

WILHELM BÜCHNER HOCHSCHULE

MASTERTHESIS

---

**Realisierung eines Source-to-Source Compilers  
zwischen Xamarin.Forms und Flutter zur  
automatisierten Transformation bestehender mobiler  
Anwendungen**

---

*Author:*

Julian Pasqué

*Betreuer:*

Dr. Thomas Kalbe

Verteilte und mobile Anwendungen

Fachbereich Informatik

Matrikelnummer: 902953

17. April 2021

# Zusammenfassung

# Abstract

# Inhaltsverzeichnis

Inhaltsverzeichnis	III
Abbildungsverzeichnis	VI
Tabellenverzeichnis	VII
Abkürzungsverzeichnis	VIII
Quellcodeverzeichnis	IX
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Ziel der Arbeit . . . . .	2
1.3 Gliederung . . . . .	3
<b>2 Compiler</b>	<b>4</b>
2.1 Grundbegriffe . . . . .	5
2.2 Compiler Struktur . . . . .	6
2.3 Lexikalische Analyse . . . . .	7
2.4 Syntaxanalyse . . . . .	8
2.5 Semantische Analyse . . . . .	9
2.6 Zwischencodeerzeugung . . . . .	10
2.7 Codeoptimierung . . . . .	10
2.8 Codeerzeugung . . . . .	11
2.9 Der .NET Compiler Roslyn . . . . .	11
<b>3 Compiler Spezifikation</b>	<b>12</b>
3.1 Funktionseingrenzung . . . . .	13
3.2 Übersetzung von verschiedenen Dateien . . . . .	14
3.3 Informationsfluss . . . . .	15
3.4 Grafische Darstellung . . . . .	16
3.5 Quelltext Optimierung . . . . .	16
<b>4 Technische Unterschiede zwischen Xamarin.Forms und Flutter</b>	<b>18</b>
4.1 Projektaufbau . . . . .	18
4.1.1 Metadaten . . . . .	18
4.1.2 Bilder und Startbildschirm . . . . .	19
4.1.3 Benutzerdefinierte Schriftarten . . . . .	20
4.1.4 Plattformspezifischer Quelltext . . . . .	20

4.2	Erweiterungen . . . . .	21
4.2.1	Interaktion mit der Hardware . . . . .	22
4.2.2	Speicherung von Daten . . . . .	22
4.2.3	Navigation zu anderen Anwendungen . . . . .	23
4.3	Lebenszyklus . . . . .	23
4.4	Ansichten . . . . .	24
4.4.1	Layouts . . . . .	24
4.4.2	Steuerelemente . . . . .	28
4.4.3	Ausrichtung von Steuerelementen . . . . .	35
4.4.4	Gesten . . . . .	36
4.4.5	Animationen . . . . .	36
4.4.6	Übersetzungsbeispiel . . . . .	37
<b>5</b>	<b>Unterschiede zwischen C# und Dart</b>	<b>40</b>
5.1	Klassendesign . . . . .	40
5.1.1	Referenz- und Wertetypen . . . . .	40
5.1.2	Datentypen . . . . .	42
5.1.3	Modifizierer . . . . .	45
5.1.4	Vererbung . . . . .	45
5.1.5	Übersetzungsbeispiel . . . . .	47
5.2	Namespaces . . . . .	48
5.3	Generische Typen . . . . .	49
5.4	Integrierte Verweistypen . . . . .	50
5.5	Asynchronität und Parallelität . . . . .	50
5.6	Bibliotheken . . . . .	51
5.6.1	Netzwerkaufrufe . . . . .	51
5.6.2	Kodierung und Dekodierung von Daten . . . . .	52
5.7	Ereignisse . . . . .	52
<b>6</b>	<b>Realisierung des Source-To-Source Compilers</b>	<b>54</b>
6.1	Programmablauf . . . . .	54
6.2	Metadaten . . . . .	56
6.2.1	Android Metadaten . . . . .	56
6.2.2	iOS Metadaten . . . . .	57
6.3	Ressourcen . . . . .	58
6.4	Übersetzung von Klassenstrukturen . . . . .	59
6.5	Übersetzung von Ansichten . . . . .	63
6.5.1	Visuellen- Zwischendarstellung . . . . .	63
6.5.2	Logische- Zwischendarstellung . . . . .	66
6.5.3	Visuelle Synthese . . . . .	67
6.6	Synthese . . . . .	67

---

6.7	Codeoptimierung . . . . .	68
6.8	Grafische Benutzeroberfläche . . . . .	68
<b>7</b>	<b>Qualitätssicherung</b>	<b>70</b>
7.1	Testobjekt . . . . .	70
7.2	Testfälle . . . . .	72
7.3	Testablauf . . . . .	73
<b>8</b>	<b>Fazit und Ausblick</b>	<b>75</b>
8.1	Open Source Veröffentlichung . . . . .	75
8.2	Beantwortung der Forschungsfrage . . . . .	75
	<b>Literaturverzeichnis</b>	<b>XI</b>
	<b>Anhang I: Gegenüberstellung von visuellen Elementen</b>	<b>XVIII</b>
	<b>Anhang II: Optimierte Flutter-LoginPage</b>	<b>XX</b>
	<b>Anhang III: Android Screenshots der Xamarin.Forms App</b>	<b>XXI</b>
	<b>Anhang IV: Android Screenshots der Flutter App</b>	<b>XXIII</b>

# Abbildungsverzeichnis

2.1	Programmiersprachen als Schnittstelle . . . . .	4
2.2	Phasen eines Compilers . . . . .	6
2.3	Lexer Beispiel . . . . .	7
2.4	Interaktion zwischen Lexer und Parser . . . . .	8
2.5	Syntaxbaum . . . . .	9
2.6	Typüberprüfung . . . . .	9
2.7	Zwischendarstellungen . . . . .	10
3.1	Umfeld des Source-To-Source Compilers . . . . .	12
3.2	Source-To-Source Compiler Aufbau . . . . .	13
3.3	Compiler-Struktur . . . . .	14
3.4	Source-To-Source Compiler Informationsfluss . . . . .	15
3.5	Mockup der grafischen Oberfläche . . . . .	16
4.1	Xamarin.Forms Pages . . . . .	24
4.2	Xamarin.Forms Layouts . . . . .	27
4.3	Darstellung der Steuerelemente ‚Checkbox‘ und ‚Switch‘ . . . . .	32
4.4	Darstellung einer exemplarischen Login-Page . . . . .	37
4.5	Layout-Baum Überführung von Xamarin.Forms zu Flutter . . . . .	38
4.6	Flutter LoginPage Screenshot und Widget-Baum . . . . .	39
6.1	Aktivitätsdiagramm . . . . .	55
6.2	Compiler Phasen für Ansichten . . . . .	63
6.3	Klassendiagramm . . . . .	64
6.4	Bereinigung von Dart Widget Platzhaltern . . . . .	66
6.5	Grafische Oberfläche des Compilers . . . . .	69
7.1	Test Objekt Screenshots I . . . . .	71
7.2	Test Objekt Screenshots II . . . . .	71
7.3	Darstellung des Testobjektes als Webseite . . . . .	74

# Tabellenverzeichnis

2.1	Token-Beispiele . . . . .	7
4.1	Unterstützte Schemata des ‚url_launcher‘ Plugins . . . . .	23
4.2	Gegenüberstellung Pages . . . . .	27
4.3	Gegenüberstellung Layouts . . . . .	28
4.4	Gegenüberstellung Darstellungssteuerelemente . . . . .	29
4.5	Gegenüberstellung ereignisauslösende Steuerelemente . . . . .	30
4.6	Gegenüberstellung textmanipulierender Steuerelemente . . . . .	31
4.7	Gegenüberstellung wertsetzender Steuerelemente . . . . .	32
4.8	Gegenüberstellung aktivitätsandeutender Steuerelemente . . . . .	33
4.9	Gegenüberstellung sammlungsanzeigender Steuerelemente . . . . .	34
4.10	Gegenüberstellung Listen . . . . .	34
5.1	Gegenüberstellung Datentypen . . . . .	42
6.1	Plist Dateien in Flutter . . . . .	57
7.1	Testfälle der Testapp . . . . .	72



# Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface
<b>GUI</b>	Graphical user interface
<b>IDE</b>	Integrated development environment
<b>JSON</b>	JavaScriptObjectNotation
<b>S2S</b>	Source-to-Source
<b>SDK</b>	Software Development Kit
<b>UI</b>	User Interface
<b>UML</b>	Unified Modeling Language
<b>URI</b>	Uniform Resource Identifier
<b>WPF</b>	Windows Presentation Foundation
<b>XAML</b>	Extensible Application Markup Language

# Quellcodeverzeichnis

3.1	Bilderauswahl in Xamarin.Forms . . . . .	17
3.2	Bilderauswahl in Dart . . . . .	17
4.1	Verwendung von Schriftsätzen in Flutter . . . . .	20
4.2	Erweiterungen in Flutter . . . . .	21
4.3	Xamarin.Forms ‚TabPage‘ Definition . . . . .	25
4.4	Xamarin.Forms ‚FlyoutPage‘ Definition . . . . .	25
4.5	Flutter ‚MaterialApp‘ Definition . . . . .	26
4.6	Flutter ‚Tab Layout‘ Definition . . . . .	26
4.7	Xamarin.Forms Button Initialisierung . . . . .	31
4.8	Xamarin.Forms Event Handler . . . . .	31
4.9	Eingabefeld mit mehreren Zeilen in Flutter . . . . .	31
4.10	Verwendung von Timepickern in Flutter . . . . .	33
4.11	Exemplarische LoginPage in Dart . . . . .	38
5.1	Null-Sicherheit in Dart bis Version 2.12 . . . . .	41
5.2	Null-Sicherheit in Dart 2.12 . . . . .	41
5.3	Objekterzeugung ohne ‚new‘ Keyword in Dart . . . . .	42
5.4	Erstellung eines Strings mit einem Zeichen in Dart . . . . .	44
5.5	Datenfelder in Dart . . . . .	44
5.6	Hashtabellen in Dart . . . . .	45
5.7	Private und Public Definitionen in Dart . . . . .	45
5.8	Vererbung in Dart . . . . .	46
5.9	Mixin's in Dart . . . . .	46
5.10	Beispielklasse in C# . . . . .	47
5.11	Beispielklasse in Dart . . . . .	48
5.12	Importieren von Paketen in Dart . . . . .	49
5.13	Generics in Dart . . . . .	49
5.14	Delegates in Dart . . . . .	50
5.15	Async und Await in Dart . . . . .	51
5.16	Flutter Network request . . . . .	52
5.17	Events in Dart . . . . .	53
6.1	Xamarin.Forms Android Launcher-Icon Name . . . . .	56
6.2	Referenzierung von Bildern in der Pubspec.yaml . . . . .	58
6.3	C# Compiler Frontend . . . . .	59
6.4	Compilierung von Eigenschaftsdeklarationen . . . . .	60

---

6.5	Austausch von C# Modifizierern . . . . .	60
6.6	Austausch von C# Datentypen . . . . .	61
6.7	Austausch des Mediaplugins . . . . .	62
6.8	Austausch des Beschleunigungssensorplugins . . . . .	62
6.9	Flutter Widget Template . . . . .	64
6.10	FontSize Umwandlung . . . . .	65
6.11	Flutter Widget mit Binding-Platzhaltern . . . . .	66

# 1 Einleitung

Die Entwicklung von verschiedenen mobilen Geräten mit unterschiedlichsten Hardwarekomponenten und Betriebssystemen hat einen stark fragmentierten Markt ergeben.<sup>1</sup> Diese Situation hat einen direkten Einfluss auf die Softwareentwicklung, da die dedizierte Programmierung für die einzelnen Plattformen ressourcenintensiv ist. Durch Realisierung von Web- und hybriden Apps können Softwareprojekte von der darunterliegenden Plattform abstrahieren und plattformübergreifend verwendet werden. Diese Anwendungen haben jedoch, wie schon ausführlich im wissenschaftlichen Diskurs ausgeführt, eine schlechtere Performance und nur begrenzten Zugriff auf die plattformspezifischen Funktionalitäten.<sup>2</sup>

Durch die Kombination der Vorteile von Web- und hybriden Anwendungen mit denen von nativen konnten Frameworks wie Xamarin.Forms und Flutter Programmierern die Möglichkeit bieten, ihre Anwendungen auf mehreren Plattformen bereit zu stellen. Diese Apps haben neben einer guten Performance auch Zugriff auf sämtliche plattformspezifischen Funktionalitäten. Durch die Abstraktion von Hardware und Betriebssystem können Apps mit einer gemeinsamen Quelltextbasis und somit mit geringerem Ressourcenaufwand entwickelt werden.<sup>3</sup>

Der Möglichkeit, Ressourcen zu sparen, steht das Risiko der Abhängigkeit gegenüber, da sich die oben genannten Frameworks zur Cross-Plattform-Entwicklung in den verwendeten Programmiersprachen sowie ihrer Arbeitsweise grundlegend unterscheiden. Ein Wechsel zwischen den einzelnen Alternativen ist daher mit enormen Arbeitsaufwänden verbunden.<sup>4</sup>

---

<sup>1</sup>Vgl. Joorabchi 2016, S. 3.

<sup>2</sup>Vgl. Keist, Benisch und Müller 2016, S. 110ff.

<sup>3</sup>Vgl. Vollmer 2017, S. 295.

<sup>4</sup>Vgl. Wissel, Liebel und Hans 2017, S. 64.

## 1.1 Motivation

Im Mai 2020 hat Microsoft mit dem Multi-platform App User Interface (.NET MAUI) einen Nachfolger für das Xamarin.Forms Framework angekündigt, der im Herbst 2021 zusammen mit der sechsten Hauptversion des .NET Frameworks veröffentlicht werden soll. Zum aktuellen Zeitpunkt ist bereits bekannt, dass der Umstieg grundlegende Änderungen mit sich bringt und Anwendungen, die mit Hilfe von Xamarin.Forms entwickelt wurden, angepasst werden müssen.<sup>5</sup>

Für Xamarin.Forms Entwickler wird es also unausweichlich sein, tiefgreifende Modifizierungen an bereits realisierten Anwendungen vorzunehmen, um in der Zukunft von Aktualisierungen zu profitieren. Unternehmen und einzelne Programmierer stehen vor der Entscheidung, ob ein Umstieg auf das leistungsfähige Flutter sinnvoller ist, als die Anpassungen für das neue noch nicht erprobte .NET MAUI, das federführend von einer Firma entwickelt wird, welche leichtfertig mit der Abhängigkeit von Entwicklern umgeht.

Ein automatisierter Umstieg auf das von Google entwickelte Framework Flutter würde also nicht nur die Anpassungen an .NET MAUI vermeiden, sondern die mobile Anwendung auf eine vermeintlich zukunftsichere Basis stellen. Denn obwohl Google in der Vergangenheit schon manche Projekte eingestellt hat, wie zum Beispiel Google Nexus oder Google Hangouts, ist damit bei Flutter aufgrund des Erfolges nicht zu rechnen. Nach offizieller Aussage von Tim Sneath, dem Produkt Manager des Frameworks, haben im Jahr 2020 mehr als zwei Millionen Entwickler Flutter verwendet und über 50.000 mobile Anwendungen programmiert.<sup>6</sup> Neben der hohen Verbreitung des Frameworks, konnte das Portal Stackoverflow in seinen jährlichen Umfragen auch eine hohe Beliebtheit unter Softwareentwicklern in den Jahren 2019<sup>7</sup> und 2020<sup>8</sup> ermitteln. Im März 2021 hat Flutter darüber hinaus die zweite Hauptversion von Flutter veröffentlicht, welche zusätzlich Support für die Entwicklung von Webseiten zur Verfügung stellt.<sup>9</sup>

## 1.2 Ziel der Arbeit

Im Rahmen dieser Arbeit soll ein Source-To-Source Compiler zwischen den Frameworks Xamarin.Forms und Flutter realisiert werden, mit dessen Hilfe die folgende

---

<sup>5</sup>Vgl. Hunter 2020, Abgerufen am 28.10.2020.

<sup>6</sup>Vgl. Sneath 2020, Abgerufen am 28.10.2020.

<sup>7</sup>Vgl. Stack Exchange Inc. 2019, Abgerufen am 28.10.2020.

<sup>8</sup>Vgl. Stack Exchange Inc. 2020, Abgerufen am 28.10.2020.

<sup>9</sup>Vgl. Sells 2021, Abgerufen am 28.10.2020.

zentrale Forschungsfrage beantwortet werden soll: "Können Apps komplett automatisiert von Xamarin.Forms zu Flutter übersetzt werden, oder sind manuelle Arbeitsschritte erforderlich?"

## 1.3 Gliederung

Um diese Forschungsfrage beantworten zu können, wird in Kapitel 2 auf die theoretischen Grundlagen von Software-Übersetzern eingegangen. Anschließend wird in Kapitel 3 auf den Entwurf des in dieser Arbeit zu implementierenden Compiler eingegangen, bevor in Kapitel 4 die Unterschiede zwischen den Frameworks Xamarin.Forms und Flutter behandelt werden. Die Unterschiede zwischen den Programmiersprachen werden in Kapitel 5 behandelt. Darauf aufbauend wird in Kapitel 6 der Source-To-Source Compiler realisiert und in dem darauf folgen Kapitel 7 getestet bevor in Kapitel 8 die Forschungsfrage beantwortet wird und ein Fazit gezogen wird.

## 2 Compiler

Programmiersprachen dienen als Verständigungsmittel zwischen Programmierern und Rechenanlagen wie z.B Smartphones. Diese Sprachen haben sich in der Vergangenheit dabei immer mehr an die Terminologie eines bestimmten Anwendungsgebietes angenähert. Durch diese Entwicklung eigneten sich Programmiersprachen direkt für die Dokumentation von entwickelten Algorithmen und Anwendungen, entfernten sich jedoch weiter von den Gegebenheiten des realen Rechners.<sup>10</sup> Die Beziehung zwischen Softwareentwicklern und Rechenanlagen mit Hilfe von Programmiersprachen werden in Abbildung 2.1 dargestellt.



Abbildung 2.1: Programmiersprachen als Schnittstelle

Für die Ausführung einer in einer problemorientierten Programmiersprache geschriebenen Anwendung ist es notwendig, die Sprache in eine maschinenorientierte Form zu überführen.<sup>11</sup> Bereits im Jahre 1952 stellte Rutishauser fest, dass Computer in der Lage sind, diesen Übersetzungsvorgang selbst durchzuführen.<sup>12</sup> Durch die Möglichkeit zur automatischen Übersetzung von problemorientierten Programmiersprachen konnten Hochsprachen entwickelt werden, die menschenfreundliche Sprachelemente anstatt Maschineninstruktionen verwenden.<sup>13</sup>

<sup>10</sup>Vgl. Schneider 1975, S. 15.

<sup>11</sup>Vgl. Schneider 1975, S. 15.

<sup>12</sup>Vgl. Rutishauser 1952, S. 312.

<sup>13</sup>Vgl. Wagenknecht und Hielscher 2014, S. 47.

## 2.1 Grundbegriffe

Diese historische Einführung zeigt, dass Software zur automatisierten Übersetzung schon seit der Mitte des letzten Jahrhunderts thematisiert wurde, so hat sich in der Wissenschaft eine einheitliche Definition ergeben. Ullman et al. beschreibt die sogenannten Compiler im Jahre 2008 wie folgt:<sup>14</sup>

### Defintion 1: Compiler

Ein Compiler ist ein Programm, welches ein anderes Programm aus einer Quellsprache in ein gleichwertiges Programm einer Zielsprache übersetzen kann.

Aus dieser Definition lässt sich ein für diese Arbeit relevanter Fakt ableiten: Compiler sind nicht ausschließlich Übersetzer zwischen problemorientierten und maschinenorientierten Programmiersprachen. Sie sind ausschließlich für die Übersetzung von einer Quellsprache in eine Zielsprache verantwortlich. Auch wenn der Begriff Programm für jedermann geläufig ist, kann es passieren, dass von verschiedenen Repräsentationen gesprochen wird. So können alle drei der folgenden Begriffe als Programm bezeichnet werden: Der Quelltext, das ausführbare Programm und der laufende Prozess auf einem Computer. Für das weitere Verständnis dieser Arbeit ist mit dem Begriff Programm die ausführbare Anwendung auf den Smartphones des Anwenders gemeint.

Neben der Übersetzung von problem- zu maschinenorientierter Sprache gibt es ebenfalls Compiler, die andere Ziele verfolgen. Dazu gehören zum Beispiel die sogenannten Binärübersetzer, die den Binärcode eines Programmes für andere Rechner übersetzen, sodass er auf diesen ausgeführt werden kann.<sup>15</sup> Ein Source-to-Source (S2S) Compiler, häufig auch als „Transpiler“ bezeichnet, ist ebenfalls eine besondere Ausprägung eines Compilers, die sich wie folgt definieren lässt.<sup>16</sup>

### Defintion 2: Source-to-Source Compiler

Ein Source-to-Source-Compiler ist ein Compiler, bei dem sowohl die Quellsprache als auch die Zielsprache eine Hochsprache ist.

Der Begriff Hochsprache ist dabei ein Synonym für die bereits eingeführten problemnahen Sprachen wie zum Beispiel C++, Java, C# oder Dart und damit für den Menschen in einer lesbaren und änderbaren Form geschrieben sind.<sup>17</sup>

<sup>14</sup>Vgl. Ullman et al. 2008, S. 1.

<sup>15</sup>Vgl. Ullman et al. 2008, S. 27.

<sup>16</sup>Vgl. Rohit, Aditi und Hardikar 2015, S. 1629.

<sup>17</sup>Vgl. Eisenecker 2008, S. 9.



## 2.2 Compiler Struktur

Zur Übersetzung von Programmen bearbeiten Compiler zwei Teilaufgaben, die Analyse und die Synthese. Während der Analyse wird das Programm in seine Bestandteile zerlegt und mit einer grammatikalischen Struktur versehen. Diese wird anschließend verwendet, um eine Zwischendarstellung zu generieren. Dabei wird überprüft, ob das Programm syntaktisch und semantisch fehlerfrei ist oder ob der Programmierer Änderungen vornehmen muss.<sup>18</sup> Der Begriff Syntax beschreibt den Aufbau eines Programmes, sie legt fest wie Sprachelemente aus anderen Sprachelementen zusammengesetzt sind. Im Gegensatz dazu beschreibt die Semantik die Bedeutung der Programmen und regelt die Bedeutung von Sprachelementen.<sup>19</sup> Außerdem werden bei der Analyse Informationen über das Quellprogramm gesammelt und in der so genannten Symboltabelle abgelegt. Die Synthese konstruiert aus der Zwischendarstellung und den Informationen aus der Symboltabelle das gewünschte Zielprogramm. Der Teil des Compilers, der sich mit der Analyse befasst wird oft als Front-End bezeichnet, derjenige der für die Synthese zuständig ist als Back-End.<sup>20</sup>

Der Vorgang des Kompilierens lässt sich basierend auf diesen zwei Teilaufgaben nach Ullman et al. in mehrere Phasen unterteilen, die in Abbildung 2.2 grafisch dargestellt sind und in diesem Abschnitt detailliert beschrieben werden.<sup>21</sup>

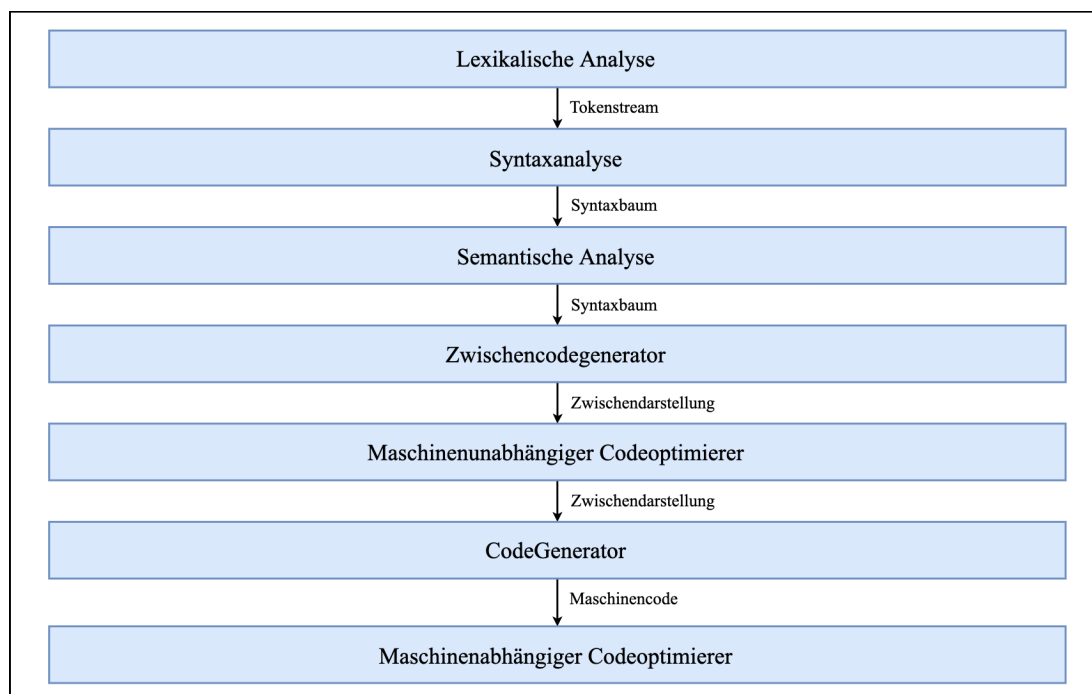


Abbildung 2.2: Phasen eines Compilers<sup>22</sup>

<sup>18</sup>Vgl. Ullman et al. 2008, S. 6f.

<sup>19</sup>Vgl. Schneider 1975, S. 36.

<sup>20</sup>Vgl. Ullman et al. 2008, S. 6f.

<sup>21</sup>Vgl. Ullman et al. 2008, S. 6.

<sup>22</sup>Abbildung in Anlehnung an Ullman et al. 2008, S.6.

## 2.3 Lexikalische Analyse

Die erste Phase eines Compilers ist die lexikalische Analyse, die den Quelltext in Lexeme untergliedert. Ein Lexem ist die Folge von Zeichen im Quellprogramm, die als Instanz eines Tokens erkannt wurden. Dabei ist ein Token ein Paar aus Namen und einem optionalen Attributwert, wobei der Name zum Beispiel ein bestimmtes Schlüsselwort, oder eine Folge von Eingabezeichen sein kann und der Attributwert auf einen Eintrag in der Symboltabelle verweist.<sup>23</sup> In Tabelle 2.1 werden einige beispielhafte Tokens aufgeführt sowie die Information darüber, aus welchen Lexemen diese extrahiert werden.

Token	Beschreibung	Lexem
if	Zeichen i,f	if
comparison	Vergleichsoperatoren	<=
id	Buchstaben	pi
number	Numerische Konstanten	3.14159

Tabelle 2.1: Token-Beispiele<sup>24</sup>

Der Teil eines Compilers, der die Lexikalische Analyse durchführt, wird als Lexer bezeichnet. Basierend auf der beschriebenen Arbeitsweise ist in 2.3 ein Beispiel dargestellt, das zeigt wie der Lexer aus einer Zeichenfolge mehrere Tokens mit den optionalen Attributwerten extrahiert.

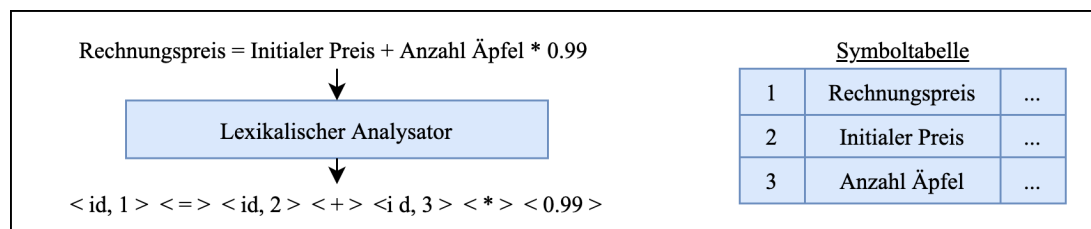


Abbildung 2.3: Lexer Beispiel<sup>25</sup>

Der Lexer interagiert mit anderen Komponenten eines Compilers, dies wird in Abbildung 2.4 visualisiert. Klassischerweise wird der Lexer über den sogenannten Parser, welcher im nächsten Abschnitt eingeführt wird, zur Übermittlung von Tokens aufgefordert, dies wird in der Abbildung mit dem Aufruf "getNextToken" dargestellt.<sup>26</sup>

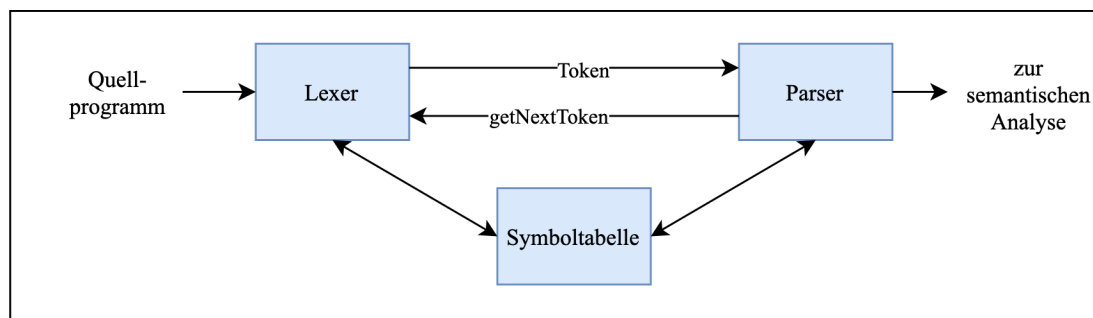
Da der Lexer derjenige Teil des Compilers ist, der den Quelltext liest, kann er neben der Identifikation von Lexemen auch weitere Aufgaben übernehmen. So eignet er sich ideal

<sup>23</sup>Vgl. Ullman et al. 2008, S. 135 f.

<sup>24</sup>Vgl. Ullman et al. 2008, S. 137.

<sup>25</sup>Abbildung in Anlehnung an Ullman et al. 2008, S.10.

<sup>26</sup>Vgl. Ullman et al. 2008, S. 135.

Abbildung 2.4: Interaktion zwischen Lexer und Parser<sup>27</sup>

zum Streichen von Kommentaren im Quelltext und zum Entfernen von Leerstellen, wie Leerzeichen und Tabulatoren. Zudem kann er gefundene Fehler den entsprechenden Zeilennummern zuordnen und dem Entwickler während der Kompilierung so einen genauen Hinweis auf den Ort des Fehlers geben.<sup>28</sup> Häufig werden Lexer daher in zwei kaskadierende Prozesse unterteilt, einen für das Löschen von Kommentaren und Zusammenfassung von Leerraumzeichen und einen für die eigentliche lexikalische Analyse.<sup>29</sup>

## 2.4 Syntaxanalyse

In der zweiten Phase, der Übersetzung, der Syntaxanalyse, werden durch den bereits erwähnten Parser auch syntaktischer Analysator genannt, die vom Lexer ausgegebenen Tokens in eine baumartige Zwischendarstellung überführt, die die grammatikalische Struktur der Tokens zeigt. Diese Darstellung wird basierend auf ihrem Aussehen häufig als Syntaxbaum bezeichnet. Die Knoten im Syntaxbaum stehen für eine Operation und die Kindknoten für die Argumente dieser Operation. Die Anordnung der Operationen stimmt mit üblichen arithmentischen Konventionen überein, wie zum Beispiel dem Vorrang der Multiplikation vor Addition.<sup>30</sup> Abbildung 2.5 zeigt die Erstellung eines Syntaxbaumes aus den Tokens der Abbildung 2.3. Anhand des Knotens <id, 1> ist jederzeit über die Symboltabelle bekannt, dass das Ergebnis der Rechnung an den Speicherort des Bezeichners Rechnungspreis abgelegt werden muss.<sup>31</sup>

<sup>27</sup>Abbildung in Anlehnung an Ullman et al. 2008, S.135.

<sup>28</sup>Vgl. Ullman et al. 2008, S. 135.

<sup>29</sup>Vgl. Ullman et al. 2008, S. 136.

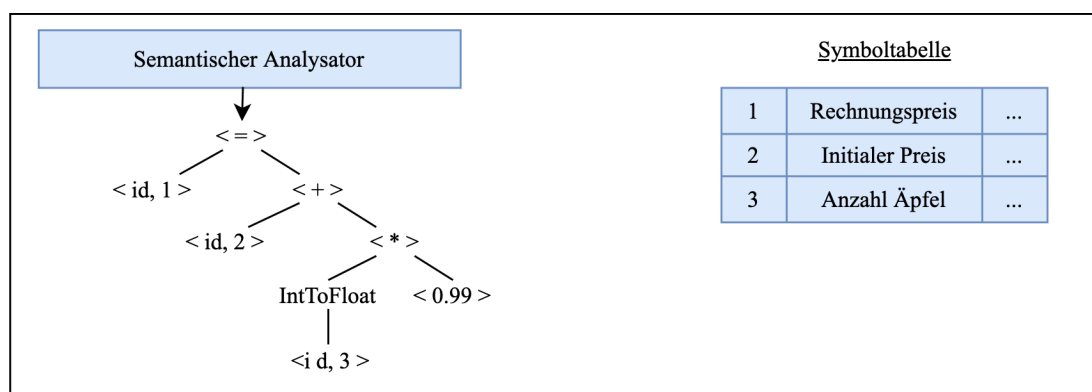
<sup>30</sup>Vgl. Ullman et al. 2008, S. 9.

<sup>31</sup>Vgl. Ullman et al. 2008, S. 9.

Abbildung 2.5: Syntaxbaum<sup>32</sup>

## 2.5 Semantische Analyse

Bei der semantischen Analyse wird der Syntaxbaum als Aufgliederung der Programmstruktur, zusammen mit den Informationen aus der Symboltabelle verwendet, um das Quellprogramm auf semantische Konsistenz mit der Sprachdefinition zu überprüfen.<sup>33</sup> Zudem werden hier Typinformationen gesammelt und zur späteren Verwendung im Syntaxbaum oder der Symboltabelle hinterlegt. Auch findet eine Typüberprüfung statt die analysiert, ob jeder Operator die passenden Operanden hat. So wird beispielsweise validiert, ob ein Index eine Ganzzahl ist. Es besteht die Möglichkeit, innerhalb des Baums Typkonvertierungen zu deponieren. So wurde in dem bisherigen Beispiel die Anzahl Äpfel als Ganzzahl behandelt und wird für die Berechnung des Preises in Abbildung 2.6 zu einer Fließkommazahl konvertiert.<sup>34</sup>

Abbildung 2.6: Typüberprüfung<sup>35</sup>

<sup>32</sup>Abbildung in Anlehnung an Ullman et al. 2008, S.10.

<sup>33</sup>Vgl. Wilhelm, Seidl und Hack 2012, S. 157.

<sup>34</sup>Vgl. Ullman et al. 2008, S. 9ff.

<sup>35</sup>Abbildung in Anlehnung an Ullman et al. 2008, S.10.

## 2.6 Zwischencodeerzeugung

Während der Übersetzung eines Programms kann der Compiler mehrere Zwischendarstellungen in unterschiedlichsten Formen, zum Beispiel wie die eines Syntaxbaums, erstellen. Nach der semantischen Analyse stellen viele Compiler eine maschinennahe Zwischendarstellung auf niedriger Abstraktionsebene her, die eigentlich für maschinenabhängige Aufgaben wie Befehlsauswahl geeignet ist. Eine Zwischendarstellung, die

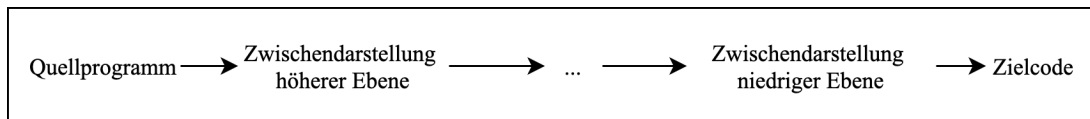


Abbildung 2.7: Zwischendarstellungen<sup>36</sup>

von Compiler zu Compiler in Auswahl oder Entwurf unterschiedlich ist, kann entweder eine tatsächliche Sprache sein, oder aus internen Datenstrukturen bestehen, die von den Phasen des Compilers gemeinsam verwendet werden. Auch wenn C eine Programmiersprache ist, wird sie häufig als eine Zwischenform verwendet, da sie flexibel ist, zu effizientem Maschinencode kompiliert werden kann und ihre Compiler weitgehend verfügbar sind.<sup>37</sup>

## 2.7 Codeoptimierung

In dieser Phase wird der Code so optimiert, dass sich daraus ein besserer, das heißt schnellerer oder ressourcenschonender Zielcode ergibt. Der Umfang der Codeoptimierung schwankt dabei von Compiler zu Compiler erheblich.<sup>38</sup> Die Codeoptimierung, die ein Compiler vornimmt, ist im Laufe der Zeit wichtiger und komplexer geworden. Grund für die zunehmende Komplexität sind die immer komplexeren Prozessorarchitekturen, die mehr Gelegenheiten bieten, die Ausführung des Codes zu verbessern. Die gestiegene Bedeutung ergibt sich Beispielsweise aus der steigenden Anzahl an Kernen in modernen Computern und der Möglichkeit, Programme parallel auszuführen.<sup>39</sup>

<sup>36</sup>Abbildung in Anlehnung an Ullman et al. 2008, S.433.

<sup>37</sup>Vgl. Ullman et al. 2008, S. 433.

<sup>38</sup>Vgl. Ullman et al. 2008, S. 11f.

<sup>39</sup>Vgl. Ullman et al. 2008, S. 20.

## 2.8 Codeerzeugung

Die Überführung aus der Zwischendarstellung in die Zielsprache nennt man Codeerzeugung. Hierbei muss die semantische Bedeutung des Quellprogramms erhalten und hochwertig dargestellt sein. Die größte Herausforderung ergibt sich aus der nicht komplett mathematischen Berechenbarkeit aller Prozesse bei der Überführung. Ein Beispiel wäre die Vergabe von Registern, die nicht effizient berechenbar sind. In der Praxis müssen heuristische Techniken ausreichen- die guten, aber nicht unbedingt optimalen Code liefern. Die Codeoptimierungs- und Codeerzeugungsphasen können mehrfach durchlaufen werden, bevor das Zielprogramm finalisiert ist.<sup>40</sup>

## 2.9 Der .NET Compiler Roslyn

Für die Arbeit mit der Programmiersprache C# steht mit Roslyn ein Compiler zur Verfügung, der sich aus modularen Bibliotheken zusammensetzt. Durch die Referenzierung dieser Bibliotheken können Programme auf den Funktionsumfang von Roslyn zugreifen. So ist es möglich, den Compiler zu verwenden, ohne das Ziel zu haben, die Programmiersprache C# in plattformnahen Code zu übersetzen. Dabei stehen die Bibliotheken über den Packetmanager Nuget für die Einbindung in eigene Projekte zur Verfügung. Um diese Funktionalität zu gewährleisten, unterteilt Roslyn die Übersetzung in mehrere Phasen, welche wiederum einige der in diesem Kapitel beschriebenen Phasen zusammenfassen. Die erste Phase ist die Erstellung des Syntaxbaums, die zweite Phase ist die semantische Analyse gefolgt von der letzten Phase der Ausgabe der so genannten Intermediate Language als Zielsprache.<sup>41</sup>

---

<sup>40</sup>Vgl. Ullman et al. 2008, S. 618f.

<sup>41</sup>Vgl. Albahari und Johannsen 2020, S. 1017.

### 3 Compiler Spezifikation

Zur Entwicklung eines möglichst effektiven Compilers ist es notwendig, die genauen Anforderungen zu ermitteln, um dann passende Softwaretechnische Lösungen zu erarbeiten.<sup>42</sup> Im speziellen Falle besteht die Anforderung darin, vom Quellframework Xamarin.Forms in das Zielframework Flutter zu übersetzen, welche beide für die Entwicklung von plattformunabhängigen Smartphone-Apps verwendet werden können. Für eine genaue Spezifikation des zu realisieren Source-To-Source Compilers ist es notwendig, einen Überblick über das Compiler-Umfeld zu erhalten. Dieses wird in Abbildung 3.1 visualisiert.

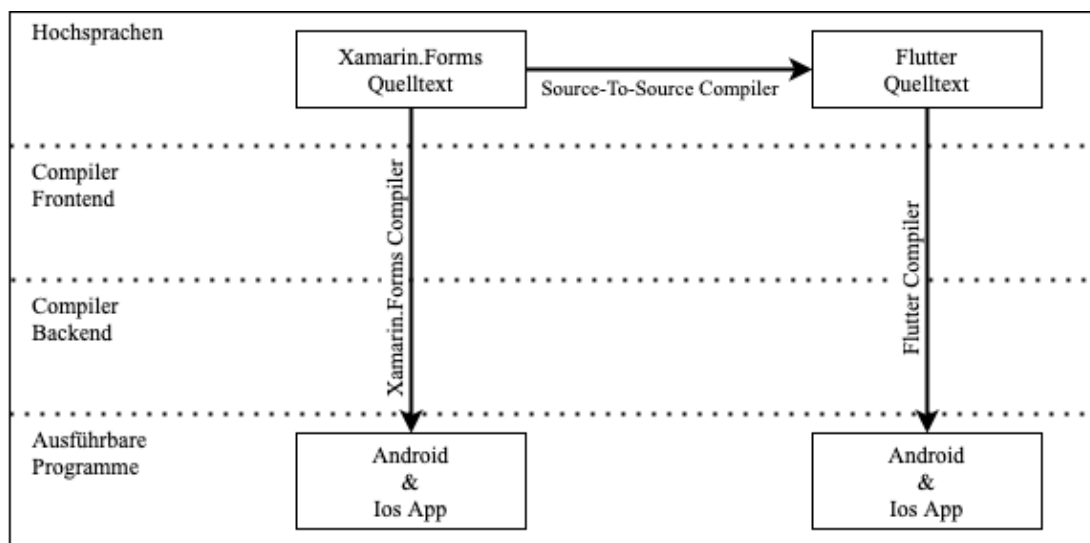


Abbildung 3.1: Umfeld des Source-To-Source Compilers

Wie auf der Abbildung erkennbar ist, durchlaufen sowohl Xamarin.Forms als auch Flutter bei der Übersetzung zu mobilen Anwendungen die in Kapitel 3.2 erläuterten Phasen, kurz dargestellt als Compiler Front- und Backend. Der Source-To-Source Compiler ist in der Abbildung horizontal dargestellt, was veranschaulichen soll, dass sich sowohl die Quelle, als auch das Ziel der Übersetzung auf einer Abstraktionsebene befinden. Auch bei dieser Übersetzung sind die Compilerphasen aus dem vorherigen Kapitel anzuwenden. So lässt sich die Abbildung 3.1, wie in 3.2 dargestellt, um ein Front- und Backend erweitern.

<sup>42</sup>Vgl. Balzert 2011, S.6.



Abbildung 3.2: Source-To-Source Compiler Aufbau

Laut Definition von Compilern, erzeugen diese ein gleichwertiges Programm in einer Zielsprache. Sowohl Xamarin.Forms als auch Flutter stellen mit Hilfe ihrer Compiler gleichwertige Programme in Form von mobilen Apps dar. Da der Source-to-Source Compiler ebenfalls eine gleichwertige Darstellung erzeugt, ist anzunehmen, dass die übersetzte Ursprungs-App gleichwertig zu der übersetzten Flutter App ist.

### 3.1 Funktionseingrenzung

Zur Beantwortung der Forschungsfrage ist es ausreichend, dass der Prototyp einen begrenzten Funktionsumfang hat. Für eine zielführende Eingrenzung eignen sich die folgenden fünf Aspekte:

- **Framework Version:** Der in dieser Arbeit zu realisierende Prototyp soll ausschließlich das offiziell von Microsoft veröffentlichte Xamarin.Forms in der Version 5.0.0.2012 zu Flutter übersetzen.
- **Erweiterungen von Dritten:** Viele Firmen und einzelne Entwickler haben Erweiterungen für Xamarin.Forms programmiert. Aufgrund der großen Anzahl und stetigen Veränderung dieser Erweiterungen, werden sie in dieser Arbeit nicht weiter betrachtet.
- **Plattformspezifischer Quelltext:** Xamarin.Forms erlaubt die Verwendung von plattformspezifischem Quelltext, der in dieser Arbeit keine Beachtung finden wird, da eine gleichwertige Darstellung in Flutter nicht garantiert werden kann.
- **User Interface (UI):** Für die Entwicklung von Benutzeroberflächen kann die Programmiersprache C# verwendet werden, jedoch hat die Alternative Extensible Application Markup Language (XAML) für Entwickler Vorteile,<sup>43</sup> weswegen die Konstruktion von Benutzeroberflächen mit C# in dieser Arbeit nicht berücksichtigt wird.

<sup>43</sup>Vgl. Microsoft Corporation 2017a, Abgerufen am 17. April 2021.



- App-Styles: Für die visuelle Darstellung wird in dieser Arbeit ausschließlich das Design-System Material von Google unterstützt. Da die Übersetzung von Darstellungsoptionen sehr aufwendig ist, und für den in dieser Arbeit zu entwickelnden Prototypen nicht notwendig ist.

Diese Eingrenzungen führen in Summe zu einer Vielzahl von nicht in Gänze übersetzbaren Xamarin.Forms Anwendungen. Durch Erweiterungen des Compilers könnte diese Limitierung in Zukunft aufgehoben werden. Im Rahmen dieser Arbeit wird eine mobile Xamarin.Forms Anwendung entworfen, die vollständig übersetzt werden kann, da sie keine der oben definierten Ausschlüsse verwendet.

## 3.2 Übersetzung von verschiedenen Dateien

Durch seinen modularen Aufbau, kann der im letzten Kapitel eingeführte Roslyn Compiler die Phasen bis zur semantischen Analyse im zu entwickelnden Prototypen übernehmen. Anschließend kann mithilfe des dabei typisierten Syntaxbaumes die Übersetzung in die Zielsprache durchgeführt werden. Die Übersetzung mit Roslyn hat Grenzen, die aus der Zusammensetzung von Xamarin.Forms Projekten resultiert. Wie in Abbildung 3.3 zu erkennen ist, setzen sich Xamarin.Forms Projektmappen aus verschiedenen Dateien zusammen, von denen ausschließlich die Klassen beinhaltenden Dateien mit der Endung ‚.cs‘ analysiert werden können.



Abbildung 3.3: Compiler-Struktur

Neben den Klassen zeigt die Abbildung 3.3, dass auch Ansichten ein Teil von Xamarin.Forms Projekten sind. Diese bestehen aus ‚XAML‘ sowie ‚XAML.cs‘ Dateien. Alle Ausgangsdateien müssen zu Dart-Dateien kompiliert werden, um ein Flutter Projekt als Ziel zu ergeben. Die Zusammenführung von ‚XAML‘ und ‚XAML.cs‘ Dateien ist dabei notwendig, weil Flutter ohne sogenannte Codebehind Dateien auskommt.

### 3.3 Informationsfluss

Neben dem Roslyn Compiler soll der zu entwickelnde Transpiler auch die Flutter Software Development Kit (SDK) als bereits bestehende Softwarekomponente verwenden. Die Flutter-SDK enthält die Pakete und Kommandozeilen-Tools die für die Entwicklung von Flutter Anwendungen notwendig sind.<sup>44</sup> Der Compiler kann diese SDK nutzen um das Zielprojekt anzulegen und die übersetzte App zu testen. Basierend auf diesen beiden existierenden Software Komponenten kann nun der Informationsfluss wie in Abbildung 3.4 dargestellt werden.

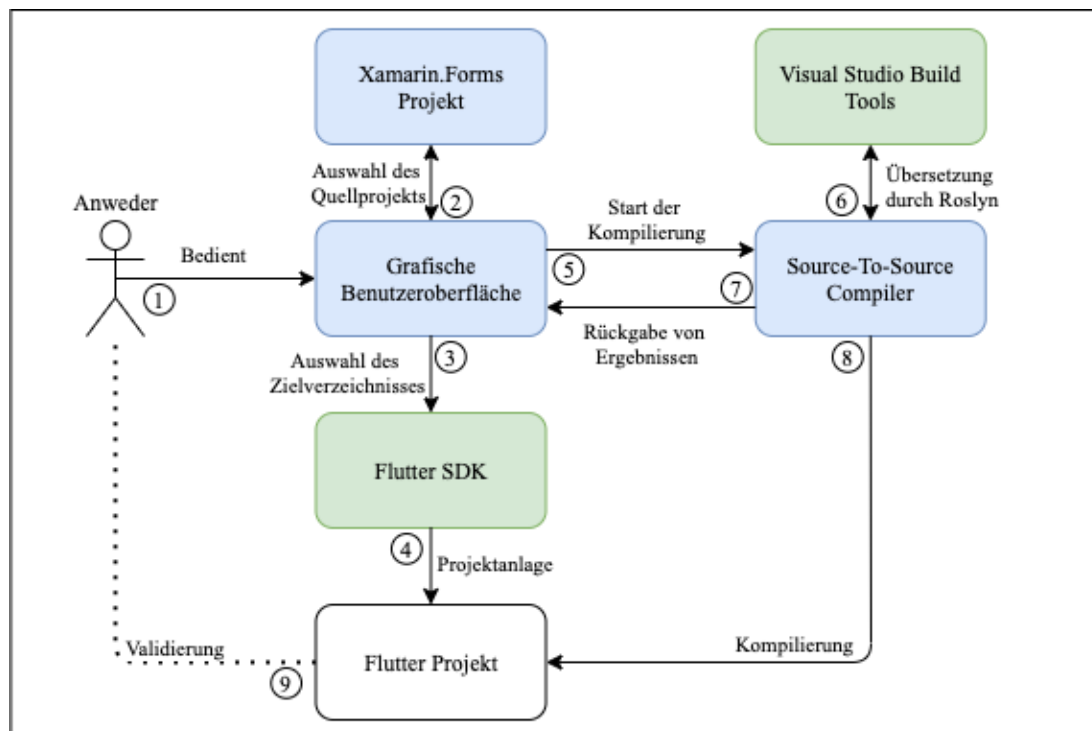


Abbildung 3.4: Source-To-Source Compiler Informationsfluss

Die in dieser Arbeit zu realisierenden Komponenten erscheinen in der Abbildung blau hinterlegt, während bereits existierende Softwarekomponenten, wie die ‚Visual Studio Build Tools‘ (die den Roslyn Compiler beinhalten) und die Flutter SDK grün gefärbt sind. Die Farbe des Xamarin.Forms Projektes bezieht sich auf die in dieser Arbeit realisierte mobile Anwendung zur Überprüfung des Prototypen. Die Zahlen innerhalb der Abbildung stellen die Zeitliche Abfolge dar. So wählt der Anwender in einem ersten Schritt das Xamarin.Forms Projekt über die grafische Benutzeroberfläche aus. Anschließend wird das Zielverzeichnis angegeben und mit Hilfe der Flutter SDK das Projekt in diesem Verzeichnis initiiert. Anschließend werden diese Informationen an den Source-To-Source Compiler übergeben, welcher zusammen mit den Visual Studio Build Tools das Ausgangsprojekt übersetzt und die Ergebnisse zurück an die Graphical user

<sup>44</sup>Vgl. Google LLC 2021d, Abgerufen am 17. April 2021.

interface (GUI) leitet. Außerdem werden die übersetzten Dart Dateien im angelegten Dart Projekt abgelegt und der Anwender kann das Flutter Projekt testen.

## 3.4 Grafische Darstellung

Damit Unternehmen und Entwickler ihre bestehenden Xamarin.Forms Anwendungen übersetzen können, muss eine Möglichkeit für die Interaktion mit dem Source-To-Source Compiler existieren. Dieser Compiler ist zur einmaligen und nicht regelmäßigen Verwendung ausgelegt und braucht somit nicht in einer Entwicklungsumgebung integrieren werden. Der Roslyn Compiler ist ausschließlich für das Betriebssystem Windows verfügbar, die zu entwickelnde Oberfläche muss dementsprechend auf Windows Computern lauffähig sein. Abbildung 3.5 zeigt einen Entwurf (engl. Mockup) der geplanten GUI.

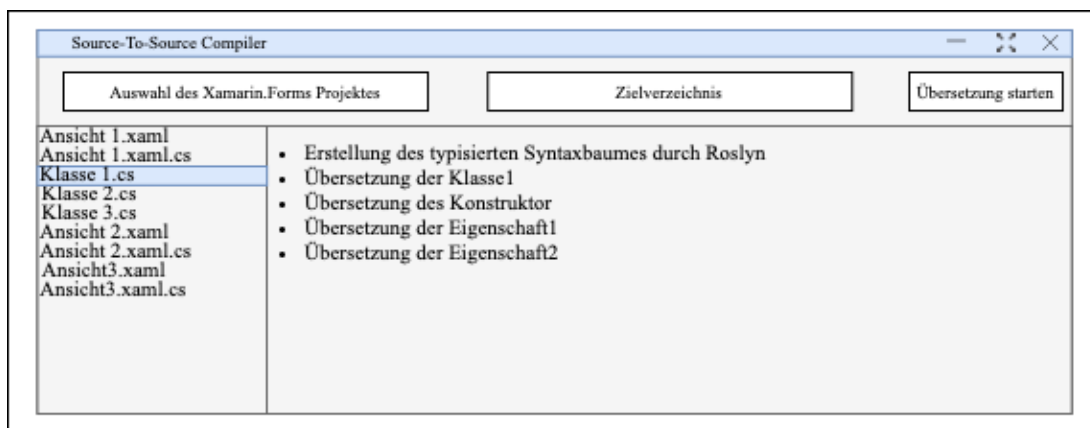


Abbildung 3.5: Mockup der grafischen Oberfläche

Im oberen Teil der GUI befindet sich eine Auswahl für das Quellprojekt und das Zielverzeichnis des Compilers. Der untere Teil der Ansicht zeigt die Ausgabe des Übersetzers, für die im linken Bereich alle bearbeiteten Dateien angezeigt werden. Bei der Auswahl einer Datei werden in dem Bereich daneben alle vorgenommen Übersetzungsschritte aufgeführt.

## 3.5 Quelltext Optimierung

Die Optimierung des Quelltextes ist, wie in Kapitel 2 beschrieben, eine Phase der Kompilierung. Im Gegensatz zu den dort beschriebenen Aspekten (Geschwindigkeit und Ressourcenschonung) sind für den Source-To-Source Compiler andere Faktoren wie der Austausch von Klassen und Methoden relevant, da diese Ressourcenoptimierung

später bei der Übersetzung durch den Flutter Compiler stattfinden. Der Bedarf zum Austausch von Klassen und Methoden resultiert aus unterschiedlichen Arbeitsweisen der Frameworks, sodass eine einfache 1:1 Übersetzung nicht möglich ist. Um dies zu visualisieren, wird folgend ein Quelltextbeispiel aus beiden Frameworks gezeigt, die die selbe Funktionalität abbilden.

```
1 var photo = await MediaPicker.PickPhotoAsync();
2
3 if(photo == null)
4 {
5     Console.WriteLine("No image selected.");
6 }
```

Quelltext 3.1: Bilderauswahl in Xamarin.Forms

```
1 final pickedFile = await picker.getImage(source: ImageSource.camera);
2
3 setState(() {
4     if (pickedFile != null) {
5         _image = File(pickedFile.path);
6     } else {
7         print('No image selected.');
```

Quelltext 3.2: Bilderauswahl in Dart

Beide Crossplatform Frameworks verwenden in diesem Beispiel unterschiedliche Klassen für die Auswahl eines Bildes aus der Smartphonegalerie. Daher ist es notwendig, beide Frameworks zu analysieren und die genauen Unterschiede zwischen den Arbeitsweisen zu verstehen. Zu diesem Zweck werden im nachfolgenden Kapitel sowohl die Frameworks als auch deren Programmiersprachen analysiert. Hieraus resultiert das Verständnis, inwiefern sich Benutzeroberflächen und Sprachen unterscheiden und wie diese übersetzt werden können.

Source-To-Source Compiler bilden eine Brücke zwischen zwei Hochsprachen. Der für die Beantwortung der Forschungsfrage geplante Compiler soll darüber hinaus auch die Arbeitsweisen des Quellprogramms in das Zielprogramm übersetzen. Das heißt, es soll versucht werden, ein frameworkbasierte App in die Form eines Zielframeworks zu überführen.

## 4 Technische Unterschiede zwischen Xamarin.Forms und Flutter

Die Unterschiede zwischen den Frameworks werden im folgenden genauer betrachtet. Für den technischen Vergleich dient Xamarin.Forms als Grundlage. Die Namen von Abschnitten und Unterabschnitten orientieren sich deshalb an dessen Terminologie. In den jeweiligen Gliederungspunkten wird anschließend genauer betrachtet, wie sich spezielle Arbeitsweisen oder Darstellungsoptionen in Flutter abbilden lassen.

### 4.1 Projektaufbau

Xamarin.Forms weist eine andere Projektstruktur auf als Flutter, das nur mit einem Projekt arbeitet. Während das Flutter Projekt alle notwendigen Inhalte für iOS und Android inkludiert,<sup>45</sup> setzt sich die sogenannte Lösung bei Xamarin.Forms aus mehreren Projekten zusammen. Es gibt für jede Plattform ein dediziertes Projekt, das den plattformspezifischen Code, Konfigurationen und Icons beinhaltet, sowie ein Projekt für den plattformunabhängigen Quelltext.<sup>46</sup> Icons und Konfigurationen werden bei Flutter in einem gleichen oder ähnlichen Format und nur in einem weiteren Projekt hinterlegt und lassen sich folglich migrieren.

#### 4.1.1 Metadaten

Zu den Metadaten einer Anwendungen gehören unter anderem der Name der Anwendung, Informationen wie die benötigte Betriebssystemversion, das ‚App Launcher Icon‘, das auf dem Smartphonebildschirm angezeigt wird und die Berechtigungen, die von

---

<sup>45</sup>Vgl. Biessek 2019, S. 113.

<sup>46</sup>Vgl. Petzold 2016, S. 25f.

der mobile Anwendung während der Ausführung beantragt werden können. Diese Eigenschaften werden bei Xamarin.Forms innerhalb der nativen Projekte verwaltet. In iOS können Änderungen mittels der Datei ‚Info.plist‘ definiert werden.<sup>47</sup> Android speichert Metadaten innerhalb der ‚AndroidManifest.xml‘.<sup>48</sup> Flutter verwendet für die Verwaltung der Metadaten die identischen Dateien, daher ist es möglich, diese aus dem Xamarin.Forms Projekt zu kopieren und innerhalb des Flutter Projektes zu sichern. Dafür muss die ‚Info.plist‘ im Verzeichnis ‚ios/Runner/‘ und die ‚AndroidManifest.xml‘ in ‚android/app/src/main/‘ gespeichert werden. In diesen Dateien wird außerdem der Identifizierer der Anwendung definiert. Durch eine Kopie der Konfigurationsdatei und Erhöhung der Versionsnummer wird eine spätere Kompilierung der Flutter-App demnach als eine Aktualisierung der Xamarin.Forms App erkannt.<sup>49</sup>

### 4.1.2 Bilder und Startbildschirm

Innerhalb von Apps können Bilder das Benutzererlebnis verbessern und helfen, eine Aktion zu veranschaulichen oder komplexe Botschaften zu verdeutlichen.<sup>50</sup> In iOS und Android werden sie in verschiedenen Auflösungen bereit gestellt. Das Betriebssystem wählt während der Laufzeit die beste Ressource basierend auf den Eigenschaften des Smartphonedisplays aus. Xamarin.Forms verwendet die Application Programming Interfaces (APIs) (auf deutsch Programmierschnittstellen) der nativen Plattformen zum Laden lokaler Bilder und unterstützt daher die plattformspezifischen Funktionalitäten.<sup>51</sup> Zur Verwendung nativer Bilddateien müssen die Bilder in Xamarin.Forms zu jedem Anwendungsprojekt hinzugefügt werden und vom gemeinsamen Xamarin.Forms-Code referenziert werden. Flutter verwendet im Gegensatz zu Xamarin.Forms keine nativen APIs, sondern ein einfaches Dichte-basiertes Format, ähnlich dem von iOS. Für die Anzeige von Bildern arbeitet Flutter mit sogenannten logischen Pixeln. Alle Bilderressourcen können sich in einem beliebigen Ordner innerhalb des Projektes befinden, da Flutter keine vordefinierte Ordnerstrukturen hat.<sup>52</sup> Der Startbildschirm (engl. Splash-screen) ist der Einstiegspunkt der mobilen App. Er dient als Ladebildschirm und wird bei Xamarin.Forms in den plattformspezifischen Projekten gespeichert und kann ähnlich wie die ‚Info.plist‘- und die ‚AndroidManifest.xml‘-Dateien in das Flutter Projekt kopiert werden, da die technische Implementierung identisch ist.<sup>53</sup>

<sup>47</sup>Vgl. Microsoft Corporation 2017c, Abgerufen am 17. April 2021.

<sup>48</sup>Vgl. Microsoft Corporation 2018a, Abgerufen am 17. April 2021.

<sup>49</sup>Vgl. Vaibhavi 2020, Abgerufen am 17. April 2021.

<sup>50</sup>Vgl. Google LLC 2020h, Abgerufen am 17. April 2021.

<sup>51</sup>Vgl. Microsoft Corporation 2020d, Abgerufen am 17. April 2021.

<sup>52</sup>Vgl. Google LLC 2020c, Abgerufen am 17. April 2021.

<sup>53</sup>Vgl. Google LLC 2020b, Abgerufen am 17. April 2021.

### 4.1.3 Benutzerdefinierte Schriftarten

Eine Schriftart (engl. Font) wird verwendet, um das Design und den Inhalt so klar und effizient wie möglich darzustellen, dafür haben Android und iOS eine Vielzahl an Fonts vorinstalliert. Wenn eine mobile App jedoch auf eine benutzerdefinierte Schriftart zurückgreifen möchte, muss diese mit der Anwendung mitgeliefert werden. In Xamarin.Forms mussten Fonts bis zum Jahre 2020 in jedem nativen Projekt referenziert werden. Seit der Version 4.5 können diese Dateien auch plattformübergreifend verwendet werden und befinden sich daher innerhalb des geteilten Projekts.<sup>54</sup> In Flutter werden die Fonts wie in der neueren Version von Xamarin.Forms in einem Ordner abgelegt.<sup>55</sup> Eine Verwendung des Schriftsatzes ‚MyCustomFont‘ in Flutter wird in Quelltext 4.1 dargestellt.

```
1 @override
2 Widget build(BuildContext context) {
3   return Scaffold(
4     appBar: AppBar(
5       title: Text("Sample App"),
6     ),
7     body: Center(
8       child: Text(
9         'This is a custom font text',
10        style: TextStyle(fontFamily: 'MyCustomFont'),
11      ),
12    ),
13  );
14 }
```

Quelltext 4.1: Verwendung von Schriftsätzen in Flutter<sup>56</sup>

### 4.1.4 Plattformspezifischer Quelltext

In den Ausschlusskriterien des dritten Kapitels wurde der plattformspezifische Quelltext von Xamarin.Forms für die Übersetzung exkludiert, der Vollständigkeit halber sollen die Unterschiede zwischen den Plattformen jedoch trotzdem erwähnt werden. Xamarin.Forms unterteilt nativen Quelltext in zwei Kategorien, die sogenannten ‚DependencyServices‘ die es erlauben, Funktionalitäten in den Plattformprojekten aus dem geteilten Code aufzurufen,<sup>57</sup> sowie ‚Custom Renderers‘, die es ermöglichen das Ausse-

<sup>54</sup>Vgl. Versluis 2020, Abgerufen am 17. April 2021.

<sup>55</sup>Vgl. Google LLC 2020o, Abgerufen am 17. April 2021.

<sup>56</sup>Quelltext in Anlehnung an Google LLC 2020o, Abgerufen am 17. April 2021.

<sup>57</sup>Vgl. Microsoft Corporation 2019c, Abgerufen am 17. April 2021.

hen und Verhalten von Steuerlementen anzupassen.<sup>58</sup> In Flutter wird für die Ausführung von plattformspezifischen Funktionalitäten auf ‚platform-channels‘ zurückgegriffen.<sup>59</sup> Für ‚Custom Renderer‘ gibt es keine Alternative, da das Framework keine nativen Steuerelemente verwendet, sondern Widgets anzeigt.

## 4.2 Erweiterungen

Erweiterungen sind Programmergänzungen, die auch von externen Entwicklern beigetragen werden können. Sie dienen der Reduzierung des Entwicklungsaufwandes, da nicht jede Funktionalität eigens implementiert werden muss. Im .NET-Ökosystem können Xamarin.Forms-Projekte auf das Paketverwaltungssystem Nuget zugreifen, um Erweiterungen zu einer App hinzuzufügen.<sup>60</sup> Bei Flutter wird für diesen Fall mit der pubspec.yaml Datei eine Referenz auf das ausgewählte Plugin gesetzt. Dabei werden in Dart geschriebene Pakete von plattformunabhängigen Plugins unterschieden. Diese Plugins können für Android (mit Kotlin oder Java) oder für iOS (mit Swift oder Objective-C) geschrieben sein.<sup>61</sup> In Quelltext 4.2 wird ein Ausschnitt der pubspec.yaml Datei gezeigt, in welcher das in Unterabschnitt 4.2.3 erwähnte Plugin ‚url\_launcher‘ geladen wird.

```
1 dependencies:  
2   flutter:  
3     sdk: flutter  
4   url_launcher: '>=5.4.0 <6.0.0'
```

Quelltext 4.2: Erweiterungen in Flutter<sup>62</sup>

In dem Beispiel muss das Plugin mindestens die Version 5.4 haben, darf jedoch die Version 6 nicht überschreiten. Aufgrund der Vielzahl an existierenden Erweiterungen für beide Frameworks ist es nicht möglich, eine vollständige Gegenüberstellung zu erstellen. Im Folgenden wird jedoch anhand von drei gängigen Anwendungsszenarien der Einsatz von Erweiterungen erläutert.

<sup>58</sup>Vgl. Microsoft Corporation 2019b, Abgerufen am 17. April 2021.

<sup>59</sup>Vgl. Google LLC 2020t, Abgerufen am 17. April 2021.

<sup>60</sup>Vgl. Microsoft Corporation 2020a, Abgerufen am 17. April 2021.

<sup>61</sup>Vgl. Google LLC 2020p, Abgerufen am 17. April 2021.

<sup>62</sup>Quelltext in Anlehnung an Google LLC 2020p, Abgerufen am 17. April 2021.



### 4.2.1 Interaktion mit der Hardware

Android und iOS nutzen einzigartige Betriebssystem- und Plattform-APIs, auf die Xamarin.Forms Entwickler zugreifen können. Mit der Erweiterung Xamarin.Essentials bietet Microsoft eine plattformübergreifende API, auf die von gemeinsamem Code aus zugegriffen und die direkt auf der Plattform ausgeführt werden kann. Der Dart Quelltext, aus dem eine Flutter-App besteht, wird nicht auf der zugrundeliegenden Plattform, sondern nativ auf dem Gerät ausgeführt. Es werden also nicht die iOS oder Android APIs benutzt. Flutter Apps können über Plattformkanäle jedoch auch mit den nativen APIs interagieren, um beispielsweise Daten von Sensoren des Gerätes abzurufen.<sup>63</sup>

### 4.2.2 Speicherung von Daten

Ein wesentlicher Bestandteil jeder mobilen Anwendung ist die Fähigkeit, Daten zu persistieren. Manchmal handelt es sich dabei um große Datenmengen, die eine Datenbank erfordern, oft sind es aber auch kleinere Daten, wie Einstellungen und Präferenzen, die zwischen den Starts der Anwendung gespeichert werden müssen.

Für die Speicherung in einer Datenbank können Xamarin.Forms Entwickler auf verschiedene Lösungsansätze zurückgreifen. Zum einen ‚SQLite‘, die am häufigsten verwendete Datenbank-Engine der Welt<sup>64</sup>, oder ‚Realm‘, eine Datenbank optimiert für mobile Endgeräte.<sup>65</sup> Beide Datenbanken stehen auch als Plugin für ‚Flutter‘ zur Verfügung, wobei SQLite ausgereift ist,<sup>66</sup> während ‚Realm‘ erst am 5 November 2020 Support für Flutter angekündigt hat und noch nicht offiziell zur Verfügung steht.<sup>67</sup>

Die Xamarin.Essentials Erweiterung stellt Entwicklern das ‚Settingsplugin‘ bereit, welches die Sicherung von Präferenzen und App-Einstellungen erlaubt.<sup>68</sup> In Flutter wird für diese Speicherung mit Hilfe des Plugins ‚shared\_preferences‘ auf die gleichen plattform-spezifischen API zugegriffen.<sup>69</sup> Die Verwendung der gleichen APIs ist ein wichtiger Faktor da die von Anwendern gespeicherten Daten auch nach einem Frameworkwechsel zu Flutter, noch verfügbar sind.

---

<sup>63</sup>Vgl. Google LLC 2020t, Abgerufen am 17. April 2021.

<sup>64</sup>Vgl. SQLite Consortium 2020, Abgerufen am 17. April 2021.

<sup>65</sup>Vgl. MongoDB Inc. 2020, Abgerufen am 17. April 2021.

<sup>66</sup>Vgl. Tekartik 2020, Abgerufen am 17. April 2021.

<sup>67</sup>Vgl. Ward 2020, Abgerufen am 17. April 2021.

<sup>68</sup>Vgl. Microsoft Corporation 2019a, Abgerufen am 17. April 2021.

<sup>69</sup>Vgl. Google LLC 2020l, Abgerufen am 17. April 2021.

### 4.2.3 Navigation zu anderen Anwendungen

Mobile Anwendungen können die Möglichkeit zur Navigation zu anderen Apps realisieren. Dafür greift Xamarin.Forms auf ein bestimmtes Uniform Resource Identifier (URI)-Schema zurück, mit dem Ressourcen eindeutig bezeichnet werden. Durch den Befehl `Launcher.OpenAsync("mailto://")`, der Bestandteil der Xamarin.Essentials Erweiterung ist, wird z.B. das Standardprogramm für E-Mails des Smartphones gestartet.<sup>70</sup> Mithilfe des Plugins `url_launcher` kann die Funktionalität zum Öffnen von anderen Anwendungen zu Flutter Apps hinzugefügt werden.<sup>71</sup> Die unterstützten URI-Schemata und ihre Aktionen werden in Tabelle 4.1 dargestellt.

URI-Schemata	Aktion
<code>http:&lt;URL&gt;</code>	URL wird im Standardbrowser geöffnet
<code>mailto:&lt;E-Mail Adresse&gt;</code>	Erstellt eine Email an die angegebene E-Mail Adresse
<code>tel:&lt;Telefonnummer&gt;</code>	Ruft die Rufnummer an
<code>sms:&lt;Telefonnummer&gt;</code>	Schreibt eine SMS an die Rufnummer

Tabelle 4.1: Unterstützte Schemata des `url_launcher` Plugins

## 4.3 Lebenszyklus

Durch das Navigieren zu anderen Anwendungen ändert sich der Status von Ausführung im Vordergrund zu Ausführung im Hintergrund. Der aktuelle Status der App bestimmt die Handlungsoptionen. Befindet sich eine App im Vordergrund, hat sie die Aufmerksamkeit des Benutzers und ist damit priorisiert beim Zugriff auf die Systemressourcen einschließlich der CPU. Gleichzeitig muss eine Hintergrund-App möglichst inaktiv sein, da sie sich außerhalb des Bildschirms befindet. Mobile Anwendungen müssen auf Änderungen des Lebenszyklus-Status reagieren können, um ihr Verhalten entsprechend anzupassen.<sup>72</sup> Xamarin.Forms bietet dafür die drei Methoden `OnStart`, `OnResume` und `OnSleep` an, die aufgerufen werden, wenn sich der Status verändert.<sup>73</sup> Bei Flutter kann auf die `didChangeAppLifecycleState` Methode zurückgegriffen werden, die bei Änderungsereignissen ausgelöst wird und die ebenfalls die drei Lebenszyklen beinhaltet.<sup>74</sup>

<sup>70</sup>Vgl. Microsoft Corporation 2020e, Abgerufen am 17. April 2021.

<sup>71</sup>Vgl. Google LLC 2020n, Abgerufen am 17. April 2021.

<sup>72</sup>Vgl. Apple Inc. 2020, Abgerufen am 17. April 2021.

<sup>73</sup>Vgl. Microsoft Corporation 2020k, Abgerufen am 17. April 2021.

<sup>74</sup>Vgl. Google LLC 2020e, Abgerufen am 17. April 2021.

## 4.4 Ansichten

Ansichten (engl. Views) sind visuelle Elemente, die in zwei Kategorien unterschieden werden können. Steuerelemente (engl. Controls) sind für die Sammlung von Benutzereingaben oder die Ausgabe von Daten zuständig, Layouts beinhalten eine Sammlung von Ansichten und sind für ihre visuelle Anordnung auf der Benutzeroberfläche verantwortlich.<sup>75</sup> Im folgenden Abschnitt werden Xamarin.Forms- und Flutter-Ansichten gegenübergestellt. Damit soll ein Konzept für die Übersetzung von grafischen Benutzeroberflächen dargelegt und die Möglichkeiten des Austausches von Ansichten für den Compiler validiert werden.

### 4.4.1 Layouts

Ähnlich wie die Ansichten lassen sich auch die Layouts in zwei Kategorien unterteilen: Ansichtsseiten (engl. Pages) sowie die generellen Layouts. Die Pages nehmen den gesamten Bildschirm ein und werden in Abbildung 4.1 dargestellt.<sup>76</sup>

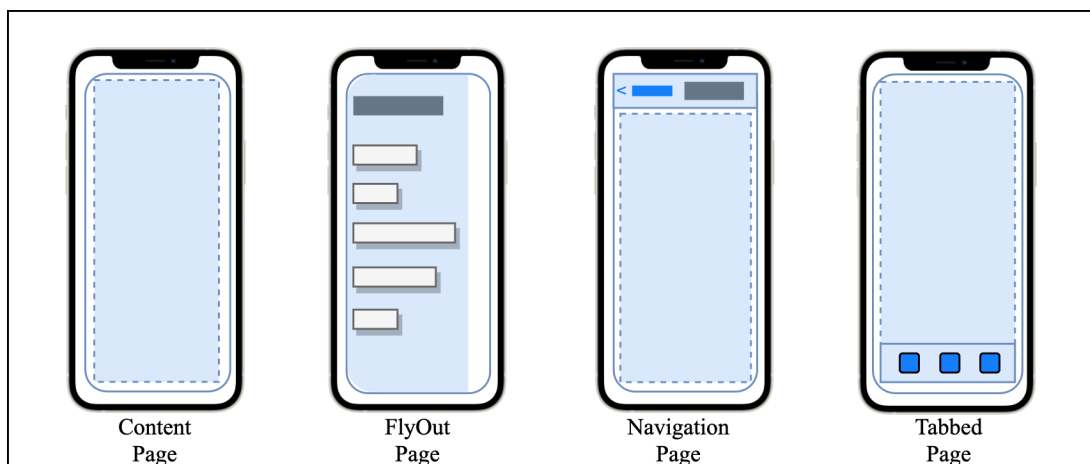


Abbildung 4.1: Xamarin.Forms Pages<sup>77</sup>

Die ‚ContentPage‘ ist ausschließlich für die Anzeige einer weiteren Ansicht verantwortlich. Die drei anderen Pages besitzen ein Navigationskonzept. Die ‚FlyOutPage‘ teilt den Bildschirm in zwei Bereiche, ein Bereich dient der Navigation. Er enthält ein Menü das, wie im Namen enthalten, einfliegen kann. Der zweite Bereich zeigt eine Detailansicht, in welcher der Inhalt der angeforderten Seite geladen wird. ‚NavigationPage‘ bietet eine Navigationsleiste, die einen Titel der aktuellen Seite und eine Navigationsschaltfläche beinhalten kann. ‚TabbedPage‘ stellt die unterschiedlichen Seiten als Registerkarten

<sup>75</sup>Vgl. Ritscher 2020, Abgerufen am 17. April 2021.

<sup>76</sup>Vgl. Microsoft Corporation 2016b, Abgerufen am 17. April 2021.

<sup>77</sup>Abbildung in Anlehnung an Microsoft Corporation 2016b, Abgerufen am 17. April 2021.

dar.<sup>78</sup> Die Ansichtsseiten befinden sich in der Regel innerhalb der XAML-Datei auf der untersten Ebene, dem so genannten Wurzelknoten. Der Quelltext 4.3 zeigt dies exemplarisch für eine ‚TabPage‘.

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <TabPage xmlns="http://xamarin.com/schemas/2014/forms"
3     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4     x:Class="MasterThesisSample.SampleTabPage">
5     <NavigationPage Title="Tab 1"/>
6     <NavigationPage Title="Tab 2"/>
7 </TabPage>

```

Quelltext 4.3: Xamarin.Forms ‚TabPage‘ Definition<sup>79</sup>

Es wird eine ‚TabPage‘ mit zwei Registerkarten entworfen. Eine Kombination mehrerer Navigationskonzepte ist möglich, das Beispiel zeigt eine Navigationsleiste innerhalb der Registerkarten.

Die verfügbaren Eigenschaften der Ansichtsseiten unterscheiden sich je nach Einsatzszenario. Im folgenden Quelltext 4.4 wird dies exemplarisch an der Realisierung einer ‚FlyoutPage‘ deutlich. Anders als bei der ‚TabPage‘, die aus einer Sammlung von Registerkarten besteht, finden sich im Quelltext die Eigenschaften ‚Flyout‘ und ‚Detail‘.

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <FlyoutPage xmlns="http://xamarin.com/schemas/2014/forms"
3     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4     x:Class="MasterThesisSample.SampleFlyoutPage">
5     <FlyoutPage.Flyout>
6         <ContentPage/>
7     </FlyoutPage.Flyout>
8     <FlyoutPage.Detail>
9         <NavigationPage/>
10    </FlyoutPage.Detail>
11 </FlyoutPage>

```

Quelltext 4.4: Xamarin.Forms ‚FlyoutPage‘ Definition<sup>80</sup>

Im Gegensatz zu Xamarin.Forms kann Flutter auf der Wurzelebene nur den Style der App, nicht aber ein Navigationskonzept definieren. Wie bereits in Kapitel 3 aufgeführt, wird in dieser Arbeit ausschließlich der Material Design Style unterstützt.<sup>81</sup> Quelltext 4.5 zeigt die Realisierung einer ‚Material Design‘ App in Flutter.

<sup>78</sup>Vgl. Microsoft Corporation 2016b, Abgerufen am 17. April 2021.

<sup>79</sup>Quelltext in Anlehnung an Microsoft Corporation 2020o, Abgerufen am 17. April 2021.

<sup>80</sup>Quelltext in Anlehnung an Microsoft Corporation 2020l, Abgerufen am 17. April 2021.

<sup>81</sup>Vgl. Google LLC 2020f, Abgerufen am 17. April 2021.

```

1 class MyApp extends StatelessWidget {
2   @override
3   Widget build(BuildContext context) {
4     return MaterialApp(
5       title: 'Flutter Demo',
6       theme: ThemeData(primarySwatch: Colors.blue,),
7       home: MyHomePage(title: 'Flutter Demo Home Page'),
8     );
9   }
10 }

```

Quelltext 4.5: Flutter ‚MaterialApp‘ Definition<sup>82</sup>

Der Vergleich zwischen den XML basierten XAML-Dateien und den bei Flutter verwendeten Dart-Dateien verdeutlicht die Unterschiede in den verwendeten Sprachen zur Benutzeroberflächenentwicklung. Die zentrale Idee hinter dem Flutter-Framework ist es, eine Benutzeroberfläche aus Widgets aufzubauen. Diese beschreiben das Aussehen der Anwendung basierend auf ihrem aktuellen Zustand. Sobald sich der Status ändert, kann das Framework den neuen mit dem alten Status vergleichen, um grafische Veränderungen möglichst effektiv vorzunehmen.<sup>83</sup> Um in Flutter ein Navigationskonzept zu definieren, können verschiedene Widgets verwendet und verschachtelt werden, wie in Quelltext 4.6 beispielhaft für eine App mit Registerkarten visualisiert wird.

```

1 appBar: AppBar(
2   bottom: TabBar(
3     tabs: [ Tab(icon: Icon(Icons.directions_transit)),
4             Tab(icon: Icon(Icons.directions_bike)), ],
5   ),
6   title: Text('Tabs Demo'),
7   ),
8   body: TabBarView(
9     children: [ Icon(Icons.directions_transit),
10                Icon(Icons.directions_bike), ],
11 ),

```

Quelltext 4.6: Flutter ‚Tab Layout‘ Definition<sup>84</sup>

Die deutlichen Unterschiede bei der Auswahl eines Navigationkonzeptes können überbrückt werden, indem zu jeder Xamarin.Forms Page das entsprechende Flutter Widget gefunden wird. Der Flutter-Widgetkatalog<sup>85</sup> und die Webseite "Flutter for Xamarin.Forms Developers"<sup>86</sup> wurde für die Recherche des Gegenstückes verwendet. Entsprechende Ergebnisse der Suche können in Tabelle 4.2 abgelesen werden.

<sup>82</sup>Quelltext in Anlehnung an Google LLC 2020s, Abgerufen am 17. April 2021.

<sup>83</sup>Vgl. Google LLC 2020j, Abgerufen am 17. April 2021.

<sup>84</sup>Quelltext in Anlehnung an Google LLC 2020r, Abgerufen am 17. April 2021.

<sup>85</sup>Vgl. Google LLC 2020q, Abgerufen am 17. April 2021.

<sup>86</sup>Vgl. Google LLC 2020f, Abgerufen am 17. April 2021.

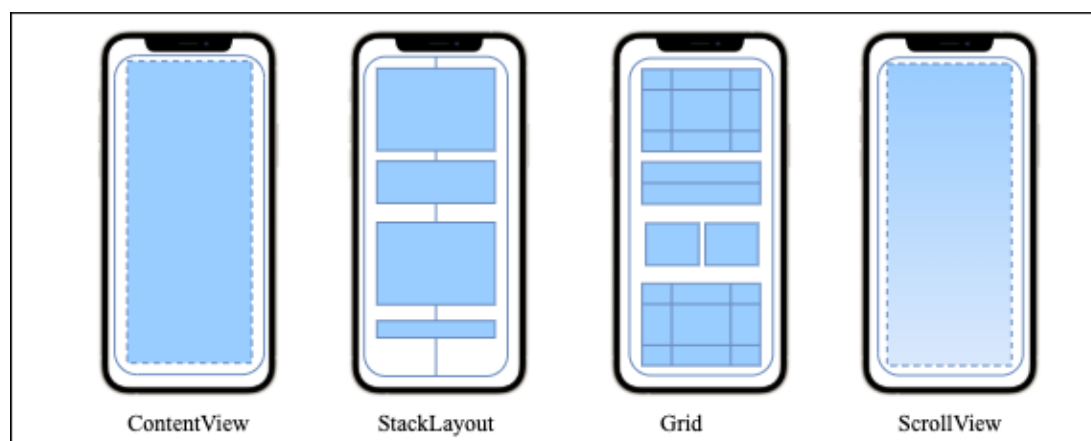
Xamarin.Forms Page	Flutter Widget
ContentPage	
FlyOutPage	MasterDetailScaffold
NavigationPage	Scaffold
TabbedPage	TabBar und TabBarView

Tabelle 4.2: Gegenüberstellung Pages

Gegenüberstellungen von Xamarin.Forms Elementen und Flutter Widgets in Tabellenform werden auch an anderer Stelle in diesem Kapitel zugunsten der Übersichtlichkeit verwendet. Flutter Widgets, die im Text nicht expliziert aufgeführt werden, sind den Xamarin.Forms Elementen in Funktionalität und Aussehen nahezu identisch. Eine vollständige Referenztabelle, die sich aus allen Einzelbetrachtungen zusammensetzt, befindet sich in Anhang I.

Die Navigation durch die Anwendung ist ein wichtiger Bestandteil und hängt wie beschrieben von der verwendeten Ansichtsseite ab. Die ‚NavigationPage‘ bietet eine hierarchische Navigation, bei der der Benutzer durch die Seiten vorwärts und rückwärts navigieren kann.<sup>87</sup> Flutter hat eine ähnliche Implementierung, welche die Widgets ‚Navigator‘ und ‚Routen‘ verwendet.<sup>88</sup>

Neben den Ansichtsseiten bietet Xamarin.Forms weitere Layouts, die Steuerelemente zu visuellen Strukturen zusammenstellen. Abbildung 4.2 präsentiert die gebräuchlichsten dieser Layouts.

Abbildung 4.2: Xamarin.Forms Layouts<sup>89</sup>

Die vorgestellten Layouts haben unterschiedliche visuellen Eigenschaften und dienen als Sprachelemente von XAML für den Entwurf von Benutzeroberflächen. ‚ContentView‘ enthält ein einzelnes untergeordnetes Ansichtselement und wird als Basisklasse

<sup>87</sup>Vgl. Microsoft Corporation 2020n, Abgerufen am 17. April 2021.

<sup>88</sup>Vgl. Google LLC 2020k, Abgerufen am 17. April 2021.

<sup>89</sup>Abbildung in Anlehnung an Microsoft Corporation 2018b, Abgerufen am 17. April 2021.

für benutzerdefinierte Darstellungen verwendet. Das Gestaltungselement ‚StackLayout‘ legt untergeordnete Elemente in einem entweder horizontal oder vertikal angeordneten Stapel ab. Ein ‚Grid‘ positioniert seine untergeordneten Elemente in einem Raster aus Zeilen und Spalten, es wird auch dafür verwendet, Layouts und Steuerelemente übereinander zu legen. Das ‚ScrollView‘ erlaubt das Verschieben von Bildschirminhalten und hat wie ein ‚ContentView‘ nur ein untergeordnetes Element. Neben diesen gängigen Layouts gibt es noch weniger verbreitete, zum Beispiel das ‚Frame‘, das einen Rahmen um ein visuelles Element zeichnet. Das ‚AbsolutLayout‘, platziert untergeordnete Elemente an bestimmten Positionen relativ zu ihrem übergeordneten Element. Das ‚RelativeLayout‘ übernimmt die gleiche Aufgabe, jedoch nur auf der Ebene des Layouts und untergeordneter Elemente.<sup>90</sup> Basierend auf diesen verfügbaren Layouts werden in Tabelle 4.3 die entsprechenden Flutter Widgets entgegengesetzt.

<b>Xamarin.Forms Layout</b>	<b>Flutter Widget</b>
AbsolutLayout	Positioned
ContentView	StatelessWidget
Frame	BoxDecoration
Grid	GridView oder Stack
ScrollView	SingleChildScrollView
StackLayout	Row und Column
RelativeLayout	Positioned

Tabelle 4.3: Gegenüberstellung Layouts

Widgets haben zum Teil erweiterte oder abweichende Funktionalitäten, sodass Optimierungen durch den Compiler notwendig sind. Damit das Layout ‚Grid‘ in Xamarin.Forms die Möglichkeit für einen Bildlauf bekommt, weil der Inhalt zu groß für die Darstellung auf einer Seite ist, wird das ‚Grid‘ in einem ‚ScrollView‘ verschachtelt. Dagegen bietet das ‚GridView‘ Widget von Flutter die Option des Scrollens automatisch an, wenn der Inhalt den sichtbaren Bereich überschreitet.<sup>91</sup> Im Rahmen der Codeoptimierung muss das ‚ScrollView‘ in diesem Anwendungsfall entfernt werden.

#### 4.4.2 Steuerelemente

Steuerelemente sind die sichtbaren Bausteine der Benutzeroberflächen, beispielsweise Schaltflächen, Beschriftungen und Textfelder. Microsoft kategorisiert die Steuerelemente innerhalb der Frameworkdokumentation anhand ihrer primären Verwendung.<sup>92</sup>

<sup>90</sup>Vgl. Microsoft Corporation 2018b, Abgerufen am 17. April 2021.

<sup>91</sup>Vgl. Google LLC 2020g, Abgerufen am 17. April 2021.

<sup>92</sup>Vgl. Microsoft Corporation 2020j, Abgerufen am 17. April 2021.

Diese Einteilung wird folgend übernommen, obwohl eine klare Abgrenzung der Steuerelemente zu Kategorien nicht uneingeschränkt möglich ist, da Einzelne zu mehreren Gruppierungen passen.

## Steuerelemente für die Präsentation

Einige Steuerelemente sind ausschließlich für die Darstellung von Inhalten vorgesehen. In Xamarin.Forms gibt es die folgenden Darstellungssteuerelemente, für die eine Flutter Repräsentation notwendig ist. Das Steuerelement ‚BoxView‘ zeigt in Xamarin.Forms ein einfarbiges Rechteck an. Für die Darstellung von Texten wird auf ‚Label‘ zurückgegriffen. Bilder können mit Hilfe des ‚Image‘ Steuerelements angezeigt werden, wobei diese aus verschiedenen Quellen, wie dem Web oder aus den Ressourcen der App geladen werden können. Das Steuerelement ‚Map‘ kann für die Anzeige von Karten innerhalb der mobilen Anwendung verwendet werden. Um Web und HTML Inhalte innerhalb einer App visualisieren zu können, steht das ‚WebView‘ Steuerelement bereit.<sup>93</sup> Für die Steuerelemente kann nun eine Gegenüberstellung zwischen Xamarin.Forms Elementen und Flutter Widgets vorgenommen werden, wie in Tabelle 4.4 dargestellt.

<b>Xamarin.Forms Steuerelement</b>	<b>Flutter Widget</b>
BoxView	SizedBox
Image	Image
Label	Text
Map	Leamaps oder Google Maps
WebView	webview_flutter
Ellipse	CustomPaint
Linie	CustomPaint
Path	CustomPaint
Polygon	CustomPaint
Polyline und Rectangle	CustomPaint
Rectangle	CustomPaint

Tabelle 4.4: Gegenüberstellung Darstellungssteuerelemente

Zu zeichnende Elemente, wie die ‚Ellipse‘, ‚Linie‘, ‚Path‘, ‚Polygon‘, ‚Polyline‘ und ‚Rectangle‘ wurden nicht gesondert aufgeführt, da diese bei Flutter auf die sogenannte Canvas der Benutzeroberfläche gezeichnet werden können.<sup>94</sup>

<sup>93</sup>Vgl. Microsoft Corporation 2018b, Abgerufen am 17. April 2021.

<sup>94</sup>Vgl. Google LLC 2020d, Abgerufen am 17. April 2021.



## Ereignisauslösende Steuerelemente

Xamarin.Forms ist ein Ereignisgesteuertes Framework. Die hier behandelten Steuerelemente stellen alle mindestens ein Ereignis zur Verfügung, das durch die in Kapitel 3 erwähnten Codebehind Klassen abonniert werden kann. Sobald ein sogenanntes Event ausgelöst wird, übermittelt das Framework diese Information an den Empfänger. Die folgenden Steuerelemente werden bei Xamarin.Forms der Kategorie ereignisauslösende Steuerelemente zugeordnet. ‚Buttons‘ sind rechteckige Objekte, die einen Text anzeigen und ein ‚clicked‘ Ereignis auslösen, nachdem sie von einem Anwender gedrückt wurden. Mit ‚ImageButton‘ steht ebenfalls eine Variante zur Verfügung, die ein Icon statt einem Text anzeigt. Bei einem ‚RadioButton‘ wird eine Option aus einer Reihe von Möglichkeiten ausgewählt und löst ein Ereignis aus, wenn sich die Benutzerauswahl ändert. Ein weiteres Steuerelement ist ‚RefreshView‘, das eine ‚PullToRefresh‘ Funktionalität für Layouts mit Bildlauf anbietet. Dabei wird durch das Herunterziehen des Seiteninhaltes der Wunsch zur Seitenaktualisierung übermittelt. Mithilfe der ‚SearchBar‘ haben Anwender die Möglichkeit, Inhalte innerhalb der App zu suchen. Nach der Eingabe von Textzeichenfolgen kann per Schaltfläche, oder Tastaturtaste, ein Ereignis ausgelöst und der eingegeben Text an die Codebehind-Datei weitergeleitet werden. Tabelle 4.5 zeigt die Ereignisauslösenden Steuerelementen von Xamarin.Forms und alternativen Flutter Widgets.

<b>Xamarin.Forms Steuerelemente</b>	<b>Flutter Widget</b>
Button	ElevatedButton
ImageButton	IconButton
RadioButton	RadioButton
RefreshView	pull_to_refresh
SearchBar	flutter_search_bar
SwipeView	flutter_slideable

Tabelle 4.5: Gegenüberstellung ereignisauslösende Steuerelemente

Flutter Widgets verhalten sich nicht exakt gleich wie die Steuerelemente von Xamarin.Forms. Ein Beispiel ist die hier erwähnte SearchBar, die bei Flutter im Gegensatz zu Xamarin.Forms nicht frei platzierbar ist, sondern immer in der Navigationsleiste angezeigt wird.

Die Beziehung zwischen Steuerelementen und Codebehind mittels Ereignissen wird in den beiden folgenden Quelltextausschnitten demonstriert. Der erste Ausschnitt zeigt XAML Quelltext, durch welchen ein Button dargestellt werden kann. Über die Eigenschaft clicked wird auf eine Methode in der XAML.cs Datei verwiesen, die in dem

zweiten Quelltextausschnitt abgebildet ist.

```

1 <Button Text="Click Me!"
2     Grid.Row="0"
3     Font="Large"
4     BorderWidth="1"
5     HorizontalOptions="End"
6     VerticalOptions="CenterAndExpand"
7     Clicked="Button_Clicked" />

```

Quelltext 4.7: Xamarin.Forms Button Initialisierung

```

1 private void Button_Clicked(object sender, EventArgs e)
2 {
3     //Button pressed
4 }

```

Quelltext 4.8: Xamarin.Forms Event Handler

## Steuerelemente zur Textmanipulation

In Xamarin.Forms stehen für die Arbeit mit Texten die Steuerelemente ‚Entry‘ zur Eingabe von einzelnen und ‚Editor‘ von mehreren Textzeilen bereit. Tabelle 4.6 zeigt die Gegenüberstellung zu Flutter Widgets.

Xamarin.Forms Steuerelemente	Flutter Widget
Entry	TextField
Editor	TextField

Tabelle 4.6: Gegenüberstellung textmanipulierender Steuerelemente

Wie in der Übersicht erkenntlich besitzt Flutter ausschließlich das Widget ‚TextField‘, das beide Funktionalitäten der Xamarin.Forms Steuerelemente bündelt. Standardmäßig bietet das ‚TextField‘ Widget die Eingabemöglichkeit für eine Zeile ähnlich dem ‚Entry‘ Steuerelement, kann aber durch das Setzen einer Eigenschaft erweitert werden. Dies wird in Quelltext 4.9 dargestellt.

```

1 TextField(
2     keyboardType: TextInputType.multiline,
3     maxLines: null,
4 )

```

Quelltext 4.9: Eingabefeld mit mehreren Zeilen in Flutter

## Steuerelemente zur Wertsetzung

Wertsetzung bedeutet das Ergänzen von Steuerelementen mit Eingaben durch den Anwender der App. Die folgenden Steuerelemente bietet Xamarin.Forms in dieser Kategorie an. Das ‚CheckBox‘ Steuerelement ermöglicht dem Benutzer die Auswahl eines booleschen Wertes (wahr, falsch). Die gleiche Funktionalität bei einem anderen visuellen Erscheinungsbild (siehe Abbildung 4.3) bietet der ‚Switch‘.



Abbildung 4.3: Darstellung der Steuerelemente ‚Checkbox‘ und ‚Switch‘

Die beiden linken Darstellungen der jeweiligen Steuerelemente zeigen den Zustand mit dem booleschen Wert falsch, die rechten wahr.

Ein ‚Slider‘ gewährt den Anwendern die Option, einen Wert aus einem kontinuierlichen Bereich, ein ‚Stepper‘ aus einem Bereich von inkrementellen Werten auszuwählen. Eine Datumsauswahl wird durch das ‚DatePicker‘ Steuerelement ermöglicht, die Zeitauswahl mit ‚TimePicker‘. Die Tabelle 4.7 präsentiert die gewohnte Gegenüberstellung von Xamarin.Forms Steuerlementen zu Flutter Widgets.

Xamarin.Forms Steuerelemente	Flutter Widget
CheckBox	Checkbox
Switch	Switch
Slider	Slider
Stepper	number_inc_dec
DatePicker	TextField mit Funktion
TimePicker	TextField mit Funktion

Tabelle 4.7: Gegenüberstellung wertsetzender Steuerelemente

Für die Steuerelemente ‚DatePicker‘ und ‚TimePicker‘ steht kein entsprechendes Widget zur Verfügung. Durch ‚TextField‘ und mithilfe einer Fingergeste wird eine Funktion aufgerufen, die den Auswahldialog für Datum und Uhrzeit öffnet und anschließend die Auswahl in das Textfeld einträgt. Quelltext 4.10 repräsentiert diese Funktion in Dart am Beispiel einer Zeitauswahl.

```

1 GestureDetector(
2     onTap: () async {
3         TimeOfDay picked = await showTimePicker(
4             context: context,
5             initialTime: TimeOfDay.now(),
6             builder: (BuildContext context, Widget child) {
7                 return MediaQuery(
8                     data: MediaQuery.of(context)
9                     .copyWith(alwaysUse24HourFormat: true),
10                    child: child,
11                );
12            },);
13    },
14    child: Text("SetTime",textAlign: TextAlign.center,)
15 );

```

Quelltext 4.10: Verwendung von Timepickern in Flutter<sup>95</sup>

Die gesetzten Werte können bei Xamarin.Forms aus den Codebehind Klassen abgefragt werden, um den Status eines Steuerelementes zu ermitteln. Das Abrufen von Informationen in Flutter wird von speziellen Widgets, beispielsweise dem ‚TextEditingController‘ durchgeführt.

## Aktivitätsandeutende Steuerelemente

In mobilen Anwendungen kann es aufgrund der limitierten Hardware Ressourcen und begrenzten Netzwerkanbindung zu zeitaufwendigen Aktionen kommen. Zur Visualisierung dieser Ladezeit stehen in Xamarin.Forms die folgenden Steuerelemente zur Verfügung. Der ‚ActivityIndicator‘ zeigt durch eine Animation, dass eine langwierige Aktivität ausgeführt wird, die ‚ProgressBar‘ kann mittels Ladebalken auch den Fortschritt darstellen. Die Tabelle 4.8 veranschaulicht die adäquaten Flutter Widgets.

Xamarin.Forms Steuerelemente	Flutter Widget
ActivityIndicator	CircularProgressIndicator
ProgressBar	LinearProgressIndicator

Tabelle 4.8: Gegenüberstellung aktivitätsandeutender Steuerelemente

<sup>95</sup>Quelltext in Anlehnung an Google LLC 2021g, Abgerufen am 17. April 2021.

## Sammlungsanzeigende Steuerelemente

Die überwiegende Anzahl von mobilen Anwendungen visualisiert Sammlungen von Daten.<sup>96</sup> Xamarin.Forms stellt hierfür ebenfalls Steuerelemente zur Verfügung. ‚Carousel-View‘ zeigt eine blätterbare Liste von Datenelementen an. ‚IndicatorView‘ stellt mithilfe von Indikatoren die Anzahl der Elemente in einer ‚CarouselView‘ dar. ‚Picker‘ bietet die Möglichkeit, eine Auswahl aus einer Sammlung zu entnehmen und anschließend in einem Textfeld auszugeben. Die Tabelle 4.9 zeigt die entsprechenden Flutter Widgets an.

<b>Xamarin.Forms Steuerelemente</b>	<b>Flutter Widget</b>
CarouselView	carousel_slider
IndicatorView	carousel_slider
Picker	flutter_material_pickers
TableView	Table

Tabelle 4.9: Gegenüberstellung sammlungsanzeigender Steuerelemente

## Listen

Listen sind Steuerelemente und dienen ebenfalls der Anzeige und Interaktion von Sammlungen. Aufgrund der langsamen Ladezeiten von ‚ListView‘ hat Microsoft im Jahre 2019 mit ‚CollectionView‘ ein zweites optimiertes Steuerelement für die Anzeige von Listen zur Verfügung gestellt. ‚SwipeView‘ erlaubt einzelne Reihen zur Seite zu schieben und darunter liegende Schaltflächen sichtbar zu machen. Tabelle 4.10 stellt die Listen aus Xamarin.Forms mit denen aus Flutter gegenüber.

<b>Xamarin.Forms Steuerelemente</b>	<b>Flutter Widget</b>
List	List
CollectionView	List
SwipeView	flutter_slideable

Tabelle 4.10: Gegenüberstellung Listen

Die Xamarin.Forms ‚ListView‘ ermittelt anhand einer Vorlage, wie eine Zeile dargestellt werden muss. Jede Reihe, die durch den Benutzer ausgewählt wird, löst ein Ereignis aus. Um dieses Verhalten in Flutter abzubilden, wird die Geste des Widgets in der Liste bereitgestellt.

<sup>96</sup>Vgl. Hindriks und Karlsson 2020, S. 180.

Damit sich die ListView im Falle von Änderung in der angezeigten Sammlung automatisch aktualisiert, ist es notwendig, die Daten in einer ‚ObservableCollection‘ vorzuhalten, da somit die Benutzeroberfläche über Änderungen informiert wird. Eine Möglichkeit, die ‚ListView‘ in Flutter zu aktualisieren, besteht darin, eine neue Instanz des Widgets zu erstellen und die Daten aus der alten in die neue Liste zu kopieren. Dieser Ansatz ist zwar einfach umsetzbar, aber für große Datensätze nicht zu empfehlen. Eine effektive Änderung für dynamische oder umfangreiche Listen ist mit dem ListView.Builder möglich.

### 4.4.3 Ausrichtung von Steuerelementen

Innerhalb von ‚Stacklayouts‘ können Xamarin.Forms Steuerelemente mit Hilfe von ‚HorizontalOptions‘ und ‚VerticalOptions‘ ausgerichtet werden. Für diese Eigenschaften können die folgenden Werte gesetzt werden:<sup>97</sup>

- ‚Start‘ positioniert die Ansicht bei einer horizontalen Ausrichtung an der linken Seite des übergeordneten Layouts, und bei vertikaler Ausrichtung am oberen Rand des übergeordneten Layouts.
- ‚Center‘ zentriert die Ansicht in der Mitte des übergeordneten Layouts.
- ‚End‘ platziert bei horizontaler Ausrichtung auf der rechten Seite und bei vertikaler Ausrichtung am unteren Rand des übergeordneten Layouts.
- ‚Fill‘ sorgt im Falle einer horizontalen Ausrichtung dafür, dass die Ansicht die Breite oder Höhe des übergeordneten Layouts ausfüllt.

Neben diesen Werten stehen auch noch Ausprägungen mit der Erweiterung ‚AndExpand‘ zur Verfügung. Die Werte ‚StartAndExpand‘, ‚CenterAndExpand‘, ‚EndAndExpand‘ und ‚FillAndExpand‘ werden verwendet, um die Ausrichtungspräferenz festzulegen und zu bestimmen, dass die Ansicht mehr Platz einnimmt, wenn dieser im übergeordneten ‚StackLayout‘ verfügbar ist.<sup>98</sup>

In Flutter kann mit den Eigenschaften ‚crossAxisAlignment‘ und ‚mainAxisAlignment‘ definiert werden, wie eine Zeile oder Spalte ihre untergeordneten Widgets ausrichtet. Bei einem ‚Row‘ Widget verläuft die Hauptachse horizontal und die Querachse vertikal bei dem ‚Column‘ Widget andersherum. Ähnlich wie bei Xamarin.Forms stehen auch

<sup>97</sup>Vgl. Microsoft Corporation 2017b, Abgerufen am 17. April 2021.

<sup>98</sup>Vgl. Microsoft Corporation 2017b, Abgerufen am 17. April 2021.

hier ‚Start‘, ‚End‘ und ‚Center‘ zur Verfügung.<sup>99</sup> Eine Zentrierung von Layouts in Flutter kann durch eine Kombination von zentrierten ‚Row‘ und ‚Column‘ Widgets erreicht werden. In der Praxis wird jedoch auf das ‚Center‘ Widget zurückgegriffen, welches die gleiche Aufgabe mit weniger Quelltext realisiert.<sup>100</sup>

#### 4.4.4 Gesten

Für die Interaktion mit der Benutzeroberfläche werden Gesten verwendet. Die Steuerelemente von Xamarin.Forms stellen Ereignisse für die häufig verwendeten Interaktionen bereit, wie im Unterpunkt ereignisauslösende Steuerelemente aufgeführt. Alternativ kann die Klasse ‚GestureRecognizer‘ verwendet werden, um seltenere Benutzerinteraktionen auf Ansichten zu erkennen, beispielsweise der Klick auf ein Steuerelement für die Darstellung.<sup>101</sup> In Flutter gibt es zwei ähnliche Möglichkeiten: Wird die Ereigniserkennung durch das Flutter-Widget, z.B. den ‚ElevatedButton‘, unterstützt, kann eine Funktion übergeben werden, in der eine Geste behandelt wird. Ist keine Ereigniserkennung mittels Widget möglich, kann es in einem ‚GestureDetector‘ verschachtelt werden, wie im Rahmen des ‚Timepickers‘ in Quelltext 4.10 dargestellt.<sup>102</sup>

#### 4.4.5 Animationen

Gut gestaltete Animationen machen eine Benutzeroberfläche intuitiver, tragen zum eleganten Erscheinungsbild einer ausgefeilten App bei und verbessern das Benutzererlebnis.<sup>103</sup> Xamarin.Forms enthält eine eigene Animationsinfrastruktur, die für die Erstellung einfacher Bildsequenzen unkompliziert, aber auch vielseitig genug ist, um komplexe Varianten zu erstellen. Die Klassen zielen auf verschiedene Eigenschaften von visuellen Elementen ab, wobei eine typische Animation eine Eigenschaft schrittweise, von einem Wert zu einem anderen, über einen bestimmten Zeitraum ändert.<sup>104</sup> In Flutter stehen viele Animationen, insbesondere für Material-Widgets, zur Verfügung. Sie sind mit den in ihrer Design-Spezifikation definierten Standard-Bewegungseffekten geliefert, wobei diese Effekte anpassbar sind.<sup>105</sup>

<sup>99</sup>Vgl. Google LLC 2021b, Abgerufen am 17. April 2021.

<sup>100</sup>Vgl. Google LLC 2021f, Abgerufen am 17. April 2021.

<sup>101</sup>Vgl. Microsoft Corporation 2020m, Abgerufen am 17. April 2021.

<sup>102</sup>Vgl. Google LLC 2020m, Abgerufen am 17. April 2021.

<sup>103</sup>Vgl. Google LLC 2020i, Abgerufen am 17. April 2021.

<sup>104</sup>Vgl. Microsoft Corporation 2020g, Abgerufen am 17. April 2021.

<sup>105</sup>Vgl. Google LLC 2020i, Abgerufen am 17. April 2021.

### 4.4.6 Übersetzungsbeispiel

Nach der Einführung aller verfügbaren Seiten, Layouts und Steuerelemente soll exemplarisch ein UI von Xamarin.Forms zu Flutter überführt werden. Als Beispielseite wurde die im linken Bereich der Abbildung 4.4 dargestellte Login-Seite mithilfe von Xamarin.Forms entwickelt. Der entsprechende XAML-Quelltext wird im rechten Bereich der Abbildung gezeigt. Die Login-Seite zeigt eine zentrierte Form mit zwei Eingabefeldern für Benutzernamen und Passwort und eine Schaltfläche für die Ausführung des Loginvorganges.



Abbildung 4.4: Darstellung einer exemplarischen Login-Page

Mithilfe der Gegenüberstellungen von UI-Elementen und Widgets, kann anschließend eine Transformation des Xamarin.Forms Layout-Baums zu dem Flutter Widget-Baum durchgeführt werden. Zur Verdeutlichung wird in Abbildung 4.5 eine Baumstruktur verwendet, welche die Verschachtelung innerhalb der Benutzeroberfläche visualisiert. Das übergeordnete ‚StackLayout‘ von Xamarin.Forms sorgt durch ‚CenterAndExpand‘ als Eigenschaftswert für eine zentrierten Darstellung. In Flutter übernimmt dies das ‚Center‘ Widget. Für alle Steuerelemente des obigen Quelltextes wird auf in diesem Kapitel verwiesene Widgets zurückgegriffen.



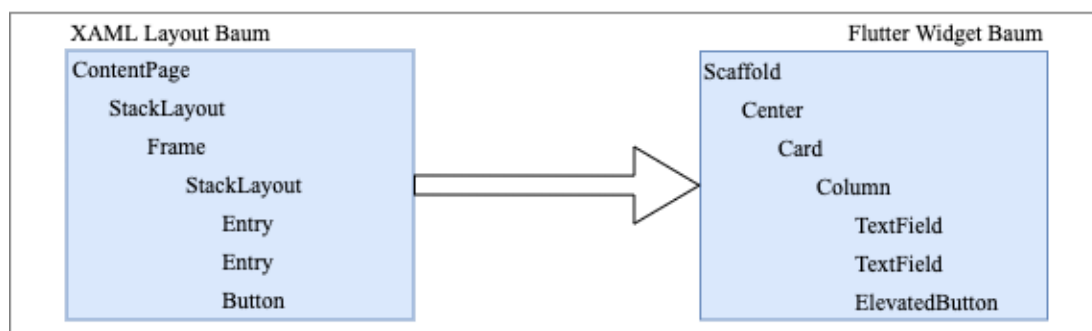


Abbildung 4.5: Layout-Baum Überführung von Xamarin.Forms zu Flutter

Um die Funktionsfähigkeit des Widget-Baums sicherzustellen, wird er innerhalb einer Flutter-App realisiert, siehe Quelltext 4.11, und das visuelle Ergebnis mit der in Abbildung 4.4 dargestellten Xamarin.Forms App verglichen.

```

1 @override
2 Widget build(BuildContext context) {
3   return Scaffold(
4     backgroundColor: Colors.grey,
5     appBar: AppBar( title: Text("LoginPage"),),
6     body: Center(
7       child: Card(
8         child: Column(
9           mainAxisAlignment: MainAxisAlignment.min,
10          children: <Widget>[
11            TextField(
12              obscureText: false,
13              decoration: InputDecoration(
14                border: OutlineInputBorder(),
15                labelText: 'Username',
16              ),
17            ),
18            TextField(
19              obscureText: true,
20              decoration: InputDecoration(
21                border: OutlineInputBorder(),
22                labelText: 'Password',
23              ),
24            ),
25            ElevatedButton(
26              child: Text('Login'),
27            ),
28          ],
29        ),
30      ),
31    );
32  };
33 }
  
```

Quelltext 4.11: Exemplarische LoginPage in Dart

Bei der Ausführung dieser App fällt auf, dass visuelle Unterschiede im Vergleich zu der vorher entwickelten Xamarin.Forms Variante existieren. Im oberen linken Teil der Abbildung 4.6 wird das zentrale Element des Login-Formulars dargestellt, wobei im Gegensatz zur Xamarin.Forms Variante die Abstände zwischen den Eingabefeldern fehlen. Außerdem ist der Login-Button nicht über die verfügbare Breite gestreckt. Abstände, die das ‚StackLayout‘ automatisch zwischen Elemente legt, müssen in Flutter durch das ‚Padding‘ Widget gesondert nachgestellt werden. Eine weitere Angleichung ist durch das Widget ‚AxisSize‘ durchzuführen, um die Breite des ‚ElevatedButtons‘ anzupassen. Somit kann der zentrale Bereich des Widget-Baums neu generiert werden, wie er im unteren linken Bereich der Abbildung 4.6 dargestellt. Der angepasste Quelltext, siehe hierzu auch Anhang II, erzeugt durch oben genannte Flutter Anpassungen eine Login-Ansicht mit dem im rechten Teil angezeigten UI, die der Xamarin.Forms Variante sehr nahekommmt.

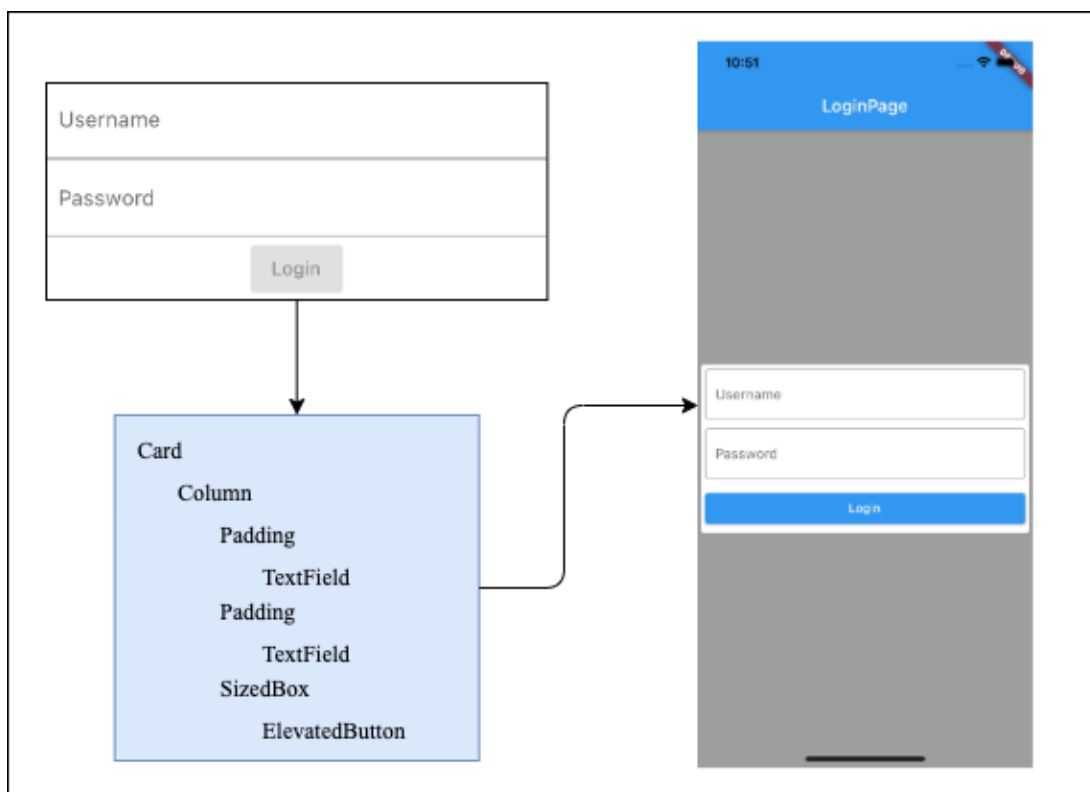


Abbildung 4.6: Flutter LoginPage Screenshot und Widget-Baum

Durch diese Validierung ist ein Beleg für die Annahme, dass der Austausch von UI-Elementen eine App mit vergleichbarer Benutzeroberfläche generieren kann, erbracht. Jedoch ist für die erfolgreiche Übersetzung von Xamarin.Forms nicht nur die Analyse der Ansichten von elementarer Bedeutung, sondern auch die Berücksichtigung der Eigenschaften und ihrer Kombinationen sowie der nicht im Quelltext ersichtlichen Standardwerte.

## 5 Unterschiede zwischen C# und Dart

Nach den in Kapitel 4 behandelten Unterschieden zwischen den Frameworks, wird in diesem Kapitel die Heterogenität zwischen den Hochsprachen C# und Dart behandelt. Durch den ähnlichen Stil und vergleichbare Syntax der beiden objektorientierten Programmiersprachen wird der Umstieg von Xamarin.Forms zu Flutter erleichtert.<sup>106</sup> Alle Unterschiede zwischen den Programmiersprachen können innerhalb des Quelltextes der mobilen Anwendung vorkommen und müssen daher bei der Übersetzung von Apps berücksichtigt werden.

### 5.1 Klassendesign

In diesem Abschnitt sollen die sogenannten Klassen und Objekte der objektorientierten Programmierung (OOP) betrachtet werden, die sowohl in C# als auch in Dart, allerdings mit vereinzelt Unterschieden bei der Realisierung, Verwendung finden. Die sogenannten Objekte werden durch bestimmte, charakteristische Merkmale beschrieben, die in der Klassendefinition festgelegt werden müssen.<sup>107</sup>

#### 5.1.1 Referenz- und Wertetypen

C# ist eine Sprache des ‚.NET Frameworks‘ und verwendet daher dessen Typ-System, welches zwischen Werte- und Referenztypen unterscheidet. Der Unterschied zwischen beiden ist in der Allokation des Systemspeichers zu finden. Eine Variable des Wertetyps enthält eine Instanz des Typs. Dies unterscheidet sich von einer Variablen des Referenztyps, der eine Adresse der Speicherzellen des Typs enthält.<sup>108</sup> C# bietet die

---

<sup>106</sup>Vgl. Pedley 2019, Abgerufen am 17. April 2021.

<sup>107</sup>Vgl. Witte 2013, S. 11f.

<sup>108</sup>Vgl. Kühnel 2019, S. 155f.

folgenden Wertetypen: ganzzahlig numerische Typen (Integer), fließkomma Typen (Float und Double), Wahrheitswerte (Boolean) und ein Zeichen (Char). Jeder Variable dieser Typen muss immer ein zum Typ passender Wert zugewiesen sein. Daraus folgt, dass zu jeder Zeit in einer Instanz des Typs Integer eine Zahl verwaltet werden muss.<sup>109</sup> Um die Bedingung dieser Wertpflicht zu umgehen, gibt es in der Hochsprache C# die sogenannte ‚nullable‘ Typen, die neben den zulässigen Werten zusätzlich den Wert ‚null‘ akzeptieren.<sup>110</sup>

Die Programmiersprache Dart war bis zu der Version 2.12 ‚nicht Null sicher‘. Da nicht gewährleistet war, dass Typen einen entsprechenden Wert zu jeder Zeit repräsentieren mussten, war zur Vermeidung von Laufzeitfehlern eine Null Prüfung, wie in Quelltext 5.1 dargestellt, bei der Arbeit mit Variablen erforderlich.

```
1 int myVariable = null;
2 if (myVariable != null)
3     myVariable = 0;
```

Quelltext 5.1: Null-Sicherheit in Dart 1.x<sup>111</sup>

Im März 2021 hat die Programmiersprache Dart den Support für Null-Sicherheit hinzugefügt. Diese gravierende Änderung für Softwareentwickler ist in der aktuellen Literatur und vielen Online-Ressourcen noch nicht veröffentlicht, muss jedoch beim Compiler-Entwurf berücksichtigt werden. Ab diesem Zeitpunkt ist ein Wert ‚null‘ nur noch möglich, wenn der Entwickler dies bewusst entscheidet, ansonsten benötigen Typen einen Wert. Mit dieser ‚Null Sicherheit‘ werden die Laufzeit-Nullreferenzfehler zu Analysefehlern, die während der lexikalischen Analyse auffallen und somit nicht mehr zwangsläufig zum Absturz der Anwendung führen.<sup>112</sup> Quelltext 5.2 visualisiert die Arbeit mit den neuen Datentypen in Dart.

```
1 int i = 42; // Inferred to be an int.
2 int? aNullableInt = null;
3 if (aNullableInt != null)
4     aNullableInt = 0;
```

Quelltext 5.2: Null-Sicherheit in Dart 2.x<sup>113</sup>

Seit dieser Veränderung verhält sich die Programmiersprache Dart analog zu C#. Auch ein Fragezeichen, das für die Definition eines nullable Typs verwendet wird, ist identisch. Spezielle Änderungen sind somit nicht mehr durch den Compiler an dieser Stelle vorzunehmen.

<sup>109</sup>Vgl. Microsoft Corporation 2020h, Abgerufen am 17. April 2021.

<sup>110</sup>Vgl. Bayer 2008, S. 167.

<sup>111</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>112</sup>Vgl. Google LLC 2021a, Abgerufen am 17. April 2021.

<sup>113</sup>Quelltext in Anlehnung an Google LLC 2021a, Abgerufen am 17. April 2021.

### 5.1.2 Datentypen

Die Unterscheidung von Referenz und Werttypen ist elementar für die Programmierung mit C#. Neben den bereits eingeführten Unterschieden bei der Speicherallokation besteht eine weitere Differenz bei der Initialisierung. Während Werttypen einfach der entsprechende Wert zugewiesen werden kann, muss bei Referenztypen explizit ein Objekt generiert oder ein bestehendes zugewiesen werden. Die Erzeugung eines neuen Objekts erfolgt durch den Operator ‚new‘.<sup>114</sup> Dart analysiert, wann ein neues Objekt initiiert werden muss und benötigt daher keinen ‚new‘ Operator, wie in Quelltext 5.3 dargestellt.<sup>115</sup>

```
1 var p1 = Point(2, 2);
2 var p2 = Point.fromJson({'x': 1, 'y': 2});
```

Quelltext 5.3: Objekterzeugung ohne ‚new‘ Keyword in Dart<sup>116</sup>

Folgend ist ein Vergleich der eingebauten Datentypen beider Sprachen in Tabelle 5.1 veranschaulicht.

Kategorien	C# Datentypen	Dart Datentypen
Ganzzahl	sbyte byte short ushort int uint long ulong	int BigInt
Fließkommazahl	double float decimal	double
Zeichenfolge	string	String
Textzeichen	char	
Datenfeld	Array	
Bool	bool	bool
Auflistung	List	List
Hashtabelle	Dictionairy	Map

Tabelle 5.1: Gegenüberstellung Datentypen

<sup>114</sup>Vgl. Eller und Kofler 2005, S. 93.

<sup>115</sup>Vgl. Google LLC 2020a, Abgerufen am 17. April 2021.

<sup>116</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

Es ist ersichtlich, dass sich einige, aber nicht alle Datentypen unterscheiden und für manche Typen in Dart eine entsprechende Repräsentation fehlt. Auf die für den Compilerbau relevanten Unterschiede wird im Folgenden genauer eingegangen.

## Ganzzahlen

Wie die Tabelle zeigt, bietet C# eine Vielzahl von Typen für die Arbeit mit Ganzzahlen. Diese haben einen Bereich von 8 bis 64 Bit und stehen jeweils mit und ohne Vorzeichen zur Verfügung. Dart besitzt keine vergleichbar umfangreiche Auswahl von Ganzzahl-Datentypen. Um Zahlen dennoch effizient im Hauptspeicher abzulegen, wird auf unterschiedliche interne Darstellungen zurückgegriffen, je nachdem, welcher Integer-Wert zur Laufzeit tatsächlich verwendet wird. Für besonders große Zahlenwerte steht jedoch der Datentyp ‚BigInt‘ zur Verfügung.<sup>117</sup> Der Compiler muss alle ganzzahligen C# Datentypen zu Dart ‚Integer‘, bis auf ‚long‘ und ‚ulong‘ die von ‚BigInt‘ repräsentiert werden, übersetzen.

## Fließkommazahlen

Fließkommazahlen stellen reelle Zahlen dar. In C# stehen dafür, wie in der Tabelle aufgeführt, verschiedene Datentypen zur Verfügung. Der ‚decimal‘-Typ behandelt Dezimalstellen am genauesten, während die Alternativen ‚double‘ und ‚float‘ nur Annäherungen an bestimmte reelle Zahl erreichen.<sup>118</sup> In Dart steht für die Arbeit mit Kommazahlen nur der Datentyp ‚double‘ zur Verfügung,<sup>119</sup> alle Fließkommazahlen müssen daher während der Kompilierung zu ‚double‘ umgewandelt werden.

## Textzeichen

In C# sind einzelne Textzeichen ein ‚Char‘. Eine Folge von Textzeichen bildet einen sogenannten ‚String‘. Dart stellt dagegen keinen Datentyp für einzelne Textzeichen zur Verfügung. Für eine entsprechende Darstellung wird auf den Datentyp ‚String‘ zurückgegriffen. Die ‚String‘ Klasse verfügt über einen Konstruktor, der einen ‚CharCode‘ als Übergabewert erwartet und das String Objekt mit einem Textzeichen erzeugt. Dies wird in Quelltext 5.4 dargestellt.

<sup>117</sup>Vgl. Star 2019, Abgerufen am 17. April 2021.

<sup>118</sup>Vgl. Microsoft Corporation 2020c, Abgerufen am 17. April 2021.

<sup>119</sup>Vgl. Google LLC 2021c, Abgerufen am 17. April 2021.

```
1 String CharContainingString;  
2 CharContainingString = String.fromCharCode([68]); // 'D'
```

Quelltext 5.4: Erstellung eines Strings mit einem Zeichen in Dart

## Datenfelder

Die vielleicht gebräuchlichsten Datensammlungen sind die sogenannten ‚Arrays‘ die auch als Datenfelder bezeichnet werden. Es sind geordnete Gruppen mit einer festen Anzahl von Objekten eines definierten Typs.<sup>120</sup> In Dart gibt es im Vergleich zu C# keine ‚Arrays‘. Alternativ wird der Datentyp ‚List‘ verwendet, sodass bei einer Übersetzung von C# eine Anpassung der entsprechenden Datentypen erfolgen muss. Die Arbeit mit Datenfeldern in Dart wird in Quelltext 5.5 dargestellt.

```
1 List<int> list = [5,10,20];  
2 print(list[1]); // 10
```

Quelltext 5.5: Datenfelder in Dart

## Auflistungen

Anders als die ‚Arrays‘ sind die Auflistungen in C# nicht statisch und kontinuierlich sondern dynamisch, was das Einfügen von Elementen vereinfacht. Diese werden von dem Datentyp ‚List‘ repräsentiert, wie es auch Dart macht. Der Datentyp ‚List‘ wird in Dart folglich sowohl für Datenfelder als auch für Auflistungen verwendet.

## Hashtabellen

Eine Hashtabelle ist ein Objekt, das Schlüssel und Werte miteinander verknüpft. Sowohl Schlüssel als auch Werte können beliebige Objekttypen sein. Jeder Schlüssel darf in einer Hashtabelle nur einmal vorkommen und ist unveränderlich, während Werte mehrfach verwendet werden können. C# bietet den Datentyp Dictionary für die Arbeit mit Hashtabellen, in Dart geschieht dies mittels Map.<sup>121</sup> Im Rahmen der Übersetzung muss der Typ Dictionary folglich durch Map ersetzt werden. Der folgende Quelltext zeigt die Verwendung einer Hashtabelle in der Programmiersprache Dart.

---

<sup>120</sup>Vgl. Kühnel 2019, S. 110f.

<sup>121</sup>Vgl. Google LLC 2020a, Abgerufen am 17. April 2021.

```
1 var Map = {'Username': 'Julian', 'Password': 'qwertz'};  
2 print(Map); //{Username: Julian, Password: qwertz}
```

Quelltext 5.6: Hashtabellen in Dart

### 5.1.3 Modifizierer

Alle Klassen und Eigenschaften verfügen in beiden Sprachen über eine Zugriffsebene, die steuert, ob Objekte von anderem Code verwendet werden können. Dabei wird in C# mithilfe der Schlüsselwörter `public`, `private`, `protected`, `internal`, `protected internal`, `private protected` der Zugriff geregelt. In Dart gibt es keine Schlüsselwörter, stattdessen wird mit einem Unterstrich (`_`) der Zugriff limitiert. Im Gegensatz zu C# wird der Modifizierer nicht vor dem Datentypen, sondern als Prefix vor dem Membernamen geführt. Dies wird in Quelltext 5.7 dargestellt.

```
1 class _PrivateClass {  
2     String _privateField;  
3 }  
4  
5 class PublicClass {  
6     String publicField;  
7 }
```

Quelltext 5.7: Private und Public Definitionen in Dart<sup>122</sup>

Unterstriche sind erlaubte Zeichen bei der Entwicklung von C# Klassen. Sie müssen bei der Übersetzung entfernt werden, da sonst fehlerhafte Übersetzungen drohen. Durch das Fehlen der anderen oben erwähnten Modifizierer können Berechtigungen in Dart nicht so feingranular definiert werden wie in C# . Der vom Compiler generierte Dart-Quelltext darf nicht mehr Zugriffe verweigern als der ursprüngliche Quelltext, daher kann es passieren, dass der Dart Quelltext weniger gute Restriktionen aufweist. Dabei fällt auf, dass der Zugriffsmodifizierer nicht mehr vor dem Typen steht sondern als Prefix vor dem Namen der jeweiligen Eigenschaft.

### 5.1.4 Vererbung

Erweiterungen oder Veränderung von Klassen entstehen durch die Vererbung, wobei aus einer bestehenden Basisklasse neue abgeleitete Klassen entwickelt werden können, die der Spezialisierung dienen.<sup>123</sup> Der folgende Quelltext zeigt eine Vererbung in Dart.

<sup>122</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>123</sup>Vgl. Microsoft Corporation 2020i, Abgerufen am 17. April 2021.



```
1 class BaseClass {  
2     void myFunction() {}  
3 }  
4  
5 class MyClass extends BaseClass {  
6     void myOtherFunction() {  
7         // do something  
8     }  
9  
10    // every function is overridable in Dart.  
11    @override  
12    void myFunction() {  
13        // do something  
14    }  
15 }
```

Quelltext 5.8: Vererbung in Dart<sup>124</sup>

Einer abgeleiteten Klasse kann exakt nur eine Basisklasse, also eine Elternklasse, zugeordnet werden, dies ist bei C# und Dart gleich definiert. Eine Mehrfachvererbung ist in C# auf Klassenebene nicht vorgesehen, kann aber durch Schnittstellenvererbung erfolgen. Dafür muss eine Implementierung von Attributen oder Methoden über Schnittstellen erfolgen.<sup>125</sup> Dart kennt im Gegensatz zu C# keine Schnittstellen, sondern verwendet stattdessen das Konzept der Mixins. Auch hier hat jede Klasse genau eine Elternklasse, jedoch kann ein Klassenkörper in mehreren Hierarchien vorkommen. Dies wird in Quelltext 5.9 visualisiert.

```
1 class MyMixin {  
2     void sayHello() => print("Hello!");  
3 }  
4  
5 class MyClass with MyMixin  
6 {  
7 }
```

Quelltext 5.9: Mixin's in Dart<sup>126</sup>

<sup>124</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>125</sup>Vgl. Kühnel 2019, S. 258f.

<sup>126</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

## 5.1.5 Übersetzungsbeispiel

Durch die herausgearbeiteten Unterschiede in der Klassenmodellierung kann exemplarisch eine C# Klasse zu Dart übersetzt werden, um einen Eindruck über die Arbeitsweise des Compilers zu vermitteln. Der folgende Quelltext veranschaulicht die Ausgangsklasse.

```
1 using System.Collections.Generic;
2
3 namespace SampleApp.Models
4 {
5     public class PublicExampleClass
6     {
7         // Konstruktor
8         public PublicExampleClass()
9         {
10             PublicChar = 'j';
11         }
12
13         // Ganzzahl
14         private int PrivateInt;
15
16         // Gleitkommazahl
17         private double PrivateDouble;
18         public float PublicFloat;
19
20         // Boolean
21         private bool PrivateBool;
22
23         // Textzeichen
24         public char PublicChar;
25
26         // Zeichenfolge
27         public string PublicString;
28
29         // Datenfeld
30         public char[] PublicCharArray;
31
32         // Auflistung
33         public List<int> _PublicIntList;
34
35         //Hashtabelle
36         public Dictionary<int, char> PublicHashTable;
37
38     }
39 }
```

Quelltext 5.10: Beispielklasse in C#

Dieser Quelltext kann nun zu Dart übersetzt werden, wie der folgende Code-Ausschnitt demonstriert.

```
1 class ExampleClass {
2     // Konstruktor
3     PublicExampleClass() {
4         PublicChar = String.fromCharCode(106);
5     }
6
7     // Ganzzahl
8     int _PrivateInt;
9
10    // Gleitkommazahl
11    double _PrivateDouble;
12    double PublicFloat;
13
14    // Boolean
15    bool _PrivateBool;
16
17    // Textzeichen
18    String PublicChar;
19
20    // Zeichenfolge
21    String PublicString;
22
23    // Datenfeld
24    List<String> PublicCharArray;
25
26    // Auflistung
27    List<int> PublicIntList;
28
29    // Hashtabelle
30    Map<int, String> PublicHashTable;
31 }
```

Quelltext 5.11: Beispielklasse in Dart

## 5.2 Namespaces

Namespaces werden häufig in C# -Programmen verwendet, um Klassen zu organisieren. Die meisten Klassen beginnen mit einem Abschnitt von using-Anweisungen, der die Namensräume einbindet. Dadurch können Entwickler auf enthaltene Methoden und Klassen zugreifen, ohne den vollqualifizierten Namen verwenden zu müssen.<sup>127</sup> Dart hat keine Namespaces, stattdessen werden Pakete und Dateien direkt importiert. Somit

<sup>127</sup>Vgl. Microsoft Corporation 2015c, Abgerufen am 17. April 2021.

kann ein direkter Zugriff auf alle Klassen und Funktionen innerhalb der Datei gewährt werden. Im Falle von Namenskonflikten, z.B. gleiche Klassennamen, können Dateien benannt werden. Quelltext 5.12 zeigt die Arbeit mit Paketen und Dateien in Dart.

```
1 // Import a core dart library
2 import 'dart:async';
3
4 // Import a package
5 import 'package:flutter/material.dart';
6
7 // Import another file in your application
8 import 'myfilename.dart' as filename;
```

Quelltext 5.12: Importieren von Paketen in Dart<sup>128</sup>

## 5.3 Generische Typen

Generics erlauben es Typen in .NET als Parameter zu übergeben. Dadurch lassen sich Klassen und Methoden designen, bei denen ein Typ durch Abstraktion verzögert übermittelt wird. So kann ein generischer Typparameter eingesetzt werden, um eine Klasse zu entwickeln, die von unterschiedlichen Methoden verwendet wird, ohne dass Kosten und Risiken durch Umwandlungsprozesse während der Laufzeit anfallen.<sup>129</sup>

Generics werden in Dart sehr ähnlich behandelt wie in C# allerdings muss keine Typ-Beschränkung übergeben werden.<sup>130</sup> Quelltext 5.13 zeigt die Implementation einer generischen State-Klasse in Dart.

```
1 class State<Type>
2 {
3   Type getValue() => null;
4 }
5 void processState(State state) {
6   dynamic value = state.getValue();
7 }
8 void processStateWithType(State<String> state) {
9   String value = state.getValue();
10 }
```

Quelltext 5.13: Generics in Dart<sup>131</sup>

<sup>128</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>129</sup>Vgl. Microsoft Corporation 2015b, Abgerufen am 17. April 2021.

<sup>130</sup>Vgl. Cheng 2019, S. 98.

## 5.4 Integrierte Verweistypen

Integrierte Verweistypen (engl. Delegates) referenzieren Methoden mit einer bestimmten Parameterliste und dem Rückgabotyp. ‚Delegates‘ werden dazu verwendet, Methoden als Parameter zu übergeben. Im ereignisgesteuerten Framework Xamarin.Forms werden die Methoden zur Behandlung von Ereignissen durch ‚Delegates‘ aufgerufen.<sup>132</sup>

In Dart kann der Typ ‚Typedef‘ verwendet werden, um eine Methodensignatur zu definieren und eine Instanz davon in einer Variablen vorzuhalten.<sup>133</sup> Dies wird in Quelltext 5.14 dargestellt.

```
1 class State<Type>
2 {
3   Type getValue() => null;
4 }
5 void processState(State state) {
6   dynamic value = state.getValue();
7 }
8 void processStateWithType(State<String> state) {
9   String value = state.getValue();
10 }
```

Quelltext 5.14: Delegates in Dart<sup>134</sup>

## 5.5 Asynchronität und Parallelität

Zur Verbesserung der Leistungsfähigkeit durch Reduktion von Blockierungen wurde das aufgabenbasierte asynchrone Programmiermodell bei C# eingeführt. Bei Synchronität wurde eine Anweisung erst komplett durchgeführt und beendet, bevor die Abarbeitung der nächsten begann. Diese strenge Reihenfolge wird durch die Asynchronität aufgehoben. Der Quelltext lässt sich trotz asynchroner Entwicklung weiterhin wie eine Folge von Anweisungen lesen, wird aber in der Praxis möglicherweise in einer deutlich komplizierteren Reihenfolge ausgeführt.<sup>135</sup>

In Dart wird auf die gleichen Schlüsselwörter ‚async‘ und ‚await‘ für die Realisierung von asynchronen Programmen zurückgegriffen. Dies wird in Quelltext 5.15 dargestellt.

<sup>131</sup> Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>132</sup> Vgl. Microsoft Corporation 2015a, Abgerufen am 17. April 2021.

<sup>133</sup> Vgl. Pedley 2019, Abgerufen am 17. April 2021.

<sup>135</sup> Vgl. Microsoft Corporation 2020b, Abgerufen am 17. April 2021.

```
1 void main() async {  
2   var result = await myFunction();  
3 }  
4  
5 Future<String> myFunction() {  
6   // Similar to Task.FromResult  
7   return Future.value('Hello');  
8 }
```

Quelltext 5.15: Async und Await in Dart<sup>136</sup>

Um die Vorteile von mehreren Prozessorkernen nutzen zu können, reicht dieses Konzept jedoch nicht aus. In Flutter müssen potentielle Hintergrundthreads, wie in C#, manuell verwaltet werden. Für den Start von rechenintensiven Arbeiten stehen die ‚Isolates‘ zur Verfügung, die dem Task Konzept aus C# ähneln. Dabei sind ‚Isolates‘ separate Ausführungsthreads, die sich keinen Speicher mit dem Hauptspeicher der Anwendung teilen und somit im Gegensatz zu ‚Task‘ nicht auf die Anwendung zugreifen können.

## 5.6 Bibliotheken

Klassenbibliotheken sind das Konzept der freigegebenen Bibliothek für .NET. Somit können nützliche Funktionalitäten auf Module verteilt und von mehreren Anwendungen verwendet werden.<sup>137</sup> Dart verfügt über eine Vielzahl von Kernbibliotheken, die für alltägliche Programmieraufgaben, wie das Arbeiten mit Objektsammlungen, das Durchführen von Berechnungen und das Kodieren sowie Dekodieren von Daten unerlässlich sind. Zusätzliche APIs sind in von der Community bereitgestellten Paketen verfügbar, die bereits im letzten Kapitel erwähnt wurden. Neben den analogen Konzepten ist auch der Inhalt einiger Bibliotheken vergleichbar. So ähnelt ‚dart:async‘ dem ‚System.Threading‘, ‚dart:Math‘ dem ‚System.Math‘ und ‚dart.io‘ dem ‚System.IO‘. Eine weitere Besonderheit von Dart ist, dass Funktionalitäten direkt Inhalt von Dateien sein können, ohne in einer Klasse oder Namespace verschachtelt zu sein.

### 5.6.1 Netzwerkaufrufe

Netzwerkaufrufe zur Datenabfrage vom Server oder Übermittlung von Benutzereingaben erfolgt in C# durch die Klasse HttpClient, Dart verwendet das http-Paket. Um eine

<sup>136</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>136</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>137</sup>Vgl. Microsoft Corporation 2016a, Abgerufen am 17. April 2021.

Netzwerkanfrage zu stellen, ist es in beiden Sprachen wichtig, die vorher eingeführten Schlüsselwörter ‚async‘ und ‚await‘ zu verwenden, damit die Benutzeroberfläche auch während der Anfrage reaktionsfähig bleibt. Ein Netzwerkanfrage in Flutter wird in 5.16 dargestellt.

```
1 import 'dart:convert';
2
3 import 'package:flutter/material.dart';
4 import 'package:http/http.dart' as http;
5 [...]
6 loadData() async {
7   String dataURL = "https://jsonplaceholder.typicode.com/posts";
8   http.Response response = await http.get(dataURL);
9   setState(() {
10     widgets = jsonDecode(response.body);
11   });
12 }
13 }
```

Quelltext 5.16: Flutter Network request

## 5.6.2 Kodierung und Dekodierung von Daten

Bei der Entwicklung von Apps wird häufig JavaScriptObjectNotation (JSON), ein kompaktes Format für den Austausch von Daten zwischen Anwendungen, verwendet.<sup>138</sup>

Kodierung, auch als Serialisierung bezeichnet, ist die Umwandlung einer Datenstruktur in eine Zeichenkette. Dekodierung, auch Deserialisierung, ist der umgekehrte Prozess, die Umwandlung einer Zeichenkette in eine Datenstruktur.<sup>139</sup> In C# kann für beide Prozesse der Namespace System.Text.Json verwendet werden,<sup>140</sup> während in Dart die Bibliothek dart:convert zur Verfügung steht.<sup>141</sup>

## 5.7 Ereignisse

Ein Ereignis ist eine Meldung, die von einem Objekt gesendet wird, um das Auftreten einer Aktion zu signalisieren. Dies wird beim ereignisgesteuerten Xamarin.Forms z.B. durch den Klick auf eine Schaltfläche ausgelöst. Events können in C# auch ohne

<sup>137</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.

<sup>138</sup>Vgl. Dobrenz, Gewinnus und Saumweber 2018, S. 684.

<sup>139</sup>Vgl. Google LLC 2021e, Abgerufen am 17. April 2021.

<sup>140</sup>Vgl. Microsoft Corporation 2020f, Abgerufen am 17. April 2021.

<sup>141</sup>Vgl. Google LLC 2021e, Abgerufen am 17. April 2021.

XAML-Dateien verwendet werden, wenn etwa die Programmlogik das Ändern eines Eigenschaftswerts verursacht. Das Objekt, von dem das Ereignis ausgelöst wird, ist der Ereignissender, dem die weitere Folge der Aktion nicht bekannt ist. Dart verwendet Streams, die ähnlich wie Ereignisse arbeiten, deren Verwendung in Quelltext 5.17 dargestellt.

```
1 StreamController streamController = new StreamController.broadcast();
2
3 void main() {
4     streamController.stream.listen((args) => {
5         // do something
6     });
7
8     streamController.add("hello");
9
10    streamController.close();
```

Quelltext 5.17: Events in Dart<sup>142</sup>

<sup>142</sup>Quelltext in Anlehnung an Pedley 2019, Abgerufen am 17. April 2021.



## 6 Realisierung des Source-To-Source Compilers

Gestützt auf das grundsätzliche Wissen über Compiler, die herausgearbeiteten Unterscheidungen von Xamarin.Forms und Flutter sowie die Differenzen der verwendeten Programmiersprachen C# und Dart, wird in diesem Kapitel die Realisierung des Source-To-Source Compilers beschrieben. Es bietet sich die Integrated development environment (IDE) (deutsch Entwicklungsumgebung) Visual Studio 2019 von Microsoft an, um das Projekt mit Roslyn Integration zu implementieren, denn auf dieser Plattform stehen Erweiterungen für Roslyn zur Verfügung. Die zu entwickelnde Projektmappe, besteht also aus dem Source to Source Compiler mit Roslyn Integration, der grafischen Benutzeroberfläche und einer Xamarin.Forms Anwendung als Testobjekt, wobei letztere jedoch erst im nächsten Kapitel thematisiert wird.

### 6.1 Programmablauf

Mithilfe des Roslyn Compilers kann der Source-To-Source Compiler Auswertungen zu den in der Xamarin.Forms App referenzierten Quelltextdateien durchführen. Somit ist es möglich eine Auflistung aller verfügbaren C# - Dateien aus einem Projekt zu extrahieren. XAML -Dateien werden nicht von Roslyn behandelt. Dies gelingt jedoch über den Umweg der Codebehind-Dateien mit Endung .XAML.cs, die ein Laden über das Dateisystem ermöglichen.

Bevor der Source-To-Source Compiler alle Quelltext- und Ansichtsdateien übersetzt, müssen die in Kapitel 4 beschriebenen Metadaten überführt werden. Anschließend kann der Quelltext optimiert und der Übersetzungsvorgang abgeschlossen werden. Zur Visualisierung des Übersetzungsvorganges kann nun ein Unified Modeling Language (UML) Aktivitätsdiagramm angelegt werden, wie es in Abbildung 6.1 dargestellt ist.



Abbildung 6.1: Aktivitätsdiagramm

Das dargestellte UML-Diagramm befindet sich aufgrund der Komplexität des Compilers auf einer hohen Abstraktionsebene. In den folgenden Abschnitten werden die einzelnen Aspekte der Kompilierung noch näher betrachtet.

## 6.2 Metadaten

Wie das UML Diagramm veranschaulicht, können die Metadaten von Android und iOS parallel in die Flutter Anwendung kopiert werden. Da die Metadaten plattformspezifische Eigenschaften der mobilen Anwendungen sind, wird im nächsten Abschnitt auf die entsprechenden Details der beiden Betriebssysteme eingegangen.

### 6.2.1 Android Metadaten

Zu den Android spezifischen Metadaten gehören die sogenannten Launcher Icons, die den Anwendern als App-Icon angezeigt werden. Xamarin.Android speichert diese grafischen Symbol in Ordnern, die die unterschiedlichen Pixeldichten der Android-Geräte unterstützen und in denen unter Umständen auch andere Bilder gespeichert sind. Innerhalb von Xamarin.Forms wird das ausgewählte Icon über die Klasse MainActivity.cs definiert, wie dies in Quelltext 6.1 dargestellt ist.

```
1 [Activity(Label = "TranspilerTestApp", Icon = "@mipmap/icon" ...]  
2 public class MainActivity : FormsAppCompatActivity
```

Quelltext 6.1: Xamarin.Forms Android Launcher-Icon Name

Nach der Extraktion des Namens können die entsprechenden Bilder kopiert werden. In der durch die Flutter SDK erzeugten App liegen bereits Bildplatzhalter für den Austausch bereit.

Der eindeutige Identifizierer, PackageID, zählt ebenfalls zu den Metadaten und dient zur eindeutigen Erkennung der Anwendungen auf dem mobilen Gerät und im GooglePlay Store. Die Information über die ID kann in Xamarin.Forms aus der Android-Manifest.XML Datei ausgelesen werden, und muss in Flutter sowohl in drei Manifest Dateien, als auch in die ‚build.gradle‘ Konfiguration geschrieben werden. Das Plugin ‚change\_app\_package\_name‘ nimmt alle notwendigen Änderungen vor. Es ist als Abhängigkeit zum Projekt hinzuzufügen und anschließend über die Kommandozeile auszuführen.

Der Anwendungsname wird, wie die PackageID, aus dem AppManifest extrahiert und in Flutter allerdings nur in eine Manifest Datei im Verzeichnis `Project/app/src/main` geschrieben. Die Berechtigungen, die während der Laufzeit der Anwendung beantragt werden, befinden sich ebenfalls innerhalb des AppManifests und können kopiert werden.

## 6.2.2 iOS Metadaten

Xamarin.iOS verwaltet die für die Übersetzung notwendigen Metadaten in einer Datei im Projektverzeichnis mit dem Namen `Info.plist`. In Flutter Apps gibt es eine entsprechende Datei, jedoch werden die Inhalte, beispielsweise die Bildnummer und der Identifizierer, aus Variablen geladen. Damit nach der Kompilierung kein erhöhter Administrationsaufwand bei der Verwaltung dieser Eigenschaften entstehen, soll dieses Schema beibehalten werden. Die `AppFrameworkInfoPlist` stellt die Quelle aller Werte, die als Variablen in die `Info.plist`, geladen werden müssen dar. Daher werden die entsprechenden Einträge aus der `Info.Plist` von Xamarin extrahiert und jeweils in die entsprechenden `Info.Plist` Dateien von Flutter kopiert. Tabelle 6.1 verdeutlicht die Zielformat für die Kopie bestimmter Werte.

Schlüssel	Info.plist	AppFrameworkInfo.plist
DevelopmentRegion		✓
ShortVersionString		✓
Version		✓
MinimumOSVersion		✓
InterfaceOrientations	✓	
BundleSignature	✓	
BundleName	✓	

Tabelle 6.1: Plist Dateien in Flutter

Die Übersicht weist nicht alle aus der iOS Entwicklung bekannten Schlüssel auf. Alle fehlenden Eigenschaften gehören in die Datei `Info.Plist`, wie die letzten drei genannten. Für das Anwendungsicoon kann das Verzeichnis `Assets.xcassets` aus dem Stammverzeichnis der Xamarin.Forms iOS App in das Verzeichnis `ios/flutter/Runner`, das alle notwendigen Bilder enthält, kopiert werden.

## 6.3 Ressourcen

Neben den Metadaten ist die Ressourcenkopie der mobilen Anwendung von Nöten. Dazu gehören die innerhalb der App angezeigten Bilder, die bei Xamarin.Forms üblicherweise innerhalb der plattformspezifischen Anwendungen abgelegt sind. Von dort aus kopiert der Compiler die Bilder aus dem iOS Projekt in das Flutter Projekt zur spätere Verwendung. Da für Bilder im Flutter Framework keine plattformspezifische Unterscheidung vorgesehen ist, werden ausschließlich die von iOS entnommen. Eine Entscheidung zugunsten der iOS Alternativen wurde entschieden, da sich die Konzepte von Xamarin.iOS und Flutter im Bezug auf Bilderressourcen ähneln und eine automatisierte Überführung möglich ist. Android spezifische Varianten werden während der Übersetzung also nicht berücksichtigt. Zur Speicherung und Darstellung höher aufgelöster Bilder existieren zwei Unterordner mit den Namen 2.0x und 3.0x im Verzeichnis Assets. Ressourcen werden je nach Skalierungsoption @2x oder @3x in den entsprechenden Verzeichnissen und ohne im Ordner Asset abgelegt. Neben dem differenzierten Vorgehen bei der Kopierung ist es erforderlich die Ressourcen in der pubspec.yaml zu referenzieren. Ein Ausschnitt aus der Pubspec.yaml, der das Einbinden von Bildern demonstriert, wird in Quelltext 6.2 dargestellt.

```
1  assets:  
2    - images/lake.jpg  
3    - images/sun.jpg
```

Quelltext 6.2: Referenzierung von Bildern in der Pubspec.yaml

Benutzerdefinierte Schriftsätze müssen ebenfalls aus dem iOS Verzeichnis kopiert und anschließend in der pubspec.yaml Datei hinzugefügt. Hier gilt, analog zu den Bildern, dass die kompilierte Flutter Anwendung ausschließlich die Schrift der iOS App verwendet und androidspezifische Schriftsätze verloren gehen.

Weitere Ressourcen bilden die Startbildschirme der verschiedenen Plattformen, die während des Ladens der App angezeigt werden und für eine Veröffentlichung im Appstore verpflichtend sind. Durch ihren simplen Aufbau entsteht keine zusätzliche Ladezeit und ihr Quelltext wird plattformspezifischen realisiert. Der Compiler verarbeitet ausschließlich einzelne Bilder, die während des Ladens der mobilen Anwendung angezeigt werden und verzichtet auf die Übersetzung des plattformspezifischen Programmcode wie in den Ausschlusskriterien erwähnt. Da die Xamarin.Forms App nicht zwangsläufig ein Bild in einem entsprechenden Format vorhält, greift der Flutter Startbildschirm auf das LauncherIcon der jeweiligen Plattformen zurück.

## 6.4 Übersetzung von Klassenstrukturen

Die Übersetzung von C# zu Dart ist ein essentieller Bestandteil des Compilers. Dafür wird zuerst die Visual Studio Projektmappe geladen. Anschließend kann fokussiert das Projekt betrachtet werden, welches den plattformunabhängigen Quelltext beinhaltet. Da dieser selbst, wie bereits in den Ausschlusskriterien erwähnt, unübersetzt bleibt, sind die entsprechenden Projekte ausschließlich für das Kopieren der Metadaten und Ressourcen notwendig. Alle Klasse durchlaufen die in Quelltext 6.3 dargestellten Ausführungsschritte.

```
1 var root = await d.GetSyntaxRootAsync();  
2  
3 var model = await d.GetSemanticModelAsync();  
4  
5 var visitor = new FlutterTranspilerVisitor(model);  
6 var res = visitor.Run(root);
```

Quelltext 6.3: C# Compiler Frontend

Für die Übersetzung von einer Programmiersprache zu einer anderen ist es notwendig, alle Knoten und Token innerhalb eines Syntaxbaums in der richtigen Reihenfolge zu betrachten. Die abstrakte Klasse ‚CSharpSyntaxWalker‘ erlaubt es, einen eigenen ‚Syntax-Walker‘ zu konstruieren, der die Knoten und Token analysiert. Dafür kann von ‚CSharpSyntaxWalker‘ geerbt und folgend die ‚Visit()‘ Methode überschrieben werden.<sup>143</sup> Der Quelltext 6.3 visualisiert die Erstellung eines ‚FlutterTranspilerVisitors‘, mithilfe des vorher ermittelten semantischen Models. Nun wird der ausgelesene Wurzelknoten des Syntaxbaumes als Parameter der ‚Run()‘ Methode übergeben, die den ‚Visitor‘ startet.

Hiermit endet das Compiler Frontend und es beginnt die Konstruktion des Dart Programmcodes. Dafür werden die einzelnen Knoten des Baumes je nach ihrem Typen zu entsprechenden Dart Synonymen umgewandelt und die einzelnen Methoden des ‚CSharpSyntaxVisitor‘ überschrieben. Quelltext 6.4 zeigt die Umwandlung von Eigenschaftsdeklarationen.

<sup>143</sup>Vgl. Varty 2014, Abgerufen am 17. April 2021.

```
1 public override void VisitPropertyDeclaration(PropertyDeclarationSyntax node)
2 {
3     var name = ToCamelCase(node.Identifier.Text);
4     var t = TranslateType(node.Type);
5
6     Write(t);
7     WriteModifiers(node.Modifiers);
8     Write(name);
9
10    if (node.Initializer != null)
11    {
12        Write(" = ");
13        Visit(node.Initializer.Value);
14    }
15
16    Write(";");
17
18    NewLine();
19 }
```

Quelltext 6.4: Compilierung von Eigenschaftsdeklarationen

Der Quelltextausschnitt veranschaulicht die generelle Arbeitsweise des Compilers, der einzelne Zeichen und Zeilen des Dart-Textdokumentes generiert und speichert. So handelt sich der Compiler durch das Dokument und übersetzt den Quelltext Schritt für Schritt, wobei die Reihenfolge, in der der Typ, der Modifizierer und der Name der Eigenschaft in die Dart Datei geschrieben wird, von entscheidender Bedeutung ist. Die Quelltextzeilen 6 bis 8 geben zuerst den Eigenschaftstyp gefolgt von dem optionalen Modifizierer und dem Namen aus. Obligatorisch kann nun ein Wert gesetzt werden bevor ein Semikolon die Deklaration beendet. Die Methode für die Generierung des Dart Modifizierers zeigt Quelltext 6.5.

```
1 private void WriteModifiers(SyntaxTokenList modifiers)
2 {
3     if (modifiers.Contains("private"))
4     {
5         Write("_");
6     }
7     if (modifiers.Contains("abstract"))
8     {
9         Write("abstract ");
10    }
11 }
```

Quelltext 6.5: Austausch von C# Modifizierern

Wie bereits in Kapitel 5 beschrieben, sind die Typen der beiden Programmiersprachen

nicht identisch und müssen daher entsprechend angepasst werden. Zur Veranschaulichung wird die realisierte Methode in Quelltext 6.6 dargestellt.

```
1 public static string GetKnownName(string name)
2 {
3     switch (name)
4     {
5         case "Void" : return "Void";
6         case "Int"  : return "Int";
7         case "byte" : return "Int";
8         case "short" : return "Int";
9         case "long" : return "BigInt";
10        case "double" : return "Double";
11        case "float" : return "Double";
12        case "char" : return "String";
13        case "string" : return "String";
14        case "List" : return "List";
15        case "Dictionary" : return "Map";
16        case "bool" : return "Boolean";
17        default: return name;
18    }
19 }
```

Quelltext 6.6: Austausch von C# Datentypen

Der Rückgabewert dieser Methode ist der entsprechende Dart Datentyp. Sollte keine entsprechende Repräsentation verfügbar sein, erfolgt die Rückgabe des ursprünglichen Typnamens. Dies ist notwendig um andere übersetzte Klassen im Quelltext verwenden zu können. Damit keine unerwünschten Typen des .Net Frameworks oder von Xamarin.Forms Erweiterungen innerhalb des Übersetzungsergebnisses erscheinen sind Bereinigungen erforderlich.

Erweiterungen ergänzen die Funktionalität der Frameworks sind jedoch herstellerabhängig mit unterschiedlichen Anforderungen programmiert, sodass es für Klassen und Methoden aus diesen Bibliotheken nicht zwangsläufig identische Repräsentationen gibt. Eine Auflösung der Inkompatibilität ist nicht mittels Automation möglich, sondern bedarf unterschiedlich umfangreicher manueller Umwandlungen. Der Komplexität ist es geschuldet, dass der Compiler nur eine Auswahl von Erweiterungen unterstützt. Nachträgliche Erweiterungen des Funktionsumfanges sind möglich. Die folgenden Anwendungsfälle skizzieren das Vorgehen.

Die Auswahl von Bildern aus der Galerie und die Aufnahme von neuen über die Kamera werden von vielen mobilen Anwendungen genutzt. Im folgenden wird der Quelltext dargestellt, der die Funktionalität des Xamarin.Forms Essentials mit dem Flutter Plugin ‚image\_picker‘ austauscht. Die notwendigen Berechtigungen wurden bereits während



der Übernahme der Metadaten gesetzt und können daher an dieser Stelle ignoriert werden.

```

1 await Cmd.ExecuteCMDCommand(TargetPath,
2     "/C \"flutter pub pub add image_picker\"");
3 await DependencyManager.ImportStatement(File,
4     "import 'package:image_picker/image_picker.dart';");
5
6 expectedResult = expectedResult.Replace("MediaPicker.capturePhotoAsync()",
7     "await picker.getImage(source: ImageSource.camera);");
8 expectedResult = expectedResult.Replace("MediaPicker.pickPhotoAsync()",
9     "await picker.getImage(source: ImageSource.gallery);");
10 await DartFileManager.UpdatePlaceholder(File, "%PROPERTIES%",
11     "final picker = ImagePicker();");

```

Quelltext 6.7: Austausch des Mediaplugins

Ein weiterer wichtiger Anwendungsfall, der nur mit Hilfe von Erweiterungen implementiert werden kann ist der Zugriff auf die Smartphone Sensoren. Der folgende Quelltext zeigt den Austausch der Beschleunigungssensorfunktionen des Xamarin.Essentials Plugins mit Hilfe des Flutter Sensor Plugins. Der Compiler übersetzt ebenfalls die Funktionalität des Gyroskops, siehe Programmcode `Ist:Sensor`, da der Quelltext jedoch analog ist, wird dieser an dieser Stelle nicht explizit aufgeführt.

```

1 int startIndex = expectedResult.IndexOf("Accelerometer.ReadingChanged +=");
2 if (startIndex != -1)
3 {
4     endIndex = Code.IndexOf("\n", startIndex);
5
6     Code = expectedResult.Substring(
7         startIndex + "Accelerometer.ReadingChanged += ".Length ,
8         endIndex - startIndex - "Accelerometer.ReadingChanged += ".Length-1);
9     Code = Code.Replace("Accelerometer.start(speed)", "");
10    Code = Code.Replace("SensorSpeed speed = SensorSpeed.UI;", "");
11    Code = Code.Replace("Accelerometer.ReadingChanged += " + EventName ,
12        DartFileManager.ReadResource("Sensors.Accelerometer.txt"));
13
14    Code = Code.Replace("%METHOD%", EventName);
15
16    Code = Code.Replace("Reading.Acceleration.X", "x;");
17    Code = Code.Replace("Reading.Acceleration.Y", "y;");
18    Code = Code.Replace("Reading.Acceleration.Z", "z;");
19
20    DependencyManager.AddDependencie("sensors");
21    await DependencyManager.ImportStatement(File,
22        "import 'package:sensors/sensors.dart';");
23 }

```

Quelltext 6.8: Austausch des Beschleunigungssensorplugins

Obwohl es keine Gewährleistung gibt, dass für jede Xamarin Forms eine entsprechende Flutter Erweiterung zur Verfügung steht, konnte bei der Implementierung des Compilers immer eine Alternative gefunden werden.

## 6.5 Übersetzung von Ansichten

Xamarin.Forms Ansichten bestehen wie bereits beschrieben aus den XAML und XAML.CS Dateien, die nach der jeweiligen Übersetzung in einer gemeinsamen Dart Datei synthetisiert werden. Die Kompilierung der beiden Dateien geschieht jeweils durch ein dediziertes Compiler Frontend, dass die Aufgaben bis zur Zwischendarstellung übernimmt. Darauf aufbauen wird ein gemeinsames Compiler-Backend für die Zusammenführung und die Code-Optimierung entwickelt. Die Abbildung 6.2 veranschaulicht den Vorgang.

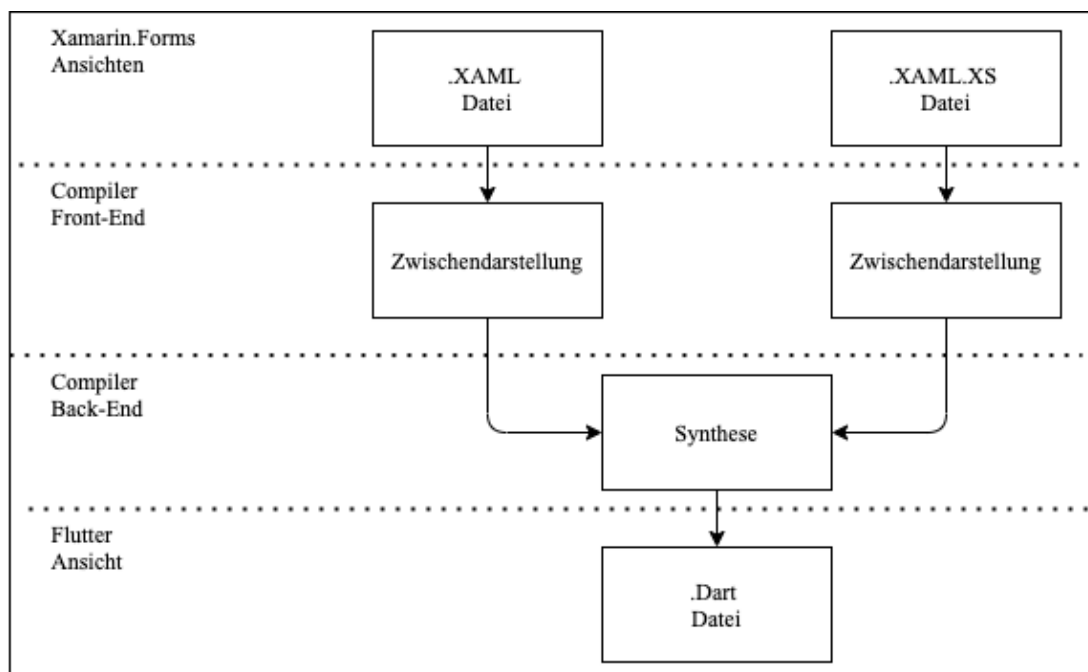


Abbildung 6.2: Compiler Phasen für Ansichten

### 6.5.1 Visuellen- Zwischendarstellung

Für die visuelle Zwischendarstellung wird in einem ersten Schritt die XML Struktur aus der XAML Datei ausgelesen und ein Syntaxbaum angelegt. Die entsprechenden Klassen für dessen Behandlung sind in Abbildung 6.3 in Form eines Klassendiagramms veranschaulicht.

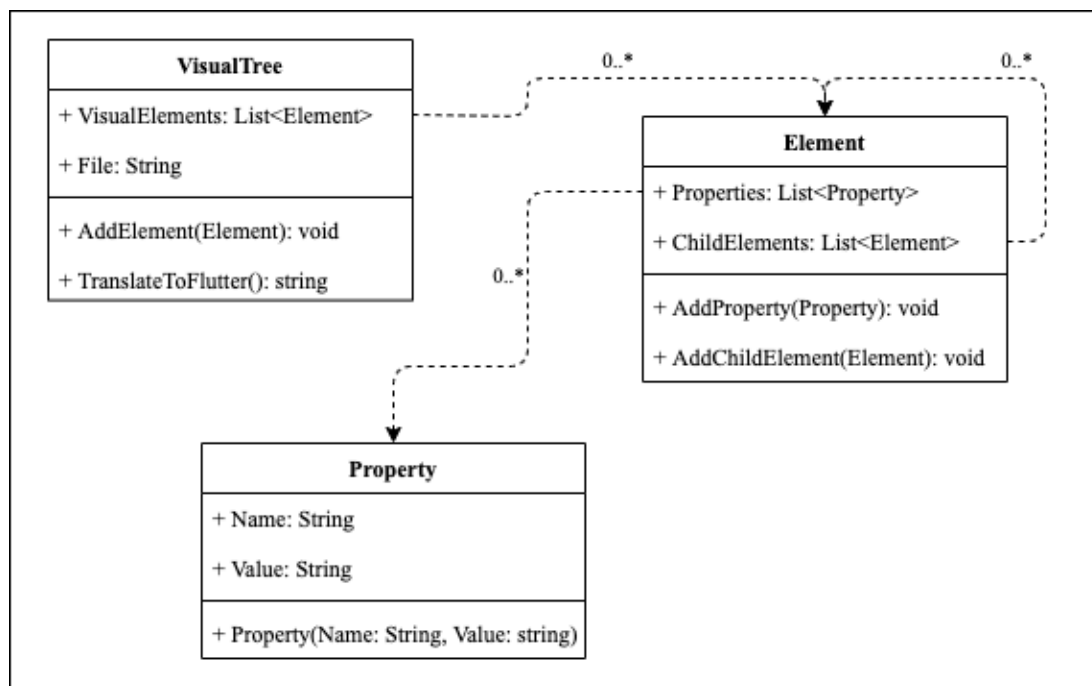


Abbildung 6.3: Klassendiagramm

Ein Objekt der Klasse **VisualTree** stellt den Layout-Baum einer speziellen XAML Datei dar. Er beinhaltet eine Auflistung von untergeordneten Elementen, die weitere Elemente beinhalten können, sodass eine Verschachtelung entsteht. Jedes Element besitzt eine Auflistung von Eigenschaften, die Attribute der einzelnen XML Knoten beschreiben. Der Xamarin.Forms Layout- Baum wird, durch die in Kapitel 4 beschriebenen Repräsentationen von Flutter Widgets, in einen Flutter Widget-Baum überführt und für jedes Widget der Quelltext zur Initialisierung in einem Template vorgehalten. Alle Eigenschaften besitzen in dieser Vorlage vorgefertigte Platzhalter wie in Quelltext 6.9 dargestellt.

```

1 Text(
2   "%TEXT%",
3   textAlign: %HORIZONTALALIGNMENT%,
4   overflow: TextOverflow.ellipsis,
5   style: TextStyle(
6     fontWeight: %FONTWEIGHT%,
7     fontSize: %FONTSIZE%,
8     fontStyle: %ITALIC%
9   ),
10 )

```

Quelltext 6.9: Flutter Widget Template

Aufgrund von Inkompatibilitäten zwischen den von Xamarin.Forms gelieferten und von Flutter erwarteten Eigenschaftswerten ist es notwendig diese umzuwandeln. Der folgende Quelltext 6.10 zeigt die Umwandlung von Textgrößen. Xamarin.Forms erwartet diesen Wert entweder als Zahl oder als sogenannte benannte Größe wie **Large** oder

Title, während Flutter ausschließlich Textgrößen mit dem Typen Double akzeptiert.

```
1 private static double GetFontSize(object _SizeValue)
2 {
3     if(_SizeValue is double)
4     {
5         return Convert.ToDouble(_SizeValue);
6     }
7     string NamedSize = (String)_SizeValue;
8
9     if (NamedSize.Equals("Caption", StringComparison.OrdinalIgnoreCase))
10        return 12;
11    if (NamedSize.Equals("Micro", StringComparison.OrdinalIgnoreCase))
12        return 10;
13    if (NamedSize.Equals("Header", StringComparison.OrdinalIgnoreCase))
14        return 34;
15    if (NamedSize.Equals("Large", StringComparison.OrdinalIgnoreCase))
16        return 34;
17    if (NamedSize.Equals("Medium", StringComparison.OrdinalIgnoreCase))
18        return 14;
19    if (NamedSize.Equals("Small", StringComparison.OrdinalIgnoreCase))
20        return 10;
21    if (NamedSize.Equals("Subtitle", StringComparison.OrdinalIgnoreCase))
22        return 16;
23    if (NamedSize.Equals("Title", StringComparison.OrdinalIgnoreCase))
24        return 24;
25    return 12;
26 }
```

Quelltext 6.10: FontSize Umwandlung

Neben dieser Umwandlung sind auch weitere Anpassungen an weiteren Eigenschaften notwendig. Dazu zählen auch Farben und die Wahl einer Textumbruchoption.

Nun erfolgt die Anlage einer Arbeitskopie des Templates, um in weiteren Compilerphasen die Werte aus der Xamarin.Forms App einzufügen. Abbildung 6.4 visualisiert, die Behandlung der Platzhalter in den einzelnen Phasen der Übersetzung.

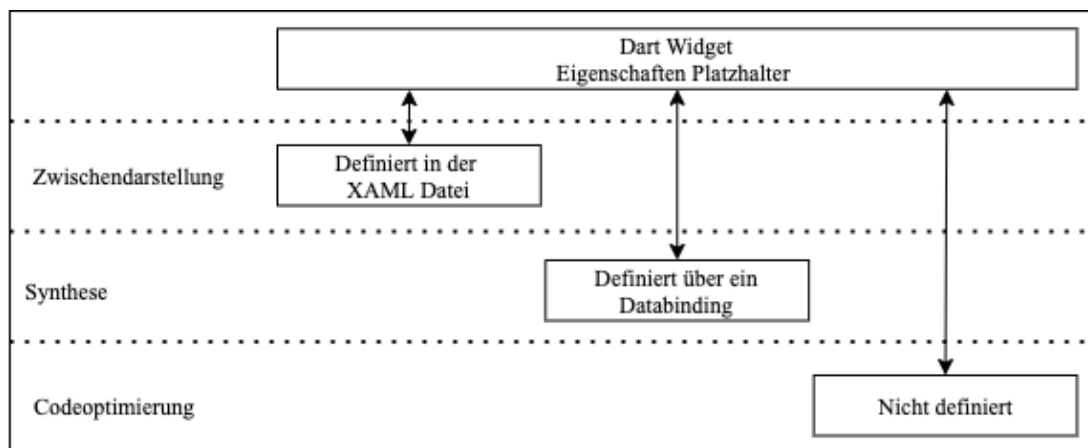


Abbildung 6.4: Bereinigung von Dart Widget Platzhaltern

Es wird deutlich, dass in der Zwischendarstellung ausschließlich XAML definierte Eigenschaftswerte importiert werden. Während der Synthese erfolgt der Austausch der sogenannten Databindings, die den Wert aus der Codebehind-Datei erhalten. Zur Übersetzung werden neue Platzhalter mit dem Prefix ‚BindingPlaceholder‘ benötigt die in der Arbeitskopie hinterlegt sein müssen. Exemplarisch wird in Quelltext 6.11 der Text-Platzhalter zu einem Binding Platzhalter umgewandelt, und der Platzhalter `textAlign` mit einem Wert versehen.

```

1 Text(
2   %BindingPlaceholder(TextProperty)%,
3   textAlign: left,
4   overflow: %OverflowPlaceholder%,
5   style: %StylePlaceholder%,
6 )

```

Quelltext 6.11: Flutter Widget mit Binding-Platzhaltern

Zu diesem Zeitpunkt sind in der Arbeitskopie noch die Binding- und ungenutzten Platzhalter vorhanden. Diese müssen wie, in der Abbildung 6.4 visualisiert, in späteren Phasen der Kompilierung entfernt werden. Grundsätzlich lässt sich aus der XAML ableiten, welche Platzhalten nicht benötigt werden, da die Bereinigung jedoch eine Teilaufgabe der Optimierung darstellt wird diese erst in der entsprechenden Phase durchgeführt.

### 6.5.2 Logische- Zwischendarstellung

Die Erzeugung der logischen Zwischendarstellung erfordert die gleiche Logik wie die der Klassenstrukturen, da es sich bei diesen Dateien ebenfalls um C# Klassen handelt. Während Xamarin.Forms Änderungen an den Eigenschaften automatisch, mithilfe der ‚`INotifyPropertyChanged`‘ Schnittstelle, an die Benutzeroberfläche übermittelt, muss bei

Flutter das Framework mithilfe der `setState()` Methode informiert werden. Notwendige Änderungen können jedoch nicht im Rahmen der Zwischendarstellungserstellung durchgeführt werden, da anhand der Codebehind Datei, nicht identifizierbar ist, welche Eigenschaft einen direkten Bezug auf die grafische Benutzeroberfläche haben.

### 6.5.3 Visuelle Synthese

Die Synthese ist eine Teilaufgabe des Compiler Backends und befasst sich mit der Zusammenführung der einzelnen Zwischendarstellungen. In diesem Falle werden jedoch die visuelle und logische Zwischendarstellung miteinander kombiniert um eine gemeinsame Zwischendarstellung zu generieren. Die in der nächsten Phase mit den übersetzten Klassenstrukturen zusammengeführt werden kann.

In einem ersten Schritt erfolgt die Kopie der übersetzten Logik unterhalb der für den Aufbau der grafischen Benutzeroberfläche beinhaltenden Build Methode. Im nächsten Schritt werden ereignisauslösende Aktivitäten, beispielsweise der Klick auf Schaltflächen, mit den übersetzten Methoden verbunden und die Platzhalter entfernt. Die in der visuellen Zwischendarstellung angelegten Binding Platzhalter geben nun Rückschluss darauf, welche Eigenschaften Änderungen der der Benutzeroberfläche vornehmen. Änderungen an diesen müssen folglich mit einem SetState Block umschlossen werden, damit sich die grafische Benutzeroberfläche auch in Flutter aktualisiert.

Eine Überprüfung welche Eigenschaften der Codebehind-Klassen am Front-End Änderungen durchgeführt haben, wird mittels Binding-Platzhaltern vollzogen. So müssen alle Bereiche im Quelltext, die Änderungen an Eigenschaften vornehmen zu denen es ein Binding-Placeholder gibt mit einem SetState Block umschlossen werden.

## 6.6 Synthese

Nach der Kombination der visuellen Zwischendarstellungen stehen folglich zwei Dart-Dateien zur Verfügung die ebenfalls miteinander kombiniert werden müssen wo dies notwendig ist. Hierfür ist es notwendig zu überprüfen welche Klassen in welchen Quelltext-Dokumenten verwendet werden. Anschließend können die entsprechenden Dateien importiert werden.

## 6.7 Codeoptimierung

Nach der erfolgreichen Zusammenführung der einzelnen Artefakte kann anschließend der Code optimiert werden. Dafür werden Sprachkomponenten entfernt, die ausschließlich für die Arbeit von `Xamarin.Forms` benötigt sind. Dazu gehört die `InitializeComponent` Methode, welche aus der Codebehind-Datei die grafische Benutzeroberfläche lädt und in Flutter nicht benötigt wird.

Ein weiterer Schritt der Bereinigung das Entfernen der `INotifyPropertyChanged` Schnittstellen Definition die in `Xamarin.Forms` für die Oberflächen- Aktualisierung zuständig ist und bereits durch `SetState`-Blöcke ersetzt wurde. Hierfür ist es notwendig, die Ereignisdefinition, den Ereignis-Handler und alle Aufrufe zu entfernen.

Jetzt folgt die Entfernung der nicht benötigten Platzhalter aus dem Dart generierten Quelltext. Damit befindet sich der Programmcode in in einem korrekten syntaktischen Zustand und kann mit Hilfe des Flutter Compilers zu einer mobilen App übersetzt und ausgeführt werden.

## 6.8 Grafische Benutzeroberfläche

Die GUI ist der zentrale Berührungspunkt von Anwendern mit dem Transpiler. Sie soll die notwendigen Eingabe-Möglichkeiten anbieten, das Ergebnis ausgeben und den Anwender auf mögliche Fehler hinweisen. Das grafische Vorbild ist das in Kapitel 3 entworfene Mockup, siehe Abbildung 3.4. Für die Erstellung einer entsprechenden Benutzeroberfläche stehen eine Vielzahl von Technologien mit verschiedenen Vor- und Nachteilen zur Verfügung. Eine Webseite erspart dem Anwender einen hohen Installationsaufwand und ermöglicht die plattformunabhängige Verwendung des Compilers. Allerdings ist zumindest ein gewisser Kontrollverlust über den eigenen Quelltext durch das Hochladen auf eine Webseite nicht ausgeschlossen. Die Bedienoberfläche dieser Arbeit soll daher eine lokale Anwendung sein, sodass kein unberechtigter Zugriff auf den Source-Code möglich ist. Zu diesem Zweck wird die GUI mit der Technologie Windows Presentation Foundation (WPF) realisiert. Dabei handelt es sich um ein UI-Framework des .NET Frameworks, das für die Erstellung von Desktopanwendungen geeignet ist die mit XAML und C# entwickelt werden.<sup>144</sup>

Anwendungen die mittels WPF programmiert werden, sind nur unter Windows als

---

<sup>144</sup>Vgl. Wenger 2012, S. 1f.

Betriebssystem ausführbar. Da der Roslyn Compiler auch nur unter Windows verfügbar ist entstehen hier keine zusätzlichen Einschränkungen.

Die folgende Darstellung 6.5 zeigt das Erscheinungsbild der Anwendung nach einer erfolgreichen Übersetzung.

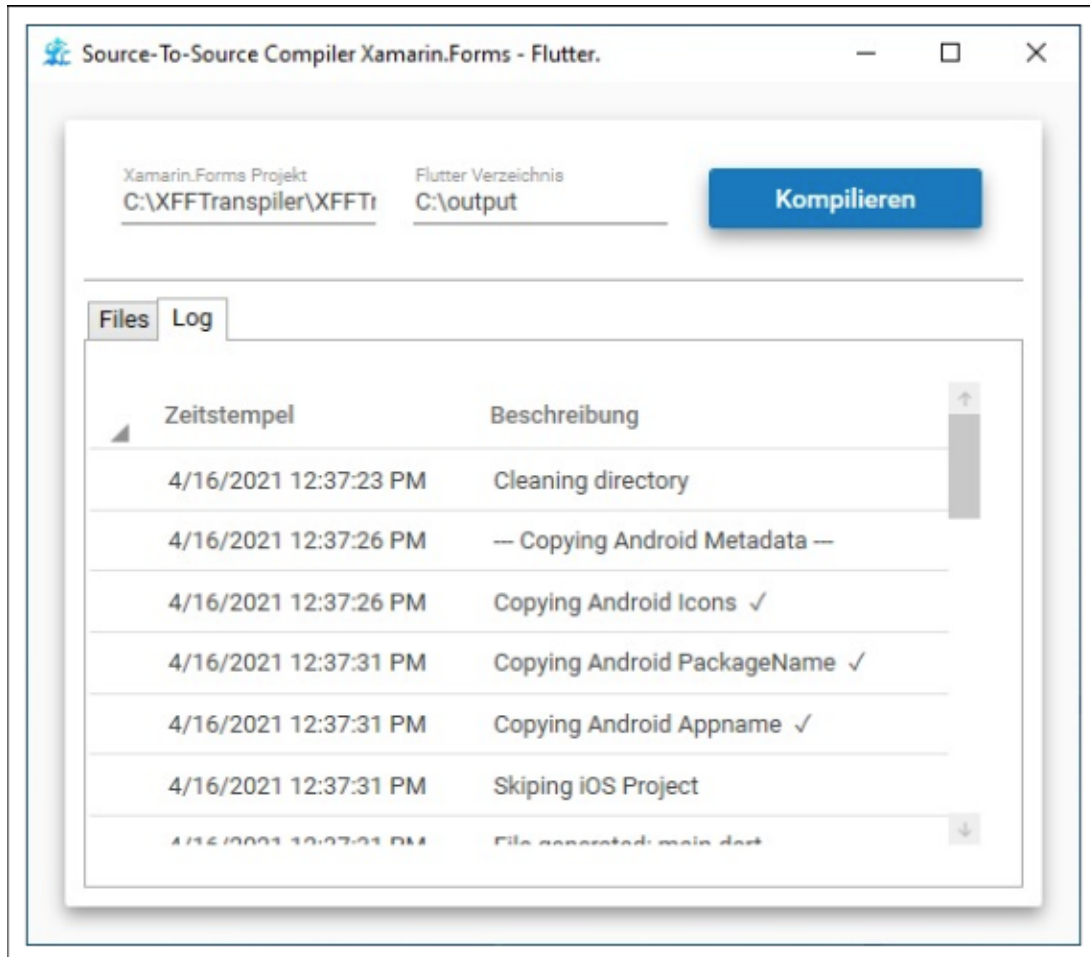


Abbildung 6.5: Grafische Oberfläche des Compilers

Wie die Ansicht veranschaulicht, werden die durchgeführten Arbeitsschritte durch Log-Einträge in der Anwendung angezeigt. Die Realisierung der grafischen Benutzeroberfläche als auch des Source-To-Source Compilers erfolgte mit .Net Technologien, sodass es möglich ist den Compiler als Abhängigkeit in das Projekt der grafischen Benutzeroberfläche zu laden und von dort aus die Übersetzung zu starten.



# 7 Qualitätssicherung

Nach der erfolgreichen Implementierung des Source-To-Source Compilers soll mithilfe einer Xamarin.Forms App überprüft werden, ob der Compiler erwartungsgemäß arbeitet.

## 7.1 Testobjekt

Die Testapp wurde mit der Version 5.0.0.2012 des Xamarin.Forms Frameworks realisiert und verwendet die Erweiterungen Xamarin. Essentials. Das Testobjekt soll möglichst viele Funktionalitäten von Xamarin.Forms abbilden, hat jedoch nicht den Anspruch einer vollständigen Testabdeckung. Auf plattformspezifische Implementationen mit Ausnahme der Metadaten und Ressourcen wurde verzichtet. Eine genaue Beschreibung der App-Funktionalitäten und -Eigenschaften folgt und wird mithilfe von iOS Screenshots veranschaulicht. Die entsprechenden Android Screenshots befinden sich in Anhang III.

Anhand des Namens, SampleApp, und eines Anwendungssicons können Anwender die Benutzer die App auf dem Startbildschirm identifizieren. Nach dem Start öffnet sich eine Menüstruktur, die Wurzel der Navigation, über die verschiedene Bereiche der mobilen Anwendung angesteuert werden können. Abbildung 7.1 zeigt im ersten Screenshoot das Icon und den Namen, im zweiten das Menu der App und im dritten die Visualisierung der verfügbaren Steuerelemente.

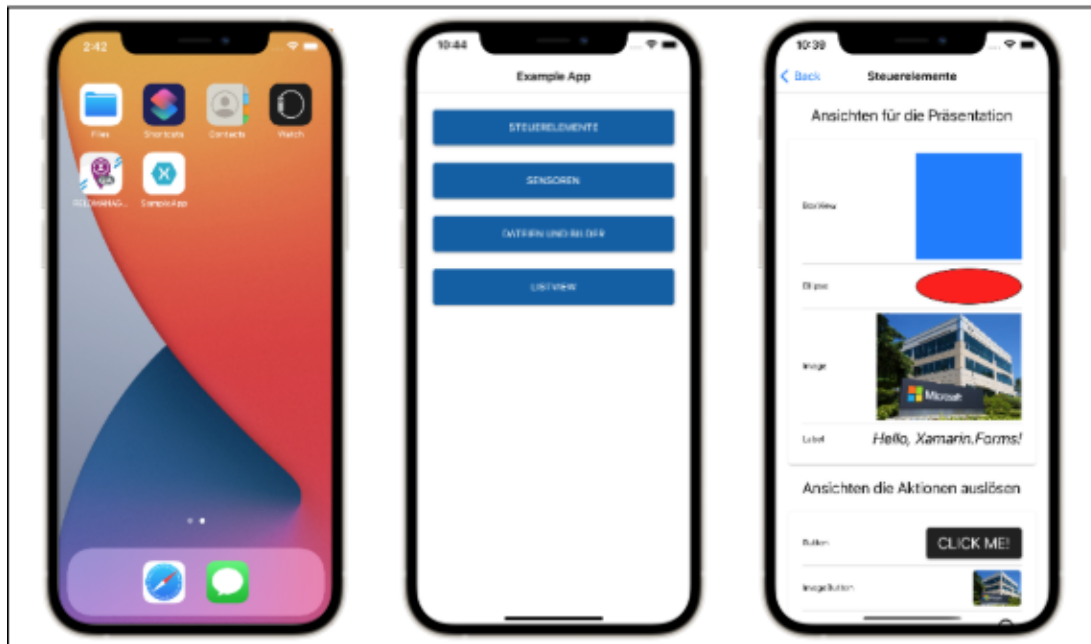


Abbildung 7.1: Test Objekt Screenshots I

Neben der Navigation zu den Steuerelemente ist es möglich zu einer Ansicht zu gelangen, die die Werte der Smartphonesensoren anzeigt. Für diesen Anwendungsfall werden sowohl der Beschleunigungssensor, als auch die Gyroscop Werte ausgegeben. Zusätzlich steht eine Seite zur Verfügung die es erlaubt auf die Kamera des Gerätes und die Bildergalerie zuzugreifen. Diese Ansichten werden in Abbildung 7.2 dargestellt.

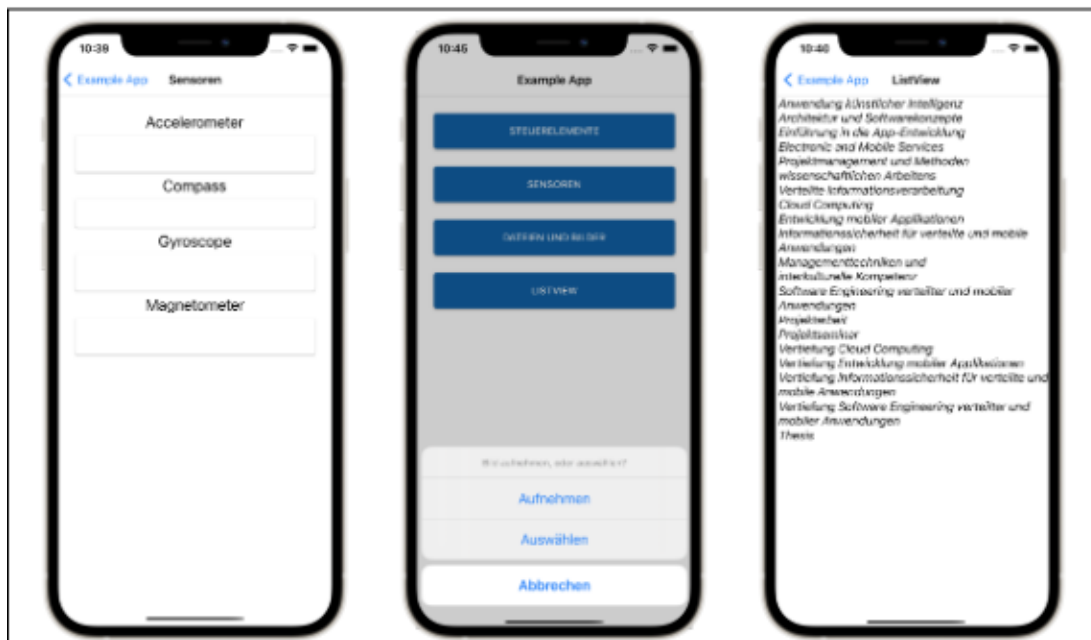


Abbildung 7.2: Test Objekt Screenshots II

## 7.2 Testfälle

Um den Compiler zu Testen ist es notwendig Testfälle zu definieren und zu überprüfen, ob der Compiler die Xamarin.Forms App so übersetzt, wie es zu erwarten. Zu diesem Zweck werden in diesem Abschnitt Testfälle definiert die sich aus den ermittelten Unterschieden zwischen beiden Frameworks, sowie der Programmiersprachen ergeben. Tabelle 7.1 beschreibt die Testfälle und ihre entsprechenden Funktionen.

ID	Komponente	Beschreibung
1	App-Icon	Prüfen ob das App-Icon übernommen wurde
2	App-Name	Prüfen ob das App-Name übernommen wurde
3	SDK Versionen	Prüfen ob die SDK Versionen übernommen wurden
4	Seitenname	In der Navigationsleiste wird der Name der aktuellen Seite angezeigt
5	Navigation	Mit Hilfe des Menüs kann navigiert werden
6	Zurück Navigation	Über die Navigationsleiste kann zurück Navigiert werden
7	Gyroscope auswerten	Die Werte werden in der App angezeigt.
8	Accelerometer auswerten	Die Werte werden in der App angezeigt.
9	Compass auswerten	Die Werte werden in der App angezeigt.
10	Magnetometer auswerten	Die Werte werden in der App angezeigt.
11	Sensor nicht verfügbar	Wenn ein Sensor nicht verfügbar ist, wird ein Fehler angezeigt
12	Steuerelemente	Alle Steuerelemente werden angezeigt
13	Ereignisse funktionieren	Steuerelemente reagieren wie gewohnt
14	Bild aus Ressourcen laden	Ein Bild aus den Ressourcen wird in der App angezeigt
15	Bild aus dem Web laden	Ein Bild aus dem Internet wird in der App

Tabelle 7.1: Testfälle der Testapp

## 7.3 Testablauf

Der Start des Testlaufs beginnt mit dem Aufruf der grafischen Benutzeroberfläche und der Auswahl sowohl der Xamarin.Forms Testapp, als auch eines leeren Ordners, der das Zielverzeichnis nach der Kompilierung dient. Aufgrund der vielen Dateizugriffe und durchzuführenden Aktionen nimmt die Übersetzung der Anwendung eine gewisse Zeit in Anspruch. Im Frontend lassen sich Informationen bezüglich der Übersetzung ablesen, wenn sie erfolgreich absolviert wurde. Anschließend kann die übersetzte Flutter Anwendung mithilfe der Flutter-SDK kompiliert und die übersetzten Ansichten kontrolliert werden. Die folgenden Screenshots zeigen das Flutterergebnis. Die entsprechenden Screenshots von Android befinden sich im Anhang dieser Arbeit.

Die Screenshots zeigen, auf den ersten Blick die erwartete visuelle Vergleichbarkeit der Anwendungen. Sowohl der Name als auch das Launchericon der Anwendung sind korrekterweise übernommen worden. Nach dem Start der Anwendung fällt auf, dass die App etwas anders aussieht als die Xamarin.Forms iOS Anwendung. Dies ist begründet durch die Verwendung der Material Design Vorlagen. Da der Compiler, wie in den Ausschlusskriterien beschrieben nicht den Style der Anwendungen übersetzt ist dieses Verhalten, das erwartete und daher an dieser Stelle akzeptabel. Das zentrale Menü der Anwendung ist auch in der Flutter Anwendung verfügbar und die Navigation über dieses funktioniert wie bei Xamarin.Forms ausschließlich die Animation ist unterschiedlich. Die Ansicht der verfügbaren Ansichten zeigt, dass sowohl die Verschachtelung der Widgets als auch ihre Funktionalitäten mit der Xamarin.Forms Variante übereinstimmen. So lassen sich beispielsweise auch die Datum- und Uhrzeitsauswahl öffnen. Die Ansicht der Sensoren zeigen, dass auch hier die Oberfläche korrekt übernommen wurde und auch, dass die Werte der Smartphone-Sensoren übernommen und im richtigen Format angezeigt sind. Im Gegensatz zu Xamarin.Forms werden diese Daten jedoch nicht im gleichen Zyklus aktualisiert. Dieser Unterschied basiert auf den Implementationen des jeweiligen Plugins und entsteht nicht durch den Übersetzer. Auch die Verwendung der Kamera und der Zugriff auf die Galerie funktionieren wie in der Xamarin.Forms Anwendung.

Mit der im März 2021 veröffentlichten Version 2.0 von Flutter können Flutter Anwendungen ebenfalls zu Web-Anwendungen kompiliert werden. Auch wenn die Erstellungen von Webseiten nicht im Fokus dieser Arbeit steht soll diese Anwendung trotzdem kurz betrachtet werden. Die grundlegende Darstellung ist identisch wie die der mobilen Anwendung. Da die Xamarin.Forms App nicht darauf ausgelegt war auch großen Monitoren dargestellt zu werden ist diese Ansicht zwar nicht für diese Darstellung optimiert aber funktionsfähig. Die Funktionalitäten der Sensoren sind jedoch nicht verfügbar, da das Plugin keinen Support für Flutter-Web bietet.

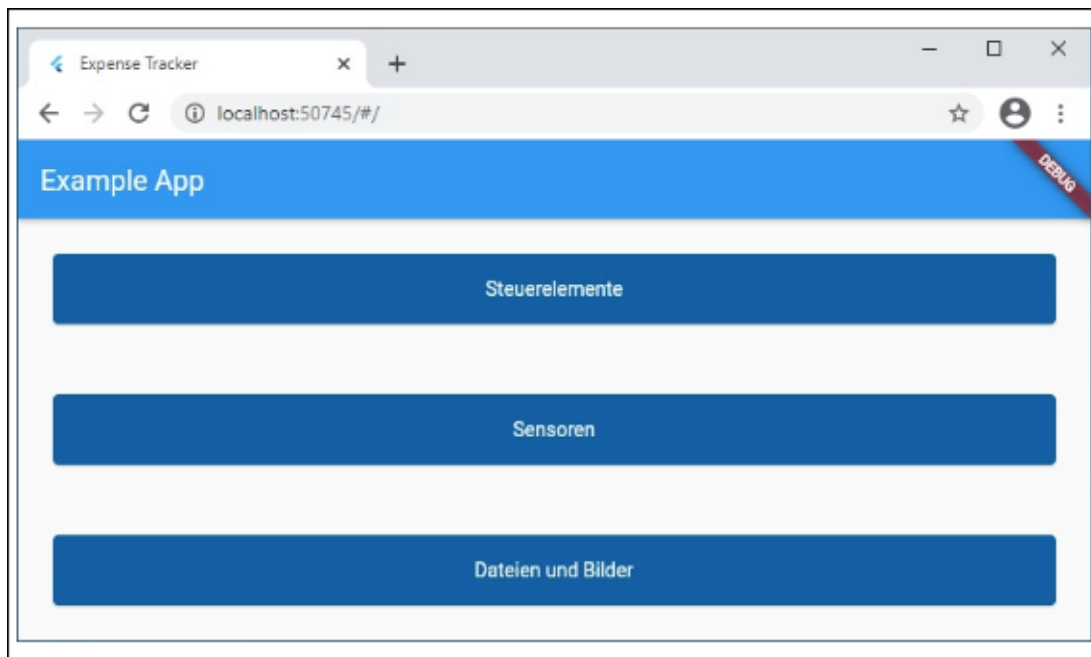


Abbildung 7.3: Darstellung des Testobjektes als Webseite

Obwohl das Testobjekt nicht alle Xamarin.Forms Funktionalitäten abbildet, konnte das Testobjekt vollständig überführt und die Funktionsfähigkeit des Prototypen bewiesen werden.

## 8 Fazit und Ausblick

Ein Source-To-Source Compiler, wie in dieser Arbeit angedacht, stellt einen Lösungsansatz für Xamarin.Forms App-Entwickler dar, die sich nach dem Ende des Supports vom Anbieter Microsoft trennen und trotzdem eine bestehende Anwendung erhalten möchten. Das generelle Problem der Abhängigkeit ist jedoch nicht durch eine Abwendung von Microsoft gelöst, da das von der Firma Google entwickelte Flutter Framework ebenfalls, wenn auch derzeit nicht absehbar, eingestellt werden kann. Der Ausbau der Idee, durch grundsätzlich automatisierte Übersetzungen Programmierern eine Wahlfreiheit und Wechseloptionen zu ermöglichen rechtfertigt eine Weiterentwicklung des Prototypen.

### 8.1 Open Source Veröffentlichung

Der Umfang des Prototypen konnte durch die in Kapitel 3 ausgeschlossenen Elemente reduziert werden, diese Präklusion besteht bei den meisten Xamarin.Forms Apps nicht. Eine Veröffentlichung des Compilers als Open Source Projekt könnte helfen, diese komplexeren mobilen Anwendungen zu übersetzen, indem Programmierer den für die Allgemeinheit verfügbaren Quelltext auf ihre speziellen Gegebenheiten anpassen und wiederum publizieren. Unternehmen wie DevExpress, Syncfusion und Telerik bieten sowohl Steuerelemente für Xamarin.Forms, als auch Widgets für Flutter und hätten die Möglichkeit ihre eigenen UI-Elemente durch den Compiler ersetzen zu lassen und somit dessen Umfang zu erhöhen. Durch Open Source Veröffentlichung könnte die Entwicklung in Geschwindigkeit und Zielgenauigkeit erhöht werden.

### 8.2 Beantwortung der Forschungsfrage

Der Source -To-Source Compiler übersetzt die Xamarin.Forms Testapp und zeigt damit die Machbarkeit einer zumindest teilautomatisierten Übersetzung zu Flutter. Zur Kompilierung von produktiven Anwendungen ist eine Weiterentwicklung nötig, sodass platt-

formspezifische Implementierungen und viele Erweiterungen für das Xamarin.Forms Framework ebenfalls umgewandelt werden können. Im aktuellen Zustand ist der Compiler schon in der Lage komplexe visuelle Darstellungen und Quelltexte zu übersetzen und somit den Entwicklungsaufwand bei einem Umstieg auf Flutter merklich zu reduzieren. Da bereits für Ende 2021 das Supportende für Xamarin.Forms beschlossen und die Einführung von MAUI angekündigt wurde, wäre eine vollständige Funktionsfähigkeit des Compilers noch in diesem Jahr nötig, um Unternehmen und Programmierern eine geeignete Alternative zu bieten.

# Literaturverzeichnis

## Gedruckte Quellen

- Albahari, Joseph und Eric Johannsen (2020). *C# in a Nutshell. The Definitive Reference*. Sebastopol: O'Reilly Media Inc.
- Balzert, Helmut (2011). *Lehrbuch der Softwaretechnik. Entwurf, Implementierung, Installation und Betrieb*. Heidelberg: Spektrum Akademischer Verlag.
- Bayer, Jürgen (2008). *Visual C# 2008. Windows-Programmierung mit dem .NET Framework 3.5*. München: Pearson Deutschland GmbH.
- Biessek, Allesandro (2019). *Flutter for Beginners. An introductory guide to building cross-platform mobile application with Flutter and Dart 2*. Birmingham: Packt Publishing Ltd.
- Cheng, Fu (2019). *Flutter Recipes. Mobile Development Solutions for iOS and Android*. Sandringham: Apress.
- Dobrenz, Walter, Thomas Gewinnus und Walter Saumweber (2018). *Visual C# 2017. Grundlagen, Profiwissen und Rezepte*. München: Carl Hanser Verlag GmbH.
- Eisenecker, Ulrich (2008). *C++: der Einstieg in die Programmierung. Strukturiert und prozedural programmieren*. Witten: W3L.
- Eller, Frank und Michael Kofler (2005). *Visual C#. Grundlagen, Programmieretechniken, Windows-Programmierung*. München: Pearson Deutschland GmbH.
- Hindrikes, Daniel und Johan Karlsson (2020). *Xamarin.Forms Projects. Build Multi-platform Mobile Apps and a Game from Scratch Using C# and Visual Studio 2019*. Birmingham: Packt Publishing Ltd.
- Joorabchi, Mona Erfani (Apr. 2016). „Mobile App Development: Challenges and Opportunities for Automated Support“. Diss. Vancouver: University of British Columbia.
- Keist, Nikolai-Kevin, Sebastian Benisch und Christian Müller (2016). „Software Engineering für Mobile Anwendungen. Konzepte und betriebliche Einsatzszenarien“. In: *Mobile Anwendungen in Unternehmen*. Hrsg. von Thomas Barton, Christian Müller und Christian Seel, S. 91–120.
- Kühnel, Andreas (2019). *C# 8 mit Visual Studio 2019. Das umfassende Handbuch*. Bonn: Rheinwerk.



- Petzold, Charles (2016). *Creating mobile Apps with Xamarin.Forms. Cross.platform C# programming for iOS, Android and Windows*. 1. Aufl. Redmond: Microsoft Press.
- Rohit, Kulkarni, Chavan Aditi und Abhinav Hardikar (2015). „Transpiler and it's Advantages“. In: *International Journal of Computer Science and Information Technologies* 6.2.
- Rutishauser, Heinz (1952). „Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen“. In: *Journal of Applied Mathematics and Physics (ZAMP)* 3, S. 312–313.
- Schneider, Hans-Jürgen (1975). *Compiler. Aufbau und Arbeitsweise*. Berlin: Walter de Gruyter.
- Ullman, Jeffrey D. et al. (2008). *Compiler. Prinzipien, Techniken und Werkzeuge*. 2. Aufl. München: Pearson Studium.
- Vollmer, Guy (2017). *Mobile App Engineering. Eine systematische Einführung - von den Requirements zum Go Live*. 1. Aufl. Heidelberg: dpunkt.
- Wagenknecht, Christian und Michael Hielscher (2014). *Formale Sprachen, abstrakte Automaten und Compiler. Lehr- und Arbeitsbuch für Grundstudium*. 2. Aufl. Wiesbaden: Springer.
- Wenger, Jörg (2012). *WPF 4.5 und XAML. Grafische Benutzeroberflächen für Windows inkl. Entwicklung von Windows Store Apps*. München: Carl Hanser Verlag GmbH.
- Wilhelm, Reinhard, Helmut Seidl und Sebastian Hack (2012). *Übersetzerbau. Syntaktische und semantische Analyse*. Bd. 2.
- Wissel, Andreas, Chrsitian Liebel und Thorsten Hans (2017). „Frameworks und Tools für Cross-Plattform-Programmierung“. In: *iX – Magazin für professionelle Informationstechnik* 2.
- Witte, Joerg (2013). *Programmieren in C#. Von den ersten Gehversuchen bis zu den Sieben-Meilen-Stiefeln*. Wiesbaden: Springer Vieweg.

## Online Quellen

- Apple Inc. (2020). *Managing Your App's Life Cycle*. URL: [https://developer.apple.com/documentation/uikit/app\\_and\\_environment/managing\\_your\\_app\\_s\\_life\\_cycle](https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle) (besucht am 27.02.2021).
- Google LLC (2020a). *A tour of the Dart language*. URL: <https://dart.dev/guides/language/language-tour#using-constructors> (besucht am 27.02.2021).

- Google LLC (2020b). *Adding a splash screen to your mobile app*. URL: <https://flutter.dev/docs/development/ui/advanced/splash-screen> (besucht am 27.02.2021).
- (2020c). *Adding assets and images*. URL: <https://flutter.dev/docs/development/ui/assets-and-images> (besucht am 27.02.2021).
  - (2020d). *Canvas class*. URL: <https://api.flutter.dev/flutter/dart-ui/Canvas-class.html> (besucht am 27.02.2021).
  - (2020e). *didChangeAppLifecycleState method*. URL: <https://api.flutter.dev/flutter/widgets/WidgetsBindingObserver/didChangeAppLifecycleState.html> (besucht am 27.02.2021).
  - (2020f). *Flutter for Xamarin.Forms developers*. URL: <https://flutter.dev/docs/get-started/flutter-for/xamarin-forms-devs> (besucht am 27.02.2021).
  - (2020g). *GridView class*. URL: <https://api.flutter.dev/flutter/widgets/GridView-class.html> (besucht am 27.02.2021).
  - (2020h). *Imagery*. URL: <https://material.io/design/communication/imagery.html> (besucht am 27.02.2021).
  - (2020i). *Introduction to animations*. URL: <https://flutter.dev/docs/development/ui/animations> (besucht am 27.02.2021).
  - (2020j). *Introduction to widgets*. URL: <https://flutter.dev/docs/development/ui/widgets-intro> (besucht am 27.02.2021).
  - (2020k). *Navigate with named routes*. URL: <https://flutter.dev/docs/cookbook/navigation/named-routes> (besucht am 27.02.2021).
  - (2020l). *Shared preferences plugin*. URL: [https://pub.dev/packages/shared\\_preferences](https://pub.dev/packages/shared_preferences) (besucht am 27.02.2021).
  - (2020m). *Taps, drags, and other gestures*. URL: <https://flutter.dev/docs/development/ui/advanced/gestures> (besucht am 27.02.2021).
  - (2020n). *url\_launcher*. URL: [https://pub.dev/packages/url\\_launcher](https://pub.dev/packages/url_launcher) (besucht am 27.02.2021).
  - (2020o). *Use a custom font*. URL: <https://flutter.dev/docs/cookbook/design/fonts> (besucht am 27.02.2021).
  - (2020p). *Using packages*. URL: <https://flutter.dev/docs/development/packages-and-plugins/using-packages> (besucht am 27.02.2021).
  - (2020q). *Widget catalog*. URL: <https://flutter.dev/docs/development/ui/widgets> (besucht am 27.02.2021).

- Google LLC (2020r). *Work with tabs*. URL: <https://flutter.dev/docs/cookbook/design/tabs> (besucht am 27.02.2021).
- (2020s). *Write your first Flutter app, part 1*. URL: <https://flutter.dev/docs/get-started/codelab> (besucht am 27.02.2021).
  - (2020t). *Writing custom platform-specific code*. URL: <https://flutter.dev/docs/development/platform-integration/platform-channels> (besucht am 27.02.2021).
  - (2021a). *Sound null safety*. URL: <https://dart.dev/null-safety> (besucht am 27.02.2021).
  - (2021b). *Center*. URL: <https://api.flutter.dev/flutter/widgets/Center-class.html> (besucht am 27.02.2021).
  - (2021c). *double class*. URL: <https://api.dart.dev/stable/2.10.5/dart-core/double-class.html> (besucht am 27.02.2021).
  - (2021d). *Flutter SDK overview*. URL: <https://flutter.dev/docs/development/tools/sdk/overview> (besucht am 27.02.2021).
  - (2021e). *JSON and serialization*. URL: <https://flutter.dev/docs/development/data-and-backend/json> (besucht am 27.02.2021).
  - (2021f). *Lay out multiple widgets vertically and horizontally*. URL: <https://flutter.dev/docs/development/ui/layout%5C#lay-out-multiple-widgets-vertically-and-horizontally> (besucht am 27.02.2021).
  - (2021g). *showTimePicker function*. URL: <https://api.flutter.dev/flutter/material/showTimePicker.html> (besucht am 27.02.2021).
- Hunter, Scott (2020). *Introducing .NET Multi-platform App UI*. URL: <https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/> (besucht am 27.02.2021).
- Microsoft Corporation (2015a). *Delegaten (C#-Programmierhandbuch)*. URL: <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/delegates/> (besucht am 27.02.2021).
- (2015b). *Generics C# – Programmierhandbuch*. URL: <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/generics/> (besucht am 27.02.2021).
  - (2015c). *Verwenden von Namespaces (C#-Programmierhandbuch)*. URL: <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/namespaces/using-namespaces> (besucht am 27.02.2021).

- Microsoft Corporation (2016a). *.NET-Klassenbibliotheken*. URL: <https://docs.microsoft.com/de-de/dotnet/standard/class-libraries> (besucht am 27.02.2021).
- (2016b). *Xamarin.Forms Pages*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/user-interface/controls/pages> (besucht am 27.02.2021).
  - (2017a). *Grundlagen zu XAML*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/xaml/xaml-basics/> (besucht am 27.02.2021).
  - (2017b). *Layoutoptionen in (Xamarin.Forms)*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/user-interface/layouts/layout-options> (besucht am 27.02.2021).
  - (2017c). *Working With Property Lists in Xamarin.iOS*. URL: <https://docs.microsoft.com/de-de/xamarin/ios/app-fundamentals/property-lists> (besucht am 27.02.2021).
  - (2018a). *Arbeiten mit dem Android-Manifest*. URL: <https://docs.microsoft.com/de-de/xamarin/android/platform/android-manifest> (besucht am 27.02.2021).
  - (2018b). *Xamarin.Forms Layouts*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/user-interface/controls/layouts> (besucht am 27.02.2021).
  - (2019a). *Xamarin.Essentials Einstellungen*. URL: <https://docs.microsoft.com/de-de/xamarin/essentials/preferences> (besucht am 27.02.2021).
  - (2019b). *Xamarin.Forms Custom Renderers*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/app-fundamentals/custom-renderer/> (besucht am 27.02.2021).
  - (2019c). *Xamarin.Forms DependencyService*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/app-fundamentals/dependency-service/> (besucht am 27.02.2021).
  - (2020a). *An introduction to NuGet*. URL: <https://docs.microsoft.com/de-de/nuget/what-is-nuget> (besucht am 27.02.2021).
  - (2020b). *Asynchrone Programmierung mit async und await*. URL: <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/async/> (besucht am 27.02.2021).
  - (2020c). *Floating-point numeric types (C# reference)*. URL: <https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types> (besucht am 27.02.2021).

- Microsoft Corporation (2020d). *Images in Xamarin.Forms*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/user-interface/images?tabs=macos> (besucht am 27.02.2021).
- (2020e). *Launcher Class*. URL: <https://docs.microsoft.com/en-us/dotnet/api/xamarin.essentials.launcher?view=xamarin-essentials> (besucht am 27.02.2021).
  - (2020f). *Serialisieren und Deserialisieren*. URL: <https://docs.microsoft.com/de-de/dotnet/standard/serialization/system-text-json-how-to?pivot=dotnet-5-0> (besucht am 27.02.2021).
  - (2020g). *Simple Animations in Xamarin.Forms*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/user-interface/animation/simple> (besucht am 27.02.2021).
  - (2020h). *Value types (C# reference)*. URL: <https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/value-types> (besucht am 27.02.2021).
  - (2020i). *Vererbung (C#-Programmierhandbuch)*. URL: <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/inheritance> (besucht am 27.02.2021).
  - (2020j). *Xamarin.Forms Ansichten*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/user-interface/controls/views> (besucht am 27.02.2021).
  - (2020k). *Xamarin.Forms App Lifecycle*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/app-fundamentals/app-lifecycle> (besucht am 27.02.2021).
  - (2020l). *Xamarin.Forms FlyoutPage*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/app-fundamentals/navigation/flyoutpage> (besucht am 27.02.2021).
  - (2020m). *Xamarin.Forms gestures*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/app-fundamentals/gestures/> (besucht am 27.02.2021).
  - (2020n). *Xamarin.Forms Navigation*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/app-fundamentals/navigation/> (besucht am 27.02.2021).
  - (2020o). *Xamarin.Forms TabbedPage*. URL: <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/app-fundamentals/navigation/tabbed-page> (besucht am 27.02.2021).

- MongoDB Inc. (2020). *Realm Mobile Database*. URL: <https://www.mongodb.com/realm/mobile/database> (besucht am 27.02.2021).
- Pedley, Adam (2019). *Moving From C# To Dart: Quick Start*. URL: <https://buildflutter.com/moving-from-csharp-to-dart-quick-start/> (besucht am 27.02.2021).
- Ritscher, Walt (2020). *Arranging Views with Xamarin.Forms Layout*. URL: <https://bit.ly/34ulgpU> (besucht am 27.02.2021).
- Sells, Chris (2021). *What's New in Flutter 2*. URL: <https://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65> (besucht am 27.02.2021).
- Sneath, Tim (2020). *Flutter Spring 2020 Update*. URL: <https://medium.com/flutter/flutter-spring-2020-update-f723d898d7af> (besucht am 27.02.2021).
- SQLite Consortium (2020). *About SQLite*. URL: <https://www.sqlite.org/about.html> (besucht am 27.02.2021).
- Stack Exchange Inc. (2019). *Developer Survey Results 2019*. URL: <https://insights.stackoverflow.com/survey/2019> (besucht am 27.02.2021).
- (2020). *Developer Survey Results 2020*. URL: <https://insights.stackoverflow.com/survey/2020> (besucht am 27.02.2021).
- Star, Ford (2019). *The Dart Language: When Java and C# Aren't Sharp Enough*. URL: <https://www.toptal.com/dart/dartlang-guide-for-csharp-java-devs> (besucht am 27.02.2021).
- Tekartik (2020). *sqflite*. URL: <https://pub.dev/packages/sqflite> (besucht am 27.02.2021).
- Vaibhavi, Rana (2020). *Change Application Name and Icon in Flutter project(Android and iOS)*. URL: <https://medium.com/@vaibhavi.rana99/change-application-name-and-icon-in-flutter-bebbec297c57> (besucht am 27.02.2021).
- Varty, Josh (2014). *Learn Roslyn Now: Part 4 CSharpSyntaxWalker*. URL: <https://joshvarty.com/2014/07/26/learn-roslyn-now-part-4-csharpsyntaxwalker/> (besucht am 27.02.2021).
- Versluis, Gerald (2020). *Embedded Fonts: Custom Fonts in Xamarin.Forms*. URL: <https://devblogs.microsoft.com/xamarin/embedded-fonts-xamarin-forms/> (besucht am 27.02.2021).
- Ward, Ian (2020). *Realm Mobile Database*. URL: <https://github.com/realm/realm-object-server/issues/55> (besucht am 27.02.2021).

# Anhang I: Gegenüberstellung von visuellen Elementen

Xamarin.Forms Element	Flutter Widget
ContentPage	
FlyOutPage	MasterDetailScaffold
NavigationPage	Scaffold
TabbedPage	TabBar und TabBarView
AbsolutLayout	Positioned
ContentView	StatelessWidget
Frame	BoxDecoration
Grid	GridView oder Stack für das Stapeln von Elementen
ScrollView	SingleChildScrollView
StackLayout	Row und Column
ReleativLayout	Positioned
BoxView	SizedBox
Image	Image
Label	Text
Map	Leamaps oder Google Maps
WebView	webview_flutter
Ellipse	CustomPaint
Linie	CustomPaint
Path	CustomPaint
Polygon	CustomPaint
Polyline und Rectangle	CustomPaint
Rectangle	CustomPaint
Button	FlatButton
ImageButton	IconButton
RadioButton	RadioButton
RefreshView	pull_to_refresh

SearchBar	flutter_search_bar
SwipeView	flutter_slideable
Entry	TextField
Editor	TextField
CheckBox	Checkbox
Switch	Switch
Slider	Slider
Stepper	number_inc_dec
DatePicker	TextField mit Funktion
TimePicker	TextField mit Funktion
ActivityIndicator	CircularProgressIndicator
ProgressBar	LinearProgressIndicator
CarouselView	carousel_slider
IndicatorView	carousel_slider
Picker	TextView mit Funktionalität
TableView	Table
List	List
CollectionView	List
SwipeView	flutter_slideable

Tabelle: Gegenüberstellung von visuellen Elementen



# Anhang II: Optimierte Flutter-LoginPage

```
1  @override
2  Widget build(BuildContext context) {
3    return Scaffold(
4      backgroundColor: Colors.grey,
5      appBar: AppBar( title: Text("LoginPage"),),
6      body: Center(
7        child: Card(
8          child: Column(
9            mainAxisAlignment: MainAxisAlignment.min,
10           children: <Widget>[
11             Padding(
12               padding: const EdgeInsets.all(5.0),
13               child: TextField(
14                 obscureText: false,
15                 decoration: InputDecoration(
16                   border: OutlineInputBorder(),
17                   labelText: 'Username', ),
18               ),
19             ),
20             Padding(padding: const EdgeInsets.all(5.0),
21               child: TextField(
22                 obscureText: true,
23                 decoration: InputDecoration(
24                   border: OutlineInputBorder(),
25                   labelText: 'Password',),
26               ),
27             ),
28             SizedBox(
29               width: double.infinity,
30               child: Padding(
31                 padding: const EdgeInsets.all(5.0),
32                 child: ElevatedButton(
33                   child: Text('Login'),),
34               ),
35             ),
36           ],
37         ),
38       ),
39     );
40   }
41 }
```

Quelltext: Optimierte Flutter-LoginPage

# Anhang III: Android Screenshots der Xamarin.Forms App

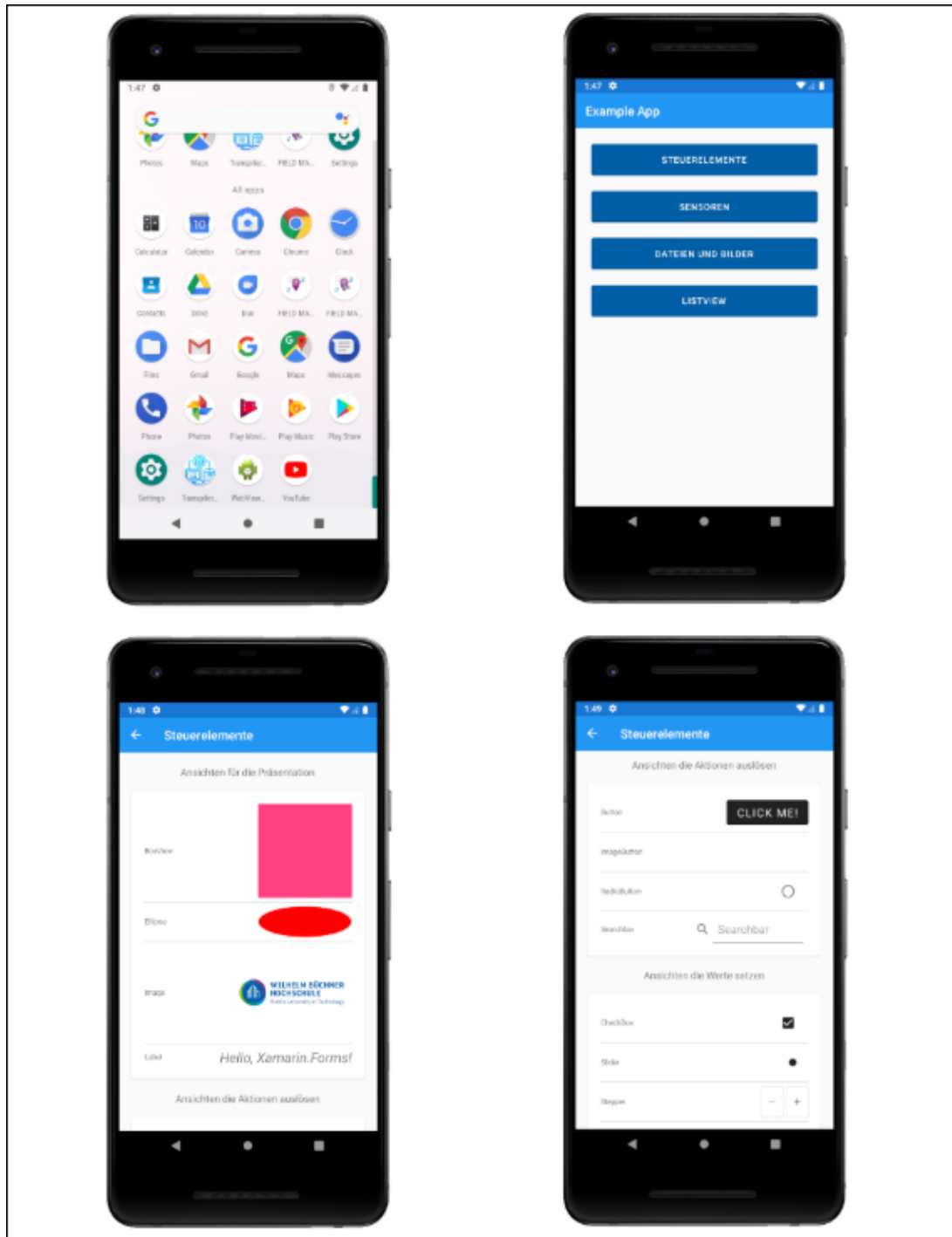


Abbildung 1: Test Objekt Screenshots I

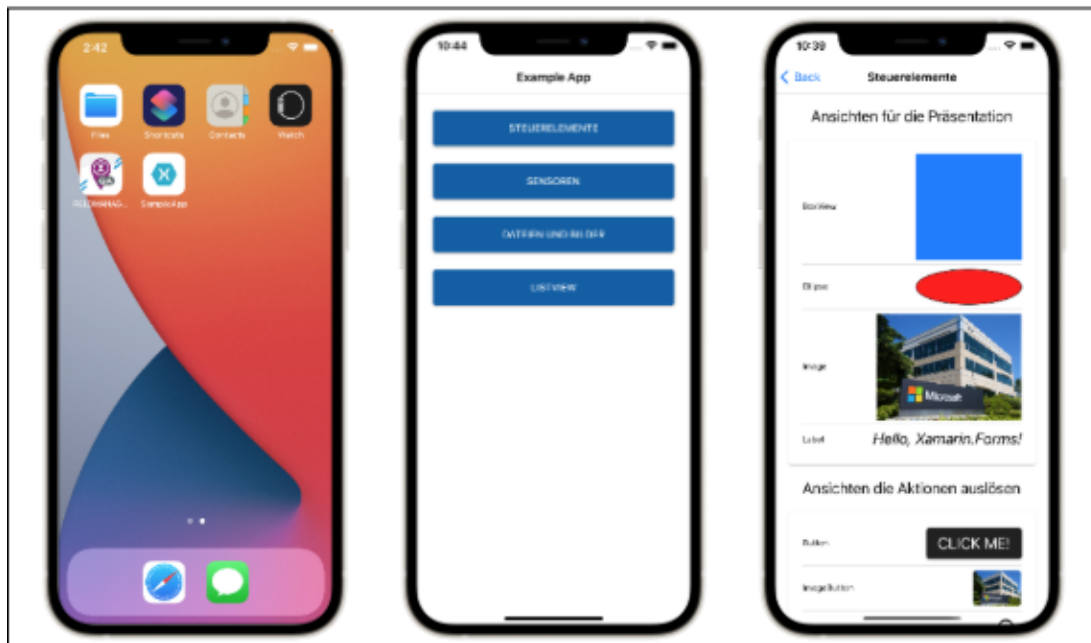


Abbildung 2: Test Objekt Screenshots III

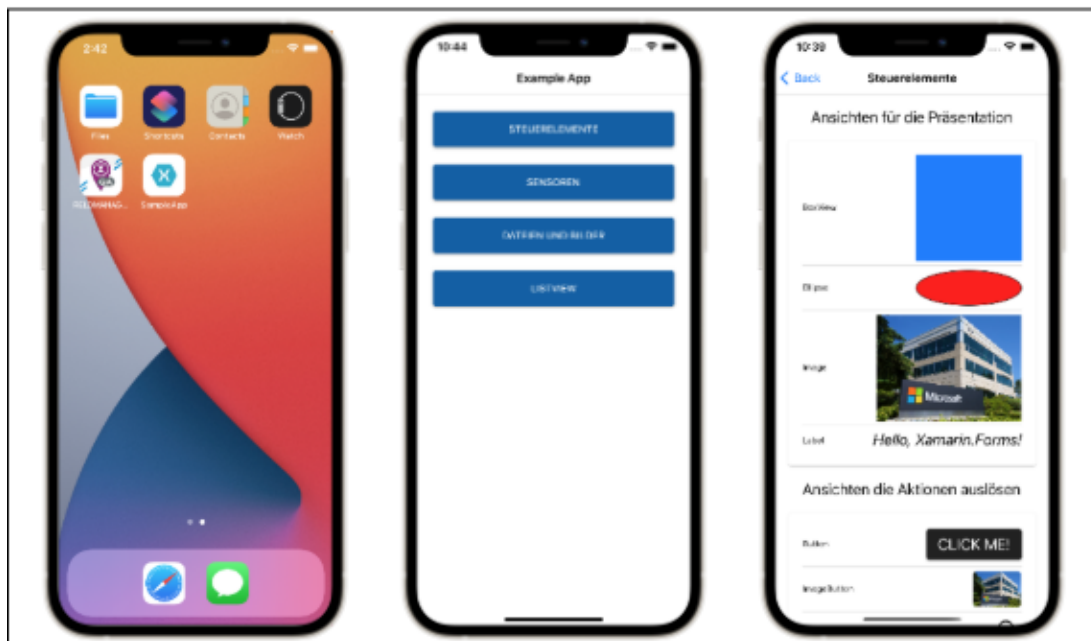


Abbildung 3: Test Objekt Screenshots IV

# Anhang IV: Android Screenshots der Flutter App



Abbildung 1: Test Objekt Screenshots I

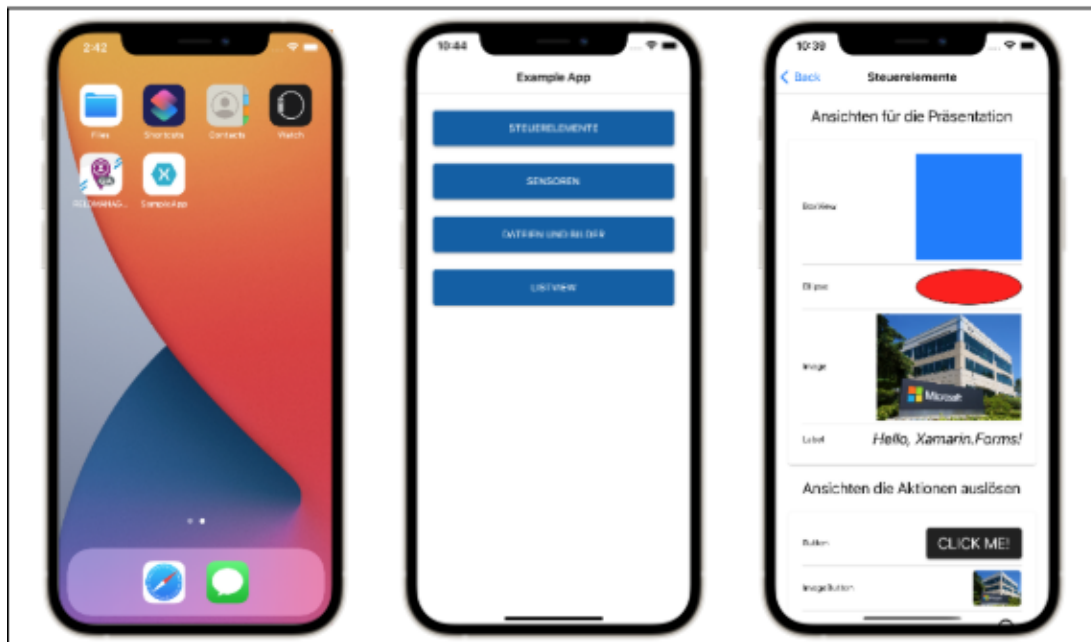


Abbildung 2: Test Objekt Screenshots II

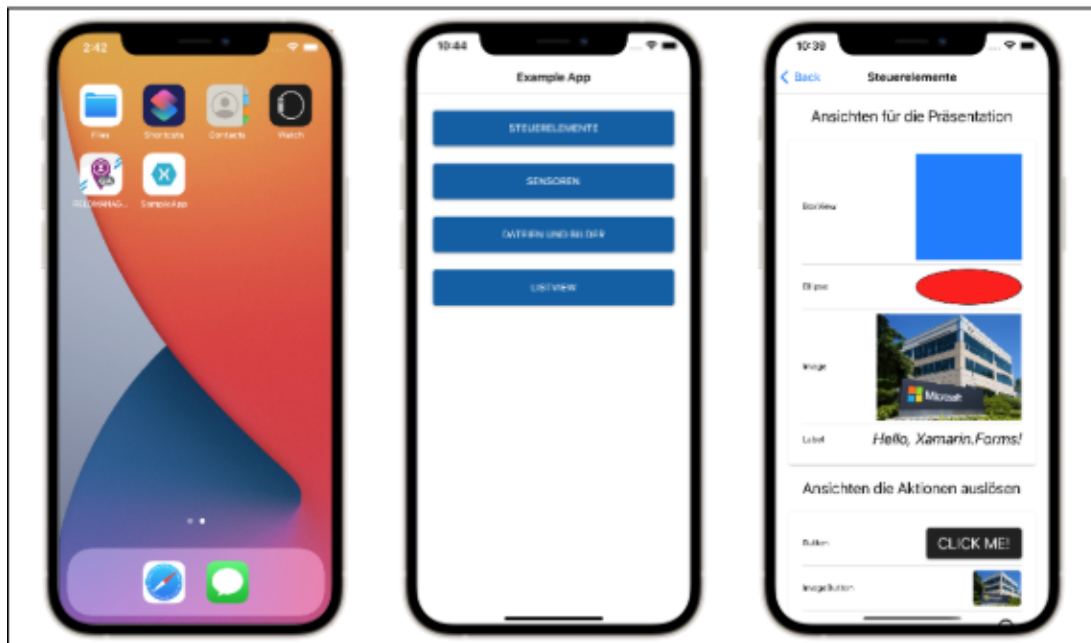


Abbildung 3: Test Objekt Screenshots III

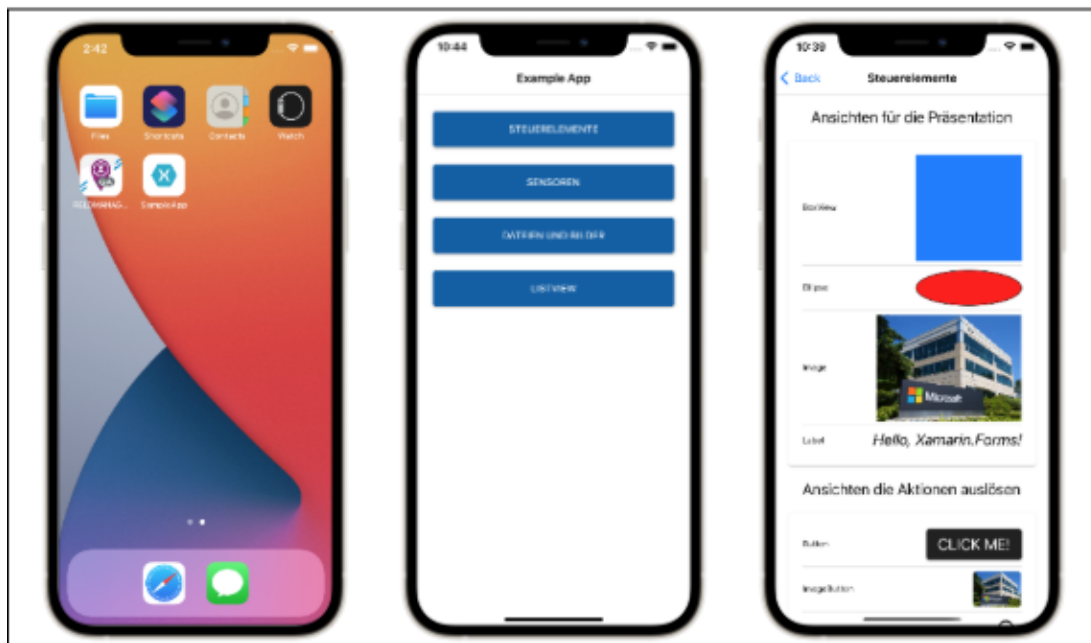


Abbildung 4: Test Objekt Screenshots IV

# Eidesstattliche Erklärung

Studierender: Julian Pasqué

Matrikelnummer: 902953

Hiermit erkläre ich, dass ich diese Arbeit selbstständig abgefasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

.....

Ort, Abgabedatum

.....

Unterschrift (Vor- und Zuname)

