. Somit ist UIKit, welches den Prozessor beim Start 29 % höher belastet 29 % schneller.
Für die Messung der Prozessorauslastung während der Nutzung der App wurde der Time Profiler des Programm Instruments genutzt (vgl. Apple, 2019c). Die Versuchsdurchführung besteht aus einer Aneinanderreihung von Interaktionen mit der App, um die intendierte Nutzung zu simulieren (siehe Anhang 5.14). Die Ergebnisse dieser Messungen wurden im Anhang beigefügt (siehe Anhang 5.2). Die Daten lassen sich jedoch nicht für die Evaluierung in dieser Arbeit nutzen, da die Instruments App über keine Funktionen verfügt, die gemessenen Daten weiter zu analysieren und auszuwerten. Die Messungen der Prozessorauslastung während des Startens der App umfasste ungefähr 23 Unterpunkte, welche manuell gezählt wurden. Die Daten der simulierten Nutzung sind sowohl mit ca. 25 Sekunden über einen 50-Mal längeren Zeitraum als auch in kleineren Staffelungen gemessen worden, wodurch das manuelle Zählen nicht mehr möglich ist. Da das Exportieren der Daten von Apple ebenfalls nicht unterstützt wird, lassen sich die Messungen nicht weiter evaluieren als die Instruments App es ermöglicht. Es lassen sich bei dem Vergleich der jeweils vier Messungen jedoch oberflächlich keine relevanten Unterschiede der Performance der beiden Frameworks erkennen.

4.9 Funktionalität

Die Funktionalität alleine kann über die Wahl eines Frameworks entscheiden, wenn benötigte Funktionen in diesem nicht umsetzbar sind. UIKit wurde zusammen mit dem damals noch iPhoneOS genannten Betriebssystem von Apple am 6 März 2008 veröffentlicht und existiert somit schon seit nahezu 13 Jahren. Die Reife von UIKit wird im Bereich der Funktionalität deutlich. Wie in Kapitel 3.1 beschrieben kann zum Zeitpunkt des Schreibens dieser Arbeit bei

der Entwicklung einer SwiftUI-App weiterhin leicht an seine Grenzen gestoßen werden. Für die Programmierung der Sustainable App sollte ein Scrollview mit der Funktion des paging und des automatischen Aktualisierens von diesem beim Auswählen eines Filters erstellt werden. Diese Funktionen sind mit SwiftUI nicht oder nur über Umwege umsetzbar und zeigen beispielhaft die noch fehlenden Funktionen von SwiftUI. Um den Vergleich der Funktionalität wissenschaftlich durchzuführen, sollten alle Views und Funktionen in UIKit, sowie die entsprechenden Gegenstücke SwiftUIs aufgelistet werden. Das ist sowohl aufgrund des Umfangs dieser Arbeit als auch wegen der unterschiedlichen Grundkonzepte nicht möglich. Der abweichende Umfang der Übersichtsseiten "Views und Kontrollelemente" beider Frameworks gibt jedoch einen Einblick in die Differenzen (vgl. Apple, 2017b; Apple, 2020g). Für UIKit werden auf dieser Seite 41 Views mit weiteren Subviews aufgelistet während in SwiftUI auf dieser Seite lediglich 22 dokumentiert werden (vgl. Apple, 2017b; Apple, 2020g). In diesem Bereich hat UIKit, da es schon seit 2008 existiert, einen deutlichen Vorsprung. Apple hat seit der Veröffentlichung in 2019, in 2020 jedoch viele Funktionen für SwiftUI hinzugefügt und weitere Updates sind zu erwarten (vgl. Apple, 2020j).

4.9.1 Zusammenspiel von SwiftUI und UIKit

Um die Funktionalitäten der beiden Frameworks zu erweitern, hat Apple es ermöglicht die beiden Frameworks in einem Zusammenspiel zu nutzen. Ein UIKit Viewcontroller lässt sich in SwiftUI nutzen, indem dieser in einen View-Struct des Types `UIViewControllerRepresentable` gesetzt wird:

```
struct IntegratedViewContoller: UIViewControllerRepresentable {
    func makeUIViewController (context:
    ↪  UIViewControllerRepresentableContext<IntegratedViewContoller>) ->
    ↪  ViewController {
        return ViewController()
    }

    func updateUIViewController(_ uiViewController: ViewController, context:
    ↪  UIViewControllerRepresentableContext<IntegratedViewContoller>) {

    }
}
```

Code-Ausschnitt 4.24: Einbinden eines UIKit-Viewscontrollers in einen SwiftUI View

Dieser ermöglicht es den Viewcontroller nun wie einen View im Code zu verwenden.

Um SwiftUI-Views in UIKit zu verwenden, müssen sie lediglich in einen `UIHostingController` gewrappt und wie ein Viewcontroller in die Applikation eingebaut werden:

```
let myViewController = UIHostingController(rootView: MySwiftUIView())
```

Code-Ausschnitt 4.25: Einbinden eines SwiftUI-Views in einen UIKit-Viewcontroller

Somit lassen sich Applikationen in einem Zusammenspiel aus beiden Frameworks kreieren und der Entwickler kann somit die besten Funktionen beider wählen.

4.9.2 Live Previews

Eine Funktion, die mit SwiftUI veröffentlicht wurde und den Entwicklungsprozess verschnellert, ist das *Live Preview*. Dadurch lassen sich Veränderungen des SwiftUI Codes unmittelbar in einer Vorschau in Xcode erkennen. Diese Funktion macht es obsolet, die App bei kleinen Veränderungen neu kompilieren und starten zu müssen. Mit Live Previews können verschiedene Gerätetypen mit unterschiedlichen Einstellungen der Schriftgrößen, Größen der Views, Farbschemen und weitere Einstellungen gleichzeitig angezeigt und simuliert werden (vgl. Apple, 2019h). Für die Entwicklung von Benutzeroberflächen kann dies bedeutend Zeit einsparen.

```
struct MyViewPreview: PreviewProvider {
    static var previews: some View {
        MyView()
            .previewDevice(.init(stringLiteral: "iPhone 12"))
            .environment(\.colorScheme, .dark)
    }
}
```

Code-Ausschnitt 4.26: Erstellen eines Live Previes in SwiftUI

Dies kann als großer Vorteil von SwiftUI über UIKit gesehen werden. Live Previews sind über die in Kapitel 4.9.1 beschriebene Methode jedoch ebenfalls in UIKit verfügbar. Somit lässt sich eine sehr nützliche Funktion von SwiftUI ebenfalls mit UIKit nutzen und die Entwicklung von Benutzeroberflächen erleichtern.
Wie zu erkennen ist, ist ein Zusammenspiel der beiden Frameworks umsetzbar, jedoch sollte Apple SwiftUIs Funktionalität weiter ausbauen um in dieser Hinsicht ebenfalls mit UIKit konkurrieren zu können.

5 Fazit

Die vorherigen Kapitel haben die Unterschiede zwischen UIKit und SwiftUI anhand zweier Apps bezüglich Umfang, Komplexität, Performanz, Funktionalität und verschiedenen Grundkonzepte aufgelistet. Im Folgenden werden die Ergebnisse zusammengeführt, rückblickend betrachtet und bewertet.

5.1 Diskussion der Ergebnisse

Ziel dieser Arbeit war es, die Unterschiede von UIKit und SwiftUI zu ermitteln. Die Ergebnisse des Vergleichs in Kapitels 4 bilden sich sowohl aus den abgewandelten Versionen der *Sustainable App*, den Applikationen von Hudson als auch der weiteren Erforschungen der beiden Frameworks. Hierbei trugen Hudsons Einblicke zur Validierung der Messungen bei. Nichtsdestotrotz gingen viele Unterschiede schon aus der Programmierung der Apps hervor und zeigten sich in vielerlei Hinsicht, wie in Kapitel 4 zu erkennen ist. Diese resultieren zum Großteil aus den unterschiedlichen Grundkonzepten. UIKits imperatives Programmierparadigma ermöglicht mehr Kontrolle über die Datenflüsse und Programmabläufe, während SwiftUIs deklarativer Ansatz dem Entwickler viele Aufgaben abnimmt. SwiftUI wickelt Funktionen wie das Erstellen der Views, Verwalten von Variablen und Benutzerinteraktionen automatisch ab. Zusammen angeordnete Views müssen nicht imperativ in einen gemeinsamen View gesetzt werden, da das Framework dies mit Hilfe des View Builders und gruppierenden Views implementiert. Aufgrund dieser Unterschiede ist für die Entwicklung einer Applikation in SwiftUI weniger Code und Aufwand nötig. Nicht nur besteht die Abwandlung der *Sustainable App* in UIKit aus 73 % mehr Zeilen Code, die Zeilen sind ebenfalls im Durchschnitt 65 % länger. Somit umfasst diese App ca. 182 % Zeichen mehr als die SwiftUI-App. Die ermittelte Komplexität des Codes nach Halteads Metrik validiert und unterstreicht diese Berechnungen.
Die Ergebnisse bestätigen den von Apple formulierten Anspruch, Apps mit SwiftUI schneller als je zuvor entwickeln zu können (siehe Zitat Kapitel 1). Die Programmierung einer IOS-Applikation erfolgt mit SwiftUI Halsteads Metrik und den Analysen der Lines of Code tatsächlich deutlich schneller. Die Unterschiede in der Performanz hingegen sind nicht so bedeutend wie erwartet und spielen mit abwechselnden Differenzen der Build Zeiten von maximal 12 % keine große Rolle. Ebenso wenig aussagekräftig waren die Diskrepanzen der Prozessorauslastung während der Startzeiten und der Simulationen der Nutzung der beiden Frameworks.
Bezugnehmend auf die Forschungsfrage lässt sich sagen, dass SwiftUI in vielen bereits erwähnten Hinsichten effizienter als der alte Standard UIKit ist. Allerdings sollte hierbei die weniger ausgeprägte Funktionalität von SwiftUI nicht außer Acht gelassen werden. Für viele einfache Anwendungen ist SwiftUI aufgrund des geringeren Entwicklungsaufwands das effizientere

Framework, jedoch kann bei größeren Applikationen schnell an die (noch) vorhandenen Grenzen von diesem gestoßen werden. In diesem Fall muss auf UIKit zurückgegriffen. Es können entweder Ausschnitte einer App mit diesem umgesetzt oder die Frameworks in einem Zusammenspiel genutzt werden.
Zusammenfassend lässt sich sagen, dass sich die Unterschiede der beiden Frameworks im Umfang des Codes, dem Entwicklungsaufwand, der Funktionalität und weniger in der Performanz und der Größe der Apps zeigen. Damit haben sowohl UIKit als auch SwiftUI einen Platz in der aktuellen Zeit.

5.2 Risiken der Validität und Reflexion

Das größte Risiko dieser Arbeit ist die Differenz der Vorkenntnisse. Die im Modul Praxisprojekt entstandene *Sustainable App* wurde zu einem Großteil mit SwiftUI entwickelt. UIKit wurde lediglich für das Umsetzen von in SwiftUI noch nicht bereitstehenden Funktionen verwendet, weswegen die Entwicklung der abgewandelten *Sustainable App* mit UIKit für diese Arbeit deutlich mehr Zeit in Anspruch nahm. Die Konzepte von UIKit mussten erst eingehend erlernt werden. Die Gefahr war, dass der Code dieser Applikation weniger komprimiert und optimiert sein würde, als der SwiftUI-Code. Dieses Risiko wurde jedoch schon während der Entwicklung aktiv versucht zu minimieren. Durch mehrere Revisionen wurde probiert, den Code der App auf das selbige Level zu heben und Redundanz zu vermeiden. Ob dies in dem erforderlichen Ausmaß gelungen ist, lässt sich wohl erst in der Zukunft mit weiteren Kenntnissen der beiden Frameworks rückblickend bewerten.

5.3 Ausblick

In der weiteren Forschung bietet sich eine Messung der Performanz und Kompilierzeiten mit einem der von Apple in 2020 veröffentlichten *Macs* an. Diese verfügen erstmals über einen selbst entwickelten Desktop Prozessor, welcher IOS-Applikationen mit Xcode laut Apple fast dreimal so schnell kompilieren kann (vgl. Apple, 2020b).
Abschließend lässt sich sagen, dass die Adoption von SwiftUI nach den gewonnenen Erkenntnissen relativ hoch zu erwarten ist. Es ist dem alten Standard UIKit in vielen Hinsichten überlegen und der Fakt, dass bereits 90 % aller IOS-Geräte SwiftUI-Apps unterstützen, bietet großes Potenzial. Wird die Funktionalität des Frameworks in Zukunft weiter ausgebaut, kann SwiftUI als Vorreiter der IOS-Entwicklung erwartet werden.

Literaturverzeichnis

[AlDanial 2015] ALDANIAL: *cloc Bibliothek*. https://github.com/AlDanial/cloc. Version: 2015, Abruf: 07.02.2021

[Apple 2009a] APPLE: *UIKit Framework*. https://developer.apple.com/documentation/uikit. Version: 2009, Abruf: 07.02.2021

[Apple 2009b] APPLE: *UIView Dokumentation*. https://developer.apple.com/documentation/uikit/uiview. Version: 2009, Abruf: 07.02.2021

[Apple 2013] APPLE: *What is Cocoa*. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html. Version: 2013, Abruf: 07.02.2021

[Apple 2014a] APPLE: *Inheritance (Vererbung)*. https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html. Version: 2014, Abruf: 07.02.2021

[Apple 2014b] APPLE: *Interface Builder*. https://developer.apple.com/xcode/interface-builder/. Version: 2014, Abruf: 07.02.2021

[Apple 2014c] APPLE: *Migrating Your Objective-C Code to Swift*. https://developer.apple.com/documentation/swift/migrating_your_objective-c_code_to_swift. Version: 2014, Abruf: 15.01.2021

[Apple 2014d] APPLE: *Subscripts*. https://docs.swift.org/swift-book/LanguageGuide/Subscripts.html. Version: 2014, Abruf: 07.02.2021

[Apple 2014e] APPLE: *Swift Basic Operators*. https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html. Version: 2014, Abruf: 07.02.2021

[Apple 2014f] APPLE: *Swift Properties*. https://docs.swift.org/swift-book/LanguageGuide/Properties.html#//apple_ref/doc/uid/TP40014097-CH14-ID262. Version: 2014, Abruf: 07.02.2021

[Apple 2014g] APPLE: *WWDC 2014 Livestream*. https://www.youtube.com/watch?v=w87fOAG8fjk. Version: 2014, Abruf: 07.02.2021

[Apple 2015a] APPLE: *App Thinning in Xcode*. https://developer.apple.com/videos/play/wwdc2015/404/. Version: 2015, Abruf: 07.02.2021

[Apple 2015b] APPLE: *IOS 9 Preview*. https://www.apple.com/au/ios/ios9-preview/. Version: 2015, Abruf: 07.02.2021

[Apple 2015c] APPLE: *NSLayoutConstraint Dokumentation*. https://developer.apple.com/documentation/uikit/nslayoutconstraint. Version: 2015, Abruf: 07.02.2021

[Apple 2015d] APPLE: *Swift 2.0*. https://developer.apple.com/swift/blog/?id=29. Version: 2015, Abruf: 07.02.2021

[Apple 2016a] APPLE: *Alerts in Swift*. https://developer.apple.com/design/human-interface-guidelines/ios/views/alerts/. Version: 2016, Abruf: 07.02.2021

[Apple 2016b] APPLE: *Optionals*. https://developer.apple.com/documentation/swift/optional. Version: 2016, Abruf: 07.02.2021

[Apple 2016c] APPLE: *Understanding Auto Layout*. https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html. Version: 2016, Abruf: 07.02.2021

[Apple 2017a] APPLE: *About App Development with UIKit*. https://developer.apple.com/documentation/uikit/about_app_development_with_uikit. Version: 2017, Abruf: 07.02.2021

[Apple 2017b] APPLE: *UIKit - Views and Controls*. https://developer.apple.com/documentation/uikit/views_and_controls. Version: 2017, Abruf: 07.02.2021

[Apple 2018a] APPLE: *Behind the Scenes of the Xcode Build Process*. https://developer.apple.com/videos/play/wwdc2018/415/. Version: 2018, Abruf: 07.02.2021

[Apple 2018b] APPLE: *Classes and Structures*. https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html. Version: 2018, Abruf: 07.02.2021

[Apple 2018c] APPLE: *Managing a Shared Resource Using a Singleton*. https://developer.apple.com/documentation/swift/cocoa_design_patterns/managing_a_shared_resource_using_a_singleton. Version: 2018, Abruf: 07.02.2021

[Apple 2018d] APPLE: *Model-View-Controller Dokumentation*. https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. Version: 2018, Abruf: 07.02.2021

[Apple 2018e] APPLE: *Using Key-Value Observing in Swift*. https://developer.apple.com/documentation/swift/cocoa_design_patterns/using_key-value_observing_in_swift. Version: 2018, Abruf: 07.02.2021

[Apple 2019a] APPLE: *App Store Connect*. https://developer.apple.com/de/support/app-store-connect/. Version: 2019, Abruf: 07.02.2021

[Apple 2019b] APPLE: *DynamicProperty Dokumentation*. https://developer.apple.com/documentation/swiftui/dynamicproperty. Version: 2019, Abruf: 07.02.2021

[Apple 2019c] APPLE: *Getting Started with Instruments*. https://developer.apple.com/videos/play/wwdc2019/411/. Version: 2019, Abruf: 07.02.2021

[Apple 2019d] APPLE: *ObservableObject Protokoll*. https://developer.apple.com/documentation/combine/observableobject. Version: 2019, Abruf: 07.02.2021

[Apple 2019e] APPLE: *Reducing Your App's Launch Time*. https://developer.apple.com/documentation/xcode/improving_your_app_s_performance/reducing_your_app_s_launch_time. Version: 2019, Abruf: 07.02.2021

[Apple 2019f] APPLE: *SwiftUI*. https://developer.apple.com/xcode/swiftui/. Version: 2019, Abruf: 07.02.2021

[Apple 2019g] APPLE: *SwiftUI Alignment*. https://developer.apple.com/documentation/swiftui/alignment. Version: 2019, Abruf: 07.02.2021

[Apple 2019h] APPLE: *SwiftUI Previews*. https://developer.apple.com/documentation/swiftui/previews. Version: 2019, Abruf: 07.02.2021

[Apple 2019i] APPLE: *SwiftUI Tutorial*. https://developer.apple.com/tutorials/SwiftUI. Version: 2019, Abruf: 07.02.2021

[Apple 2019j] APPLE: *SwiftUI ViewBuilder*. https://developer.apple.com/documentation/swiftui/viewbuilder. Version: 2019, Abruf: 07.02.2021

[Apple 2019k] APPLE: *ViewModifier Protokoll*. https://developer.apple.com/documentation/swiftui/viewmodifier. Version: 2019, Abruf: 07.02.2021

[Apple 2020a] APPLE: *App Store Information*. https://developer.apple.com/support/app-store. Version: 2020, Abruf: 07.02.2021

[Apple 2020b] APPLE: *Apple Event 2020*. https://www.apple.com/de/apple-events/november-2020/. Version: 2020, Abruf: 07.02.2021

[Apple 2020c] APPLE: *EnvironmentObject Property Wrapper*. https://developer.apple.com/documentation/swiftui/environmentobject. Version: 2020, Abruf: 07.02.2021

[Apple 2020d] APPLE: *Published Property Wrapper*. https://developer.apple.com/documentation/combine/published. Version: 2020, Abruf: 07.02.2021

[Apple 2020e] APPLE: *State and Data Flow Dokumentation*. https://developer.apple.com/documentation/swiftui/state-and-data-flow. Version: 2020, Abruf: 07.02.2021

[Apple 2020f] APPLE: *StateObject Property Wrapper*. https://developer.apple.com/documentation/swiftui/stateobject. Version: 2020, Abruf: 07.02.2021

[Apple 2020g] APPLE: *SwiftUI - Views and Controls*. https://developer.apple.com/documentation/swiftui/views-and-controls. Version: 2020, Abruf: 07.02.2021

[Apple 2020h] APPLE: *SwiftUI Binding Dokumentation*. https://developer.apple.com/documentation/swiftui/binding. Version: 2020, Abruf: 07.02.2021

[Apple 2020i] APPLE: *SwiftUI State Dokumentation*. https://developer.apple.com/documentation/swiftui/state. Version: 2020, Abruf: 07.02.2021

[Apple 2020j] APPLE: *What's new in SwiftUI*. https://developer.apple.com/videos/play/wwdc2020/10041/. Version: 2020, Abruf: 07.02.2021

[Boyter 2020] BOYTER, Ben: *scc Bibliothek*. https://github.com/boyter/scc. Version: 2020, Abruf: 07.02.2021

[Coleman u. Liau 1975] COLEMAN, Meri ; LIAU, T. L.: A computer readability formula designed for machine scoring. In: *Journal of Applied Psychology* 60 (1975), S. 283–284

[Flesch u. Gould 1949] FLESCH, Rudolf ; GOULD, Alan J.: *The art of readable writing*. Bd. 8. Harper New York, 1949

[González García u. a. 2015] GONZÁLEZ GARCÍA, Cristian ; ESPADA, Jordán ; PELAYO GARCÍA-BUSTELO, B. ; CUEVA LOVELLE, Juan: Swift vs. Objective-C: A New Programming Language. In: *International Journal of Artificial Intelligence and Interactive Multimedia* 3 (2015), 01, S. 74–81. http://dx.doi.org/10.9781/ijimai.2015.3310. – DOI 10.9781/ijimai.2015.3310

[Google 2021] GOOGLE: *UIKit vs. SwiftUI*. https://trends.google.com/trends/explore?date=2018-02-01%202021-02-02&q=UIKit,SwiftUI. Version: 2021, Abruf: 09.01.2021

[Gunning u. a. 1952] GUNNING, Robert u. a.: Technique of clear writing. (1952)

[Halstead 1977] HALSTEAD, Maurice H.: Elements of software science. (1977)

[Hudson 2020] HUDSON, Paul: *Reimplementing SwiftUI's environment for UIKit*. https://www.youtube.com/watch?v=UUGk2HPbtMg. Version: 2020, Abruf: 07.02.2021

[Husdon 2019] HUSDON, Paul: *UIKit vs SwiftUI App Comparison*. https://www.youtube.com/watch?v=qk2y-TiLDZo. Version: 2019, Abruf: 07.02.2021

[McCabe 1976] MCCABE, T. J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), Nr. 4, S. 308–320. http://dx.doi.org/10.1109/TSE.1976.233837. – DOI 10.1109/TSE.1976.233837

[McCall 2019] MCCALL, John: *Function builders (draft proposal)*. https://github.com/apple/swift-evolution/blob/9992cf3c11c2d5e0ea20bee98657d93902d5b174/proposals/XXXX-function-builders.md. Version: 2019, Abruf: 07.02.2021

[OkCupid 2020] OKCUPID: *Login App (UIKit and SwiftUI)*. https://github.com/OkCupid/LoginSample. Version: 2020, Abruf: 07.02.2021

[Overflow 2015] OVERFLOW, Stack: *2015 Developer Survey*. https://insights.stackoverflow.com/survey/2015. Version: 2015, Abruf: 07.02.2021

[Overflow 2016] OVERFLOW, Stack: *2016 Developer Survey*. https://insights.stackoverflow.com/survey/2016. Version: 2016, Abruf: 07.02.2021

[Overflow 2017] OVERFLOW, Stack: *2017 Developer Survey*. https://insights.stackoverflow.com/survey/2017. Version: 2017, Abruf: 07.02.2021

[Overflow 2018] OVERFLOW, Stack: *2018 Developer Survey*. https://insights.stackoverflow.com/survey/2018. Version: 2018, Abruf: 07.02.2021

[Overflow 2019] OVERFLOW, Stack: *2019 Developer Survey*. https://insights.stackoverflow.com/survey/2019. Version: 2019, Abruf: 07.02.2021

[Overflow 2020] OVERFLOW, Stack: *2020 Developer Survey*. https://insights.stackoverflow.com/survey/2020. Version: 2020, Abruf: 07.02.2021

[Schmalbach 2020] SCHMALBACH, Bastian: *Sustainable App Code*. https://github.com/bschmalb/praxisProjekt. Version: 2020, Abruf: 07.02.2021

[Schmalbach 2021] SCHMALBACH, Bastian: *UIKit vs SwiftUI App Code*. https://github.com/bschmalb/UIKitvsSwiftUI. Version: 2021, Abruf: 07.02.2021

Anhang

Estimated App Store file sizes for Build 1.0 (1)

DEVICE TYPE	DOWNLOAD SIZE	INSTALL SIZE
Universal	611 KB	847 KB
iPhone XS	314 KB	617 KB
iPhone XS Max	314 KB	617 KB
iPhone 12 Pro Max	314 KB	617 KB
iPhone SE (1st generation)	281 KB	581 KB
iPhone 12 Pro	314 KB	617 KB
iPhone 6S	281 KB	581 KB
iPhone XS Max	314 KB	617 KB
iPhone 12	314 KB	617 KB
iPhone 12 mini	314 KB	617 KB
iPhone 6S Plus	314 KB	617 KB
iPad Pro (12.9-inch) (4th generation) Wi-Fi + Cellular	306 KB	609 KB
iPad Pro (12.9-inch) (4th generation)	306 KB	609 KB
iPad Pro (11-inch) (2nd generation) Wi-Fi + Cellular	306 KB	609 KB
iPhone XR	295 KB	597 KB

Done

Estimated App Store file sizes for Build 1.0 (1)

DEVICE TYPE	DOWNLOAD SIZE	INSTALL SIZE
Universal	459 KB	667 KB
iPhone XS	279 KB	564 KB
iPhone XS Max	279 KB	564 KB
iPhone 12 Pro Max	279 KB	564 KB
iPhone SE (1st generation)	265 KB	548 KB
iPhone 12 Pro	279 KB	564 KB
iPhone 6S	265 KB	548 KB
iPhone XS Max	279 KB	564 KB
iPhone 12	279 KB	564 KB
iPhone 12 mini	279 KB	564 KB
iPhone 6S Plus	279 KB	564 KB
iPad Pro (12.9-inch) (4th generation) Wi-Fi + Cellular	276 KB	560 KB
iPad Pro (12.9-inch) (4th generation)	276 KB	560 KB
iPad Pro (11-inch) (2nd generation) Wi-Fi + Cellular	276 KB	560 KB
iPhone XR	271 KB	556 KB

Done

Abbildung 5.1: Geschätzte Größen der UIKit-App und SwiftUI-App

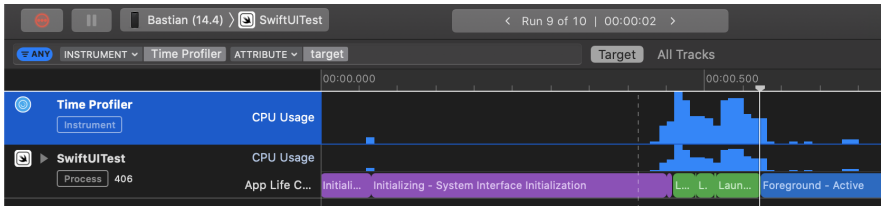

Abbildung 5.2: UIKit Launch Profiler Messung 1

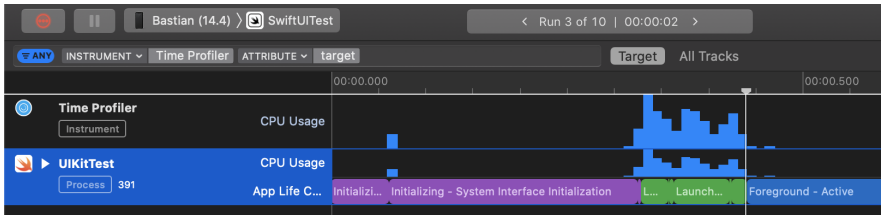

Abbildung 5.3: UIKit Launch Profiler Messung 2

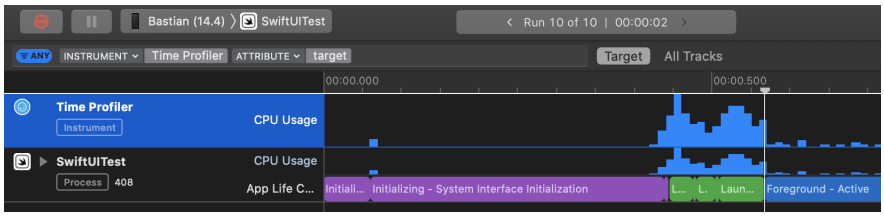

Abbildung 5.4: UIKit Launch Profiler Messung 3

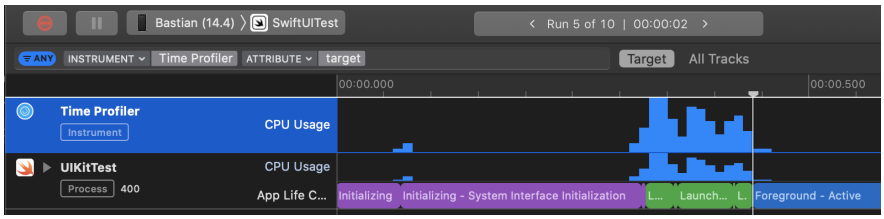

Abbildung 5.5: UIKit Launch Profiler Messung 4

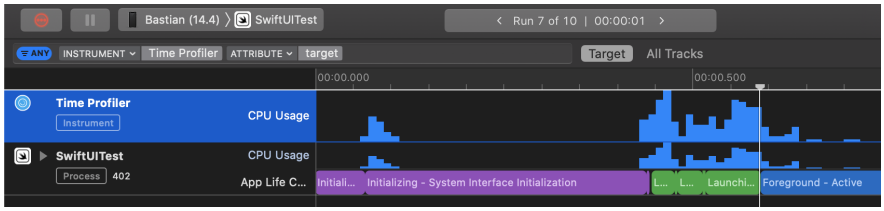

Abbildung 5.6: SwiftUI Launch Profiler Messung 1

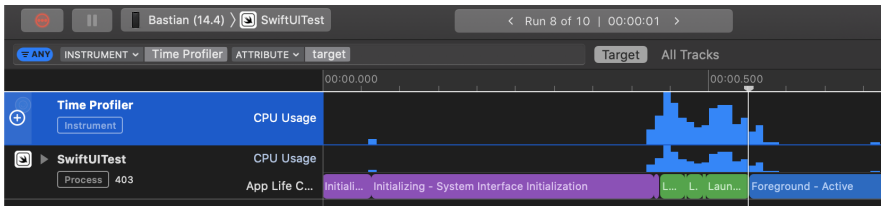

Abbildung 5.7: SwiftUI Launch Profiler Messung 2

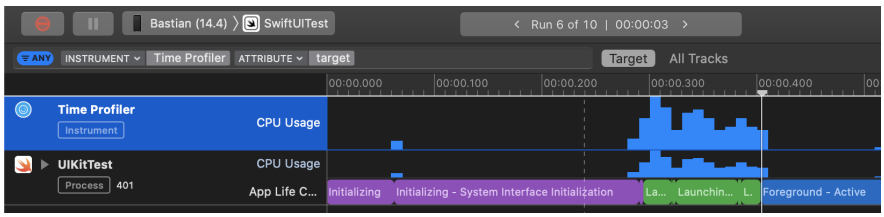

Abbildung 5.8: SwiftUI Launch Profiler Messung 3

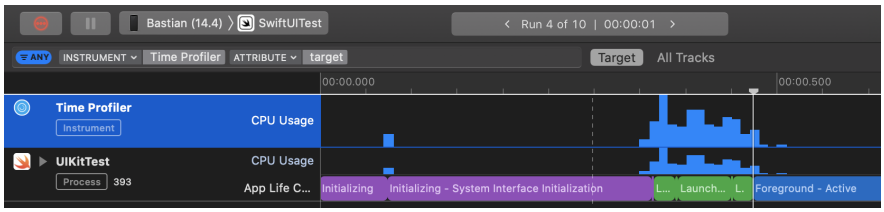

Abbildung 5.9: SwiftUI Launch Profiler Messung 4

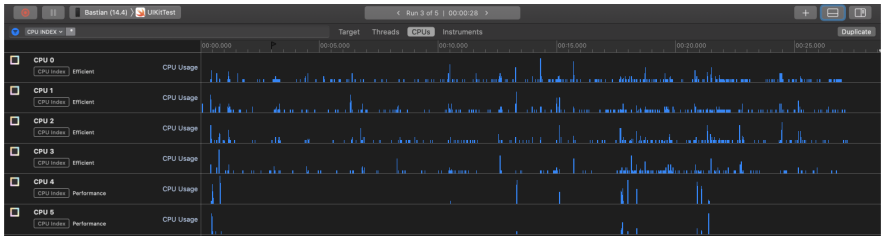

Abbildung 5.10: UIKit Performance Messung

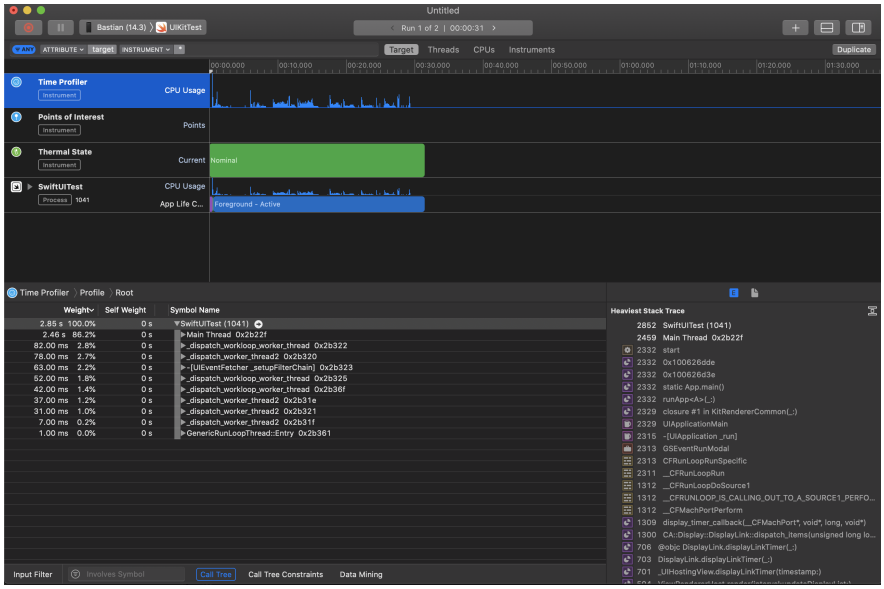

Abbildung 5.11: UIKit Performance Messung mit Prozessen

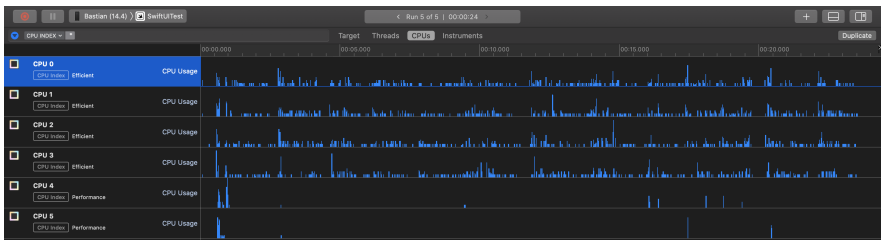

Abbildung 5.12: SwiftUI Performance Messung

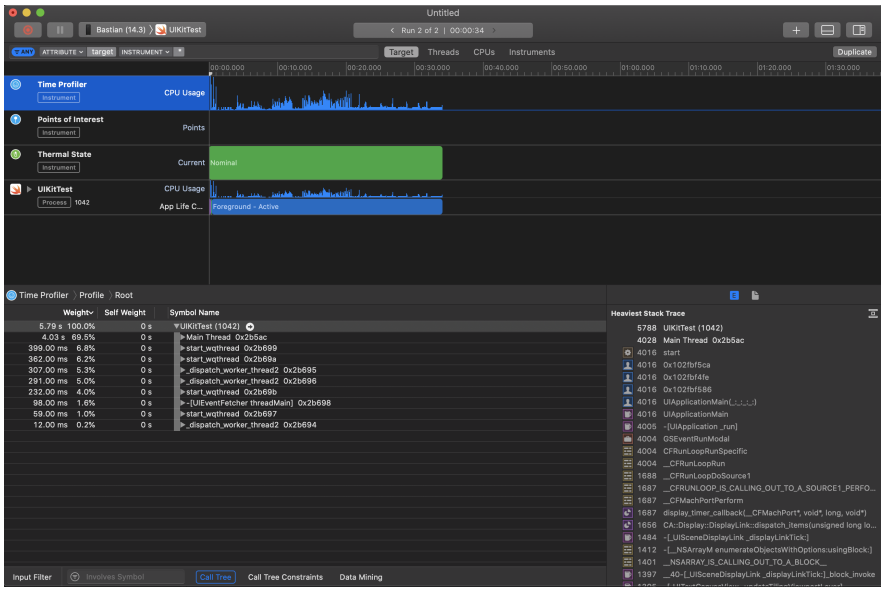

Abbildung 5.13: SwiftUI Performance Messung mit Prozessen

Versuchsdurchführung für die Messung der Prozessorauslastung:

1. Launch App
2. Tipp abhaken
3. Tipp abhaken Rückgängig
4. Optionen klicken, dass Card slided
5. Card wieder schließen
6. Filter Scrollen
7. Filter abwählen
8. Filter Auswählen
9. Tipps 3 Positionen Scrollen
10. Dann nochmals
11. Optionen klicken, dass Card slided
12. Card wieder schließen

Abbildung 5.14: Versuchsdurchführung für die Messung der Prozessorauslastung