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

25. April 2021

Zusammenfassung

Die Einführung von .NET Multi-platform App User Interface zwingt Entwickler tiefgreifende Modifizierungen an bereits realisierten Xamarin.Forms Anwendungen, die zukünftig nicht von Aktualisierungen profitieren werden, vorzunehmen.

Die Abhängigkeit von Microsoft kann durch einen automatisierten Umstieg auf das von Google entwickelte Flutter Framework gelöst und die betroffenen Apps auf eine bewährte, leistungsfähigere Basis gestellt werden. In dieser Arbeit wurde analysiert, ob eine rein automatisierte Transformation möglich ist, oder manuelle Arbeitsschritte notwendig sind. Zur Beantwortung der daraus resultierten Forschungsfrage wurde ein Source-To-Source Compiler entwickelt, der Projekte von Xamarin.Forms zu Flutter übersetzt. Erster Schritt der Kompilierung war die Analyse von Ansichten und deren dahinterstehende Logiken. Visuelle Elemente von Xamarin.Forms wurden im Anschluss durch Flutter Widgets ersetzt, die zu diesem Zweck in Vorlagen mit Platzhaltern gespeichert wurden. Nach einer Umwandlung der visuellen Eigenschaften konnten diese Platzhalter befüllt werden, um die Benutzeroberfläche zu finalisieren. Im letzten Schritt konnten mithilfe des Roslyn Compilers, die in C# geschriebene Anwendungslogik zu einem Syntaxbaum transformiert und nach dessen Durchlauf der Dart Quelltext zu konstruiert werden. Die ähnliche Syntax beider Sprachen erleichtert den Wechsel zwischen den Frameworks. Unterschiede bei den verfügbaren Typen und Modifizieren, bedurften jedoch einer genaueren Behandlung.

Zur Überprüfung ob der Prototyp das erwartete Ergebnis erzeugt, wurde eine speziell für diesen Zweck programmierte Xamarin.Forms App übersetzt und die erzeugte Flutter App mit der Ursprungsvariante verglichen. Das Resultat dieses Tests beweist, dass Xamarin.Forms Anwendungen automatisiert, ohne Funktionsverlust, nur mit leichten Designabweichungen zu Flutter überführt werden können und nur wenige manuelle Nacharbeitung erforderlich sind. Einschränkungen des Prototypens, wie die Ausklammerung der Übersetzung des plattformspezifischen Quelltexts und des Styles der Anwendung, könnten zukünftig durch eine Weiterentwicklung als Open-Source Projekt realisiert werden.

Abstract

The introduction of .NET Multi-platform App User Interface forces developers to make profound modifications to already realised Xamarin.Forms applications that will not benefit from future updates.

The dependency on Microsoft can be resolved by an automated switch to the Flutter Framework developed by Google and put the affected apps on a more on a proven basis with better performance. In this thesis it has been analysed whether a purely automated transformation is possible or whether manual steps are necessary. To answer the resulting research question, a source-to-source compiler was developed that translates projects from Xamarin.Forms to Flutter. The first step of the compilation was the analysis of views and their underlying logics. Visual elements of Xamarin.Forms were then replaced by Flutter widgets, which for this purpose were stored in templates with placeholders. After a conversion of the visual properties, these placeholders could be filled in in order to finalise the user interface. In the last step, with the help of the Roslyn compiler, the application logic written in C# was transformed into a syntax tree and after running through it, the Dart source code could be constructed. The similar syntax of both languages makes it easier to switch between the frameworks. Differences in the available types and modifiers, however, required more detailed treatment.

In order to check whether the prototype produces the expected result, a specially Xamarin.Forms app specially programmed for this purpose was translated and the generated flutter app app was compared with the original version. The result of this test proves that Xamarin.Forms applications can be transferred to Flutter automatically, without loss of function, with only slight design deviations. . Limitations of the prototype, such as the exclusion of the the translation of the platform-specific source code and the style of the application, could be be realised in the future through further development as an open-source Project.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface
GUI Graphical user interface
IDE Integrated development environment
JSON JavaScriptObjectNotation
MAUI Multi-platform App User Interface
SDK Software Development Kit
UI User Interface
UML Unified Modeling Language
URI Uniform Resource Identifier
WPF Windows Presentation Foundation
XAML Extensible Application Markup Language

Quellcodeverzeichnis

1 Einleitung

Die Entwicklung von verschiedenen mobilen Geräten mit unterschiedlichsten Hardwarekomponenten und Betriebssystemen hat einen stark fragmentierten Markt ergeben.[1] 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 begrenzt Zugriff auf die plattformspezifischen Funktionalitäten.[2]

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.[3]

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.[4]

[1] Vgl. Joorabchi 2016, S. 3.
[2] Vgl. Keist, Benisch und Müller 2016, S. 110ff.
[3] Vgl. Vollmer 2017, S. 295.
[4] Vgl. Wissel, Liebel und Hans 2017, S. 64.

1.1 Motivation

Im Mai 2020 hat Microsoft mit dem .NET Multi-platform App User Interface (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.[5]

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 zukunftssichere 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 Milliotatanen Entwickler Flutter verwendet und über 50.000 mobile Anwendungen programmiert.[6] Neben der hohen Verbreitung des Frameworks, konnte das Portal Stackoverflow in seinen jährlichen Umfragen auch eine hohe Beliebtheit unter Softwareentwicklern in den Jahren 2019[7] und 2020[8] ermitteln. Im März 2021 hat Google darüber hinaus die zweite Hauptversion von Flutter veröffentlicht, welche zusätzlich Support für die Entwicklung von Webseiten zur Verfügung stellt.[9]

[5] Vgl. Hunter 2020, Abgerufen am 28.10.2020.
[6] Vgl. Sneath 2020, Abgerufen am 28.10.2020.
[7] Vgl. Stack Exchange Inc. 2019, Abgerufen am 28.10.2020.
[8] Vgl. Stack Exchange Inc. 2020, Abgerufen am 28.10.2020.
[9] Vgl. Sells 2021, Abgerufen am 28.10.2020.

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 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 Forschunsgfrage 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 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 bestimmtes 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.[10] Die Beziehung zwischen Softwareentwicklern und Rechenanlagen mit Hilfe von Programmiersprachen wird in Abbildung 2.1 dargestellt.

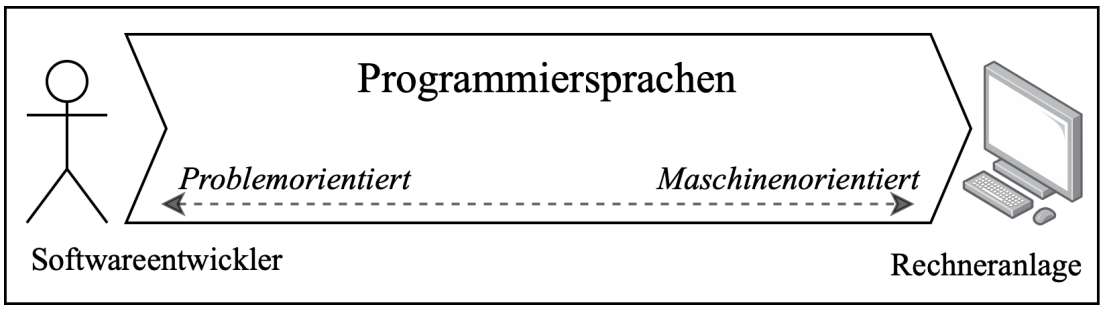


Abbildung 2.1: Programmiersprachen als Schnittstelle

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

[10] Vgl. Schneider 1975, S. 15.
[11] Vgl. Schneider 1975, S. 15.
[12] Vgl. Rutishauser 1952, S. 312.
[13] 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:[14]

Defintion 1: Compiler

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

Aus der 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 überführen, sodass er auf diesen ausgeführt werden kann.[15] Ein Source-to-Source Compiler, häufig auch als „Transpiler“ bezeichnet, ist ebenfalls eine besondere Ausprägung eines Compilers, die sich wie folgt definieren lässt.[16]

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, die für den Menschen les- und änderbar sind.[17]

[14] Vgl. Ullman et al. 2008, S. 1.
[15] Vgl. Ullman et al. 2008, S. 27.
[16] Vgl. Rohit, Aditi und Hardikar 2015, S. 1629.
[17] 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.[18] Die Syntax bezeichnet den Aufbau eines Programms, sie legt fest wie Sprachelemente aus anderen Sprachelementen zusammengesetzt sind. Im Gegensatz dazu beschreibt die Semantik die Bedeutung der Sprachelemente.[19] 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.[20]

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.[21]

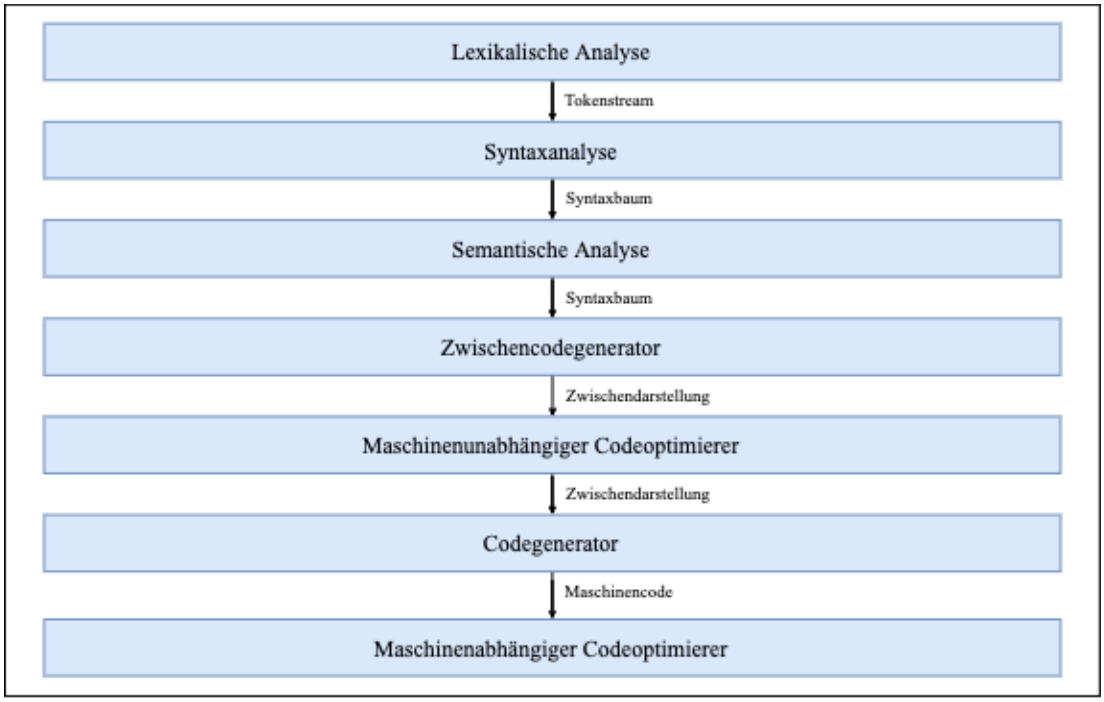


Abbildung 2.2: Phasen der Kompilierung[22]

[18] Vgl. Ullman et al. 2008, S. 6f.
[19] Vgl. Schneider 1975, S. 36.
[20] Vgl. Ullman et al. 2008, S. 6f.
[21] Vgl. Ullman et al. 2008, S. 6.
[22] 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.[23] 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[24]

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

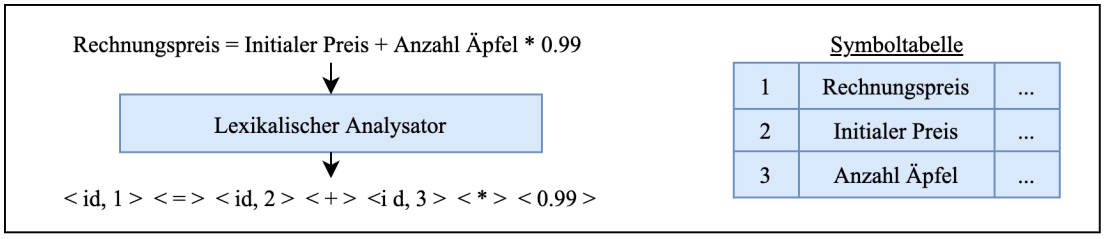


Abbildung 2.3: Exemplarische lexikalische Analyse[25]

Der Lexer interagiert mit anderen Komponenten eines Compilers. Klassischerweise wird der Lexer über den sogenannten Parser, welcher im nächsten Abschnitt eingeführt wird, zur Übermittlung von Tokens aufgefordert. Diese schematische Kommunikation wird in Abbildung 2.4 dargestellt.[26]

[23] Vgl. Ullman et al. 2008, S. 135 f.
[24] Vgl. Ullman et al. 2008, S. 137.
[25] Abbildung in Anlehnung an Ullman et al. 2008, S.10.
[26] Vgl. Ullman et al. 2008, S. 135.

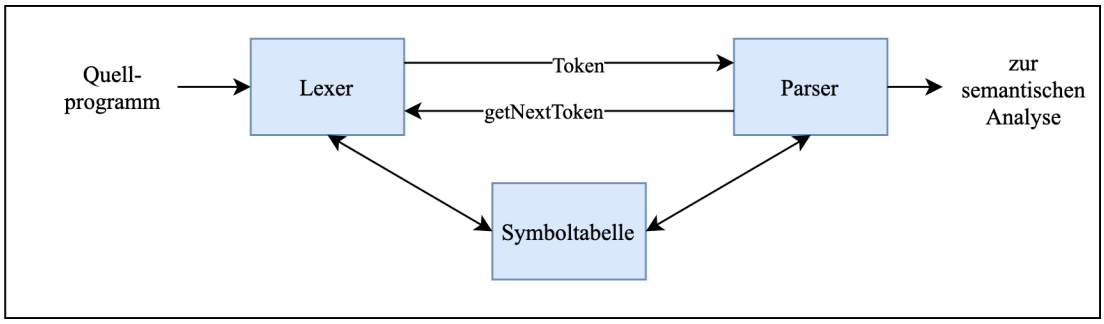


Abbildung 2.4: Interaktionen des Lexers[27]

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 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 Kompilation so einen genauen Hinweis auf den Ort des Fehlers geben.[28] 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.[29]

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.[30] 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 ablesbar, dass das Ergebnis der exemplarischen Rechnung an den Speicherort des Bezeichners Rechnungspreis abgelegt werden muss.[31]

[27] Abbildung in Anlehnung an Ullman et al. 2008, S.135.
[28] Vgl. Ullman et al. 2008, S. 135.
[29] Vgl. Ullman et al. 2008, S. 136.
[30] Vgl. Ullman et al. 2008, S. 9.
[31] Vgl. Ullman et al. 2008, S. 9.

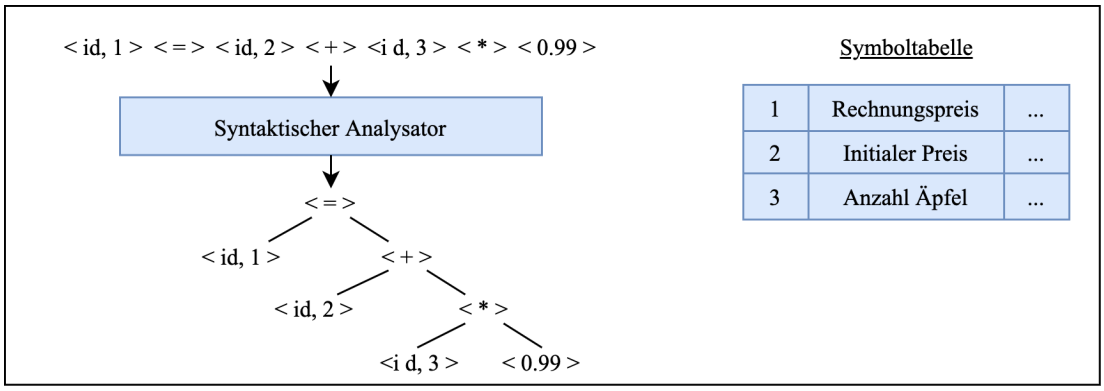


Abbildung 2.5: Exemplarischer Syntaxbaum[32]

2.5 Semantische Analyse

Bei der semantischen Analyse wird der Syntaxbaum als Aufgliederung der Programmkonstruktur, zusammen mit den Informationen aus der Symboltabelle verwendet, um das Quellprogramm auf semantische Konsistenz mit der Sprachdefinition zu überprüfen.[33] 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.[34]

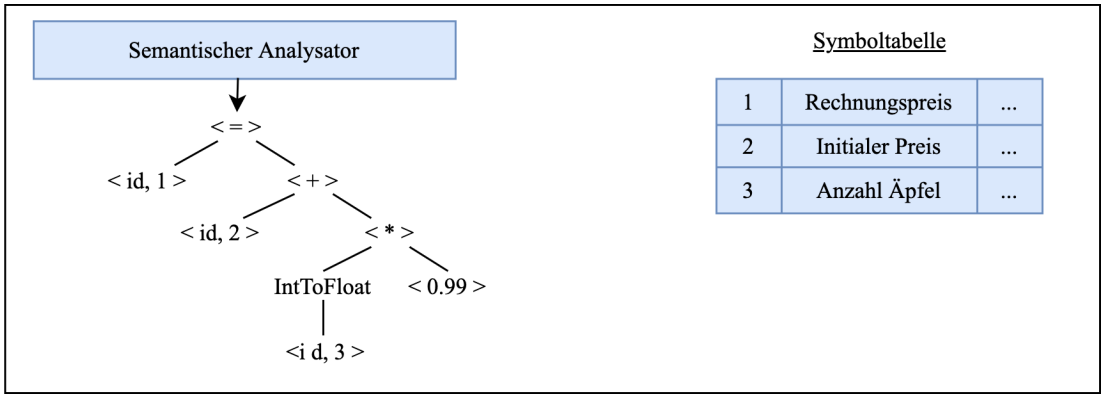


Abbildung 2.6: Exemplarische Typüberprüfung[35]

[32] Abbildung in Anlehnung an Ullman et al. 2008, S.10.
[33] Vgl. Wilhelm, Seidl und Hack 2012, S. 157.
[34] Vgl. Ullman et al. 2008, S. 9ff.
[35] 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 die eines Syntaxbaums, erstellen. Nach der semantischen Analyse stellen viele Compiler eine maschinennahe Zwischendarstellung auf niedriger Abstraktionsebene her, die eigentlich für maschinenabhängige Aufgaben wie die Befehlsauswahl geeignet ist. Eine Zwischendarstellung, die 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.[36] Die variable Anzahl von Zwischendarstellungen bei der Kompilierung werden in Abbildung 2.7 skizziert.

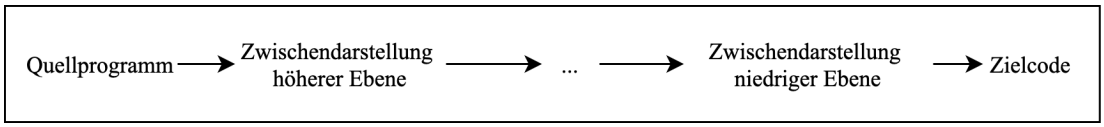


Abbildung 2.7: Zwischendarstellungen[37]

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.[38] Die Codeoptimierung, die ein Compiler vornimmt, ist im Laufe der Zeit wichtiger und umfangreicher geworden. Grund für die zunehmenden Anforderungen 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.[39]

[36] Vgl. Ullman et al. 2008, S. 433.
[37] Abbildung in Anlehnung an Ullman et al. 2008, S.433.
[38] Vgl. Ullman et al. 2008, S. 11f.
[39] 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.[40]

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 Rosyln 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.[41]

[40] Vgl. Ullman et al. 2008, S. 618f.
[41] 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.[42] 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.

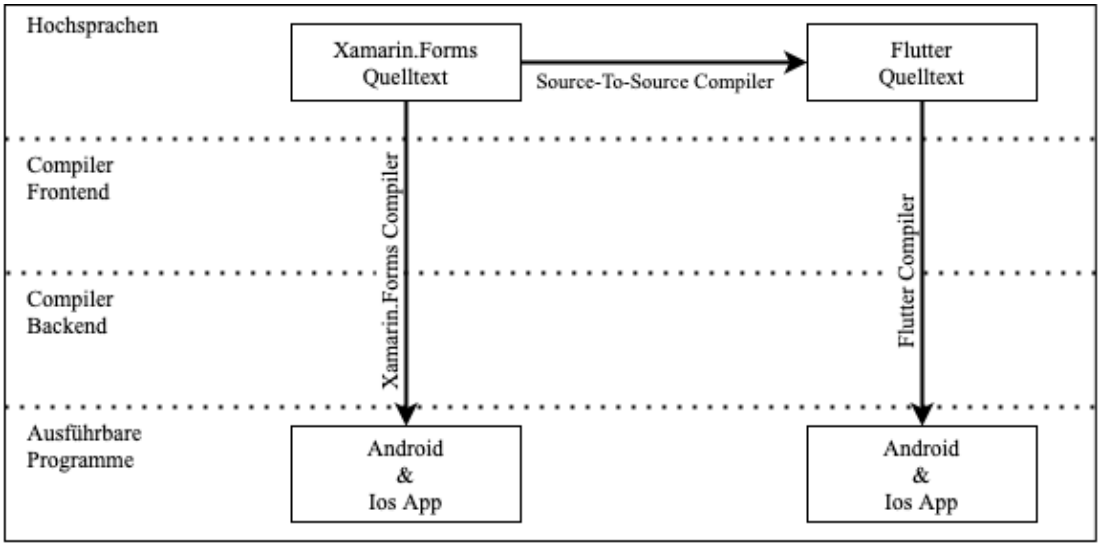


Abbildung 3.1: Umfeld des Source-To-Source Compilerns

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.

[42] Vgl. Balzert 2011, S.6.

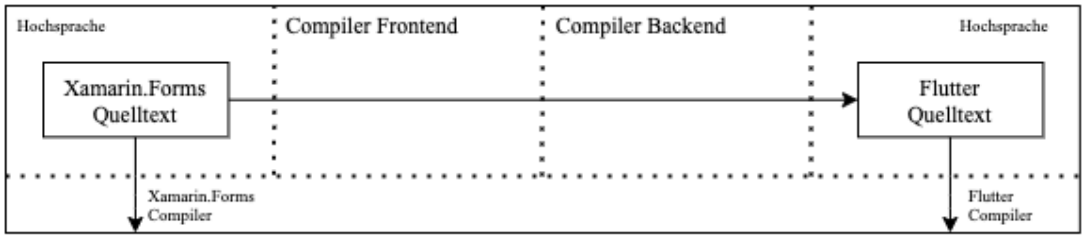


Abbildung 3.2: Source-To-Source Compiler Aufbau

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 veröffentliche Xamarin.Forms in der Version 5.0.0.2012 zu Flutter übersetzten.
- 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,[43] weswegen die Konstruktion von Benutzeroberflächen mit C# in dieser Arbeit nicht berücksichtigt wird.

[43] Vgl. Microsoft Corporation 2017a, Abgerufen am 25. 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, setzten sich Xamarin.Forms Projektmappen aus verschiedenen Dateien zusammen, von denen ausschließlich die Klassen beinhaltenden Dateien mit der Endung ‚cs' analysiert werden können.

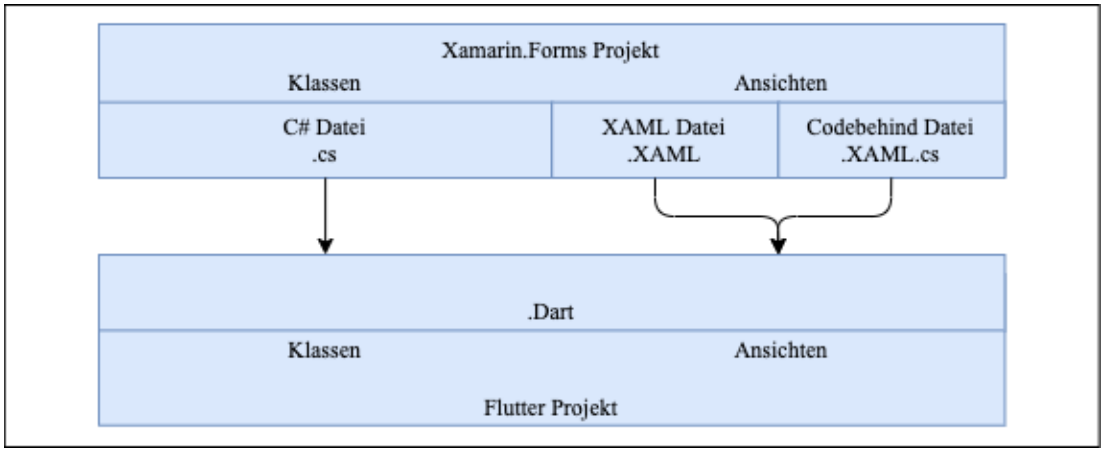


Abbildung 3.3: Compiler-Struktur

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.[44] 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.

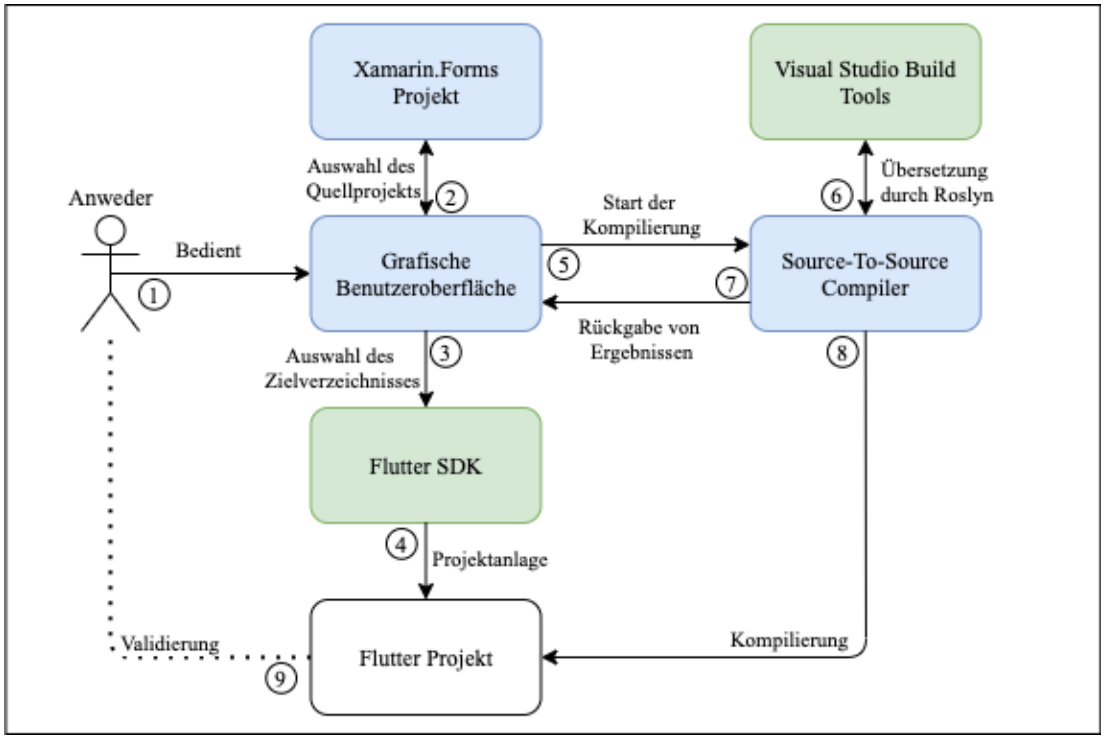


Abbildung 3.4: Source-To-Source Compiler Informationsfluss

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

[44] Vgl. Google LLC 2021d, Abgerufen am 25. 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.

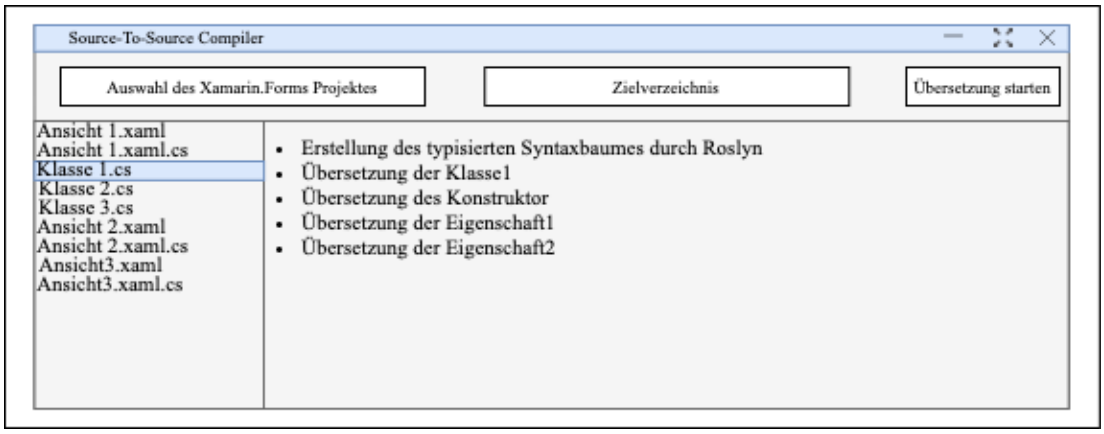


Abbildung 3.5: Mockup der grafischen Oberfläche

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.

```
var photo = await MediaPicker.PickPhotoAsync();

if(photo == null)
{
 Console.WriteLine("No image selected.");
}
```

Quelltext 3.1: Bilderauswahl in Xamarin.Forms

```
final pickedFile = await picker.getImage(source: ImageSource.camera);

setState(() {
    if (pickedFile != null) {
          _image = File(pickedFile.path);
    } else {
          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. Wärend das Flutter Projekt alle notwendigen Inhalte für iOS und Android inkludiert,[45] 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.[46] 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

[45] Vgl. Biessek 2019, S. 113.
[46] 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.[47] Android speichert Metadaten innerhalb der ‚AndroidManifest.xml'.[48] 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.[49]

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.[50] 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.[51] Zur Verwendung nativer Bilddateien müssen die Bilder in Xamarin.Forms zu jedem Anwendungsprojekt hinzugefügt werden und vom gemeinsamen Xamarin.Forms-Code referenziertwerden. 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.[52] Der Startbildschirm (engl. Splashscreen) ist der Einstiegspunkt der mobilen App. Er dient als Ladebildschrim 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.[53] ,

[47] Vgl. Microsoft Corporation 2017c, Abgerufen am 25. April 2021.
[48] Vgl. Microsoft Corporation 2018a, Abgerufen am 25. April 2021.
[49] Vgl. Vaibhavi 2020, Abgerufen am 25. April 2021.
[50] Vgl. Google LLC 2020h, Abgerufen am 25. April 2021.
[51] Vgl. Microsoft Corporation 2020d, Abgerufen am 25. April 2021.
[52] Vgl. Google LLC 2020c, Abgerufen am 25. April 2021.
[53] Vgl. Google LLC 2020b, Abgerufen am 25. 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.[54] In Flutter werden die Fonts wie in der neueren Version von Xamarin.Forms in einem Ordner abgelegt.[55] Eine Verwendung des Schriftsatzes ‚MyCustomFont' in Flutter wird in Quelltext 4.1 dargestellt.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Sample App"),
    ),
    body: Center(
      child: Text(
        'This is a custom font text',
        style: TextStyle(fontFamily: 'MyCustomFont'),
      ),
    ),
  );
}
```

Quelltext 4.1: Verwendung von Schriftsätzen in Flutter[56]

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, Funktionenalitäten in den Plattformprojekten aus dem geteilten Code aufzurufen,[57] sowie ‚Custom Renderers', die es ermöglichen das Ausse-

[54] Vgl. Versluis 2020, Abgerufen am 25. April 2021.
[55] Vgl. Google LLC 2020o, Abgerufen am 25. April 2021.
[56] Quelltext in Anlehnung an Google LLC 2020o, Abgerufen am 25. April 2021.
[57] Vgl. Microsoft Corporation 2019c, Abgerufen am 25. April 2021.

hen und Verhalten von Steuerlementen anzupassen.[58] In Flutter wird für die Ausführung von plattformspezifischen Funktionalitäten auf ‚platform-channels' zurückgegriffen.[59] Für ‚Custom Renderer' gibt es bei 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.[60] 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.[61] In Quelltext 4.2 wird ein Auschnitt 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[62]

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.

[58] Vgl. Microsoft Corporation 2019b, Abgerufen am 25. April 2021.
[59] Vgl. Google LLC 2020t, Abgerufen am 25. April 2021.
[60] Vgl. Microsoft Corporation 2020a, Abgerufen am 25. April 2021.
[61] Vgl. Google LLC 2020p, Abgerufen am 25. April 2021.
[62] Quelltext in Anlehnung an Google LLC 2020p, Abgerufen am 25. 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.[63]

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[64], oder ‚Realm', eine Datenbank optimiert für mobile Endgeräte.[65] Beide Datenbanken stehen auch als Plugin für ‚Flutter' zur Verfügung, wobei SQLite ausgereift ist,[66] während ‚Realm' erst am 5 November 2020 Support für Flutter angekündigt hat und noch nicht offiziell zur Verfügung steht.[67]

Die Xamarin.Essentials Erweiterung stellt Entwicklern das ‚Settingsplugin' bereit, welches die Sicherung von Präferenzen und App-Einstellungen erlaubt.[68] In Flutter wird für diese Speicherung mit Hilfe des Plugins ‚shared_preferences' auf die gleichen plattformspezifischen API zugegriffen.[69] Die Verwendung der gleichen APIs ist ein wichtiger Faktor da die von Anwendern gespeicherten Daten auch nach einem Frameworkwechsel zu Flutter, noch verfügbar sind.

[63] Vgl. Google LLC 2020t, Abgerufen am 25. April 2021.
[64] Vgl. SQLite Consortium 2020, Abgerufen am 25. April 2021.
[65] Vgl. MongoDB Inc. 2020, Abgerufen am 25. April 2021.
[66] Vgl. Tekartik 2020, Abgerufen am 25. April 2021.
[67] Vgl. Ward 2020, Abgerufen am 25. April 2021.
[68] Vgl. Microsoft Corporation 2019a, Abgerufen am 25. April 2021.
[69] Vgl. Google LLC 2020l, Abgerufen am 25. 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.[70] Mithilfe des Plugins ‚url_launcher' kann die Funktionalität zum Öffnen von anderen Anwendungen zu Flutter Apps hinzugefügt werden.[71] Die unterstützten URI-Schemata und ihre Aktionen werden in Tabelle 4.1 dargestellt.

URI-Schemata	**Aktion**
http:<URL>	URL wird im Standardbrowser geöffnet
mailto:<E-Mail Adresse>	Erstellt eine Email an die angegebene E-Mail Adresse
tel:<Telefonnummer>	Ruft die Rufnummer an
sms:<Telefonnummer>	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.[72] Xamarin.Forms bietet dafür die drei Methoden ‚OnStart' , ‚OnResume' und ‚OnSleep' an, die aufgerufen werden, wenn sich der Status verändert.[73] Bei Flutter kann auf die ‚didChangeAppLifecycleState' Methode zurückgegriffen werden, die bei Änderungsereignissen ausgelöst wird und die ebenfalls die drei Lebenszyklen beinhaltet.[74]

[70] Vgl. Microsoft Corporation 2020e, Abgerufen am 25. April 2021.
[71] Vgl. Google LLC 2020n, Abgerufen am 25. April 2021.
[72] Vgl. Apple Inc. 2020, Abgerufen am 25. April 2021.
[73] Vgl. Microsoft Corporation 2020k, Abgerufen am 25. April 2021.
[74] Vgl. Google LLC 2020e, Abgerufen am 25. 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 Anordung auf der Benutzeroberfläche verantwortlich.[75] 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.[76]

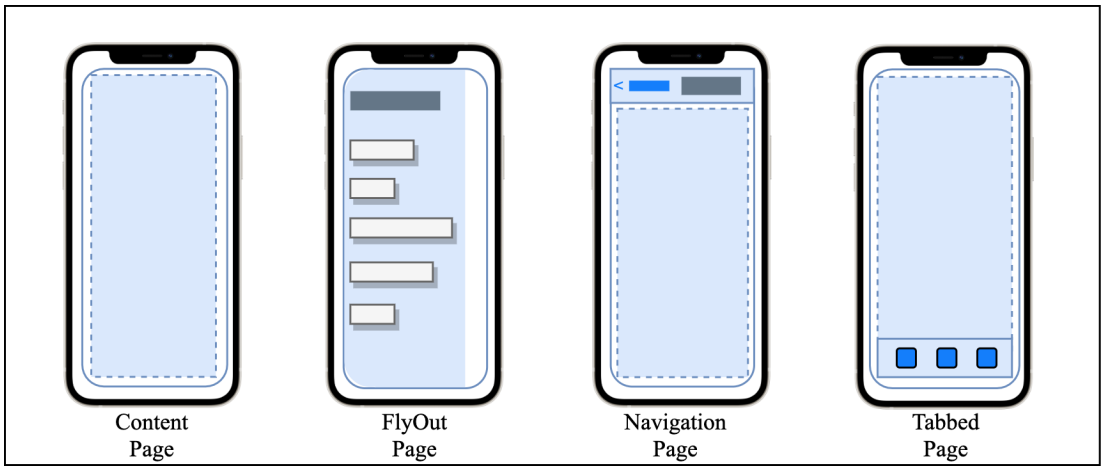


Abbildung 4.1: Xamarin.Forms Pages[77]

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

[75] Vgl. Ritscher 2020, Abgerufen am 25. April 2021.
[76] Vgl. Microsoft Corporation 2016b, Abgerufen am 25. April 2021.
[77] Abbildung in Anlehnung an Microsoft Corporation 2016b, Abgerufen am 25. April 2021.

dar.[78] 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 ‚TabbedPage'.

```
<?xml version="1.0" encoding="UTF-8" ?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            x:Class="MasterThesisSample.SampleTabbedPage">
    <NavigationPage Title="Tab 1"/>
    <NavigationPage Title="Tab 2"/>
</TabbedPage>
```

Quelltext 4.3: Xamarin.Forms ‚TabbedPage' Definition[79]

Es wird eine ‚TabbedPage' 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 ‚TabbedPage', die aus einer Sammlung von Registerkarten besteht, finden sich im Quelltext die Eigenschaften ‚Flyout' und ‚Detail'.

```
<?xml version="1.0" encoding="UTF-8" ?>
<FlyoutPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            x:Class="MasterThesisSample.SampleFlyoutPage">
    <FlyoutPage.Flyout>
        <ContentPage/>
    </FlyoutPage.Flyout>
    <FlyoutPage.Detail>
        <NavigationPage/>
    </FlyoutPage.Detail>
</FlyoutPage>
```

Quelltext 4.4: Xamarin.Forms ‚FlyoutPage' Definition[80]

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.[81] Quelltext 4.5 zeigt die Realisierung einer ‚Material Design ' App in Flutter.

[78] Vgl. Microsoft Corporation 2016b, Abgerufen am 25. April 2021.
[79] Quelltext in Anlehnung an Microsoft Corporation 2020o, Abgerufen am 25. April 2021.
[80] Quelltext in Anlehnung an Microsoft Corporation 2020l, Abgerufen am 25. April 2021.
[81] Vgl. Google LLC 2020f, Abgerufen am 25. April 2021.

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(primarySwatch: Colors.blue,),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

Quelltext 4.5: Flutter ‚MaterialApp' Definition[82]

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.[83] 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.

```
appBar: AppBar(
    bottom: TabBar(
        tabs: [ Tab(icon: Icon(Icons.directions_transit)),
                Tab(icon: Icon(Icons.directions_bike)),],
    ),
    title: Text('Tabs Demo'),
    ),
    body: TabBarView(
    children: [ Icon(Icons.directions_transit),
                Icon(Icons.directions_bike), ],
),
```

Quelltext 4.6: Flutter ‚Tab Layout' Definition[84]

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[85] und die Webseite "Flutter for Xamarin.Forms Developers"[86] wurde für die Recherche des Gegenstückes verwendet. Entsprechende Ergebnisse der Suche können in Tabelle 4.2 abgelesen werden.

[82] Quelltext in Anlehnung an Google LLC 2020s, Abgerufen am 25. April 2021.
[83] Vgl. Google LLC 2020j, Abgerufen am 25. April 2021.
[84] Quelltext in Anlehnung an Google LLC 2020r, Abgerufen am 25. April 2021.
[85] Vgl. Google LLC 2020q, Abgerufen am 25. April 2021.
[86] Vgl. Google LLC 2020f, Abgerufen am 25. 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.[87] Flutter hat eine ähnliche Implementierung, welche die Widgets ‚Navigator' und ‚Routen' verwendet.[88]

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

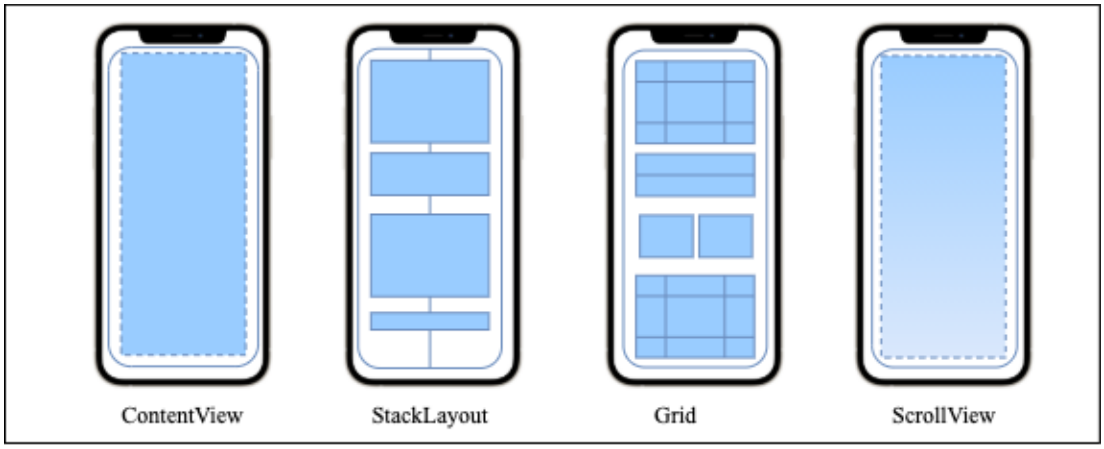


Abbildung 4.2: Xamarin.Forms Layouts[89]

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

[87] Vgl. Microsoft Corporation 2020n, Abgerufen am 25. April 2021.
[88] Vgl. Google LLC 2020k, Abgerufen am 25. April 2021.
[89] Abbildung in Anlehnung an Microsoft Corporation 2018b, Abgerufen am 25. 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.[90] Basierend auf diesen verfügbaren Layouts werden in Tabelle 4.3 die entsprechenden Flutter Widgets entgegengesetzt.

Xamarin.Forms Layout	**Flutter Widget**
AbsolutLayout	Positioned
ContentView	StatelessWidget
Frame	BoxDecoration
Grid	GridView oder Stack
ScrollView	SingleChildScrollView
StackLayout	Row und Column
ReleativLayout	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.[91] 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.[92]

[90] Vgl. Microsoft Corporation 2018b, Abgerufen am 25. April 2021.
[91] Vgl. Google LLC 2020g, Abgerufen am 25. April 2021.
[92] Vgl. Microsoft Corporation 2020j, Abgerufen am 25. 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.[93] Für die Steuerelemente kann nun eine Gegenüberstellung zwischen Xamarin.Forms Elementen und Flutter Widgets vorgenommen werden, wie in Tabelle 4.4 dargestellt.

Xamarin.Forms Steuerelement	**Flutter Widget**
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.[94]

[93] Vgl. Microsoft Corporation 2018b, Abgerufen am 25. April 2021.
[94] Vgl. Google LLC 2020d, Abgerufen am 25. 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.

Xamarin.Forms Steuerelemente	**Flutter Widget**
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.

```
<Button Text="Click Me!"
    Grid.Row="0"
    Font="Large"
    BorderWidth="1"
    HorizontalOptions="End"
    VerticalOptions="CenterAndExpand"
    Clicked="Button_Clicked" />
```

Quelltext 4.7: Xamarin.Forms Button Initialisierung

```
private void Button_Clicked(object sender, EventArgs e)
{
 //Button pressed
}
```

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.

```
TextField(
  keyboardType: TextInputType.multiline,
  maxLines: null,
)
```

Quelltext 4.9: Eingabefeld mit mehreren Zeilen in Flutter

Steuerelemente zur Wertsetzung

Wertsetzung bedeutet das Ergänzen von Steuerelmenten 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.

```
GestureDetector(
    onTap: () async {
        TimeOfDay picked = await showTimePicker(
        context: context,
        initialTime: TimeOfDay.now(),
        builder: (BuildContext context, Widget child) {
           return MediaQuery(
           data: MediaQuery.of(context)
           .copyWith(alwaysUse24HourFormat: true),
           child: child,
           );
        },);
     },
     child: Text("SetTime",textAlign: TextAlign.center,)
);
```

Quelltext 4.10: Verwendung von Timepickern in Flutter[95]

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 ausführt 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

[95] Quelltext in Anlehnung an Google LLC 2021g, Abgerufen am 25. April 2021.

Sammlungsanzeigende Steuerelemente

Die überwiegende Anzahl von mobilen Anwendungen visualisiert Sammlungen von Daten.[96] Xamarin.Forms stellt hierfür ebenfalls Steuelemente zur Verfügung. ‚CarouselView' 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.

Xamarin.Forms Steuerelemente	**Flutter Widget**
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 Fluttter gegenüber.

Xamarin.Forms Steuerelemente	**Flutter Widget**
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.

[96] Vgl. Hindrikes 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 ‚HorzintalOptions' und ‚VerticalOptions' ausgerichtet werden. Für diese Eigenschaften können die folgenden Werte gesetzt werden:[97]

- ‚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.[98]

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

[97] Vgl. Microsoft Corporation 2017b, Abgerufen am 25. April 2021.
[98] Vgl. Microsoft Corporation 2017b, Abgerufen am 25. April 2021.

hier ‚Start', ‚End' und ‚Center' zur Verfügung.[99] 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 zurrückgegriffen, welches die gleiche Aufgabe mit weniger Quelltext realisiert.[100]

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.[101] 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.[102]

4.4.5 Animationen

Gut gestaltete Animationen machen eine Benutzeroberfläche intuitiver, tragen zum eleganten Erscheinungsbild einer ausgefeilten App bei und verbessern das Benutzererlebnis.[103] 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.[104] 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.[105]

[99] Vgl. Google LLC 2021b, Abgerufen am 25. April 2021.
[100] Vgl. Google LLC 2021f, Abgerufen am 25. April 2021.
[101] Vgl. Microsoft Corporation 2020m, Abgerufen am 25. April 2021.
[102] Vgl. Google LLC 2020m, Abgerufen am 25. April 2021.
[103] Vgl. Google LLC 2020i, Abgerufen am 25. April 2021.
[104] Vgl. Microsoft Corporation 2020g, Abgerufen am 25. April 2021.
[105] Vgl. Google LLC 2020i, Abgerufen am 25. 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.

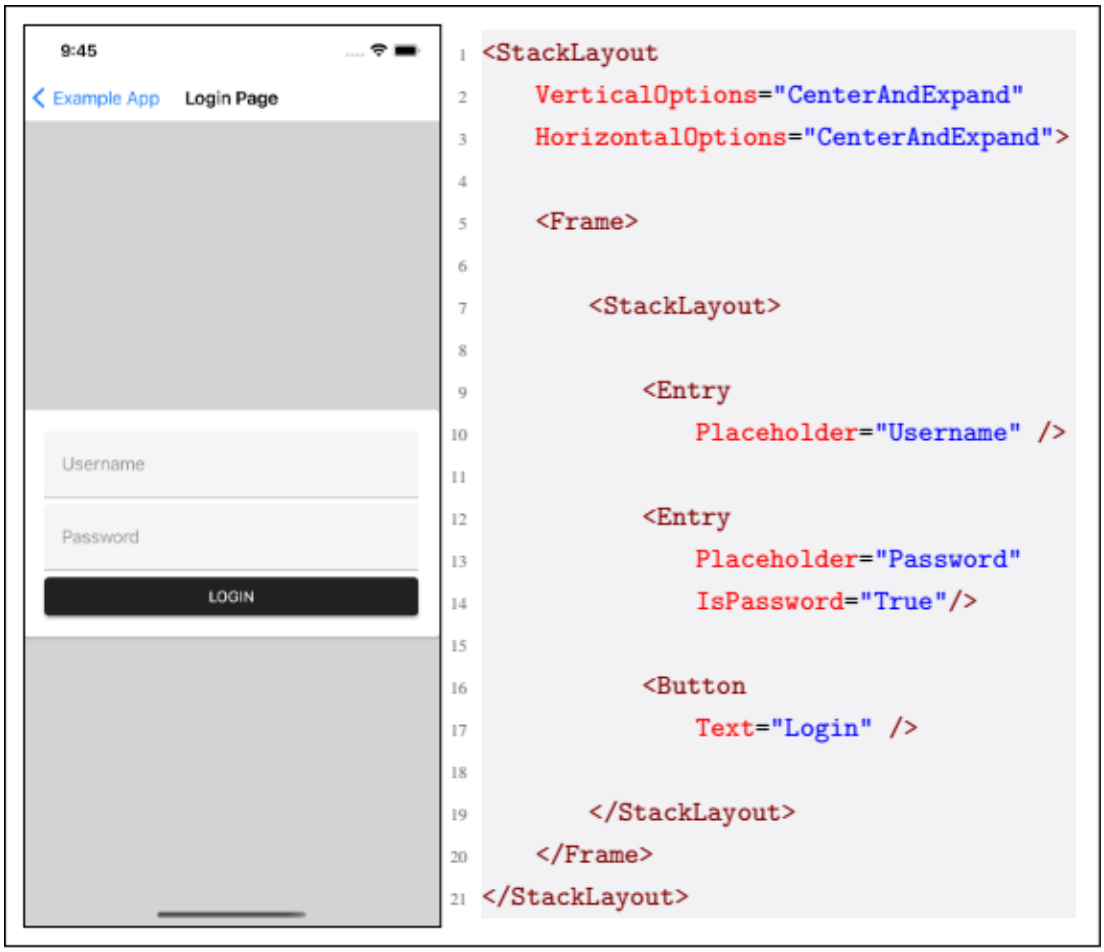


Abbildung 4.4: Darstellung einer exemplarischen Login-Page

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.

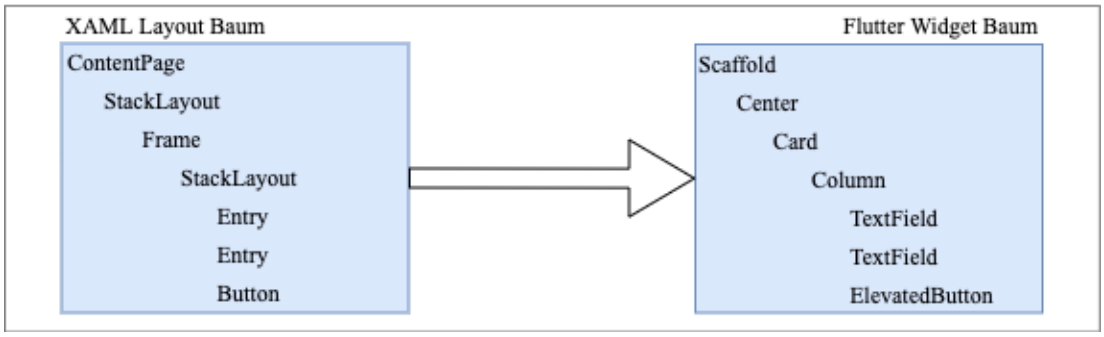


Abbildung 4.5: Layout-Baum Überführung von Xamarin.Forms zu Flutter

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.

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.grey,
      appBar: AppBar( title: Text("LoginPage"),),
      body: Center(
        child: Card(
          child: Column(
            mainAxisSize: MainAxisSize.min,
            children: <Widget>[
              TextField(
                obscureText: false,
                decoration: InputDecoration(
                  border: OutlineInputBorder(),
                  labelText: 'Username',
                ),
              ),
              TextField(
                obscureText: true,
                decoration: InputDecoration(
                  border: OutlineInputBorder(),
                  labelText: 'Password',
                ),
              ),
              ElevatedButton(
                child: Text('Login'),
              ),
            ],
          ),
        ),
      ),
    );
  }
```

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 ‚SizedBox' 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 nahekommt.

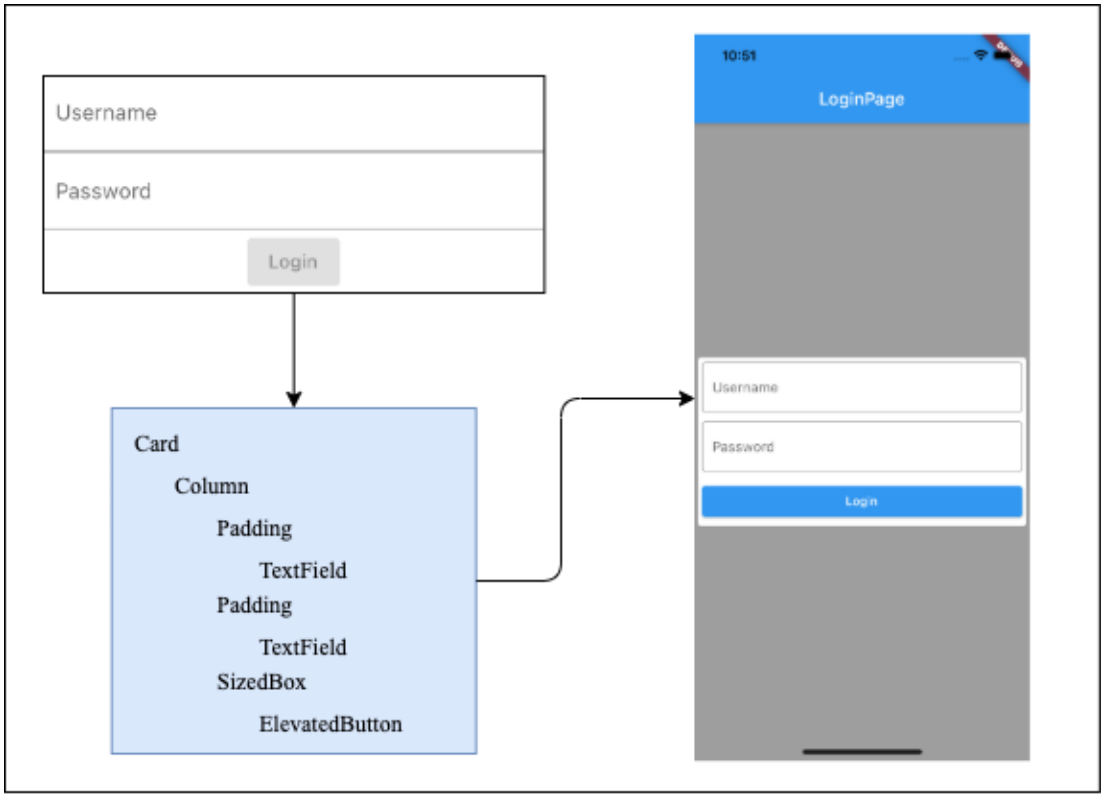


Abbildung 4.6: Flutter LoginPage Screenshot und Widget-Baum

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.[106] 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 vereinzelten Unterschieden bei der Realisierung, Verwendung finden. Die sogenannten Objekte werden durch bestimmte, charakteristische Merkmale beschrieben, die in der Klassendefinition festgelegt werden müssen.[107]

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.[108] C# bietet die

[106] Vgl. Pedley 2019, Abgerufen am 25. April 2021.
[107] Vgl. Witte 2013, S. 11f.
[108] 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.[109] 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.[110]

Die Programmiersprache Dart war bis zu der Version 2.12 ‚nicht Null sicher'. Da nicht gewährleistet war, das 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.

```
int myVariable = null;
if (myVariable != null)
    myVariable = 0;
```

Quelltext 5.1: Null-Sicherheit in Dart 1.x[111]

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.[112] Quelltext 5.2 visualisiert die Arbeit mit den neuen Datentypen in Dart.

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

Quelltext 5.2: Null-Sicherheit in Dart 2.x[113]

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

[109] Vgl. Microsoft Corporation 2020h, Abgerufen am 25. April 2021.
[110] Vgl. Bayer 2008, S. 167.
[111] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[112] Vgl. Google LLC 2021a, Abgerufen am 25. April 2021.
[113] Quelltext in Anlehnung an Google LLC 2021a, Abgerufen am 25. April 2021.

5.1.2 Datentypen

Die Unterscheidung von Referenz und Wertetypen 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 expliziert ein Objekt generiert oder ein bestehendes zugewiesen werden. Die Erzeugung eines neuen Objekts erfolgt durch den Operator ‚new'.[114] Dart analysiert, wann ein neues Objekt initiiert werden muss und benötigt daher keinen ‚new' Operator, wie in Quelltext 5.3 dargestellt.[115]

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

Quelltext 5.3: Objekterzeugung ohne ‚new' Keyword in Dart[116]

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

[114] Vgl. Eller und Kofler 2005, S. 93.
[115] Vgl. Google LLC 2020a, Abgerufen am 25. April 2021.
[116] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. 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.[117] Der Compiler muss alle ganzzahligen C# Datentypen zu Dart ‚Integer', bis auf ‚long' und ‚ulong' die von ‚BigInt' repräsentiert werden, übersetzten .

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.[118] In Dart steht für die Arbeit mit Kommazahlen nur der Datentyp ‚double' zur Verfügung,[119] 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 ‚CharacterCode' als Übergabewert erwartet und das String Objekt mit einem Textzeichen erzeugt. Dies wird in Quelltext 5.4 dargestellt.

[117] Vgl. Star 2019, Abgerufen am 25. April 2021.
[118] Vgl. Microsoft Corporation 2020c, Abgerufen am 25. April 2021.
[119] Vgl. Google LLC 2021c, Abgerufen am 25. April 2021.

```
String CharContainingString;
CharContainingString = String.fromCharCodes([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.[120] 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.

```
List<int> list = [5,10,20];
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.[121] 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.

[120] Vgl. Kühnel 2019, S. 110f.
[121] Vgl. Google LLC 2020a, Abgerufen am 25. April 2021.

```dart
var Map = {'Username':'Julian','Password':'qwertz'};
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.

```dart
class _PrivateClass {
  String _privateField;
}

class PublicClass {
  String publicField;
}
```

Quelltext 5.7: Private und Public Definitionen in Dart[122]

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.[123] Der folgende Quelltext zeigt eine Vererebung in Dart.

[122] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[123] Vgl. Microsoft Corporation 2020i, Abgerufen am 25. April 2021.

```dart
class BaseClass {
  void myFunction() {}
}

class MyClass extends BaseClass {
  void myOtherFunction() {
    // do something
  }

  // every function is overridable in Dart.
  @override
  void myFunction() {
    // do something
  }
}
```

Quelltext 5.8: Vererbung in Dart[124]

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.[125] 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.

```dart
class MyMixin {
  void sayHello() => print("Hello!");
}

class MyClass with MyMixin
{
}
```

Quelltext 5.9: Mixin's in Dart[126]

[124] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[125] Vgl. Kühnel 2019, S. 258f.
[126] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. 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.

```
using System.Collections.Generic;

namespace SampleApp.Models
{
    public class PublicExampleClass
    {
        // Konstruktor
        public PublicExampleClass()
        {
            PublicChar = 'j';
        }

        // Ganzzahl
        private int PrivateInt;

        // Gleitkommazahl
        private double PrivateDouble;
        public float PublicFloat;

        // Boolean
        private bool PrivateBool;

        // Textzeichen
        public char PublicChar;

        // Zeichenfolge
        public string PublicString;

        // Datenfeld
        public char[] PublicCharArray;

        // Auflistung
        public List<int> _PublicIntList;

        //Hashtabelle
        public Dictionary<int, char> PublicHashTable;

    }
}
```

Quelltext 5.10: Beispielklasse in C#

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

```
class ExampleClass {
  // Konstruktor
  PublicExampleClass() {
    PublicChar = String.fromCharCode(106);
  }

  // Ganzzahl
  int _PrivateInt;

  // Gleitkommazahl
  double _PrivateDouble;
  double PublicFloat;

  // Boolean
  bool _PrivateBool;

  // Textzeichen
  String PublicChar;

  // Zeichenfolge
  String PublicString;

  // Datenfeld
  List<String> PublicCharArray;

  // Auflistung
  List<int> PublicIntList;

  // Hashtabelle
  Map<int, String> PublicHashTable;
}
```

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.[127] Dart hat keine Namespaces, stattdessen werden Pakete und Dateien direkt importiert. Somit

[127] Vgl. Microsoft Corporation 2015c, Abgerufen am 25. 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.

```
// Import a core dart library
import 'dart:async';

// Import a package
import 'package:flutter/material.dart';

// Import another file in your application
import 'myfilename.dart' as filename;
```

Quelltext 5.12: Importieren von Paketen in Dart[128]

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.[129]

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

```
class State<Type>
{
  Type getValue() => null;
}
void processState(State state) {
  dynamic value = state.getValue();
}
void processStateWithType(State<String> state) {
  String value = state.getValue();
}
```

Quelltext 5.13: Generics in Dart[131]

[128] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[129] Vgl. Microsoft Corporation 2015b, Abgerufen am 25. April 2021.
[130] Vgl. Cheng 2019, S. 98.

5.4 Integrierte Verweistypen

Integrierte Verweistypen (engl. Delegates) referenzieren Methoden mit einer bestimmten Parameterliste und dem Rückgabetyp. ‚Delegates' werden dazu verwendet, Methoden als Parameter zu übergeben. Im ereignisgesteuerten Framework Xamarin.Forms werden die Methoden zur Behandlung von Ereignissen durch ‚Delegates' aufgerufen.[132]

In Dart kann der Typ ‚Typedef' verwendet werden, um eine Methodensignatur zu definieren und eine Instanz davon in einer Variablen vorzuhalten.[133] Dies wird in Quelltext 5.14 dargestellt.

```
class State<Type>
{
  Type getValue() => null;
}
void processState(State state) {
  dynamic value = state.getValue();
}
void processStateWithType(State<String> state) {
  String value = state.getValue();
}
```

Quelltext 5.14: Delegates in Dart[134]

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.[135]

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

[131] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[132] Vgl. Microsoft Corporation 2015a, Abgerufen am 25. April 2021.
[133] Vgl. Pedley 2019, Abgerufen am 25. April 2021.
[135] Vgl. Microsoft Corporation 2020b, Abgerufen am 25. April 2021.

```dart
void main() async {
  var result = await myFunction();
}

Future<String> myFunction() {
  // Similar to Task.FromResult
  return Future.value('Hello');
}
```

Quelltext 5.15: Async und Await in Dart[136]

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.[137] 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

[136] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[136] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[137] Vgl. Microsoft Corporation 2016a, Abgerufen am 25. 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.

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
[...]
  loadData() async {
    String dataURL = "https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = jsonDecode(response.body);
    });
  }
}
```

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.[138] 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.[139] In C# kann für beide Prozesse der Namespace System.Text.Json verwendet werden,[140] während in Dart die Bibliothek dart:convert zur Verfügung steht.[141]

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

[137] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. April 2021.
[138] Vgl. Dobrenz, Gewinnus und Saumweber 2018, S. 684.
[139] Vgl. Google LLC 2021e, Abgerufen am 25. April 2021.
[140] Vgl. Microsoft Corporation 2020f, Abgerufen am 25. April 2021.
[141] Vgl. Google LLC 2021e, Abgerufen am 25. 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.

```
StreamController streamController = new StreamController.broadcast();

void main() {
  streamController.stream.listen((args) => {
        // do something
      });

  streamController.add("hello");

  streamController.close();
```

Quelltext 5.17: Events in Dart[142]

[142] Quelltext in Anlehnung an Pedley 2019, Abgerufen am 25. 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, da auf dieser Plattform Erweiterungen für Roslyn zur Verfügung stehen. Die zu entwickelnde Projektmappe besteht also aus dem Source to Source Compiler mit Rosyln 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 dient das in Abbildung 6.1 dargestellte Unified Modeling Language (UML)-Aktivitätsdiagramm.

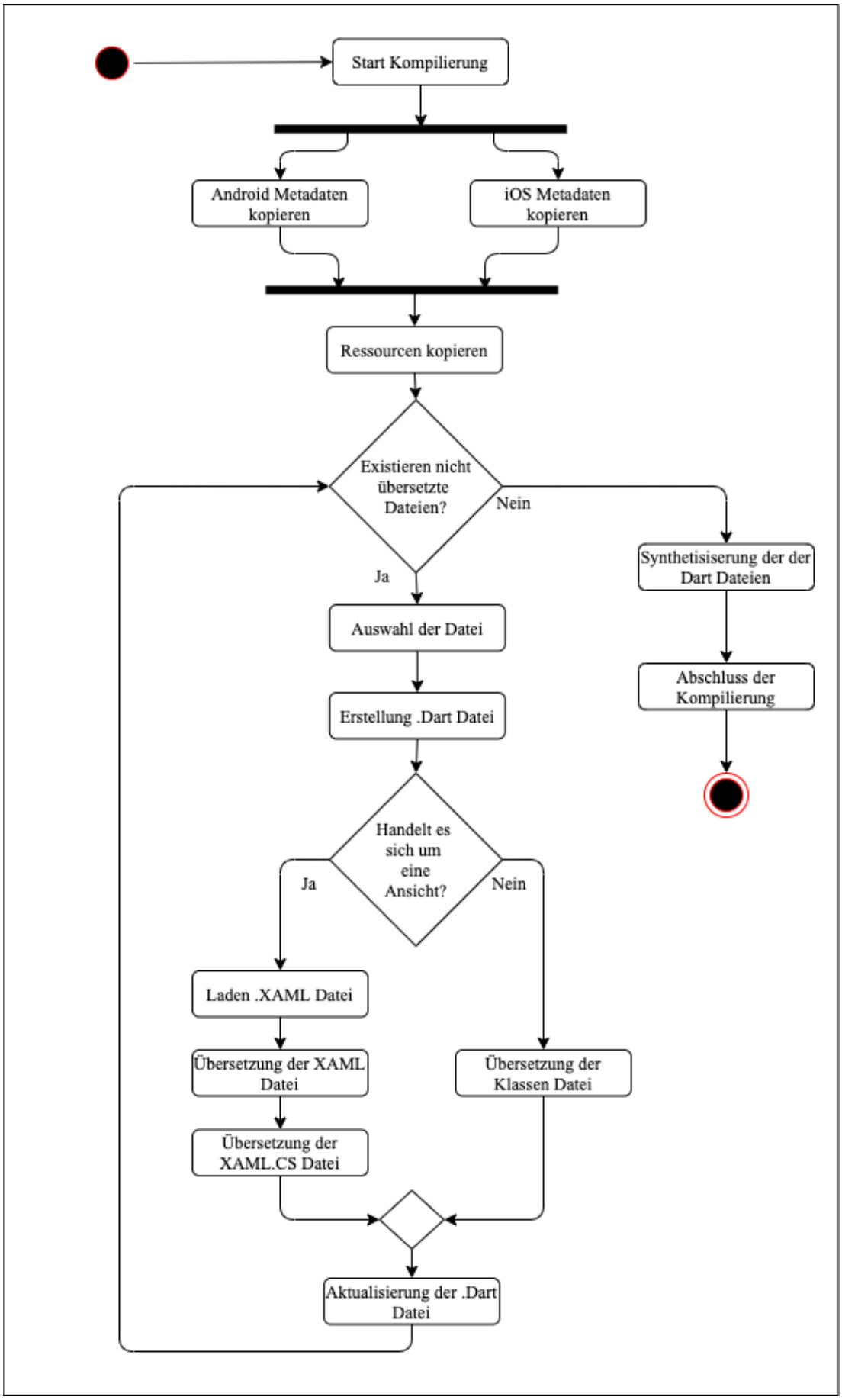


Abbildung 6.1: Aktivitätsdigramm

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 Symbole 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 in Quelltext 6.1 dargestellt.

```
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 der eindeutigen Erkennung einer Anwendung 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 von hier 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 erhöhrter Administrationsaufwand bei der Verwaltung dieser Eigenschaften entsteht, 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 zeigt die Zieldatei für die Kopie bestimmter Werte.

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

Tabelle 6.1: Info.plist Einträge 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 Anwendungsicon 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 vonnöten. 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 späteren Verwendung. Da für Bilder im Flutter Framework keine plattformspezifische Unterscheidung vorgesehen ist, werden ausschließlich die Bilder aus der iOS App entnommen. Eine Entscheidung zugunsten der iOS Alternativen wurde getroffen, 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 Option im Ordner Asset abgelegt. Neben dem differenzierten Vorgehen bei der Kopie 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 werden. 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 plattformspezifisch realisiert. Der Compiler verarbeitet ausschließlich einzelne Bilder, die während des Ladens der mobilen Anwendung angezeigt werden und verzichtet auf die Übersetzung des plattformspezischen Programmcodes, 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.[143] 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.

[143] Vgl. Varty 2014, Abgerufen am 25. 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      if (node.Initializer != null)
10     {
11         Write(" = ");
12         Visit(node.Initializer.Value);
13     }
14     Write($";");
15     NewLine();
16 }
```

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 hangelt 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. Zusätzlich 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.

```
public static string GetKnownName(string name)
{
    switch (name)
    {
        case "void" : return "Void";
        case "int" : return "Int";
        case "byte" : return "Int";
        case "short" : return "Int";
        case "long" : return "BigInt";
        case "double" : return "Double";
        case "float" : return "Double";
        case "char" : return "String";
        case "string" : return "String";
        case "List" : return "List";
        case "Dictionary" : return "Map";
        case "bool" : return "Boolean";
        default: return name;
    }
}
```

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.

Mobile Anwendungen nutzen Bilder, die neu über die Kamera aufgenommen werden oder aus der bestehenden Galerie stammen. 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.

```
await Cmd.ExcecuteCMDCommand(TargetPath,
          "/C \"flutter pub pub add image_picker\"");
await DepencyManager.ImportStatement(File,
          "import 'package:image_picker/image_picker.dart';");

expectedResult = expectedResult.Replace("MediaPicker.capturePhotoAsync()",
                  "await picker.getImage(source: ImageSource.camera);");
expectedResult = expectedResult.Replace("MediaPicker.pickPhotoAsync()",
                  "await picker.getImage(source: ImageSource.gallery);");
await DartFileManager.UpdatePlaceholder(File, "%PROPERTIES%",
    "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 6.8, da der Quelltext jedoch analog ist, wird er an dieser Stelle nicht explizit aufgeführt.

```
int startIndex = expectedResult.IndexOf("Accelerometer.ReadingChanged +=");
if (startIndex != -1)
{
    EndIndex = Code.IndexOf("\n", startIndex);

    Code = expectedResult.Substring(
        startIndex + "Accelerometer.ReadingChanged += ".Length ,
        EndIndex - startIndex - "Accelerometer.ReadingChanged +=".Length-1);
    Code = Code.Replace("Accelerometer.start(speed)", "");
    Code = Code.Replace("SensorSpeed speed = SensorSpeed.UI;", "");
    Code = Code.Replace("Accelerometer.ReadingChanged += " + EventName ,
        DartFileManager.ReadResource("Sensors.Accelerometer.txt"));

    Code = Code.Replace("%METHOD%", EventName);

    Code = Code.Replace("Reading.Acceleration.X", "x;");
    Code = Code.Replace("Reading.Acceleration.Y", "y;");
    Code = Code.Replace("Reading.Acceleration.Z", "z;");

    DepencyManager.AddDependencie("sensors");
    await DepencyManager.ImportStatement(File,
        "import 'package:sensors/sensors.dart';");
}
```

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, das die Aufgaben bis zur Zwischendarstellung übernimmt. Darauf aufbauend wird ein gemeinsames Compiler-Backend für die Zusammenführung und die Code-Optimierung entwickelt. Die Abbildung 6.2 veranschaulicht den Vorgang.

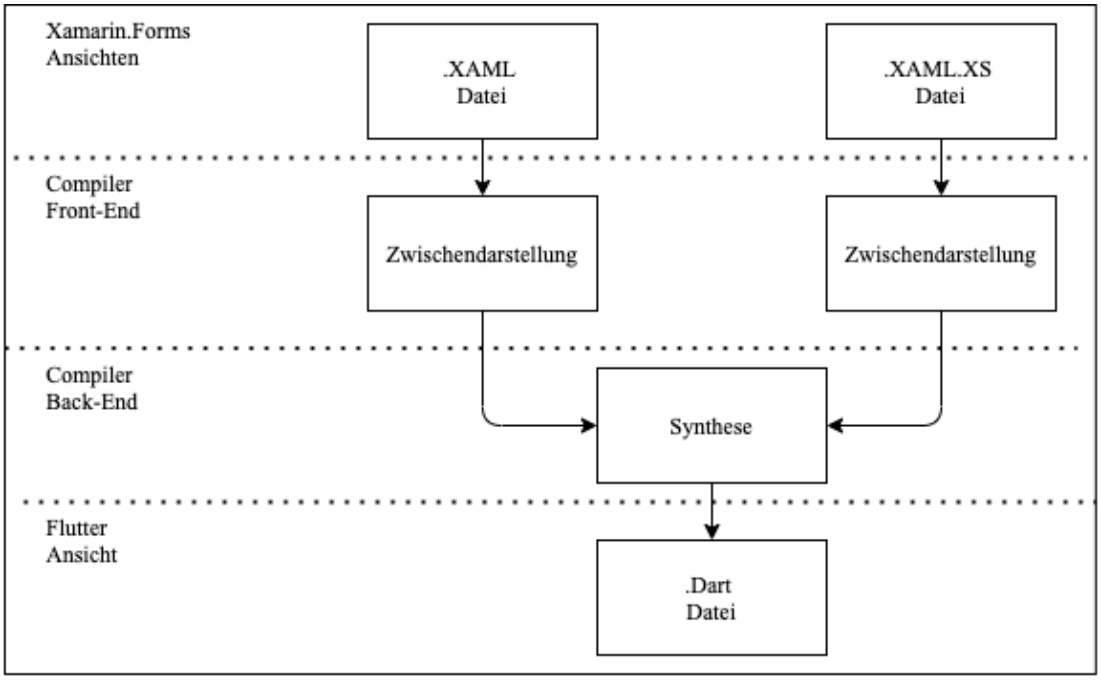


Abbildung 6.2: Compiler Phasen für Ansichten

6.5.1 Visuelle- Zwischendarstellung

Für die visuelle Zwischendarstellung wird in einem ersten Schritt die XML Struktur aus der XAML Datei ausgelesen und als Syntaxbaum interpretiert. 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. Die baumartige Struktur des Xamarin.Forms Layouts 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.

```
Text(
  "%TEXT%",
  textAlign: %HORIZONTALALIGNMENT%,
  overflow: TextOverflow.ellipsis,
  style: TextStyle(
    fontWeight: %FONTWEIGHT%,
    fontSize: %FONTSIZE%,
    fontStyle: %ITALIC%,
    color: %TEXTCOLOR%
 ),
)
```

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 Modifizierung von Textgrößen. Xamarin.Forms fordert den Wert entweder als Zahl oder als sogenannte benannte Größe, wie Large oder Title, während Flutter ausschließlich Textgrößen mit dem Typen Double akzeptiert.

```
private static double GetFontSize(object _SizeValue)
{
    if(_SizeValue is double)
        return Convert.ToDouble(_SizeValue);

    string NamedSize = (String)_SizeValue;

    switch (NamedSize)
    {
        case "Caption" : return 12;
        case "Micro" : return 10;
        case "Header" : return 34;
        case "Large" : return 34;
        case "Medium" : return 14;
        case "Small" : return 10;
        case "Subtitle" : return 16;
        case "Title" : return 24;
        default: return 12;
    }
}
```

Quelltext 6.10: FontSize Umwandlung

Mittels vergleichbarer Implementierung sind auch an anderen Eigenschaften, z.B. Farbwahl oder Textumbruchoptionen, Anpassungen erforderlich.

Nun erfolgt die Anlage einer Arbeitskopie des Templates, um in weiteren Compilerphasen die Werte aus der Xamarin.Forms App einzufügen. Abbildung 6.3 visualisiert die Behandlung der Platzhalter in den einzelnen Phasen der Übersetzung.

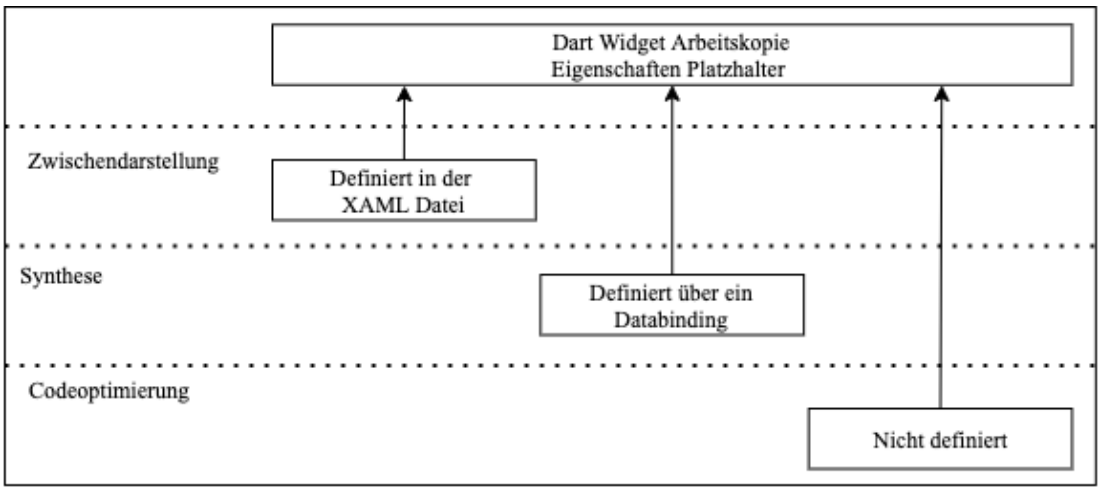


Abbildung 6.3: Bereinigung von Dart Widget Platzhaltern

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.

```
Text(
  %BindingPlaceholder(TextProperty)%,
  textAlign: left,
  overflow: %OverflowPlaceholder%,
  style: TextStyle(
    fontWeight: %FONTWEIGHT%,
    fontSize: %FONTSIZE%,
    fontStyle: %ITALIC%,
    color: %TEXTCOLOR%
  ),
)
```

Quelltext 6.11: Flutter Widget mit Binding-Platzhaltern

Zu diesem Zeitpunkt sind in der Arbeitskopie noch die Binding- und die ungenutzten Platzhalter vorhanden. Diese müssen, wie in der Abbildung 6.3 visualisiert, in späteren Phasen der Kompilierung entfernt werden. Grundsätzlich lässt sich aus der XAML Datei ableiten, welche Platzhalter nicht benötigt werden. Da die Bereinigung jedoch eine Teilaufgabe der Optimierung darstellt, wird sie erst in der entsprechenden Phase im Compiler-Backend durchgeführt.

6.5.2 Logische Zwischendarstellung

Die Erzeugung der logischen Zwischendarstellung, basierend auf den Codebehind Dateien, erfordert die gleiche Implementierung mit ‚CSharpSyntaxWalker' wie die der Klassenstrukturen, da es sich bei diesen Dateien ebenfalls um C# Klassen handelt.

Xamarin.Forms übermittelt Änderungen an Eigenschaften automatisch, mithilfe der ‚INotifyPropertyChanged' Schnittstelle, an die Benutzeroberfläche. Bei Flutter erhält das Framework durch die ‚setState())' Methode entsprechende Informationen. Der Compiler kann notwendige Änderungen nicht im Rahmen der Zwischendarstellung durchführen, da anhand der Codebehind Dateien nicht identifizierbar ist, welche Eigenschaften einen direkten Bezug auf die graphische Benutzeroberfläche haben.

6.5.3 Visuelle Synthese

Die Zusammenführung der visuellen und logischen Zwischendarstellung erfolgt während der visuellen Synthese im Compiler-Backend. In einem ersten Schritt erfolgt die Kopie der übersetzen Logik unterhalb der für den Aufbau der grafischen Benutzeroberfläche beinhalteten Build Methode. Im nächsten Schritt werden ereignisauslösende Aktivitäten, beispielsweise der Klick auf Schaltflächen wie in Quelltext 6.12 dargestellt, mit den übersetzen Methoden verbunden und die Platzhalter entfernt. Dabei wird der Name der behandelten Methode aus der XML Struktur extrahiert und der entsprechende Binding-Platzhalter mit dem Namen und dem Parameter ‚Context' ersetzt.

```
1 var ClickedEvent = GetValueForAttribute(_Node, "Clicked");
2 if (!String.IsNullOrEmpty(ClickedEvent))
3     SourceCode = SourceCode.Replace("%CLICK%", ClickedEvent + "(context);");
```

Quelltext 6.12: Synthese von ereignisauslösenden Aktivitäten

Die Sicherstellung der automatischen Aktualisierung der Benutzeroberfläche kann an dieser Stelle durch die Verwendung von SetState-Blöcken realisiert werden. Dafür wird der Eigenschaftsname aus dem Binding-Platzhalter extrahiert und im Quelltext überprüft, wo die entsprechende Eigenschaft manipuliert wird. Die ermittelten Zeilen werden anschließend mit einem SetState-Block eingeschlossen.

6.6 Referenzierung

Im Anschluss an die Zwischendarstellung, in der die Generierung von Logik und Ansichten erfolgte, müssen nun die entstandenen Dartartefakte untereinander referenziert werden. Darunter wird in diesem Zusammenhang die Verknüpfung der entsprechenden Quelltextdokumente verstanden. Diese Verlinkung ist erforderlich, damit Komponenten auf Klassen zugreifen können, die in anderen Dateien definiert sind. Im Gegensatz zu C# können Dateien in Dart nicht über Namespaces, sondern nur über ihren Dateinamen eingebunden werden. Folglich ist es erforderlich, Dateien in Quelltextdokumente einzubinden, die verwendete Klassendefinition enthalten. Die Information, in welcher Datei sich eine Klasse befindet, wurde durch den ‚CSharpSyntaxWalker' festgehalten. Diese Referenzen werden ebenfalls benötigt, um durch die Anwendung zu den Navigationszielen zu gelangen, da es sich bei der Navigation um die Erzeugung eines Objektes des entsprechenden Widgets handelt.

6.7 Codeoptimierung

Nach der erfolgreichen Zusammenführung der einzelnen Artefakte kann der Code optimiert werden. Dafür werden Sprachkomponenten einschließlich der Ereignisdefinition, der Ereignishandler und aller Aufrufe entfernt, die nur für die Arbeit von Xamarin.Forms benötigt wurden. Die Löschung betrifft die ‚InitializeComponent' Methode, die aus der Codebehind-Datei die grafische Benutzeroberfläche geladen hat und die bereits durch SetState-Blöcke ersetzte ‚INotifyPropertyChanged' Schnittstellen Definition, die für die Oberflächenaktualisierung zuständig war. Jetzt folgt die Entfernung der nicht benötigten Platzhalter mit dem in Quelltext 6.13 dargestellten Algorithmus, aus dem generierten Dart Quelltext.

```
1  public static List<string> RemovePlaceholder(List<String> _LinesOfCode){
2      var LinesToDelete = new List<string>();
3      foreach (string L in _LinesOfCode)
4      {
5          if (L.IndexOf('%') != L.LastIndexOf('%') && L.IndexOf('%') != -1)
6              LinesToDelete.Add(L);
7      }
8      foreach (string S in LinesToDelete)
9          _LinesOfCode.Remove(S);
10     return _LinesOfCode;
11 }
```

Quelltext 6.13: Bereinigung von ungenutzten Platzhaltern

Damit befindet sich der Programmcode in einem korrekten syntaktischen Zustand und kann mit Hilfe des Flutter Compilers zu einer mobilen App übersetzt und ausgeführt werden. Es wird jedoch noch ein weiterer Schritt in der Codeoptimierung durchgeführt, der die Lesbarkeit des Quelltextes durch eine einheitliche Formatierung verbessert. Durch die Verwendung der Templates ist der generierte Widget-Baum innerhalb der Build Methode nicht eingerückt. Die Flutter SDK bietet jedoch die Möglichkeit, Quelltext-Dokumente über die Kommandozeilen Schnittstelle zu formatieren. Quelltext 6.14 zeigt die Methode zur Quelltextformatierung mit Hilfe der Flutter SDK.

```
1 public static async Task FormatDocument(string _Path, string _DartDocument)
2 {
3     await Cmd.ExcecuteCMDCommand(
4         _Path,
5          "/C \"flutter format " + _DartDocument + "\"");
6 }
```

Quelltext 6.14: Methode zur Quelltextformatierung

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 mit XAML und C# entwickelten Desktopanwendungen geeignet ist.[144]

Anwendungen, die mittels WPF programmiert werden, sind nur unter Windows als Betriebssystem ausführbar. Da der Roslyn Compiler auch nur unter Windows verfügbar ist, entstehen hier keine zusätzlichen Einschränkungen. Die folgende Darstellung 6.4 zeigt das Erscheinungsbild der Anwendung nach einer erfolgreichen Übersetzung.

[144] Vgl. Wenger 2012, S. 1f.

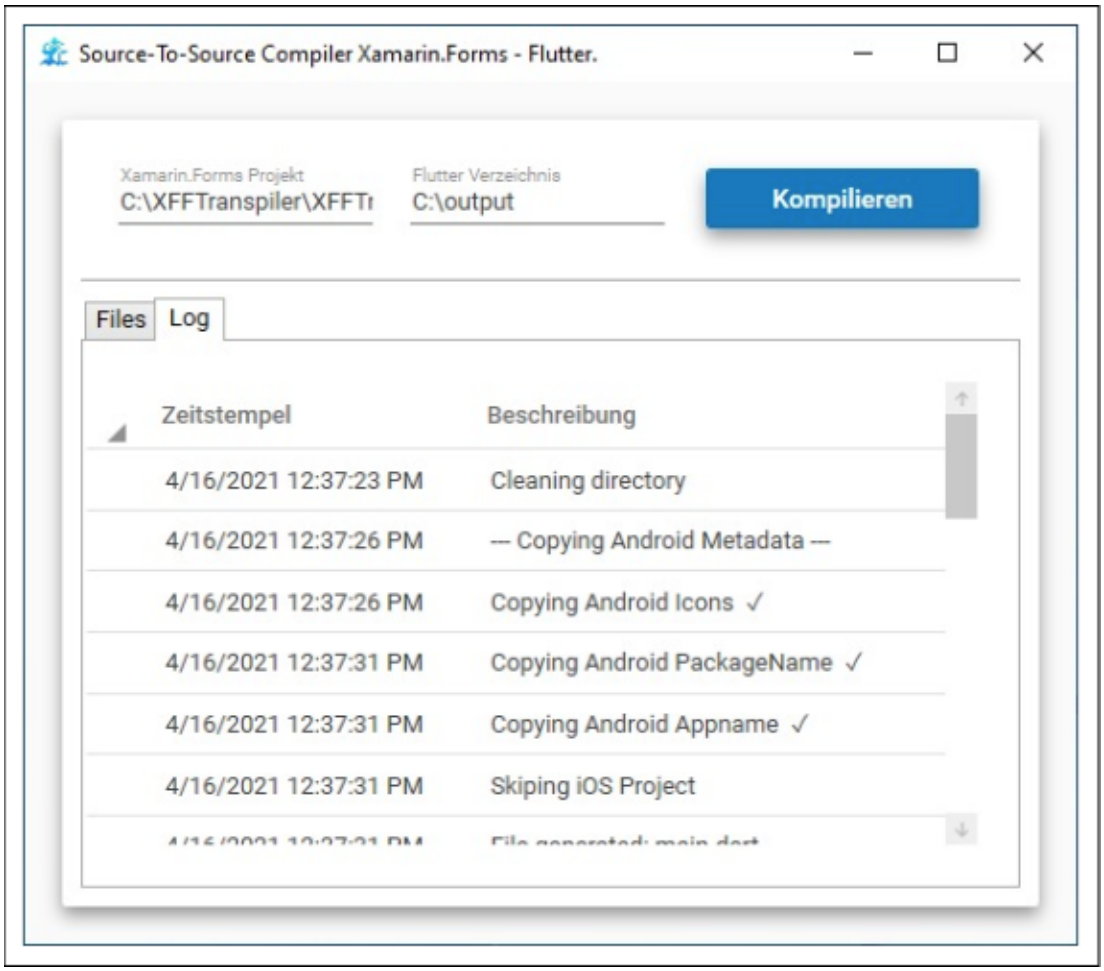


Abbildung 6.4: Grafische Oberfläche des Compilers

Wie die Ansicht veranschaulicht, werden die durchgeführten Arbeitsschritte durch Log-Einträge in der Anwendung angezeigt. Die Realisierung der grafischen Benutzeroberfläche und 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

Die Qualität des Source-To-Source Compilers muss während und nach der erfolgreichen Implementierung sichergestellt werden. Im ersten Abschnitt dieses Kapitels erfolgt die Vorstellung der Komponententests als Qualitätssicherungsmaßnahme im Lauf der Entwicklung. Anschließend wird die Funktionsweise des Compilers mithilfe einer speziell entwickelten Xamarin.Forms App überprüft. Anhand dieser Maßnahmen kann anschließend festgestellt werden, ob der Compiler erwartungsgemäß arbeitet.

7.1 Komponententests

Komponententests unterteilen ein Programm in einzelne, testfähige Einheiten und validieren deren Funktionalität. Wenn sie integraler Bestandteil des Entwicklungspozesses sind, optimieren sie die Qualität des Programms. In Visual Studio können diese Testfälle, bei Änderungen am Quelltext automatisiert durchgeführt werden, was garantiert, dass die Modifikation am Code kein unerwünschtes Verhalten erzeugt. Sobald eine Klasse oder Methode geschrieben ist, beginnt die Analyse, die das Verhalten des Codes nach Eingabe von gültigen, falschen und grenzwertig gültigen Daten überprüft.[145] Diese testgesteuerte (engl. testdriven) Entwicklung entspricht der Qualitätssicherung bei der Programmierung des Source-To-Source Compilers, denn auch hier wurden die Komponententests vor dem Quelltext geschrieben und dienen damit der funktionalen Spezifikation der Anforderungen.

Dieses Testverfahren soll exemplarisch anhand der Umwandlung von Farben veranschaulicht werden. App-Entwickler können die Farbe mit einem gültigen Farbnamen oder als Hexadezimalzahl angeben. Hexadezimale Darstellungen können jeweils über drei, vier, sechs oder acht Ziffern mit einem optionalen Präfix # angegeben werden.[146] Diese verschieden verfügbaren Eingabeformate machen es notwendig, dass der Compiler eine vollständige Umwandlung vornimmt. Anhand dieser Anforderungen können nun die in Quelltext 7.1 gezeigten Testfälle entworfen werden.

[145] Vgl. Microsoft Corporation 2021, Abgerufen am 25. April 2021.
[146] Vgl. Microsoft Corporation 2021, Abgerufen am 25. April 2021.

```
[Test]
public void TransformColorName()
{
    Assert.IsTrue(S2SConverter.GetColor("White").Equals("ffffffff"));
    Assert.IsTrue(S2SConverter.GetColor("Black").Equals("ff000000"));
    Assert.IsTrue(S2SConverter.GetColor("Blue").Equals("ff0000FF"));
    Assert.IsTrue(S2SConverter.GetColor("Black").Equals("ff000000"));
}

[Test]
public void TransformColor()
{
    Assert.IsTrue(S2SConverter.GetColor("#123").Equals("ff112233"));
    Assert.IsTrue(S2SConverter.GetColor("123").Equals("ff112233"));

    Assert.IsTrue(S2SConverter.GetColor("#1234").Equals("11223344"));
    Assert.IsTrue(S2SConverter.GetColor("1234").Equals("11223344"));

    Assert.IsTrue(S2SConverter.GetColor("#ffffff").Equals("ffffffff"));
    Assert.IsTrue(S2SConverter.GetColor("ffffff").Equals("ffffffff"));

    Assert.IsTrue(S2SConverter.GetColor("#ffffffff").Equals("ffffffff"));
    Assert.IsTrue(S2SConverter.GetColor("ffffffff").Equals("ffffffff"));
}

[Test]
public void TransformInvalidTextTrowsException()
{
    Assert.Throws<ArgumentException>(
        () => { S2SConverter.GetColor("Test"); });

    Assert.Throws<ArgumentException>(
        () => { S2SConverter.GetColor("TestTest"); });

    Assert.Throws<ArgumentException>(
    () => { S2SConverter.GetColor("#TestTest"); });

     Assert.Throws<ArgumentException>(
    () => { S2SConverter.GetColor(string.Empty); });
}
```

Quelltext 7.1: Testfälle für die Umwandlung von Farben

Durch die in dem Quelltext aufgeführten Komponententests konnte nun der in Quelltext 7.2 dargestellte Algorithmus realisiert werden.

```
public static string GetColor(string _Color)
{
    if(Color.FromName(_Color).IsKnownColor)
        return string.Format("{0:x6}", color.ToArgb());

    if (_Color[0] == '#')
            _Color = _Color.Remove(0, 1);

    int res = 0;
    if (int.TryParse(_Color, NumberStyles.HexNumber,
                CultureInfo.InvariantCulture, out res))
    {
        if(_Color.Length == 3)
        {
            return "ff" + _Color[0] + _Color[0] +
                _Color[1] + _Color[1] +
                _Color[2] + _Color[2];
        }
        else if(_Color.Length == 4)
        {
            return "" + _Color[0] + _Color[0] +
                _Color[1] + _Color[1] +
                _Color[2] + _Color[2] +
                _Color[3] + _Color[3];
        }
        else if(_Color.Length == 6)
        {
            return "ff" + _Color;
        }
        else if(_Color.Length == 8)
        {
            return _Color;
        }
    }
    throw new ArgumentException("Could not identify a Color", _Color);
}
```

Quelltext 7.2: Algorithmus für die Umwandlung von Farben

7.2 Manuelles Testing

Zur Überprüfung der Funktionalität des Compilers muss eine Xamarin.Forms Anwendung übersetzt und anhand von definierten Testfällen das Ergebnis validiert werden. In diesem Abschnitt wird ein Testobjekt eingeführt, bevor die Testfälle spezifiziert werden und die Anwendung übersetzt und anschließend überprüft wird.

7.2.1 Testobjekt

Die Testapp wurde mit der Version 5.0.0.2012 des Xamarin.Forms Frameworks realisiert und verwendet die Erweiterungen Xamarin. Essentials. Als Testobjekt soll diese mobile Anwendung 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.

Abbildung 7.1 zeigt im ersten Screenshoot den Startbildschirm mit dem Namen und Icon der Testapp und im zweiten den Hauptbildschirm.

Abbildung 7.1: Test Objekt Screenshots I

Anhand des Namens, SampleApp, und eines Anwendungsicons können Anwender die App auf dem Startbildschirm identifizieren. Nach ihrem Start öffnet sich eine

Menüstruktur, die Wurzel der Navigation, über die verschiedene Bereiche, der mobilen Anwendung angesteuert werden können. In einer vertikalen Ansicht werden dafür Schaltflächen als Ereignisauslöser für die Navigation bereitgestellt, die zu den Seiten, Übersicht der Steuerelemente, Ausgabe von Smartphonesensoren, Arbeit mit Bildern und Ansicht einer Listview führen.

Abbildung 7.2 zeigt die Übersicht der Steuerelemente sowie die Werte der Smartphone-Sensoren.

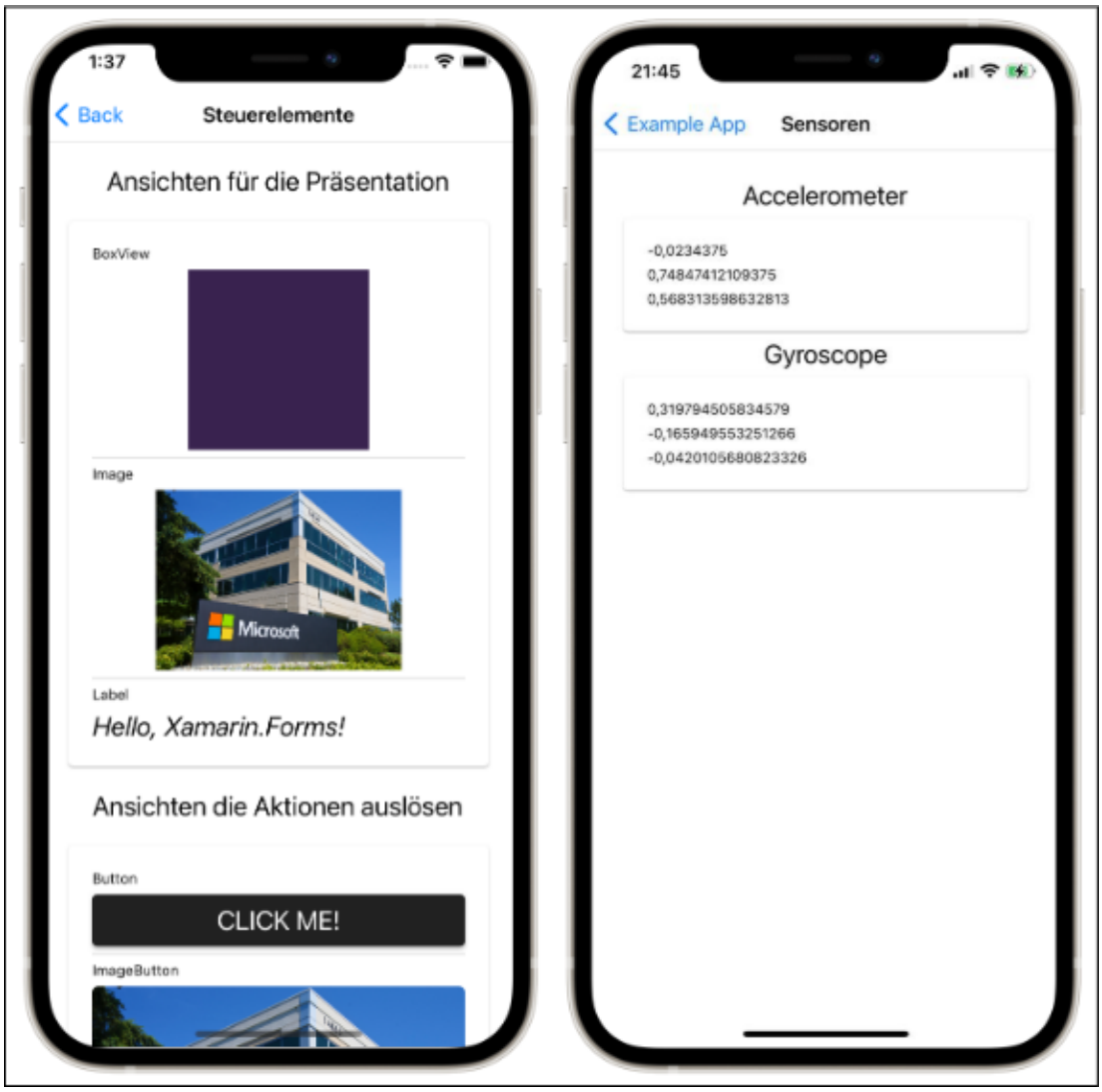


Abbildung 7.2: Test Objekt Screenshots II

Die Darstellung visualisiert die gängigsten Steuerelemente, die anhand der in Kapitel 4 aufgeführten Kategorien gruppiert und vertikal angeordnet wurden. Zu den angezeigten Elementen gehören unter anderem eine Schaltfläche, ein Bild und Auswahlmöglichkeiten für Datum und Uhrzeitwerte. Der zweite Screenshot zeigt eine Übersicht der aktuellen Werte von Beschleunigungssensors und Gyroskopes, die beide über das Plugin Xamarin.Essentials in der Codebehind-Datei ausgelesen und über Databindings im UI angezeigt werden.

Die nächsten beiden Screenshots in 7.3 zeigen das Menü für die Arbeit mit Bildern und einem zusätzlichen Berechtigungsdialog rechts.

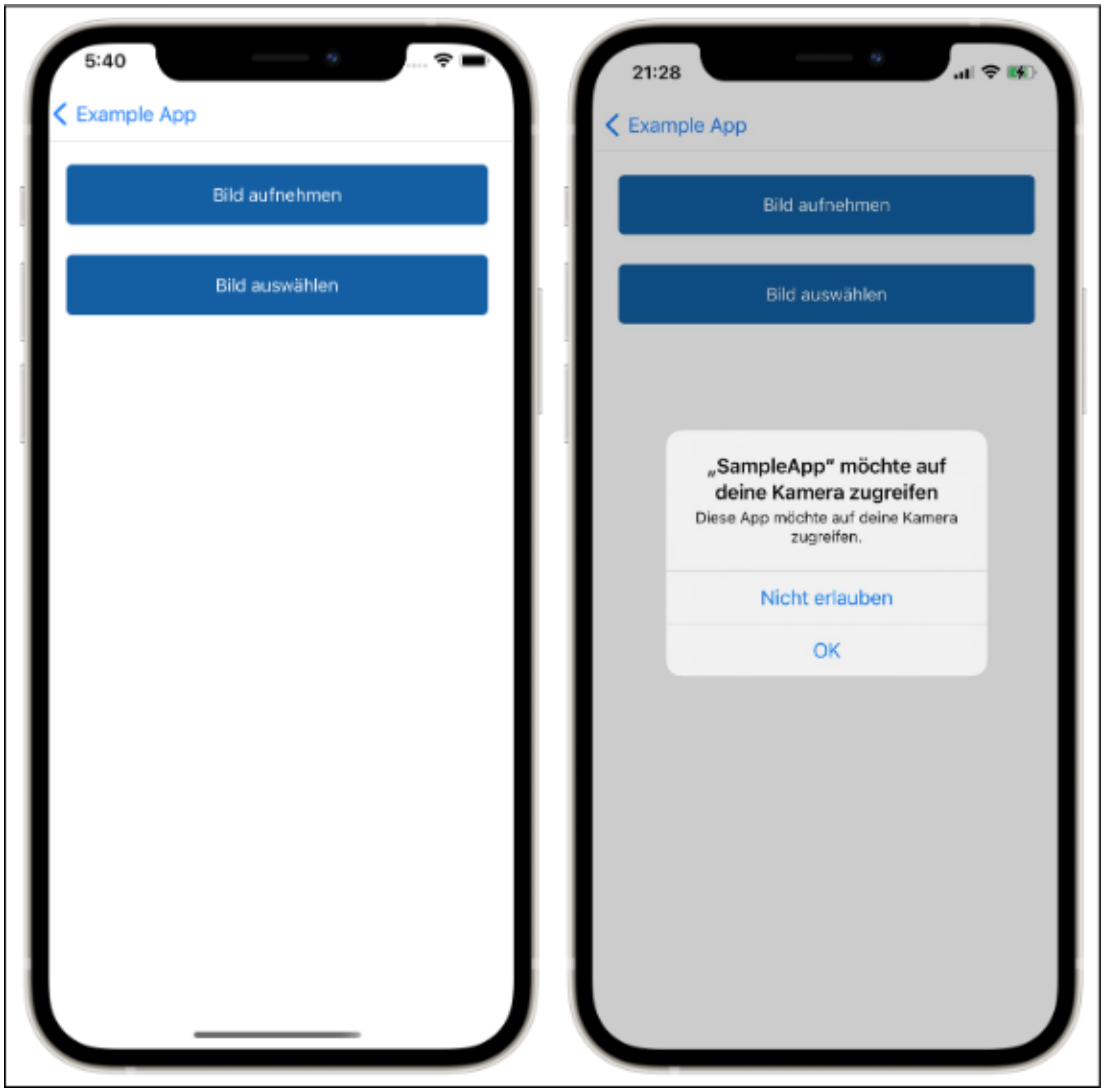


Abbildung 7.3: Test Objekt Screenshots III

Das Menü hat Ähnlichkeiten mit dem der Hauptseite. Statt einer Navigation werden hier jedoch plattformspezifische Funktionalitäten über das Xamarin.Essentials Plugin ausgeführt, die einen Zugriff auf die Kamera und die Galerie des Smartphones bieten. Der Dialog, der nach der Zugriffsberechtigung fragt, zeigt in der iOS Version einen Eintrag aus den Metadaten an. Die nächste Abbildung stellt im linken Screenshot die Auswahlfunktion von Bildern aus der Galerie dar und rechts eine ListView mit Namen und E-Mail-Adressen von fiktiven Benutzern. Die Klasse ‚Benutzer' wird innerhalb des Quelltextes der mobilen Anwendung definiert und über den Namespace in die Codebehind-Datei referenziert. Anschließend werden exemplarisch zehn Objekte dieser Klasse innerhalb der Codebehind Datei in einer Auflistung vorgehalten und mithilfe eines Databindings in der ListView angezeigt.

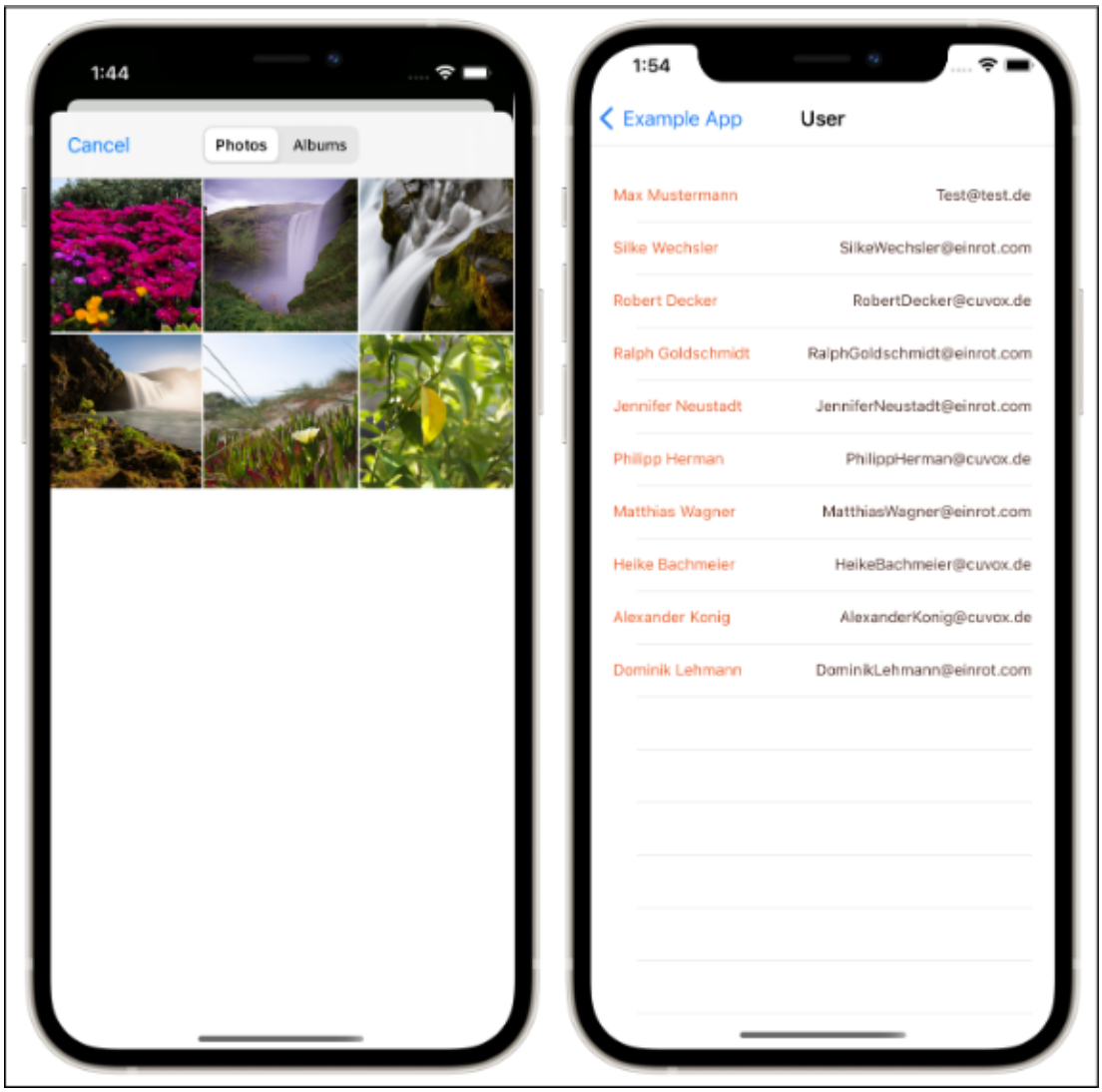


Abbildung 7.4: Test Objekt Screenshots IV

7.2.2 Testfälle

Zur Überprüfung der Funktionalität des Compilers müssen Testfälle definiert werden, mit denen die übersetzte mobile Anwendung manuell validiert wird. Relevante Prüfobjekte lassen sich sowohl aus dem visuellen Erscheinungsbild als auch aus dem Funktionsspektrum des Testobjekts ableiten und sind in Tabelle 7.1 aufgeführt. Dieser Testkatalog kann erweitert werden, ist jedoch für eine Qualitätssicherung zur Beantwortung der Forschungsfrage ausreichend.

Prüfobjekt	Prüfvorgang
App-Icon	Wird das App-Icon übernommen
App-Name	Wird der App-Name übernommen
Seitenname	Wird in der Navigationsleiste der Name der aktuellen Seite angezeigt
Menü	Werden alle 4 Menüpunkte angezeigt
Navigation	Wird mithilfe des Menüs navigiert
Navigation	Wird über die Zurück-Schaltfläche in der Navigationsleiste zurück zur letzten Seite navigiert
Schriftgrößen	Werden Schriftgrößen in der Anwendungen übernommen
Farben	Werden Farben in der Anwendung korrekt übernommen
Layouts	Werden alle Layouts durch die entsprechenden Widgets ersetzt.
Steuerelemente	Werden alle Steuerelemente durch die entsprechenden Widgets ersetzt.
Beschleunigungssensor	Werden die Werte des Beschleunigungssensor ausgelesen
Gyroskop	Werden die Werte des Gyroskopes ausgelesen
Kamera	Können Bilder über die Kamera aufgenommen werden
Galerie	Können Bilder aus der Galerie ausgewählt werden
Berechtigungen	Wird nach einer Zugriffsberechtigungen gefragt beim Zugriff auf Kamera und Galerie
ListView	Werden in der Listview die generierten Daten angezeigt

Tabelle 7.1: Testfälle der Testapp

7.2.3 Testablauf

Die Analyse der Softwarequalität erfolgt nicht ausschließlich automatisiert. In diesem Abschnitt werden die oben genannten Testfälle manuell durchgeführt und ihr Ergebnis dokumentiert. Der Start des Testlaufs beginnt mit dem Aufruf der grafischen Benutzeroberfläche, der Auswahl der Xamarin.Forms Testapp und eines leeren Ordners, der als 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 über den Status, den Verlauf und mögliche Fehlerquellen nach der Übersetzung ablesen. Anschließend kann die fertige Flutter Anwendung mithilfe der SDK kompiliert und ihre Ansichten kontrolliert werden. Die im Folgenden visualisierten Screenshots zeigen die generierte iOS Flutter-App, die entsprechenden Androidrepräsenationen befinden sich in Anhang IV dieser Arbeit. Die ausgewöhlten Screenshots der übersetzten App zeigen die gleichen Bildschirmstatus der Ausgangsapp wie die der Ausgangsapp um eine Vergleichbar zu gewähren. Das nahezu

identische Erscheinungsbild belegt bereits auf den ersten Blick eine hohe Qualität des Prototypen.

In Abbildung 7.5 wird links der Startbildschirm mit dem Namen und Icon und rechts das zentrale Menü der App dargestellt.

Abbildung 7.5: Flutter App Screenshots I

Sowohl der Name als auch das Launchericon der Anwendung sind korrekt übernommen worden. Nach dem Start der Anwendung fallen jedoch Designabweichungen auf, die durch die Verwendung der Material Design Vorlagen in Flutter begründet sind. Da der Compiler nicht, wie in den Ausschlusskriterien festgelegt, den Style der Anwendung übersetzt, entspricht das Ergebnis der Erwartungshaltung und ist, weil daraus keine funktionelle Störung entstehet, an dieser Stelle akzeptabel. Das zentrale Menü mit seiner Navigationsfunktion entspricht in der Ziel- dem der Testapp, unterscheidet sich jedoch, wenn auch nicht in der Abbildung ersichtlich, in der Animation.

Abbildung 7.6 stellt die Übersicht der Steuerelemente sowie die Werte der Smartphone-

Sensoren dar.

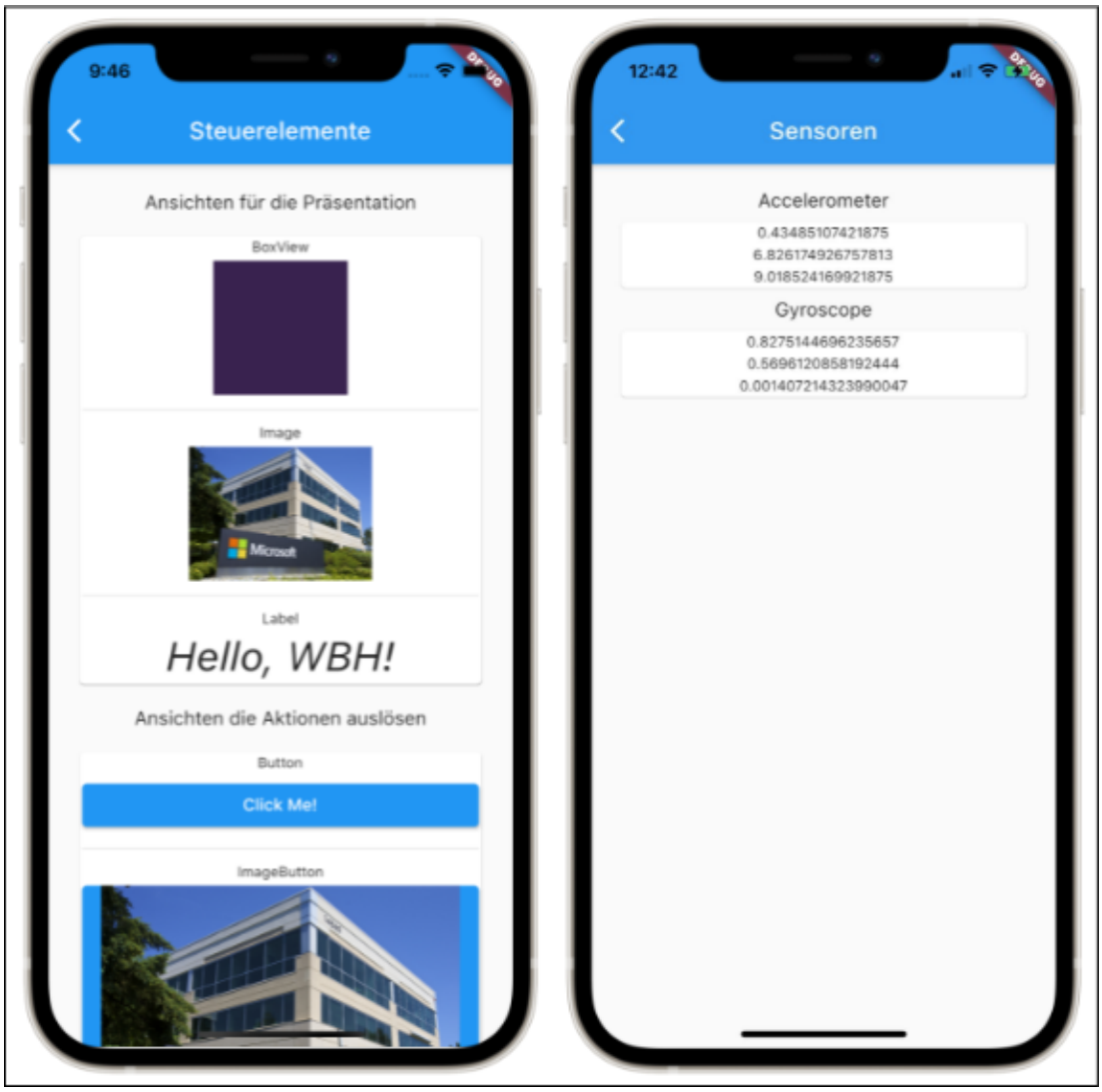


Abbildung 7.6: Flutter App Screenshots II

Die Übersicht der Steuerelemente stimmt bezüglich der Verschachtlungen als auch der Funktionalitäten mit denen der Xamarin.Forms Variante überein, das durch die Öffnung der Datums- und Uhrzeitauswahl validiert werden kann. Ebenfalls korrekt übernommen wurde die Oberfläche der Sensorenansicht und alle Werte der Smartphonesensoren befinden sich im richtigen Format. Im Gegensatz zu der Xamarin.Forms App werden diese Daten jedoch nicht im gleichen Zyklus aktualisiert. Dieser Unterschied basiert auf den Implementationen des jeweiligen Plugins und entsteht nicht durch den Übersetzer. In der Xamarin.Forms App werden die Werte über Databindings geladen und mithilfe der INotifyPropertyChanged Schnittstelle die Benutzeroberfläche im Falle von Änderungen aktualisiert. Diese Schnittstelle wurde in der Flutter Anwendung entfernt und der Teil des Quelltextes, der Änderungen am UI vornimmt mit einem Setstate-Block umschlossen, damit sich die Oberfläche aktualisiert.

Die in Abbildung 7.7 gezeigten Screenshots zeigen ein Menü zur Arbeit mit Bildern,

wobei der rechte zusätzlich den Berechtigungsdialog für die Kameraverwendung präsentiert.

Abbildung 7.7: Flutter App Screenshots III

Der Berechtigungsdialog entspricht nach Kompilierung in Darstellung und Funktion dem der Xamarin.Forms Version. Auch der Zugriff auf die Kamera und die Galerie wurde, allerdings mithilfe der Erweiterung image_picker, erfolgreich übersetzt.

Die Screenshots aus Abbildung 7.8 zeigen, links die Galerie zur Bildauswahl und rechts die Darstellung der Benutzer in einer ListView.

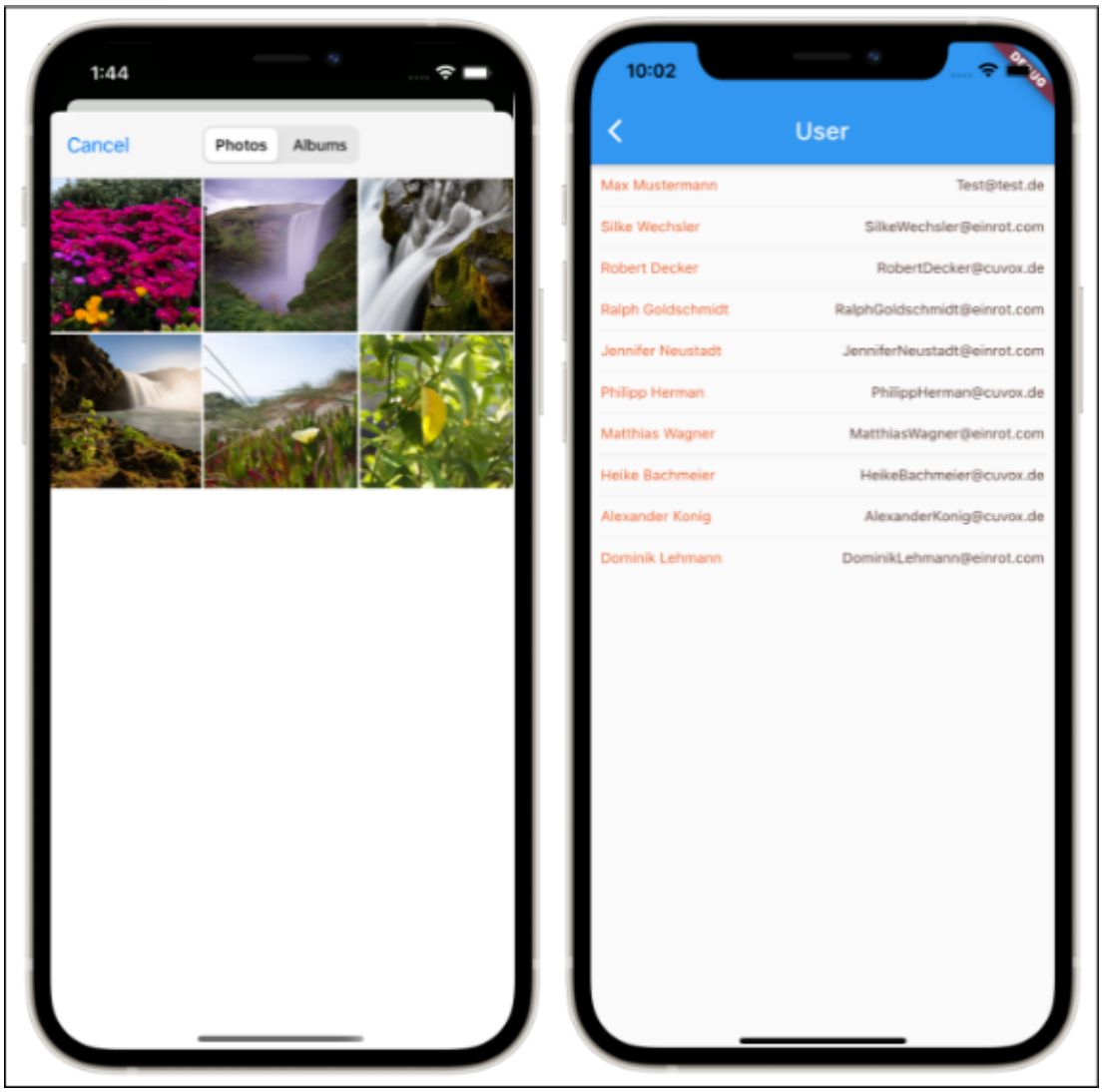


Abbildung 7.8: Flutter App Screenshots IV

Die korrekte Darstellung der ListView beweist, dass auch die Klassendefinition und ihr Konstruktor korrekt übersetzt wurden, denn nur unter dieser Voraussetzung konnte die Liste mit den passenden Daten gefüllt werden.

Das Testobjekt konnte also vollständig überführt werden, was grundsätzlich die Funktionsfähigkeit des Prototypen beweist. Bei einer Übersetzung aller in der Praxis möglichen Funktionalitäten von Xamerin.Forms bedarf es jedoch Erweiterungen des Compilers.

8 Fazit

Ein Source-To-Source Compiler, wie in dieser Arbeit entwickelt, 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 könnte. Die Idee durch automatisierte Übersetzungen Programmierern eine Wahlfreiheit und Wechseloptionen zu ermöglichen rechtfertigt eine Weiterentwicklung des Prototypen.

8.1 Open Source Veröffentlichung

Der Umfang des Prototypen wurde durch die in Kapitel 3 ausgeschlossenen Aspekte reduziert, da diese für die Beantwortung der Forschungsfrage eine untergeordnete Rolle spielen. Diese Einschränkungen besteht bei den meisten Xamerin.Forms Apps nicht. Eine Veröffentlichung des Compilers als Open Source Projekt könnte helfen, diese komplexeren mobilen Anwendungen zu übersetzen, indem Programmierer den dann für die Allgemeinheit verfügbaren Quelltext auf spezielle 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 ersetzten zu lassen und somit dessen Umfang zu erhöhen. Die Open-Source Lizenz GPL 3.0 würde garantieren, das Veränderungen am Source-To-Source Compuler wiederum Quelloffen zu seien haben. Durch die Open Source Veröffentlichung könnte also die Compilerentwicklung in Geschwindigkeit und Zielgenauigkeit erhöht werden.

8.2 Beantwortung der Forschungsfrage

Es besteht die Möglichkeit Apps von Xamarin.Forms zu Flutter vollautomatisiert, ohne manuelle Arbeitsschritte zu übersetzten. Zwar überträgt der Source-To-Source Compiler die Xamarin.Forms Testapp und zeigt damit die Machbarkeit einer automatisierten Transformation, damit jedoch auch produktive Anwendungen vollständig, inklusive plattformspezifischer Implementierungen oder anwendungsspezifischer Erweiterungen, ohne manuelles Zutun übersetzt werden können, sind Weiterentwicklungen beispielsweise durch eine Open-Source Veröffentlichung nötig. 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. Kritisch angemerkt sei, dass manuelle Arbeitsschritte bei der Übersetzung notwendig werden, sobald eines der ausgeschlossenen Aspekte verwendet wird.

8.3 Ausblick

Bereits für Ende 2021 ist das Supportende für Xamarin.Forms beschlossen und die Einführung von .NET MAUI angekündigt. Eine vollständige Funktionsfähigkeit des Compilers wäre noch in diesem Jahr wünschenswert, um Unternehmen und Programmierern eine rechtzeitige Alternative zu bieten.

Anwendungsentwickler bekommen durch den Umstieg von Xamarin.Forms zu Flutter seit März 2021 die Möglichkeit ihre Anwendung als Webseite zu veröffentlichen und somit eine weitere Plattform zu nutzen. Jedoch sind zusätzliche Implementierungen in der Flutter App notwendig um die Webdarstellung auf großen Displays zu optimieren. Flutter treibt seine Frameworkentwicklung durch Neuerungen, wie den Support der Webnutzung, weiter voran, sodass derzeit nicht der Eindruck entsteht, dass ein Supportende in naher Zukunft zu befürchten ist. Ob die Einführung von .Net MAUI wegweisende Innovationen mit sich bringt, ist bisher genau so unbekannt wie der Programmieraufwand für den Umstieg von Xamarin.Forms. Der Source-To-Source Compiler stellt eine berechenbare Alternative zum Umstieg auf .NET MAUI zur Verfügung.

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, Programmiertechniken, Windows-Programmierung*. München: Pearson Deutschland GmbH.

Hindrikes, Daniel und Johan Karlsson (2020). *Xamarin.Forms Projects. Build Multiplatform 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` (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%5C_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 (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). *Layouts in Flutter*. URL: https://flutter.dev/docs/development/ui/layout (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).

– (2016a). *.NET-Klassenbibliotheken*. URL: https://docs.microsoft.com/de-de/dotnet/standard/class-libraries (besucht am 27. 02. 2021).

Microsoft Corporation (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).

– (2020d). *Images in Xamarin.Forms*. URL: https://docs.microsoft.com/de-de/xamarin/xamarin-forms/user-interface/images (besucht am 27. 02. 2021).

Microsoft Corporation (2020e). *Launcher Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/xamarin.essentials.launcher (besucht am 27. 02. 2021).

– (2020f). *Serialisieren und Deserialisieren*. URL: https://docs.microsoft.com/de-de/dotnet/standard/serialization/system-text-json-how-to (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).

– (2021). *Color Struktur*. URL: https://docs.microsoft.com/de-de/dotnet/api/xamarin.forms.color (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://www.codemag.com/Article/1605071/Arranging-Views-with-Xamarin.Forms-Layout (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

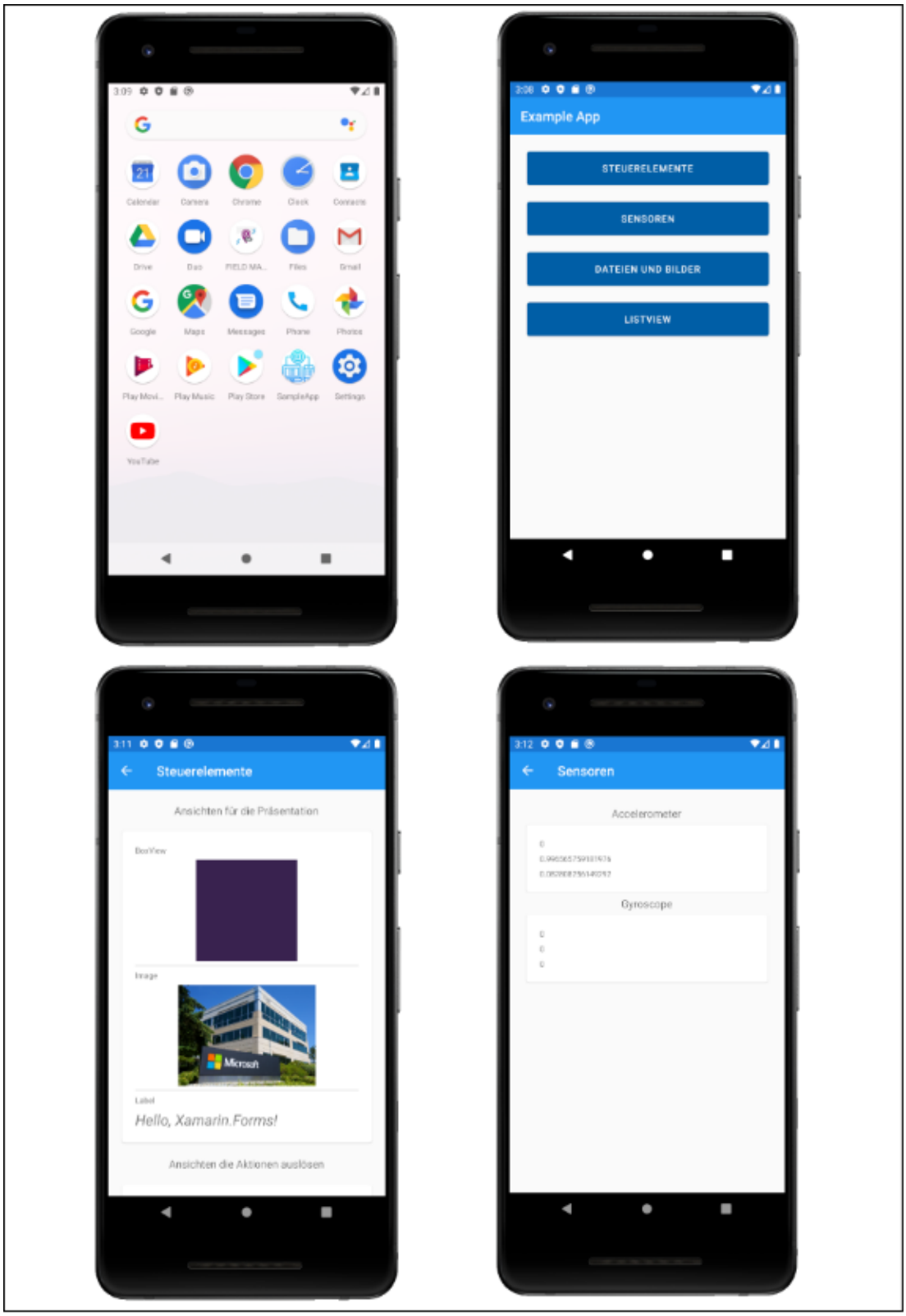
Tabelle: Gegenüberstellung von visuellen Elementen

Anhang II: Optimierte Flutter-LoginPage

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.grey,
    appBar: AppBar( title: Text("LoginPage"),),
    body: Center(
      child: Card(
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: <Widget>[
            Padding(
              padding: const EdgeInsets.all(5.0),
              child: TextField(
                obscureText: false,
                decoration: InputDecoration(
                  border: OutlineInputBorder(),
                  labelText: 'Username', ),
              ),
            ),
            Padding(padding: const EdgeInsets.all(5.0),
              child: TextField(
                obscureText: true,
                decoration: InputDecoration(
                  border: OutlineInputBorder(),
                  labelText: 'Password',),
              ),
            ),
            SizedBox(
              width: double.infinity,
              child: Padding(
                padding: const EdgeInsets.all(5.0),
                child: ElevatedButton(
                    child: Text('Login'),),
              ),
            ),
          ],
        ),
      ),
    ),
  );
}
```

Quelltext: Optimierte Flutter-LoginPage

Anhang III: Android Screenshots der Xamarin.Forms App



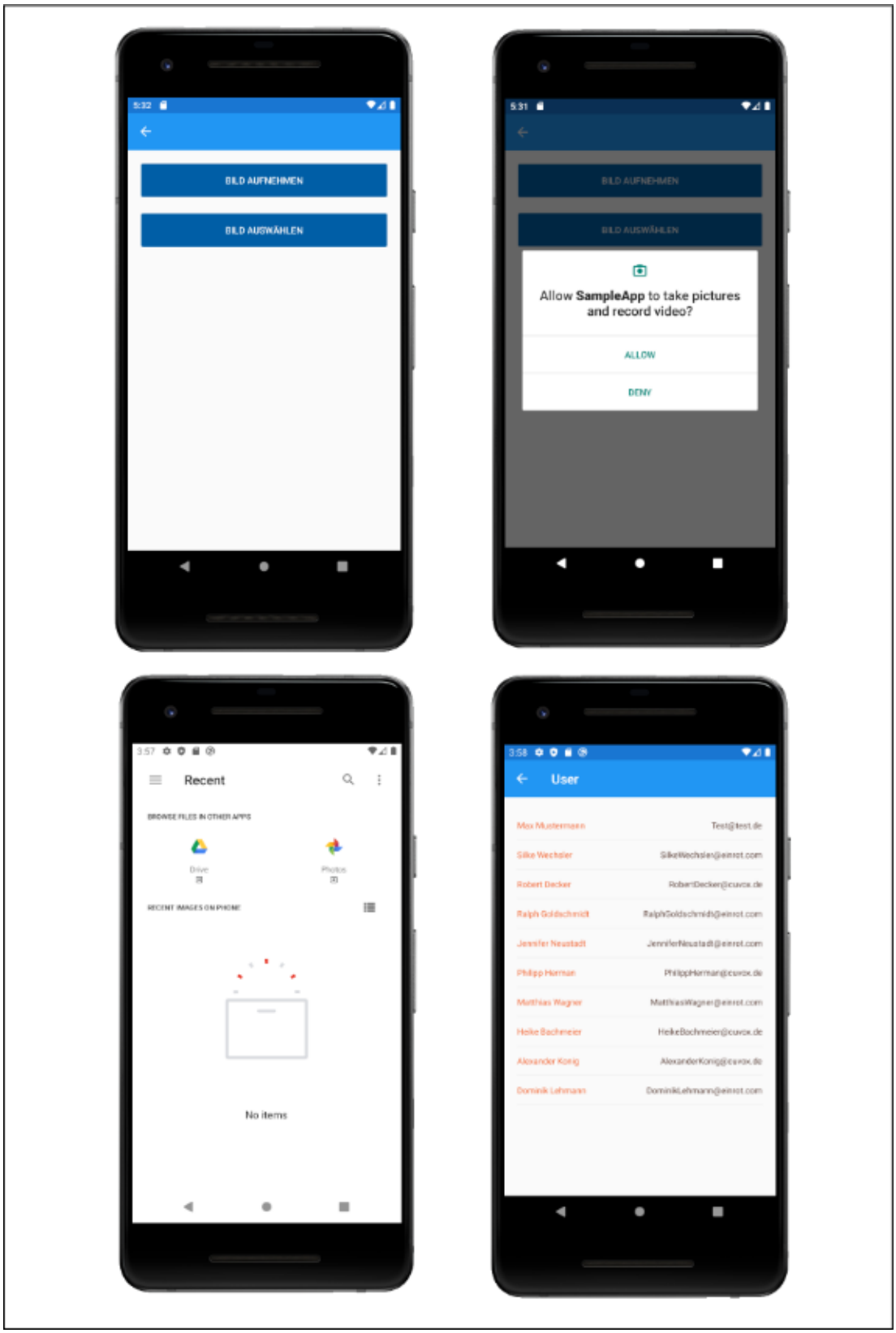
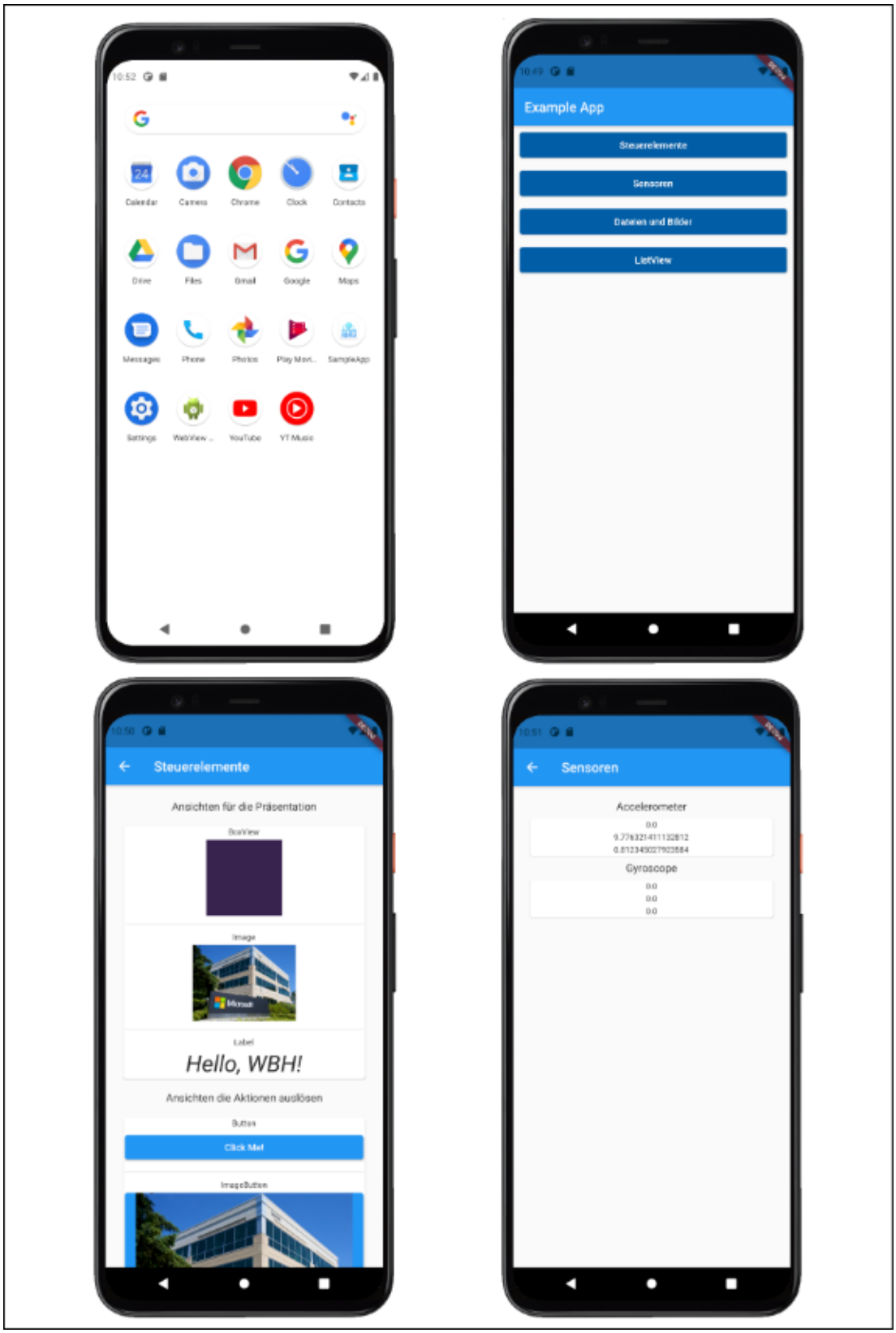


BILD AUFNEHMEN
BILD AUSWÄHLEN
Allow SampleApp to take pictures and record video?
ALLOW
DENY
Recent
No items
User

Anhang IV: Android Screenshots der Flutter App



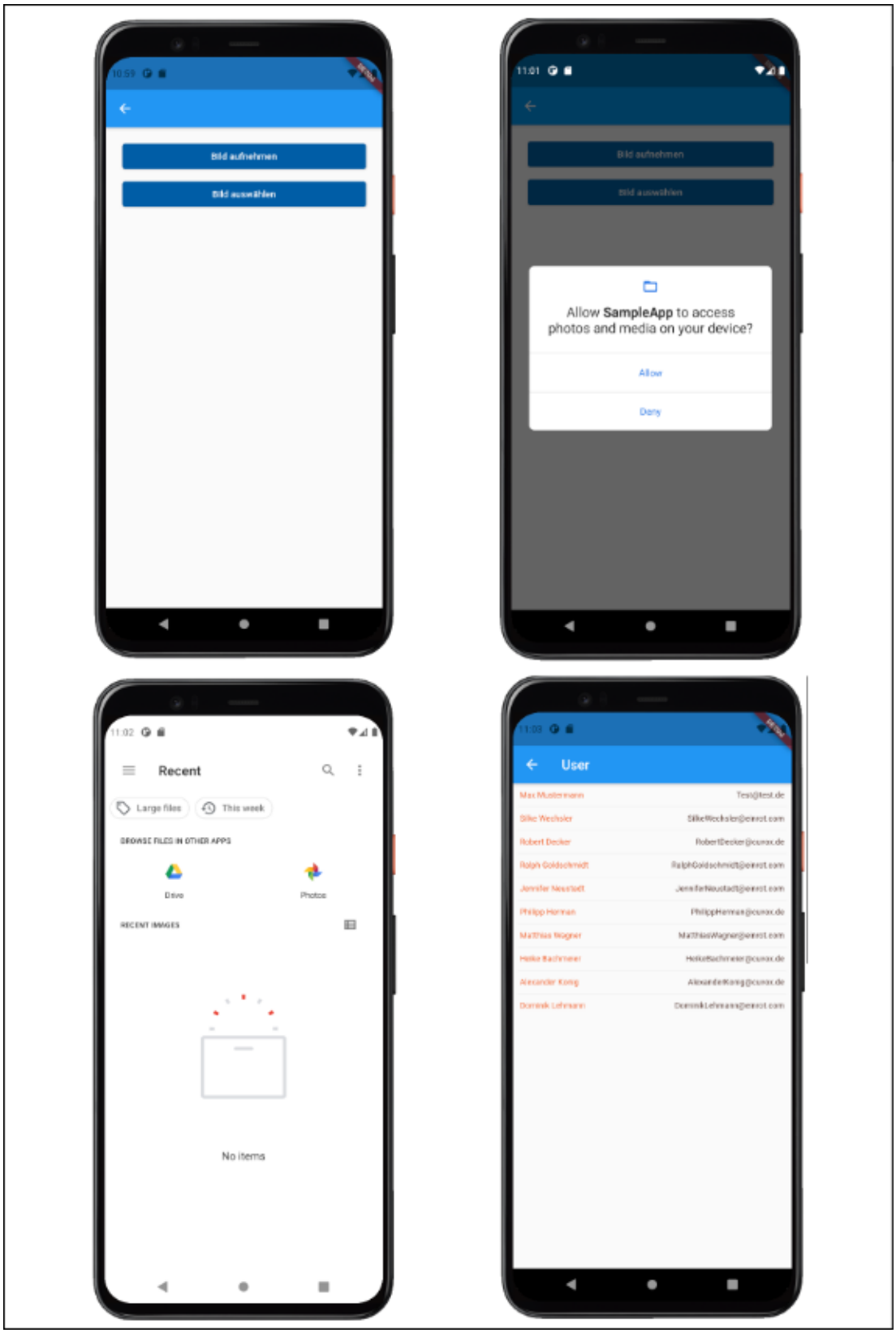


Bild aufnehmen
Bild auswählen
Allow SampleApp to access photos and media on your device?
Allow
Deny
Recent
Large files
This week
BROWSE FILES IN OTHER APPS
Drive
Photos
RECENT IMAGES
No items
User

Anhang V: Installationsanleitung

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)