THM TECHNISCHE HOCHSCHULE MITTELHESSEN | CAMPUS FRIEDBERG | IEM Informationstechnik-Elektrotechnik-Mechatronik

Plattformübergreifende Entwicklung mobiler Anwendungen mit besonderer Berücksichtigung von Windows 10

Masterarbeit

Studiengang Medieninformatik

vorgelegt von

Fabian Timm, 827445

durchgeführt bei der Firma

SDX AG, Frankfurt am Main

Referent:	Prof. Dr. Stephan Euler
Korreferent:	Ing. Felix Rupp, M. Sc.
Eingereicht am:	30.03.2016

Danksagung

An dieser Stelle möchte ich all jenen meinen Dank aussprechen, die mich in Vorbereitung und während der Erstellung dieser Masterarbeit unterstützt und motiviert haben.

Zunächst möchte ich mich bei Herrn Prof. Dr. Stephan Euler für die sehr gute Betreuung und die zahlreichen Ratschläge bedanken sowie bei Herrn Ing. Felix Rupp, M. Sc., dass er sich bereit erklärt hat, bei dieser Arbeit Korreferent zu sein.

Mein besonderer Dank gilt der SDX AG, bei der ich stets vollste Unterstützung erfahren habe, hier insbesondere Herrn Alexander Jung, der mir immer mit größtmöglichem Engagement zur Seite stand.

Dank sei auch meinen Korrektoren und meiner Familie gesagt.

Vielen Dank!

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und ohne unerlaubte Hilfe durch Dritte verfasst habe. Für die Erstellung wurden ausschließlich die angegebenen Hilfsmittel und benannte Fachliteratur verwendet. Wörtlich und sinngemäß zitierte Stellen sind als solche gekennzeichnet.

_____

Fabian Timm, Friedberg, März 2016

Abstract

Mobile Endgeräte gehören heute zum Alltag, sodass diese auch immer mehr Beachtung im Enterprise Bereich erhalten. So müssen moderne Enterprise Anwendungen häufig nicht nur für den Desktop, sondern auch auf Smartphones und Tablets bereitgestellt werden. Dadurch, dass Anwendungen nicht nur auf Desktops zur Verfügung stehen sollen, ergeben sich immer größere Herausforderungen für deren Entwicklung. Diese sind, neben den unterschiedlichen Betriebssystemen (iOS, Android und Windows), auch die immer größer werdende Anzahl von Formfaktoren (Smartphone, Phablet, Tablet, Laptop, Desktop und Fernsehgeräte). Für diese Probleme werden schon lange Zeit verschiedenste Cross-Plattform-Entwicklungs-Ansätze und -Frameworks entwickelt und eingesetzt. Ein Ansatz ist dabei der Übersetzungsansatz, welcher eine besonders native User Experience verspricht. Xamarin ist hierfür ein Framework, dass die Cross-Plattform-Entwicklung für Android, iOS, und Windows Phone (Smartphone, Phablet, Tablet) ermöglicht. Neben diesen hat Microsoft für die Windows Plattform die Universal Windows Platform auf den Markt gebracht. Sie ermöglicht es für alle Windows Geräte, oder anders gesagt, für alle Windows Betriebssysteme und Formfaktoren, von IoT über Smartphones, Tablets und Desktops bis hin zum Surface Hub, eine Anwendung zu entwickeln. Diese beiden Ansätze vereint könnten damit so gut wie alle Geräte abdecken. Daher befasst sich diese Arbeit damit, wie gut diese Ansätze zusammengebracht werden können und wieviel Aufwand sich einsparen lässt.

Inhaltsverzeichnis

Danksagung iii
Selbstständigkeitserklärung v
Abstract vii
1 Einleitung 1
1.1 Motivation und Problemstellung 1
1.2 Zielsetzung 2
1.3 Einsatzszenarien 3
1.4 Grundlegender Aufbau der Arbeit 3
2 Verwandte Arbeiten & Stand der Technik 5
2.1 Verwandte Arbeiten 5
2.1.1 Universal Windows Platform 5
2.1.2 Xamarin 6
2.1.3 Cross-Plattform-Entwicklung 6
2.1.4 Fazit 6
2.2 Stand der Technik 7
3 Architektur Grundlagen 9
3.1 Architektur-Pattern 9
3.1.1 Layer-Modell 9
3.1.2 Onion-Modell 10
3.1.3 Unterschiede und Relativierung 11
3.2 Design-Pattern 12
3.2.1 MVVM 12
3.2.2 Dependency Injection 13
3.2.3 Fassade 14
3.2.4 Data-Transfer-Object & Data-Access-Object 15
4 Grundlagen 17
4.1 Überblick verwendeter Frameworks 17
4.2 Xamarin 17
4.2.1 Kompilierung 18
4.2.2 Xamarin.Forms 18
4.2.3 MvvmCross 20
4.3 Universal Windows Platform 20
4.3.1 Gerätefamilien 21
4.3.2 Formfaktor 22

4.4 Code-Sharing-Strategien ... 25
4.4.1 Shared Project ... 25
4.4.2 Portable Class Library ... 26
4.5 Microsoft Azure ... 26
4.5.1 Azure Active Directory ... 27
4.5.2 Azure Mobile Services ... 27
5 Methodik ... 29
5.1 Konkrete Ziele ... 29
5.2 Die Prototypen ... 29
5.2.1 ShowTime ... 29
5.2.2 Meeting-App ... 31
5.2.3 Serien-App ... 31
5.3 Konzeption und Design ... 33
5.3.1 Unterschiedliche Betriebssysteme ... 33
5.3.2 Verschiedene Gerätefamilien ... 37
5.3.3 Mögliche Herangehensweisen ... 41
5.4 Architektur ... 42
5.4.1 Design-Pattern ... 42
5.4.2 Geplante Architektur ... 44
6 Implementierung ... 47
6.1 Projektsetup ... 47
6.2 Dependency Injection ... 48
6.3 Services ... 50
6.3.1 Authentifizierung ... 50
6.3.2 Kommunikation mit einer REST-API ... 52
6.3.3 Serialisieren und Deserialisieren von Daten ... 53
6.3.4 Lokale Datenhaltung mit SQLite ... 53
6.4 ViewModels ... 57
6.4.1 Command ... 58
6.4.2 DataBinding ... 61
6.5 Userinterface ... 62
6.5.1 Formfaktoren ... 64
6.5.2 Custom Renderer ... 65
6.6 Navigationskonzept ... 66
6.6.1 Rudimentäre Navigation ... 67
6.6.2 Navigation mit verschiedensten Formfaktoren ... 70
6.6.3 Navigation mit einem Split-View bzw. Master-Detail-View ... 74

6.7 Mobile Service ... 76
7 Ergebnisse ... 79
7.1 Architektur ... 79
7.2 Code-Sharing-Strategien ... 79
7.3 Shared-Code-Anteil ... 80
8 Fazit und Ausblick ... 83
8.1 Fazit ... 83
8.2 Ausblick ... 83
8.2.1 Erweiterung auf WPF (Windows 7) ... 83
8.2.2 Xamarin.Forms UWP im Vergleich zum untersuchten Ansatz ... 84
8.2.3 Xamarin Übernahme durch Microsoft ... 85
Abkürzungsverzeichnis ... 87
Abbildungsverzeichnis ... 90
Referenzen ... 93
Anhang CD ... 97

Kapitel 1

1 Einleitung

1.1 Motivation und Problemstellung

Mobile Endgeräte gehören heute zum Alltag, so haben in Deutschland 46[1] Millionen und weltweit sogar 1,76[2] Milliarden Menschen ein Smartphone. Neben der immer stärkeren Verbreitung unter der Bevölkerung müssen sich auch immer mehr Unternehmen der Herausforderung der Digitalisierung stellen, diese wird zum Teil mit Hilfe von mobilen Anwendungen gelöst. Der digitale Wandel in der Gesellschaft und den Unternehmen bringt viele Herausforderungen mit sich, eine der größten ist die hohe Fragmentierung von Geräten. Dabei unterscheiden sich die Geräte durch die drei größten Betriebssysteme Android (Smartphone - Deutschland, Tablet - Welt, 2015) (72,6%, 67%), iOS (20,2%, 24,5%) und Windows (6,4%, 8,5%)[3] [4], ihre unterschiedlichen Formfaktoren und verfügbaren Sensoren. Neben mobilen Anwendungen spielen gerade bei der Digitalisierung der Unternehmen Desktop-Anwendungen eine entscheidende Rolle, bei denen das Betriebssystem Windows im Gesamten einen Marktanteil von (Deutschland, Januar 2016) 76,63%[5] hat.

Um alle Mobile und Desktop-Geräte erreichen zu können, müsste eine Anwendung für jedes Betriebssystem einzeln entwickelt werden. Damit dies nicht notwendig ist, gibt es unterschiedliche entwicklungsmethodische Ansätze, diese sind der Webbasierter-, der Hybrider-, der Interpretations- und der Übersetzungsansatz. Jeder dieser Ansätze hat unterschiedliche Vor- und Nachteile, die sich wiederum, je nach angewandtem Framework, unterscheiden. Diese große Anzahl an verschiedenen Ansätzen und eine unüberschaubare Anzahl von Frameworks zeigen den immensen Bedarf an plattformübergreifenden Lösungen. Zwei führende Ansätze sind dabei der Hybride mit Cordova (PhoneGap) und der Übersetzungsansatz mit Xamarin.

Xamarin ist ein mögliches Framework, es folgt dem Übersetzungsansatz. Mit Hilfe von Xamarin können Android-, iOS- und Windows Phone-Anwendungen mit einem hohen Anteil an Shared-Code entwickelt werden. Die Userinterfaces sind dabei für jede einzelne Plattform zu implementieren. Um den Anteil an Shared-Code weiter zu erhöhen und das Userinterface der einzelnen Plattformen zu abstrahieren

[1] Vgl. [13]
[2] Vgl. [12]
[3] Vgl. [31]
[4] Vgl. [32]
[5] Vgl. [33]

wurde Xamarin.Forms entwickelt. Mit Xamarin.Forms muss auch das Userinterface, bis auf kleinere gerätespezifische Unterschiede, nur einmal implementiert werden.

Für Windows Geräte hat Microsoft mit Windows 10 und der Universal Windows Platform (UWP) einen weiteren interessanten Ansatz geschaffen. Die UWP bietet allen Windows 10 Geräten dieselben Core-API's, wodurch es möglich ist, eine Universal Windows Application (UWA) zu entwickeln, die auf allen Windows Geräten gleichermaßen läuft, also von IoT Geräten bis hin zum Surface Hub. Windows 10 wird bei den Unternehmen zwar nicht kurz, aber sicher mittel- oder langfristig zum Einsatz kommen, da dies das offiziell letzte Betriebssystem von Microsoft ist und ältere Windowsversionen abgekündigt sind. Außerdem haben viele Unternehmen Windows 8 übersprungen, sodass derzeit ein Upgrade-Druck vorhanden ist. Auch auf dem Consumer Markt verfolgt Microsoft eine aggressive Upgrade-Politik mit einem kostenlosen Upgrade von Windows 7 und 8 auf Windows 10. Diese aggressive Strategie spiegelt sich in den Marktanteilen von Windows 10 wider, durch die bereits im Januar 2016 ein Marktanteil (Deutschland) von 17,9%[6] erreicht wurde.

Die Gemeinsamkeit dieser beiden Ansätze ist nicht nur, eine gemeinsame Codebasis über unterschiedlichste Plattformen hinweg zu haben, sondern auch, dass beide in C# und mit Hilfe des .NET Frameworks entwickelt werden. Des Weiteren basiert Xamarin.Forms auf vielen Ideen der Microsoftwelt, wie dem MVVM-Pattern (Model-View-ViewModel) und Bindings. Durch die starken Ähnlichkeiten lassen sich beide Ansätze kombinieren, inwieweit, wird diese Arbeit zeigen.

1.2 Zielsetzung

Wie geschildert, stellt der digitale Wandel die Entwickler vor die Herausforderung, für unterschiedlichste Geräte und Betriebssysteme Lösungen zu entwickeln. Dabei müsste eine Anwendung, die sich auf den einzelnen Geräten nicht oder nur marginal unterscheiden soll, für jedes Betriebssystem einzeln entwickelt werden. Um diese redundante Entwicklung zu vermeiden oder zu reduzieren, wurden schon mögliche Ansätze aufgezeigt. Mit diesen, den Ansätzen Xamarin.Forms und UWP, würde durch zwei Entwicklungen ein großes Spektrum abgedeckt werden, doch wäre weiterhin eine redundante Entwicklung notwendig. Daher befasst sich die Arbeit damit, inwieweit diese beiden Ansätze kombinierbar sind und wie viel Redundanz sich einsparen lässt, ohne dabei Funktionalität zu verlieren.

[6] Vgl. [33]

1.3 Einsatzszenarien

Für Cross-Plattform-Anwendungen mit Xamarin.Forms und UWP als Kombination gibt es unterschiedlichste Einsatzszenarien. Besonders von der Kombination könnten dabei zwei Szenarien profitieren.

Ein Szenario ist, eine bestehende Desktopanwendung auch für die mobile Anwendung zur Verfügung zu stellen, oder umgekehrt, bestehende mobile Xamarin Anwendungen auch für den Desktop zu veröffentlichen. So könnte die Reichweite einer bestehenden Anwendung durch einen geringen Aufwand stark erweitert werden. Außerdem können so momentan beliebte Strategien wie „Bring Your Own Device“ (BYOD) wesentlich preisgünstiger umgesetzt werden.

Ein zweites Einsatzszenario ist eine komplexe Desktopanwendung, von der einzig eine Teilmenge auf den mobilen Geräten verfügbar sein soll, also eine Anwendung, bei der Daten auf der Mobile- sowie Desktopgerätefamilie angezeigt werden, die Daten aber lediglich auf der Desktopgerätefamilie eingepflegt werden.

1.4 Grundlegender Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel gegliedert.

Das erste Kapitel erläutert die Motivation und Problemstellung, die sich diese Arbeit stellt. Außerdem wird das Ziel der Arbeit formuliert und mögliche Einsatzgebiete aufgezeigt.

Im Kapitel „Verwandte Arbeiten & Stand der Technik“ wird zunächst vorgestellt, welche Arbeiten dieser ähneln. Anschließend wird ein Statement zum Stand der Technik dieser Arbeit abgegeben.

Das Kapitel „Architektur Grundlagen“ erklärt die Architekturmodelle Layer und Onion sowie die für diese Arbeit wichtigsten Design-Pattern MVVM, Dependency Injection, Fassade, Data-Transfer-Object und Data-Access-Object. Einfache und/oder für die Cross-Plattform-Entwicklung weniger relevante Pattern, wie beispielsweise das Singleton-Pattern, werden hier nicht erwähnt.

Im Kapitel „Grundlagen“ wird auf die für diese Arbeit wichtigen Themen Code-Sharing-Strategien, Xamarin, Windows Universal Platform und Microsoft Azure eingegangen. Dabei wird besonders intensiv auf die für Cross-Plattform-Entwicklung relevanten Faktoren eingegangen. Diese sind Shared-Code und Formfaktoren.

Das Kapitel „Methodik“ schildert die Herausforderungen bei der Herangehensweise und Planung einer Cross-Plattform-Anwendung. Dabei werden im ersten Abschnitt kurz die Ziele der Arbeit wiederholt. Im Anschluss folgt eine kurze Aufführung der Prototypen. Danach werden wichtige konzeptionelle und

designtechnische Problemstellungen und Entscheidungen erläutert. Außerdem wird auf eine mögliche Architektur eingegangen.

Im Kapitel „Implementierung“ werden die wichtigsten Erkenntnisse und Gedankengänge der Implementierungsphase zusammengefasst, sowie relevante Ausschnitte des daraus resultierenden Quellcodes erläutert. Dabei werden die einzelnen Abschnitte des Kapitels in die Layer der zuvor vorgestellten Architektur sowie in weitere logische Abschnitte unterteilt. Innerhalb der Abschnitte wird der Anteil an Shared-Code verdeutlicht und es wird aufgezeigt, welche Code-Sharing-Strategie verwendet wurde. Die Code-Passagen und die gewählten Beispiele beanspruchen dabei keine Funktionstüchtigkeit oder Vollständigkeit. Sie sind gewählt, um die wichtigsten Aspekte des Gesamtbildes herauszustellen und aufzuzeigen.

Das Kapitel „Ergebnisse“ bewertet, wie gut die architektonischen Überlegungen und die Code-Sharing-Strategien während der Implementierungsphase funktioniert haben. Es werden auch Code-Metriken ausgewertet und die daraus resultierenden quantitativen Daten analysiert und bewertet.

Den Abschluss bilden ein Fazit und Ausblick In diesem Kapitel wird ein kurzes Fazit zu dem untersuchten Ansatz gezogen. Es wird auch ein Ausblick gegeben, wie dieser noch erweitert werden kann. Zusätzlich wird kurz auf die Preview Version von Xamarin.Forms UWP sowie auf die Übernahme von Xamarin durch Microsoft eingegangen.

Kapitel 2

2 Verwandte Arbeiten & Stand der Technik

Das Kapitel Verwandte Arbeiten und Stand der Technik stellt zunächst vor, welche Arbeiten dieser ähneln. Anschließend wird ein Statement zum Stand der Technik dieser Arbeit gegeben.

2.1 Verwandte Arbeiten

Im Vorfeld und Nachgang dieser Arbeit wurde geprüft, welche verwandten Arbeiten es gibt. Dabei wurde zu den Themen Universal Windows Platform, Xamarin und Cross-Plattform-Entwicklung mit den Schlüsselwörtern Universal Windows Platform, UWP, Universal Windows Application, Universal Windows App, UWA, Xamarin, Xamarin.Forms, Cross Plattform Entwicklung, Cross Platform Development gesucht. Die Suche ergab zahlreiche Abhandlungen und Bücher, davon sind im Folgenden zu jedem Thema eine Abhandlungen oder ein Buch aufgeführt.

2.1.1 Universal Windows Platform

Da der Begriff Universal Windows Platform ein Teilgebiet gut eingrenzt und das Thema ein sehr neues ist, gibt es verhältnismäßig wenige Treffer.

Besonders hervorgetan hat sich das Buch “Real World Windows 10 Development”[7]. Wie am Namen schon zu erkennen, behandelt dieses Werk die Entwicklung unter Windows 10 und der Universal Windows Platform. Diese Arbeit hat also bei der reinen UWP Entwicklung gewisse Überschneidungen. Dabei behandelt es aber kein Xamarin, Xamarin.Forms oder intensiver das Teilen von Code zwischen verschiedenen Betriebssystemen. Daher werden auch keine Probleme behandelt, welche bei der Kombination dieser beiden Frameworks auftreten.

Neben diesem Buch wurden noch einige andere Abhandlungen gefunden, die diese Arbeit jedoch inhaltlich noch entfernter tangieren.

[7] [34]

2.1.2 Xamarin

Da Xamarin und zum Teil auch Xamarin.Forms keine Neuheit mehr ist, gibt es unzählige Arbeiten, welche sich mit unterschiedlichsten Themen beschäftigen. Die gefundenen Arbeiten und Bücher decken dabei praktisch alle gängigen Xamarin und Xamarin.Forms Themen ab. In dem Buch "Creating Mobile Apps with Xamarin.Forms" von Charles Petzold[8], welches in Kooperation mit Xamarin und Microsoft entstanden ist, wird sehr ausführlich das UI-Framework Xamarin.Forms beschrieben.

2.1.3 Cross-Plattform-Entwicklung

Wie schon in der Motivation zu lesen war, ist das Thema Cross-Plattform-Entwicklung schon seit langer Zeit ein sehr wichtiges geworden, daher gibt es hierzu eine nahezu unbegrenzte Anzahl an Arbeiten. Diese beschäftigen sich mit allgemeinen Themen der Cross-Plattform-Entwicklung, wie, welche Ansätze gibt es, welche architektonischen Möglichkeiten und vieles mehr. Außerdem handeln viele auch von einem konkreten Ansatz und häufig auch von einem konkreten Framework für diesen Ansatz.

Eine sehr gelungene Arbeit ist die von Martin Dobrev mit dem Titel "Untersuchung hybrider Ansätze zur plattformübergreifenden App-Entwicklung"[9]. In dieser Abhandlung werden der Webbasierter-, der Hybrider-, der Interpretations- und der Übersetzungsansatz beschrieben und verglichen. Außerdem wird ein Prototyp konzeptioniert und mit Xamarin implementiert.

2.1.4 Fazit

In den verschiedensten Arbeiten, Büchern, aber auch in diversen Blog-Beiträgen und Tutorials werden die Themen Universal Windows Platform, Xamarin und Cross-Plattform-Entwicklung ausführlich analysiert. Es wurde aber keine Quelle gefunden, die sich damit beschäftigt, wie gut Xamarin.Forms und UWP miteinander vereinbar sind, ohne dabei wichtige Features der Frameworks zu verlieren. Daher kann sich die Arbeit dahingehend komplett von den anderen abheben. Dennoch wird es zwangsläufig einige Schnittmengen mit den zahllosen vorhandenen Büchern und Abhandlungen geben.

[8] [35]
[9] [36]

2.2 Stand der Technik

Die Arbeit handelt vom Stand der Technik zur Zeit der jeweiligen Implementierung und Anfertigung der einzelnen Passagen. So kann es an einigen Stellen dazu kommen, dass bspw. die konkrete Implementierung zur Zeit der Abgabe schon wieder anders aussehen würde. Das nicht alles aktuell gehalten werden kann, ist dem Fakt geschuldet, dass es sich, gerade bei UWP, um eine brandneue Technologie handelt, die sich im schnellen Wandel befindet. Des Weiteren ist es so, dass sich allgemein die Update-Zyklen in der IT-Branche immer stärker verkürzen, wodurch es nahezu unmöglich ist, in allen Belangen den aktuellen Stand widerzuspiegeln.

Dennoch wird darauf geachtet, dass die getroffenen Aussagen und Schlussfolgerungen im Kern die gleichen bleiben. Außerdem werden sehr wichtige Neuerungen im Fazit oder Ausblick aufgegriffen.

Kapitel 3

3 Architektur Grundlagen

Dieses Kapitel erklärt die Architekturmodelle Layer und Onion sowie die für diese Arbeit wichtigsten Design-Pattern MVVM, Dependency Injection, Fassade, Data-Transfer-Object und Data-Access-Object. Einfache und/oder für die Cross-Plattform-Entwicklung weniger relevante Pattern, wie beispielsweise das Singleton-Pattern, werden hier nicht erwähnt.

3.1 Architektur-Pattern

3.1.1 Layer-Modell

Beim Layer-Modell handelt es sich um ein hierarchisches Architektur-Modell. Die Umsetzung des Modells kann strikt oder flexibel erfolgen. Bei einem strikten Layering ist der obere Layer abhängig von den jeweils darunter befindlichen, es existieren aber keine weiteren Abhängigkeiten über die Layer hinweg. Beim flexiblen Layering sind Abhängigkeiten über Layer hinweg möglich. Das flexible Layering ermöglicht eine schnellere Implementierung und ggf. bessere Performance durch reduziertes Mapping zwischen den Layern. Das strikte Layering garantiert eine bessere Wartbarkeit und Erweiterbarkeit. Durch das Layering wird eine Datenkapselung erreicht, eine Koppelung zwischen den Layern bleibt teilweise bestehen. Gerade beim flexiblen Layering kann es zu komplizierten Abhängigkeiten kommen. Dies ist allerdings ein sehr starres Bild des Layer-Modells, in der Relativierung wird beschrieben, wie das Layer-Modell auch angewandt werden kann.[10]

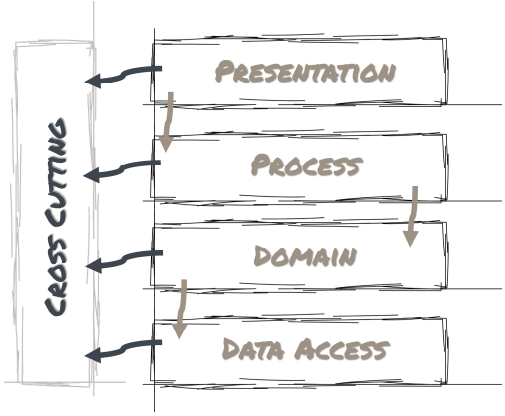

Abbildung 1: Layer Überblick [1]

[10] Vgl. [19]

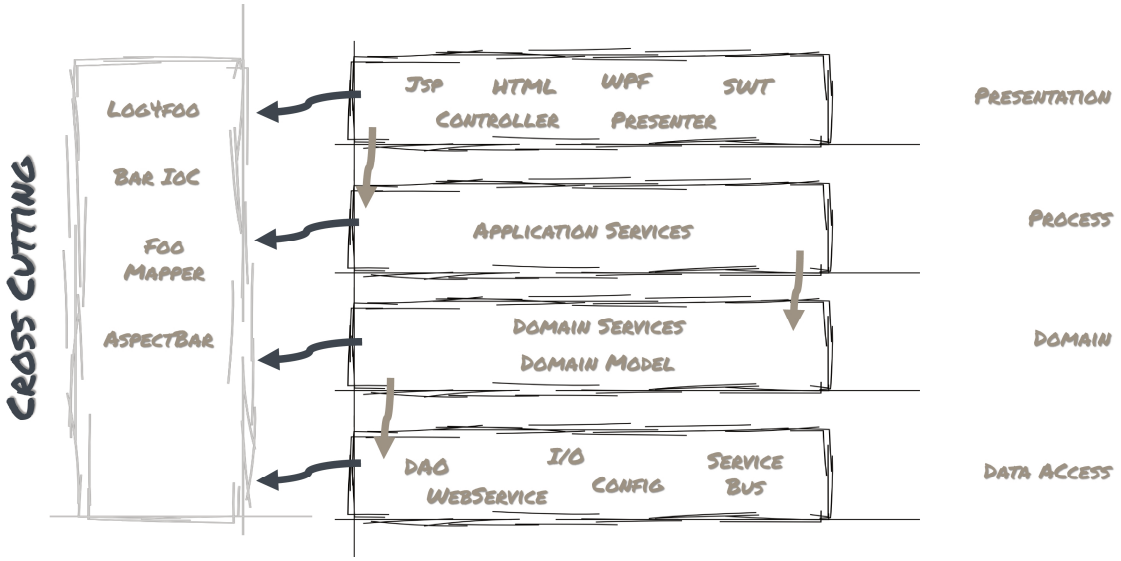

Abbildung 2:Layer Detail [1]

3.1.2 Onion-Modell

Das Onion-Modell ist ein hierarchisches Architektur-Modell, bei dem die Abhängigkeiten von außen nach innen gehen. Bei der Umsetzung werden Komponenten, die sich häufiger ändern, möglichst weit nach außen verlagert und so Abhängigkeiten von diesen weitestgehend vermieden. Außerdem wird viel mit Interfaces gearbeitet, um die Komponenten voneinander zu entkoppeln. Durch das Onion-Modell wird neben der Datenkapselung ein hoher Grad an Entkoppelung erreicht.[11]

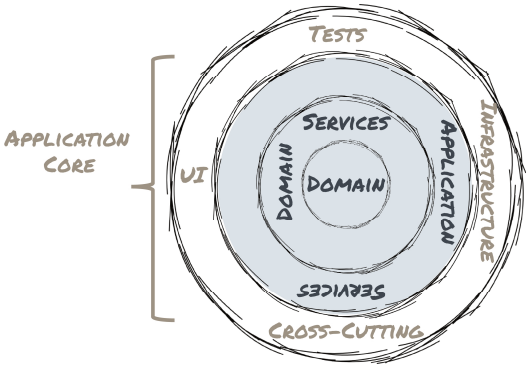

Abbildung 4: Onion Überblick [19]

Abbildung 3: Onion Abhängigkeiten [1]

[11] Vgl. [19]

3.1.3 Unterschiede und Relativierung

Die Modelle sind sich sehr ähnlich, sie sind beide hierarchisch und kapseln die Daten mit Hilfe von Layern. Sie unterscheiden sich allerdings von der Sichtweise her. Während das Layer-Modell in der Regel die Layer je nach Abstand von der Hardware definiert, werden beim Onion-Modell der Abstand zur Businesslogik und der Wahrscheinlichkeit von Veränderungen in der jeweiligen Komponente definiert. Durch die andere Herangehendweise bei einem Onion-Modell kann theoretisch eine höhere Entkopplung der Layer erreicht werden.

Bei mobilen Anwendungen trägt die Businesslogik allerdings meist nur einen kleinen Teil zur gesamten Anwendung bei. Wenn das Layer-Modell in der Entwicklung der Architektur weniger starr genutzt wird und Abhängigkeiten nicht nur von unten nach oben entstehen, verringert sich die Koppelung zwischen den Layern. Zusätzlich können Abhängigkeiten durch Interfaces und Dependency Injection reduziert werden. Ein nicht zu verachtender Vorteil des Layer-Modells ist die einfache Darstellung, welche auch von Fachfremden verstanden wird.[12]

[12] Vgl. [19]

3.2 Design-Pattern

3.2.1 MVVM

Das MVVM-Pattern gehört zur Familie des MVC-Pattern und hat sich im Windows Presentation Foundation (WPF) Umfeld entwickelt. Es ist mittlerweile das Standard-Pattern in allen XAML-basierten UI-Technologien und wird immer häufiger auch im JavaScript-Umfeld eingesetzt. Das MVVM-Pattern ermöglicht eine besonders starke Entkoppelung zwischen UI und Logik, wodurch eine Anwendung besser zu testen, pflegen und erweitern ist. Die wesentlichen Vorteile gegenüber dem MVC-Pattern ergeben sich durch das Databinding. Durch Databinding entsteht eine stärkere Entkoppelung zwischen Userinterface und dessen Logik und Daten, der Anteil an Shared-Code kann erhöht werden und es ermöglicht einen besseren Design-Workflow. Die stärkere Entkoppelung wird dadurch erreicht, dass in einem View (UI) ein BindingContext gesetzt wird, wodurch der View ein ViewModel kennt, für das ViewModel wiederum ist es irrelevant von welchem View es konsumiert wird. Das Model (Daten) wird ausschließlich vom ViewModel verwendet. Für eine Interaktion zwischen ViewModel und View werden im View Bindings gesetzt, diese Bindings können Properties (Daten) oder Commands (Events) sein, welche in beide oder nur in eine Richtung gebunden sind. Das ViewModel schickt Notifications, die vom View entgegen genommen werden. In einem View können unterschiedliche Kontexte (ViewModels) gesetzt sein und ein ViewModel kann von mehreren Views konsumiert werden.[13]

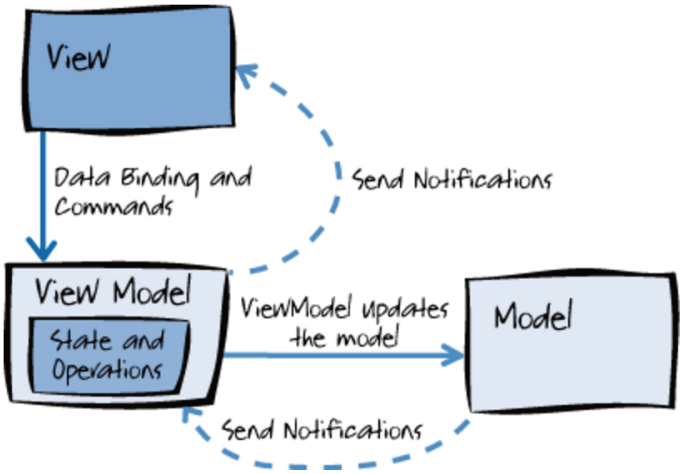

Abbildung 5: The MVVM-Pattern [2]

[13] Vgl. [15]

3.2.2 Dependency Injection

Das Dependency Injection (DI) Pattern erlaubt es, die Erzeugung von abhängigen Objekten (ServiceA) in einer Klasse (ClassA) mit Hilfe eines Interfaces (IServiceA) an einen zentralen Ort (Builder) auszulagern, also das Abhängigkeitsnetzwerk zentral aufzubauen. Die verwendeten Klassen und Eigenschaften müssen also nicht direkt angefordert, sondern nur bekannt gemacht werden. Dafür wird im Konstruktor der Klasse oder bei der Deklaration einer Eigenschaft ein Interface angegeben, welches dafür sorgt, dass das benötigte Objekt bei der Erzeugung der Klasse oder Eigenschaft injiziert wird. So werden Klassen von ihren Abhängigkeiten und der Verwaltung deren Lebenszyklen entkoppelt. Hinzu kommt, dass die konkrete Implementierung einer Abhängigkeit zur Compilezeit nicht bekannt sein muss, da diese erst zur Laufzeit injiziert wird. So kann eine injizierte Klasse oder Eigenschaft je nach Kontext eine komplett andere Implementierung aufweisen. Durch das DI-Pattern können Klassen besser isoliert getestet und Anwendungen einfacher gepflegt und erweitert werden.[14]

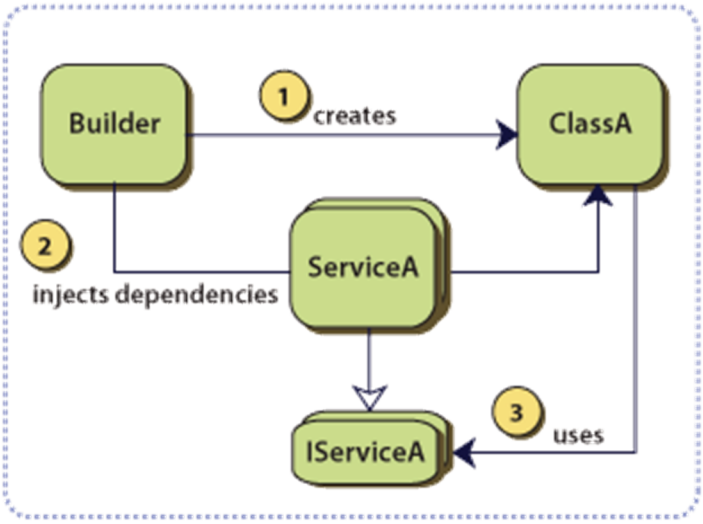

Abbildung 6: Dependency Injection [3]

[14] Vgl. [38]

3.2.3 Fassade

Die Fassade reduziert die Abhängigkeiten zwischen den Komponenten einer Anwendung. Sie dient als Schnittstelle zwischen den Komponenten und verhindert zirkulare Abhängigkeiten. Werden andere Komponenten benötigt, wird eine Anfrage an die Fassade gestellt, welche die nötigen Aufgaben an die einzelnen Komponenten delegiert. Da nur die Fassade die einzelnen Komponenten kennt, müssen bei Änderungen oder Austausch einer Komponente nur in der Fassade Änderungen stattfinden. Die restlichen Komponenten müssen nicht angepasst werden, die Komponenten sind also voneinander entkoppelt.[15]

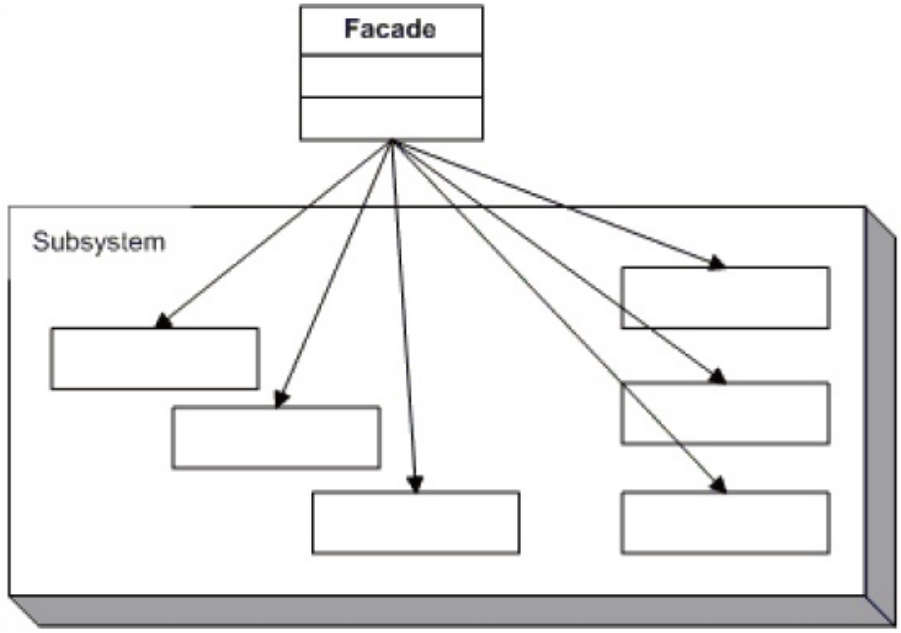

Abbildung 7: Fassade [4]

[15] Vgl. [18]

3.2.4 Data-Transfer-Object & Data-Access-Object

Historisch wird das Data-Transfer-Object (DTO) in verteilten Systemen eigesetzt, um Daten nach Bedarf zu clustern, wodurch die nötigen Zugriffe von Client und Server reduziert werden können. Des Weiteren ermöglicht es die eigentliche Datenhaltung vom Server und Client zu entkoppeln und reduziert den Aufwand von Serialisierung und Deserialisierung. Es stellt eine Schnittstelle zwischen Client und Server dar.[16]

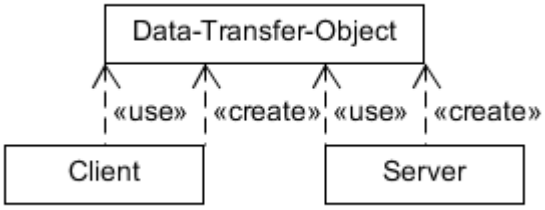

Abbildung 8: Data Transfer Object

Praktisch immer dann, wenn JSON oder SOAP eingesetzt werden, wird ein DTO verwendet. Das DTO hat in der modernen Anwendungsarchitektur auch an Bedeutung gewonnen, da DTOs auch zur Kommunikation zwischen den Layern verwendet werden.

Ein Data-Access-Object (DAO) kommt häufig in verteilten Systemen vor und dient als Persistenzschicht zwischen Businesslogik und Datenquelle. Es kapselt die Businesslogik bzw. deren Daten so von der Datenquelle, dass diese gegen jede beliebige ausgetauscht werden kann. Wenn das DAO-Pattern zur Verwendung kommt, findet dies häufig in Kombination mit einem DTO statt. Es stellt also eine Schnittstelle zwischen Businesslogik und Datenquelle dar.[17]

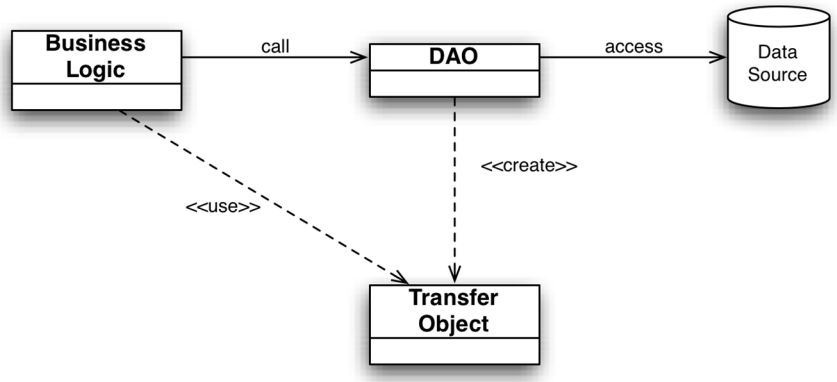

Abbildung 9: Data Access Object [5]

[16] Vgl. [16]
[17] Vgl. [17]

Kapitel 4

4 Grundlagen

Im Kapitel Allgemeine Grundlagen wird auf die für diese Arbeit wichtigen Themen Code-Sharing-Strategien, Xamarin, Windows Universal Platform und Microsoft Azure eingegangen. Dabei wird besonders intensiv auf die für Cross-Plattform-Entwicklung relevanten Faktoren eingegangen. Diese sind Shared-Code und Formfaktoren.

4.1 Überblick verwendeter Frameworks

In der Einleitung konnte bereits ein grober Einblick in die Frameworks Universal Windows Platform und Xamarin gewonnen werden, wichtig ist aber zu verstehen, wie diese zusammenhängen. Beide Frameworks werden mit der Programmiersprache C# entwickelt und basieren beide auf dem .NET-Framework. Wobei Xamarin auf Mono basiert, was ein plattformunabhängiger Open-Source .NET Klon ist. Des Weiteren basieren Xamarin.Forms und UWP auf XAML, so dass das MVVM-Pattern eingesetzt werden kann.

4.2 Xamarin

Xamarin ist ein Cross-Plattform-Entwicklungsansatz, der auf dem Mono-Projekt basiert. Mit Xamarin.Android und Xamarin.iOS ist es möglich, Anwendungen für Android, iOS und Windows Phone in C# zu entwickeln. Dabei wird aus C# heraus auf die jeweils plattformspezifischen Funktionen und Controls zugegriffen, das UI ist also plattformabhängig. Bei der Entwicklung kann auf Teile des .NET Frameworks und auf einige NuGet Packages zurückgegriffen werden. Als IDE stehen Xamarin Studio und Visual Studio zur Verfügung.

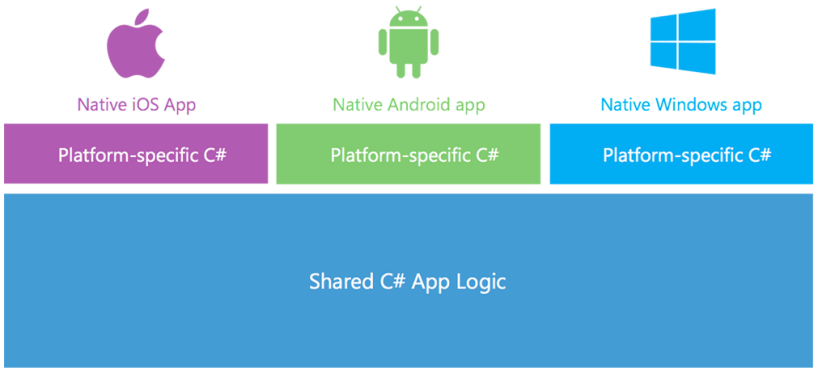

Abbildung 10: Xamarin Architektur [6]

4.2.1 Kompilierung

Die Kompilierung des C# Codes erfolgt für alle drei Plattformen unterschiedlich.[18]

Android:

Für Android wird der Code zunächst in Intermediate Language (IL) kompiliert. Der IL Code wird dann mit einen JIT-Compiler (Just-In-Time) auf dem System installiert. Der IL Code läuft in einer Mono Execution Environment (.NET APIs, Android Bindings), welche parallel zur Android Runtime läuft. Die beiden verbindet ein Java Native Interface (JNI), über das sie jeweils die APIs der jeweilig anderen Runtime aufrufen können.

iOS:

Bei iOS wird der Code mit einem AOT-Compiler (Ahead-of-time) direkt in Apples ARM Assembly Sprache übersetzt, auf dem System installiert und ausgeführt.

Windows Phone:

Für Windows Phone wird der Code in IL kompiliert, installiert und zur Laufzeit ausgeführt.

4.2.2 Xamarin.Forms

Xamarin.Forms ist eine Erweiterung von Xamarin und ermöglicht einen höheren Anteil an Shared-Code. Bei Xamarin.Forms handelt es sich um eine Abstraktion der UI-Elemente von Android, iOS und Windows Phone. Mit Hilfe von Xamarin.Forms können die UI's in XAML oder C# definiert werden. Dieser Code wird zur Laufzeit in das äquivalente native Element übersetzt.[19]

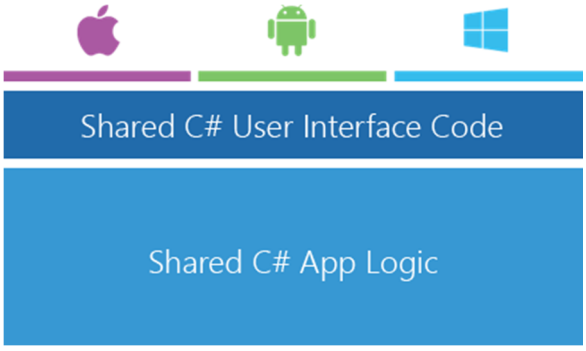

Abbildung 11: Xamarin.Forms Architektur [7]

Ein gutes Beispiel hierfür ist der DatePicker, deklariert man diesen in XAML:

```
<DatePicker Date="{Binding MyDateTimeProperty}" />
```

[18] Vgl. [20]
[19] Vgl. [22]

wird dieser in das jeweilige native Element übersetzt.

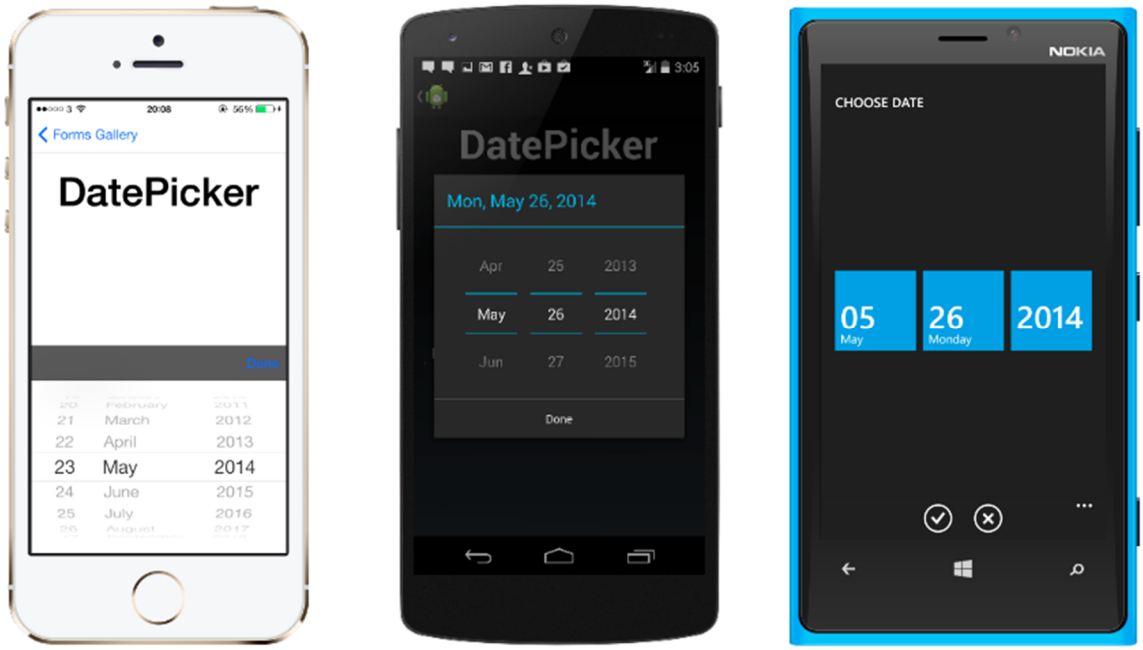

Abbildung 12: DatePicker [8]

Eine große Schwäche von Xamarin.Forms ist, dass zurzeit noch kein Designer existiert, das Projekt muss daher jedes Mal deployed werden, um das implementierte Userinterface zu überprüfen. Durch den fehlenden Design-Workflow wird das Entwerfen von Userinterfaces aufwändiger. Außerdem stellt Xamarin.Forms momentan nur die nötigsten UI-Elemente und die dazu nötigsten Attribute zur Verfügung.

Durch die zurzeit limitierte Anzahl an Attributen ist es gelegentlich nicht möglich, das geplante Design mit den vorhandenen Xamarin.Forms Elementen umzusetzen, für diesen Fall gibt es Custom Renderers. Custom Renderer ermöglichen den plattformspezifischen Zugriff auf das native Element und so auch auf dessen Attribute. Durch die bestehende Zugriffsmöglichkeit auf die nativen Elemente ist jedwedes Design, welches nativ möglich ist, auch unter Xamarin.Forms möglich. Custom Renderer müssen für jede Plattform einzeln erstellt werden und reduzieren so die Vorteile von Xamarin.Forms.

Außerdem können Xamarin.Forms UI's auf die jeweiligen Formfaktoren angepasst werden. Dafür besitzt die Klasse Device eine Eigenschaft Idiom, welche Auskunft über den Gerätetyp (Phone, Tablet) gibt. Durch eine Direktive kann das Layout für einen Phone- oder Tablet-Formfaktor individuell angepasst werden.

4.2.3 MvvmCross

MvvmCross ist ein MVVM Open Source Framework für die Cross-Plattform-Entwicklung mit Xamarin. Es ermöglicht das DataBinding in Xamarin.Android und Xamarin.iOS, wodurch auch hier das MVVM-Pattern eingesetzt und so ein höherer Wiederverwendungsgrad erzielt werden kann. Neben dem reinen DataBinding stellt MvvmCross auch eine große Anzahl von Erweiterungen zur Verfügung, durch die ein höherer Anteil an Shared-Code erreicht werden kann. Es wird beispielsweise eine SQLite-PCL zur Verfügung gestellt, durch die es möglich ist, eine lokale Datenbank plattformübergreifend zu erstellen und pflegen, die Erweiterungen werden meist mit Hilfe des DI-Patterns eingebunden.

MvvmCross Erweiterungen können auch in Kombination mit Xamarin.Forms eingesetzt werden. Zusätzlich gibt es nützliche Erweiterungen der UI-Elemente von Xamarin.Forms.

4.3 Universal Windows Platform

Schon seit Silverlight arbeitet Microsoft darauf hin, die .NET Plattformen zu vereinheitlichen. Mit Windows 8 und dann Windows 8.1 hat Microsoft die Systeme Schritt für Schritt vereinheitlicht. Im Sommer 2015 wurde mit Windows 10 die Universal Windows Platform eingeführt. Die UWP ermöglicht es, eine Universal Windows Application zu schreiben, welche auf jedem Windows 10 Gerät läuft.

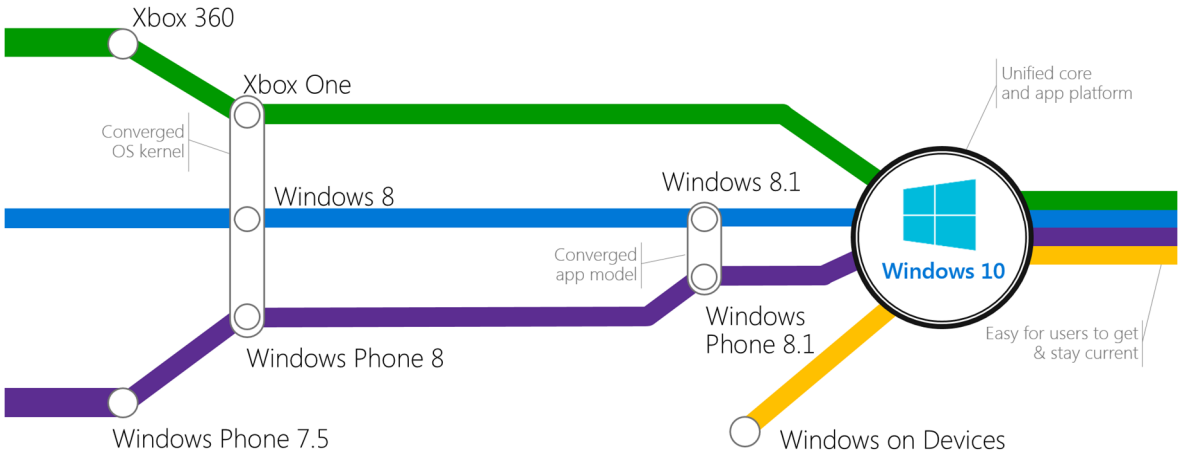

Abbildung 13: Schritte zu Windows 10 [9]

Die Entwicklung einer UWA ist möglich, da die UWP einen Core API Layer anbietet. Neben den Core API's können bei Bedarf auch weiterhin gerätespezifische API's eingebunden werden. Es kann also weiterhin für eine bestimmte Gerätefamilie oder ein Set an Gerätefamilien entwickelt werden. UWA werden mit C#, C++, .NET, XAML oder HTML/JS entwickelt.[20]

[20] Vgl. [23]

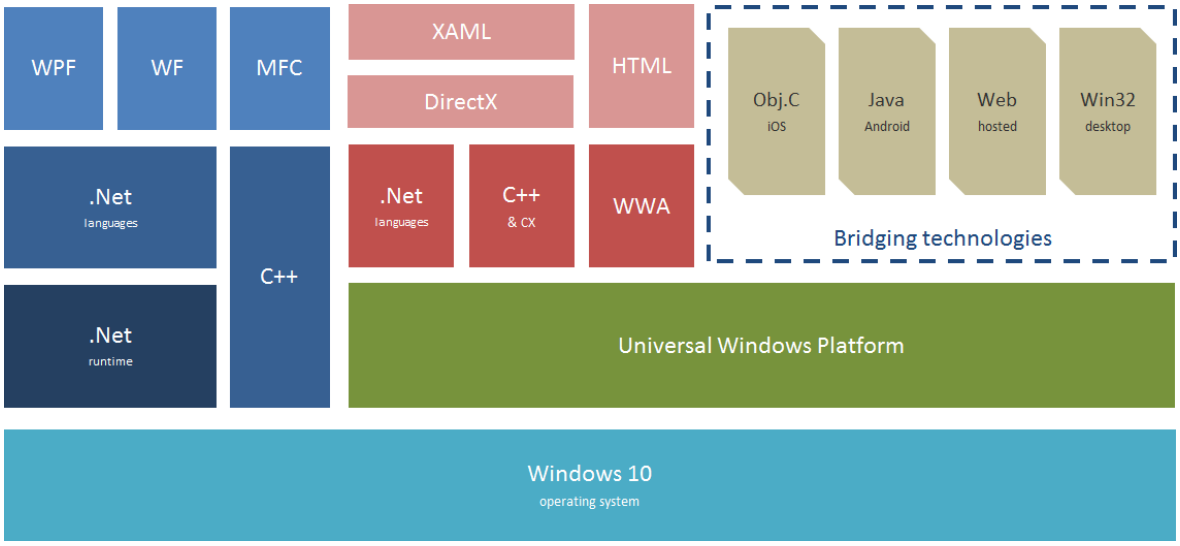

Abbildung 14: Windows Universal Platform Architektur [9]

4.3.1 Gerätefamilien

Unter Windows 10 läuft künftig eine große Anzahl an Geräten, die alle unterschiedliche Formfaktoren und Sensoren haben. Daher gibt es für jede Gerätefamilie ein eigenes OS und API Set. Darauf sollte bei der Planung einer Anwendung genau geachtet werden und abgewogen werden, für welche Gerätefamilie/-n entwickelt werden soll. Dies ist stark vom jeweiligen Bedarf abhängig. Microsoft hat dabei folgende Gerätefamilien festgelegt:[21] [22]

- Universal Device Family: Alle Geräte
- Desktop Device Family: Tablets, Laptops, Desktop PCs
- Mobile Device Family: Smartphones, Phablets
- IoT Device Family: Wearables, Haushaltsgeräte
- Surface Hub / Xbox Device Family: Surface Hub, Xbox

[21] Vgl. [23]
[22] Vgl. [24]

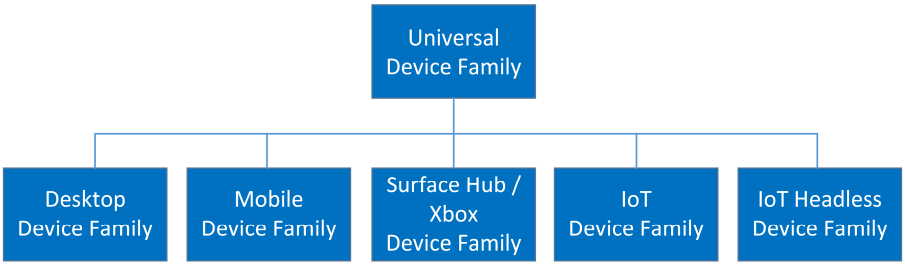

Abbildung 15: Gerätefamilien

Bei der Universal Device Family handelt es sich um eine besondere Gerätefamilie, sie hat als einzige kein eigenes Betriebssystem. Sie ist der Core API Layer, der die Gemeinsamkeiten der anderen API Sets ableitet. Diese Arbeit beschränkt sich dabei im weiteren Verlauf auf die Gerätefamilien Desktop und Mobile.

4.3.2 Formfaktor

Mit Hilfe des Core API Layers sind schon viele Probleme der Cross-Plattform-Entwicklung gelöst worden, eines bleibt jedoch, der Formfaktor. Eine Anwendung über mehrere Gerätefamilien hinweg bedienbar und ansprechend zu gestalten, ist ein schwieriges Unterfangen, da sich diese stark von ihrer Dimension und der Auflösung unterscheiden.

Um den unterschiedlichen Dimensionen der Geräte gerecht zu werden, muss eine Anwendung Informationen auf dem Smartphone, im Gegensatz zum PC, entweder reduziert oder anders strukturiert anzeigen. Für dieses Problem wendet das Web bereits die Konzepte des Adaptive- und Repsonsive-Design an, welche von Microsoft in der Universal Windows Platform adaptiert wurden. Mit Hilfe von AdaptiveTrigger ist es möglich, Elemente ein- und auszublenden, zu verschieben und vieles mehr. Zusätzlich wurde ein RelativePanel eingeführt, welches es ermöglicht, Elemente relativ zueinander auszurichten. So können folgende Effekte erzielt werden:

Ändern der Position

Abbildung 16: Formfaktor - Position ändern [10]

Ändern der Größe

Abbildung 17: Formfaktor - Größe ändern [10]

Neuanordnen

Abbildung 18: Formfaktor – Neuanordnen [10]

Einblenden

Abbildung 19: Formfaktor – Einblenden [10]

Ersetzen

Abbildung 20: Formfaktor – Ersetzen [10]

Ändern des Navigtionsablaufs

Abbildung 21: Formfaktor - Architektur ändern [10]

Neben der Dimension eines Gerätes ist auch dessen Auflösung bzw. Pixeldichte wichtig. Wird das nicht berücksichtigt, wirkt ein Element auf niedriger Auflösung sehr groß und auf hoher Auflösung sehr klein. Um diesen Effekt zu vermeiden, werden UI's in effektiven Pixel und nicht in physischen Pixel entwickelt. Die Pixelzahl eines Elements passt sich je nach Gerät und Auflösung an. Ein Element das am Handy 150x150 Pixel groß ist, ist dann am Laptop beispielsweise 175x175 Pixel groß.[23]

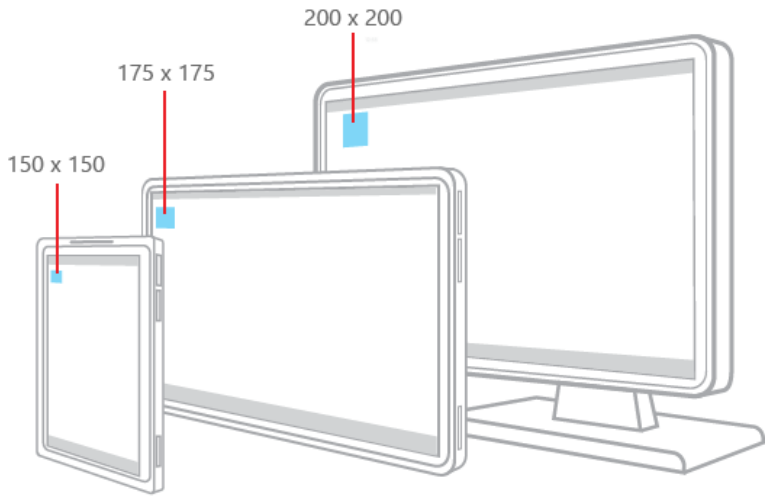

Abbildung 22: Effektive Pixel [10]

4.4 Code-Sharing-Strategien

Um zwischen Xamarin und UWP möglichst viel Code zu teilen und gleichzeitig weiterhin plattformspezifische Anpassungen vornehmen zu können, bietet das .NET Framework zwei mögliche Code-Sharing-Strategien an, zum einen ein Shared Project und zum anderen Portable Class Libraries (PCL).

4.4.1 Shared Project

Ein Shared Project ist, eine simple Variante für mehrere Plattformen zu entwickeln, die es einem Entwickler erlaubt, mit Compilerdirektiven plattformspezifischen Code zu schreiben. Dabei hält das Shared Project den Code nur vor, die eigentliche Übersetzung findet dann auf den Plattform-Projekten statt, so dass ein Assembly je Plattform entsteht.[24]

[23] Vgl. [10]
[24] Vgl. [21]

Positiv	Negativ
• Compilerdirektiven können genutzt werden • Es können plattformspezifische Referenzen genutzt werden	• Compilerdirektiven machen den Code weniger übersichtlich • Refactoring wird erschwert • Aus einem Shared Project kann keine DLL erzeugt werden • Schlechte Testbarkeit

4.4.2 Portable Class Library

Eine Portable Class Library (PCL) ist eine architektonisch etwas kompliziertere Code-Sharing-Strategie. Bei einer PCL wird im Vorfeld eine Kombination von Zielplattformen ausgewählt, woraus ein Profil generiert wird. Dieses Profil definiert, welche APIs dem Entwickler in dieser Assembly zugängig sind. Es entsteht ein Assembly, das auf allen vom Profil adressierten Plattformen verwendet werden kann.[25]

Positiv	Negativ
• Übersichtlicherer Code • Einfacheres Refactoring • Andere Projekte, die kompatibel zum Profil sind, können das Projekt referenzieren • Es kann genau eine DLL erzeugt werden • UnitTests testen den tatsächlich verwendeten Code	• Es können nicht überall plattformspezifische Referenzen genutzt werden • Je nach Profil kann nur ein Teil des .NET Frameworks benutzt werden • Zur Nutzung plattformspezifischer Dienste sind komplexere Patterns und Abstraktionen notwendig

4.5 Microsoft Azure

Cloud-Computing Dienste sind für die meisten mobilen Anwendungen existenziell wichtig. Ein immer größer werdender Teil der Logik moderner Anwendungen wird in die Cloud ausgelagert. Dieser Prozess nimmt immer mehr Tempo auf. Daher wird im Rahmen dieser Arbeit auch auf zwei Cloud Dienste zugegriffen und die damit verbundenen Synergieeffekte aufgezeigt.

Ein möglicher Cloud-Computing Dienst ist Azure von Microsoft. Microsoft Azure umfasst eine große Anzahl an Diensten, welche die volle Palette des Cloud-Computing abdeckt. Jeder Dienst ist dabei optimiert für eine Aufgabe und zum Teil kombinierbar. Besonders hervorzuheben ist, dass je nach Service

[25] Vgl. [21]

und Virtual-Machine-Role nicht nur mit .NET (C# und Visual Basic) entwickelt werden kann, sondern auch mit C++, PHP, Ruby, Python oder Java. Eine weitere Besonderheit ist, dass alte On-Premise Lösungen in einer Virtual Machine gehostet und als Cloud-Service zur Verfügung gestellt werden können. Für Entwickler ist meist ein Remote-Debugging möglich.

4.5.1 Azure Active Directory

Azure Active Directory (AAD oder Azure AD) ist ein Verzeichnis- und Identitätsverwaltungsdienst von Microsoft. Er bietet dabei alle Identitätsverwaltungsfunktionen, von mehrstufiger Authentifizierung, über rollenbasierte Zugriffssteuerung, bis hin zu Sicherheitsüberwachung und -warnungen. Sehr interessant für mobile Anwendungen sind dabei die Authentifikationsmöglichkeiten von Azure AD. Mit Hilfe von Azure AD kann der Nutzer sich für eigene Azure Services und für die Cloud Produkte von Microsoft, wie Office 365, authentifizieren. Außerdem lassen sich Azure AD Accounts mit denen von Drittanbietern wie Facebook, Dropbox oder Xing verknüpfen, wodurch ein Single Sign-on, also eine einmalige Authentifizierung für unterschiedliche Dienste, möglich ist.

4.5.2 Azure Mobile Services

Der Azure Mobile Service ist dafür optimiert, ein Cloud-Back-End für Anwendungen zur Verfügung zu stellen. Mit Hilfe des Azure Mobile Services lässt sich beispielsweise schnell und einfach eine REST-API implementieren. Dabei durchläuft der Entwickler einen Wizzard. Aus den dort getätigten Vorgaben und Einstellungen wird eine Datenbank mit passender REST-API generiert, welche dann nach Belieben angepasst werden kann. Ein Remote-Debugging ist möglich.

Kapitel 5

5 Methodik

Das Kapitel 5 – Methodik schildert die Herausforderungen bei der Herangehensweise und Planung einer Cross-Plattform-Anwendung. Dabei werden im ersten Abschnitt kurz die Ziele der Arbeit wiederholt. Im Anschluss folgt eine kurze Aufführung der Prototypen. Danach werden wichtige konzeptionelle und designtechnische Problemstellungen und Entscheidungen erläutert. Außerdem wird auf eine mögliche Architektur eingegangen.

5.1 Konkrete Ziele

Das Ziel ist es, durch Prototypen zu überprüfen, inwieweit es möglich ist, für Windows Desktop, Tablet, Phone, Android und iOS eine Anwendung nur einmalig implementieren zu müssen. Dabei soll neben der Implementierung der Fokus auch auf die Architektur gelegt werden. Zum Einsatz kommen die Frameworks Xamarin.Forms und Windows Universal Platform. Es soll dabei für die einzelnen Layer einer Anwendung evaluiert werden, welches Code-Sharing Potential sie aufweisen. Zusätzlich soll ein für Code-Sharing-Projekte allgemeingültiger Architekturüberblick gegeben werden. Außerdem sollen die Möglichkeiten der einzelnen Technologien möglichst nicht beschnitten werden. Unrealistisch, weil zu aufwändig, wäre es die UI-Elemente zu abstrahieren, daher werden hier zwei Implementierungen der UI's in Kauf genommen. Die Überlegungen, Konzepte und Implementierungen werden im Kontext von prototypischen Anwendungen erarbeitet.

5.2 Die Prototypen

Im Folgenden werden die während der Implementierung entstandenen Prototypen beschrieben. Des Weiteren wird aufgezeigt, für welchen Testzweck die einzelnen Prototypen erstellt wurden.

5.2.1 ShowTime

ShowTime ist der Show Case der SDX AG. Bei ShowTime handelt es sich um ein Zeiterfassungssystem, welches verschiedene Backends, sowie unterschiedlichste Clients mit den unterschiedlichsten Technologien aufweist. Der Prototyp ermöglicht es dem Nutzer, Tasks und Zeiteinträge anzuzeigen, sowie neue Zeiteinträge zu erstellen.

ShowTime diente als erste Spielwiese. Hier wurde zunächst ein Projektsetup ermittelt und anschließend die eventuellen Code-Sharing Probleme innerhalb der ViewModels, sowie die sich aus den verschiedenen Formfaktoren ergebenden Navigations-Herausforderungen untersucht. Außerdem wurde hier die Implementierung unterschiedlicher Formfaktoren erprobt. Es wurden auch eine Authentifikation sowie ein REST-API Zugriff zu dem Backend implementiert.

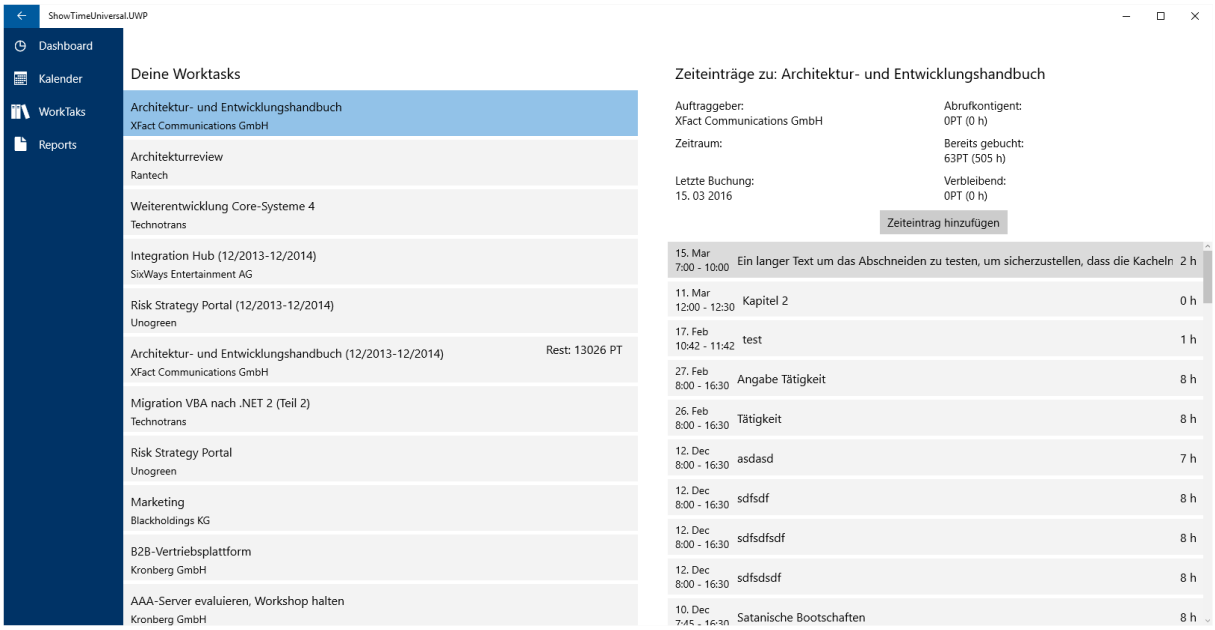

Abbildung 23: ShowTime UWP Desktop

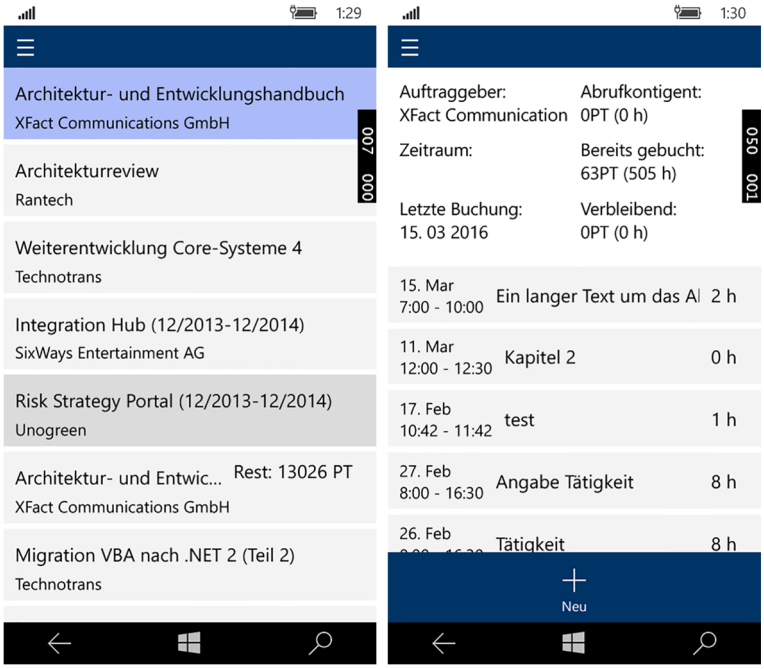

Abbildung 24: ShowTime UWP Mobile

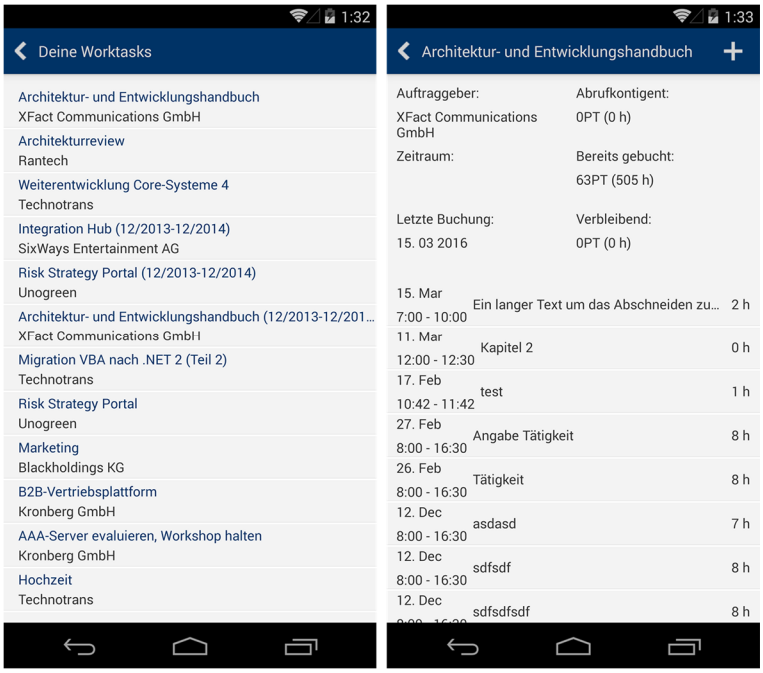

Abbildung 25: ShowTime Android

5.2.2 Meeting-App

Die Meeting-App ist eine Anwendung, die im Rahmen der werkstudentischen Tätigkeit bei der SDX AG entstanden ist. Sie ermöglicht es, Meetings mit Hilfe von unterschiedlichen Kriterien und einem Freitext zu bewerten.

Mit Hilfe der Meeting-App wurde eine Authentifikation mit einem externen Service, sowie das beziehen dessen Daten, durch eine REST-API durchexerziert. Außerdem wird diese Anwendung im Abschnitt Shared-Code-Anteil verwendet, da diese ein produktives Projekt ist und so quantitative Daten ermittelt werden können.

5.2.3 Serien-App

Die Serien-App ist unabhängig von der SDX AG entstanden und war ein privates Projekt. Der Prototyp bietet einem Nutzer die Möglichkeit, Serien hinzuzufügen, zu löschen oder deren Inhalte zu aktualisieren. Er ermöglicht es auch, Folgen als gesehen zu markieren und zeigt dem Nutzer wichtige Informationen, wie bspw. das Ausstrahlungsdatum der nächsten Folge an.

An der Serien-App wurde die lokale Datenhaltung mit einer SQLite Datenbank implementiert. Außerdem dient die App auch zur Erläuterung der Navigationsprobleme. Hier wurde am Ende festgestellt, dass anhand dieses Beispiels das Verständnis für das Problem einfacher vermittelt werden kann.

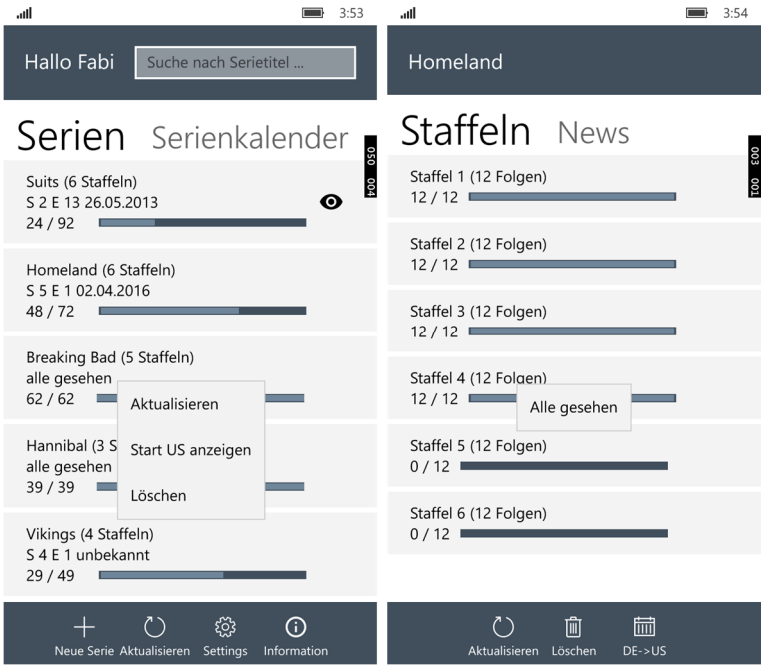

Abbildung 26: Serien-App UWP

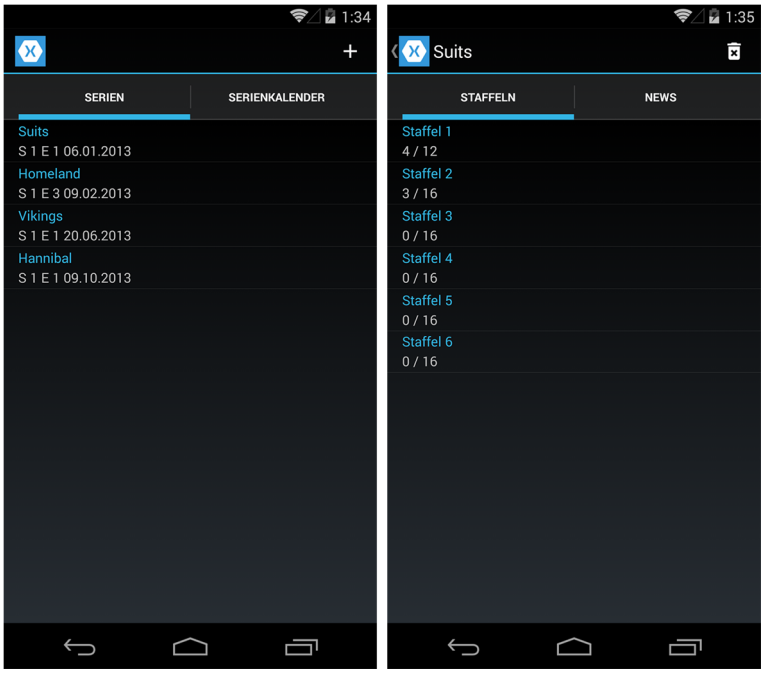

Abbildung 27: Serien-App Android

5.3 Konzeption und Design

Die Konzeption und das Design einer Cross-Plattform-Anwendung beherbergen unterschiedliche Herausforderungen. Dies sind zum einen die unterschiedlichen Gerätefamilien, welche unterschiedlichste Hardware und verschiedene Formfaktoren aufweisen. Zum anderen gehören hierzu die verschiedenen UI-Elemente und Design-Guidelines der Betriebssysteme.

5.3.1 Unterschiedliche Betriebssysteme

Jedes Betriebssystem bringt seine eigenen UI-Elemente und Design-Guidelines mit, an die die Nutzer der jeweiligen Betriebssysteme gewöhnt sind. Daher sollten diese für eine optimale Usability und einen guten Eindruck eingehalten werden. Gerade durch die unterschiedlichen Design-Guidelines sind viele Bedienelemente einer App, je nach Betriebssystem, an unterschiedlichen Stellen zu finden. Werden die Guidelines nicht eingehalten, kann es dazu führen, dass der Nutzer längere Zeit suchen muss um ein Bedienelement zu finden, wodurch die Anwendung schlechter zu bedienen ist.

Design-Guidelines

Die wichtigsten Unterschiede der Design-Guidelines sind die folgenden:

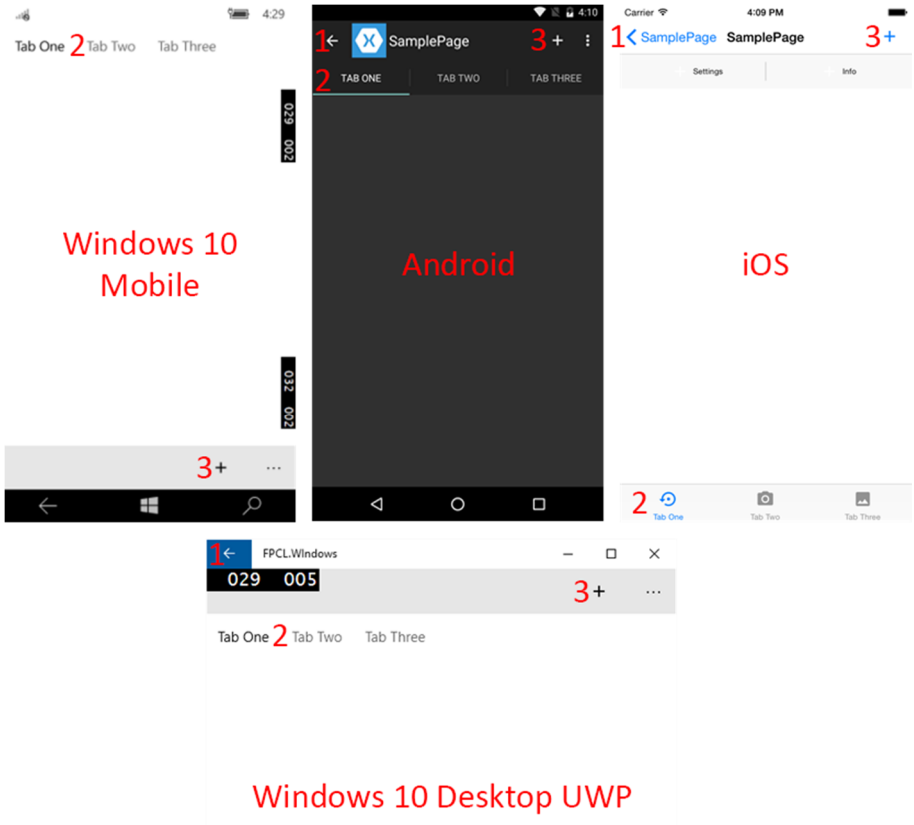

Abbildung 28: Design-Guidelines 1-3

1. Backbutton

Bei Android, iOS und Windows Desktop befindet sich ein Backbutton oben links, bei Windows Mobile gibt es keinen, da es hierfür immer einen Hardwarebutton gibt (oder eine vom Betriebssystem bereitgestellte Alternative).

2. Tabs

Bei Android und Windows befinden sich die Tabs oberhalb der in dem Tab befindlichen Elemente. Bei iOS sind die Tabs hingegen am unteren Bildschirmrand zu finden.

3. Bedienelemente

Die Bedienelemente befinden sich bei Android und Windows Desktop immer oben in der NavigationBar. Bei iOS sowie Windows Mobile sind diese in der Regel in einer Leiste am unteren Bildschirmrand, bei iOS können aber auch Bedienelemente oben rechts platziert werden.

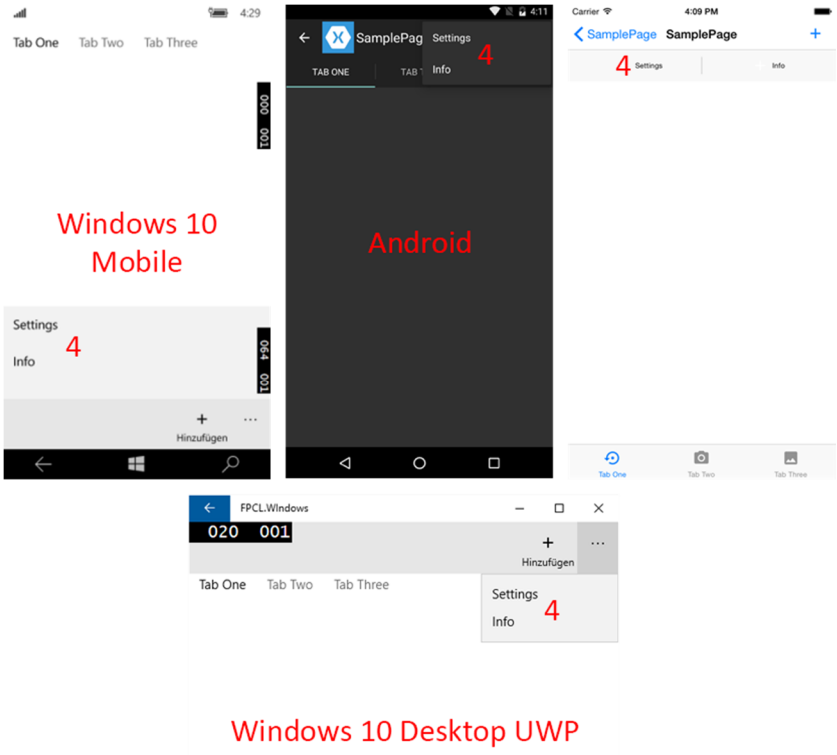

Abbildung 29: Design-Guidelines 4

4. „Mehr“ Bedienelemente

Wenn mehr Bedienelemente vorhanden sind, als angezeigt werden können, wird unter Android das in der Abbildung 29 zu sehende Bedienelement eingesetzt. Drückt der Nutzer darauf, öffnet sich eine Liste mit den Bedienelementen. Unter iOS wird ein „Mehr“ Button zu den Bedienelementen hinzugefügt. Beim Click auf diesen öffnet sich eine Liste mit weiteren Bedienelementen, hier liegt einer der sehr seltenen Fälle vor, bei dem sich Xamarin.Forms nicht automatisch an die Richtlinien hält. Bei Windows kann der Nutzer den Button mit den drei horizontal angeordneten Punkten anklicken, wodurch über der Bedienleiste (Desktop darunter) eine Liste mit weiteren Bedienelementen eingeblendet wird.

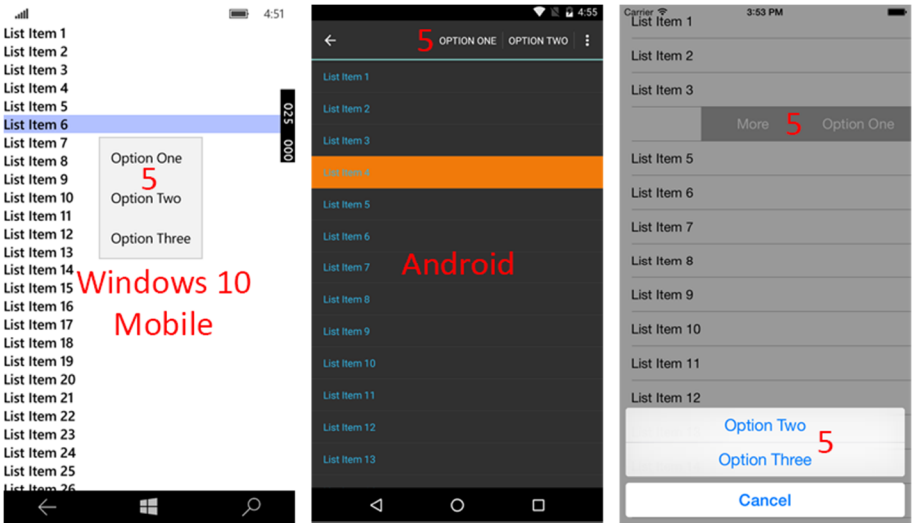

Abbildung 30: Design-Guidelines 5

5. ListView Kontextmenü

Android bietet für das Kontextmenü zwei Möglichkeiten, die beide durch ein “long-tap” auf das List-Element ausgelöst werden. Eine Möglichkeit ist ein Overlay über die NavigationBar mit den Bedienelementen, eine andere ein Overlay über das List-Element. Bei Windows gibt es ein Overlay über das List-Element, welches durch ein “long-tap” auf das List-Element ausgelöst wird. Bei iOS muss der Nutzer nach links swipen und bekommt dann zwei Bedienelemente im List-Element angezeigt. Sind zwei Bedienelemente nicht ausreichend, dann wird über ein Bedienelement More ein Kontextmenü geöffnet.

UI-Elemente

Die UI-Elemente der Betriebssysteme unterscheiden sich fast alle voneinander, auch unterscheiden sich die UI-Elemente zwischen den Betriebssystemversionen. Ein gutes Beispiel dafür ist die Picker-Familie (iOS 9, Android 6 und Windows 10 UWP):

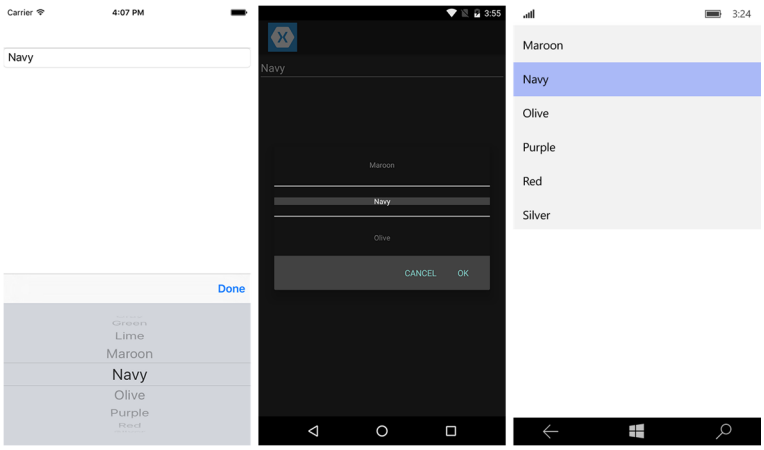

Abbildung 31: UI-Elemente Picker

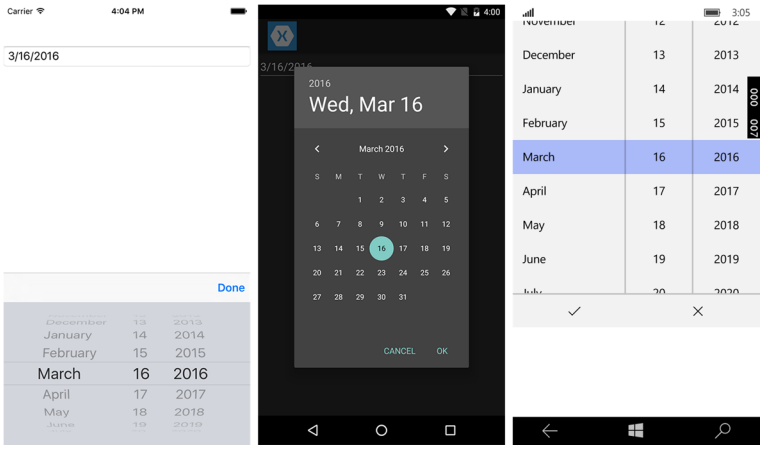

Abbildung 32: UI-Elemente DatePicke

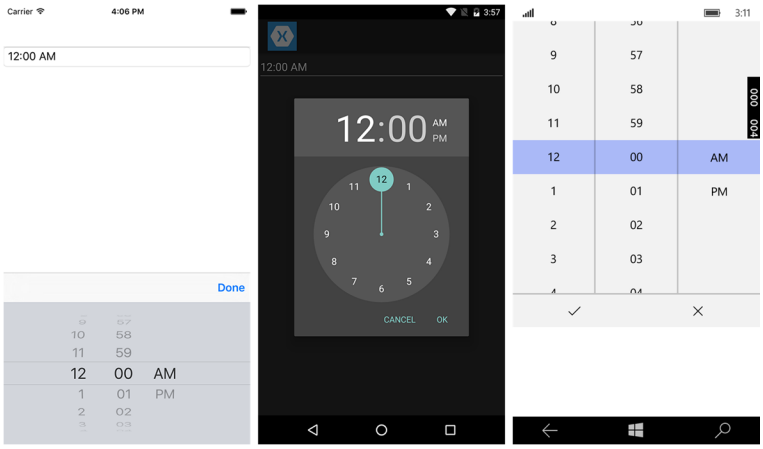

Abbildung 33: UI-Elemente TimePicker

5.3.2 Verschiedene Gerätefamilien

Die Betriebssysteme iOS, Android und Windows bringen unterschiedlichste Gerätefamilien und Formfaktoren mit sich, für die eine Anwendung nach Möglichkeit optimiert sein sollte. Am auffälligsten sind diese Unterschiede bei Windows 10. Hier sollte eine Anwendung nicht nur für Smartphones und Tablets optimiert werden, sondern auch für große Bildschirme von PC Monitoren bis hin zu TV-Geräten oder einem Surface Hub. Außerdem müssen im Gegensatz zu iOS und Android dynamische Größenänderungen mitbedacht werden, da ein Anwender auf dem Desktop das Fenster beliebig groß ziehen kann. Des Weiteren kann es durch Neuheiten wie Continuum dazu kommen, dass eine Smartphone-Anwendung auch auf einem größeren Monitor angewendet wird.

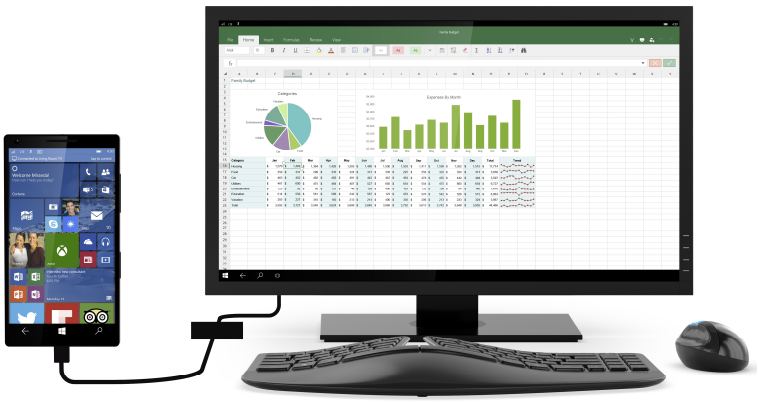

Abbildung 34: Continuum [11]

Natürlich muss eine Anwendung, je nach Verwendungszweck, nur für einige dieser Gerätefamilien optimiert werden, doch selbst eine Optimierung für Smartphones, Tablets und PC Monitore stellt einen Entwickler vor einige Herausforderungen.

Eines dieser Probleme ist es, dass verschiedenste Gerätefamilien unterschiedlichste Sensoren und somit andersartige Funktionen unterstützen. Ein gutes Beispiel hierfür ist die native Telefonie, welche von jedem Smartphone unterstützt wird, aber von den wenigsten Tablets oder PCs. Ein weiteres gutes Beispiel ist die Ortung via GPS. Die meisten Smartphones und Tablets haben einen GPS Sensor, aber die wenigsten Laptops. Hierzu muss aber gesagt werden, dass sich die Unterschiede in den letzten Jahren stark reduziert haben, gerade bei Laptops, Smartphones und Tablets. Viele der modernen Laptops haben bereits Touchscreens und einige auch einen GPS Sensor. Außerdem war es beispielsweise lange Zeit nicht möglich, eine Maus oder Tastatur an ein Handy anzuschließen, was jetzt teilweise möglich ist.

Gerätefamilie / Funktionen	Smartphone	Tablet	Desktop
GPS Tracking	immer	meistens	selten
Profilbilder machen	immer	immer	teilweise
NFC	immer	selten	nie

Abbildung 35: Gerätefamilien Matrix

Um diese Herausforderung zu bewältigen, sollte im Vorfeld der Entwicklung ein möglichst genauer Funktionsumfang erarbeitet werden. Dieser kann dann mit Hilfe einer Matrix, wie in Abbildung 35 beispielhaft, ausgewertet werden. Aus der Matrix ist dann abzulesen, ob es möglich ist, eine Anwendung für gewisse Gerätefamilien zu entwickeln und wenn es möglich ist, welche Einschränkungen sich gegebenenfalls für einige der Gerätefamilien ergeben.

Eine weitere Schwierigkeit ist es, die Informationen und Funktionen einer Anwendung auf den unterschiedlichsten Formfaktoren in einem logischen und ansprechenden Format bereitzustellen.

Hierfür gibt es generell drei Ansätze, die Microsoft wie folgt definiert hat:

Responsive Design

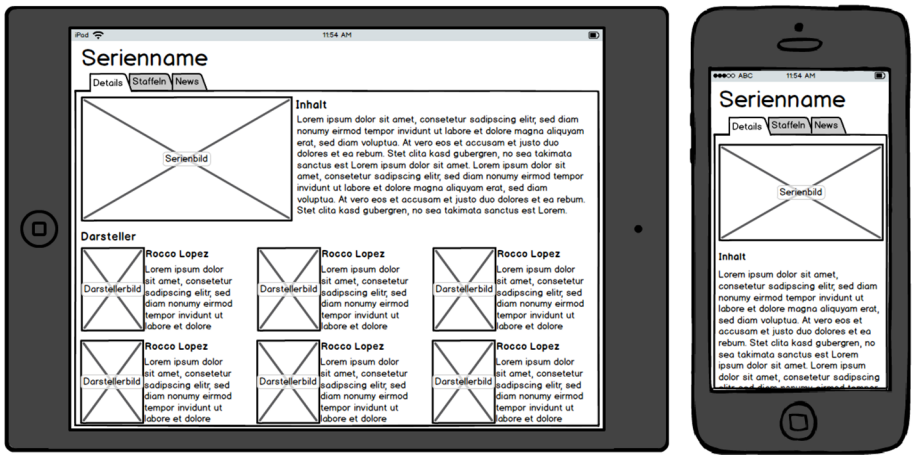

Abbildung 36: Responsive Design Mockup

Beim Responsive Design finden durch Formfaktoränderungen kleinere Anpassungen der Elemente statt. Die Elemente verhalten sich abhängig vom Formfaktor, je nach Formfaktor ändern sie ihre Größe und ordnen sich leicht anders an.

Adaptive Design

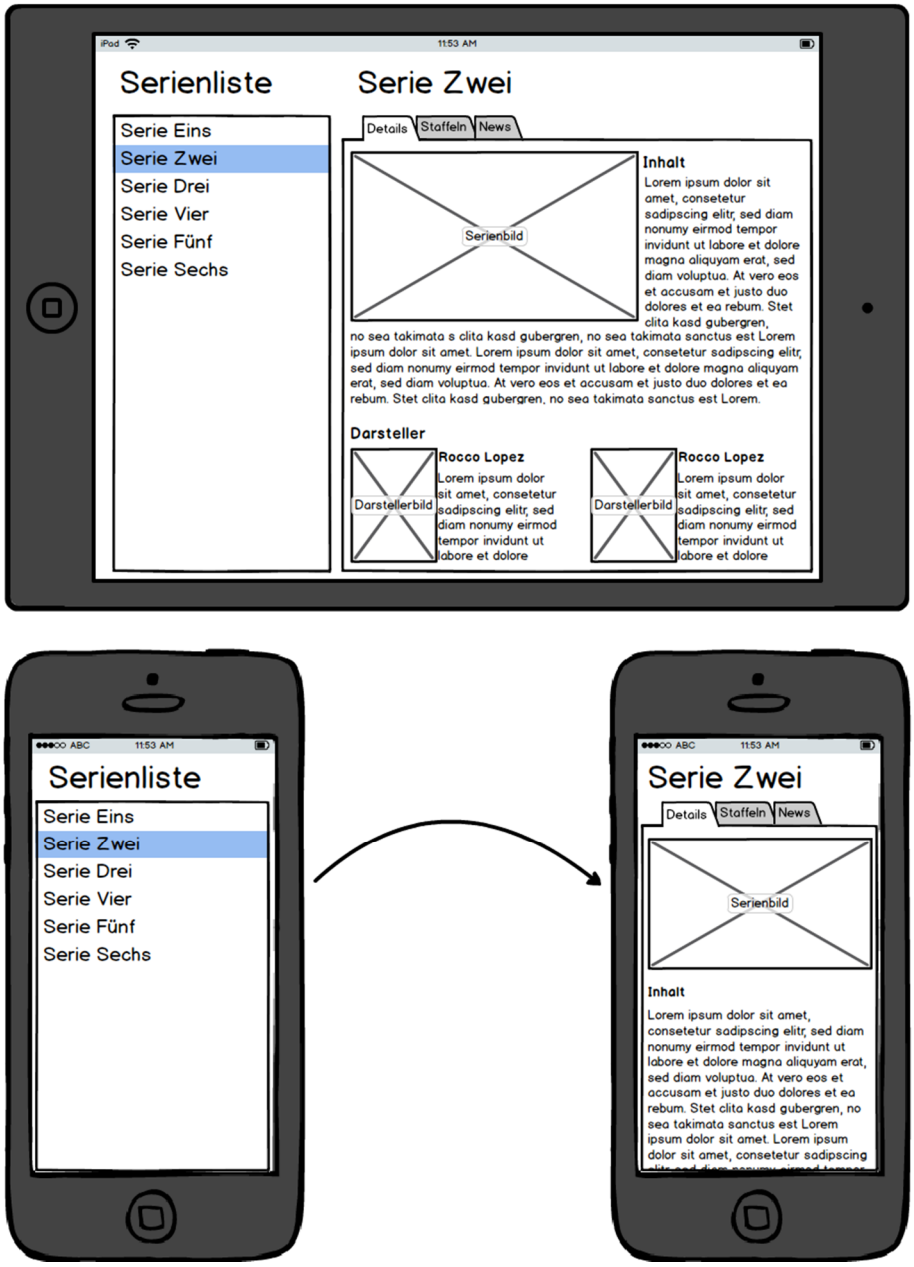

Abbildung 37: Adaptive Design Mockup

Das Adaptive Design geht in seinen Formfaktor abhängigen Änderungen deutlich weiter. Hier werden Informationen und Bedienelemente nicht nur leicht in Ihrer Ordnung und Größe verändert dargestellt, sondern, je nach Formfaktor, auch ausgeblendet und an komplett anderen Stellen angezeigt. Neben diesen Änderungen kann sich auch der Navigationsfluss verändern, indem bspw. auf einem großen Display mehrere Einzelseiten der Smartphone Ansicht in einer zusammengefasst werden. Der Nutzer wechselt also nicht mehr von Seite 1 auf Seite 2, sondern hat beide in einer Übersicht (Abbildung 37).

Tailored Design

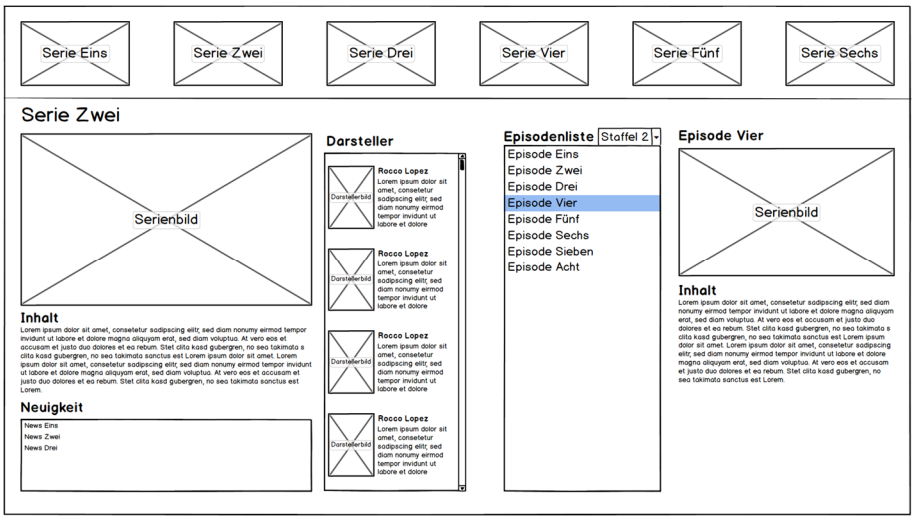

Abbildung 38: Tailored Design Mockup

Unter Tailored Design wird verstanden, dass es für jeden Formfaktor einen komplett eigenen View gibt. Das Tailored Design kommt zum Einsatz, wenn sich der Informations- und Funktionsumfang, also das dargestellte Design, zwischen den Formfaktoren gewollt oder aufgrund der eingeschränkten Möglichkeiten einer Gerätefamilie sehr stark voneinander unterscheiden. Es wird also im Vorfeld entschieden, dass die Nutzung der Views von kleineren Formfaktoren keinen Sinn macht.

5.3.3 Mögliche Herangehensweisen

Für die Konzeption einer Anwendung für verschiedene Betriebssysteme und Gerätefamilien gibt es verschiedene Herangehensweisen, die für jede Ansicht ggf. auch unterschiedlich gewählt werden kann. Allgemein kann man auch sagen, dass das Betriebssystem für die Konzeption irrelevant ist, da der Infromations- und Funktionsumfang gleich bleibt. Beim Design muss aber auf die Design-Guidelines der jeweiligen Plattform geachtet werden. Xamarin.Forms spart hier auch viel Arbeit, da es die Elemente automatisch in das jeweilige native UI-Element umwandelt.

Eine Herangehensweise während der Konzeption ist es, die Ansicht zunächst für den Formfaktor Smartphone (kleinster Formfaktor) zu entwerfen und die einzelnen Ansichten dann für immer größere Bildschirme immer weiter zusammenzufassen.

Ein weiterer Ansatz ist es, zunächst die Desktop Ansicht zu entwerfen und von dieser die Funktionen und Informationen zu priorisieren und dann ggf. Informationen oder Funktionen zu reduzieren, oder die Informationen und Funktionen, je nach Priorisierung, auf Unterseiten oder Tabs zu verteilen.

Generell ist zu sagen, dass sich gegenwärtig der mobile first Ansatz durchsetzt. Dennoch sollte von Projekt zu Projekt entschieden werden, welche Herangehensweise die bessere ist.

5.4 Architektur

Bei einer Cross-Plattform-Anwendung ist eine gute Architektur von essentieller Bedeutung. Sie ist ausschlaggebend für die Wartbarkeit, Testbarkeit, Erweiterbarkeit sowie Performance einer Anwendung. Die richtige Architektur entscheidet auch darüber, wie hoch der Anteil an Shared-Code ist oder wie viel im Umkehrschluss plattformspezifisch umgesetzt werden muss. Die Architektur entscheidet also über zeitliche und finanzielle Aufwände für die Implementierung, Wartung und Erweiterung der Anwendung.

5.4.1 Design-Pattern

Der Einsatz der richtigen Design-Pattern innerhalb einer Architektur ist ebenso entscheidend wie die Architektur an sich. Dabei lösen die Design-Pattern Teilprobleme einer Architektur. Genauso wichtig ist es Design-Pattern richtig einzusetzen, denn sonst werden diese schnell zu Antipattern.

MVVM

Das MVVM-Pattern ermöglicht durch seinen hohen Grad an Kapselung zwei verschiedene Userinterfaces (Views) zu implementieren, aber die Navigation, Steuerung und Daten der Views in einem ViewModel abzubilden (Abbildung 39). So wird ein höherer Anteil an Shared-Code erreicht und die Aufwände reduziert.

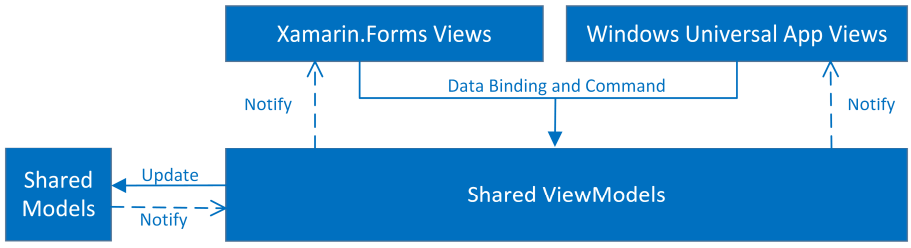

Abbildung 39: Methodik - MVVM

Dependency Injection und Service Fassade

Dadurch, dass beim Dependency Injection Pattern die Objekte und das Abhängigkeitsnetzwerk zentral erzeugt werden, können unter anderem plattformspezifische Services einfacher implementiert und verwendet werden. Dies trifft aber auch auf die anderen Services zu.

In die Service Fassade werden die einzelnen Services injiziert. Sie stellt die Schnittstelle zwischen den Services untereinander, sowie zwischen Services und ViewModel (Abbildung 40). Sie entkoppelt also die Services untereinander und die ViewModels von den Services.

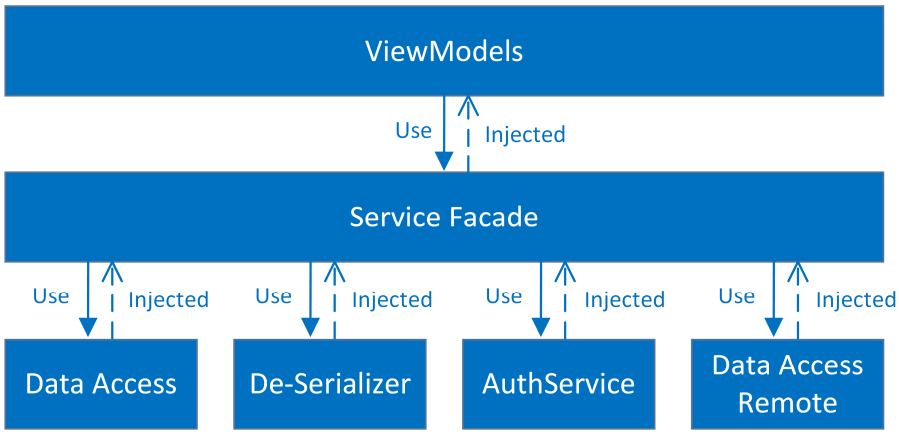

Abbildung 40: Methodik - Service Fassade & DI

Für ein besseres Verständnis ist in Abbildung 41 ein Sequenzdiagramm zu sehen, welches die möglichen Abläufe darstellt, wenn ein neuer Datensatz erstellt wird.

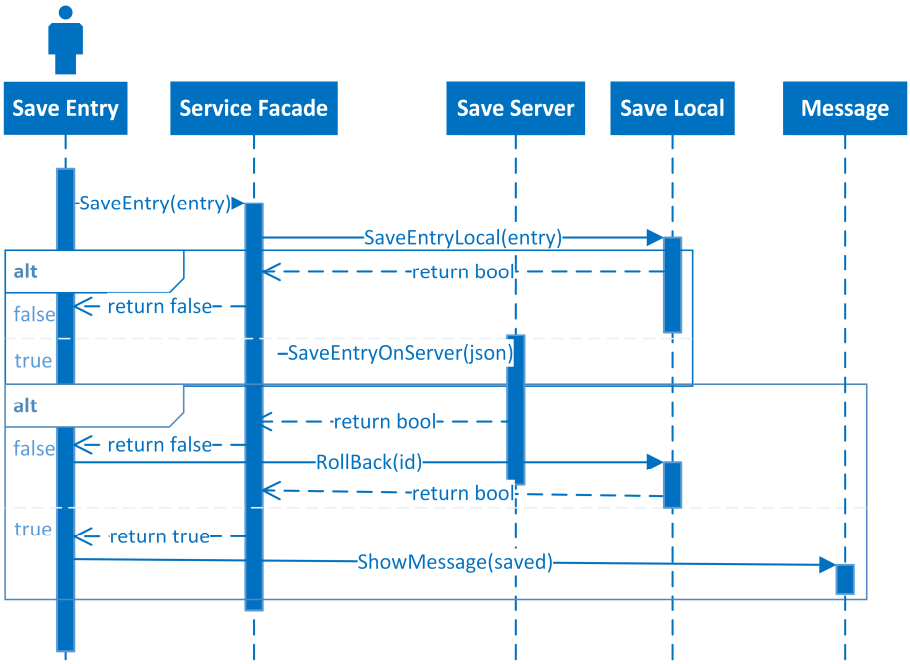

Abbildung 41: Methodik - SF & DI Sequenzdiagramm

Data-Access-Object und Data-Transfer-Object

Wenn die serverseitige Implementierung ein Azure Cloud Service ist, dann können Data-Access-Object und Data-Transfer-Object über eine PCL geteilt werden. Dies verstärkt die in den Grundlagen erwähnten Vorteile, vereinfacht die Serialisierung und Deserialisierung weiter und es kann ein größerer Anteil an Shared-Code erreicht werden.

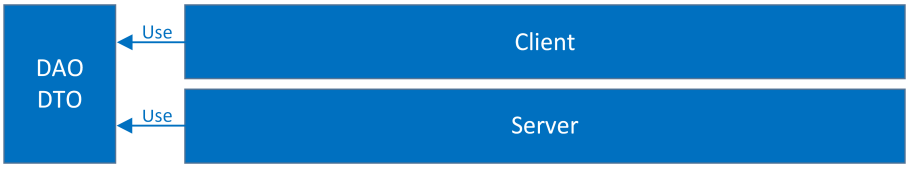

Abbildung 42: Methodik - DAO & DTO

5.4.2 Geplante Architektur

Die verwendeten Design-Patterns zeigen bereits einzelne Abschnitte der geplanten Architektur. In Abbildung 43 wird die Architektur als Layer-Modell dargestellt, dabei werden hellblau gekennzeichnete Layer unter UWP und Xamarin.Forms gesondert implementiert. Die Graublauen müssen dann bis auf die Xamarin.Android und Xamarin.iOS Ebene plattformspezifisch implementiert werden.

Allgemein wurde hier das Layer-Modell weniger strikt gehalten. Es gelingt durch quer eingeschobene Layer (Abstraktion, Models), den DI-Pattern und einer Service-Fassade, die befürchtete stärkere Kopplung zwischen den Layern, gegenüber dem Onion-Modell, zu vermeiden.

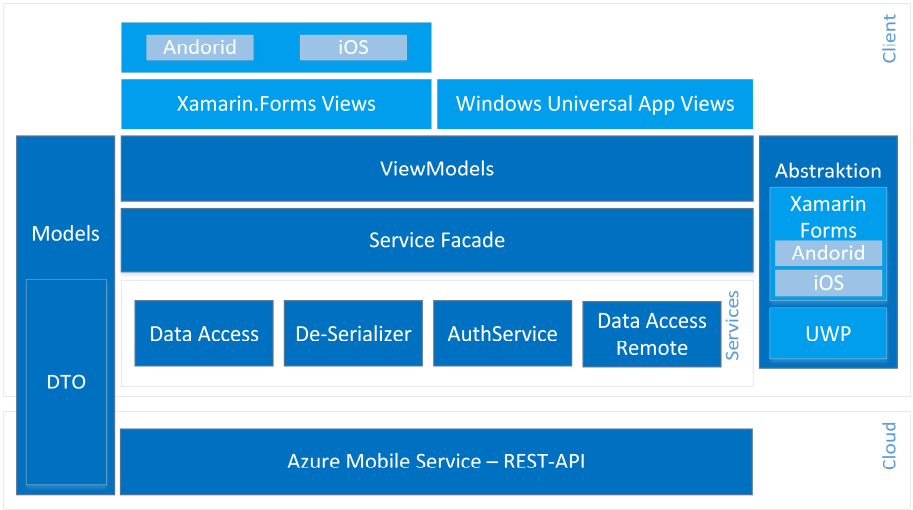

Abbildung 43: Methodik – Gesamtarchitektur

Die Layer Services, Models (DTO), Service-Fassade, und ViewModels wurden bereits mit Hilfe der Design-Patterns abgehandelt.

Azure Mobile Service Layer

Dieser Layer befindet sich in der Azure Cloud. Hier ist die Serverseitige Authentifikation und REST-API zu finden.

Abstraktion

Im quer eingeschobenen Layer Abstraktion befinden sich Abstraktionen wie die Navigation. Die Abstraktionen ermöglichen es in den PCL-Layern auf plattformspezifische Funktionalitäten zuzugreifen.

Views - Xamarin.Forms und UWP

In den jeweiligen Layern befinden sich die Views separiert nach Framework. Über dem Xamarin.Forms Views Layer befindet sich außerdem noch ein Layer mit plattformspezifischen Anpassungen der UI-Elemente, den Custom Renderers.

Kapitel 6

6 Implementierung

In diesem Kapitel werden die wichtigsten Erkenntnisse und Gedankengänge der Implementierungsphase zusammengefasst, sowie relevante Ausschnitte des daraus resultierenden Quellcodes erläutert. Dabei werden die einzelnen Abschnitte des Kapitels in die Layer der zuvor vorgestellten Architektur sowie in weitere logische Abschnitte unterteilt. Innerhalb der Abschnitte wird der Anteil an Shared-Code verdeutlicht und es wird aufgezeigt, welche Code-Sharing-Strategie verwendet wurde. Die Code-Passagen und die gewählten Beispiele beanspruchen dabei keine Funktionstüchtigkeit oder Vollständigkeit. Sie sind gewählt, um die wichtigsten Einblicke des Gesamtbildes herauszunehmen und aufzuzeigen.

6.1 Projektsetup

Als erstes wird eine Solution und pro Layer je ein PCL Projekt erstellt. Dabei ist es sehr wichtig, die Kompatibilität so zu wählen, dass das richtige PCL-Profil erstellt wird. Wird hier etwas falsch eingestellt, kann entweder das Xamarin.Forms-Projekt, das UWP-Projekt oder später einige NuGet Packages nicht hinzufügt werden. Besonders ist darauf zu achten, dass das .NET Framework 4.5 und kein neueres ausgewählt wird. Dies ist wichtig, da Xamarin auf .NET 4.5 aufbaut und noch keine neueren Versionen unterstützt.

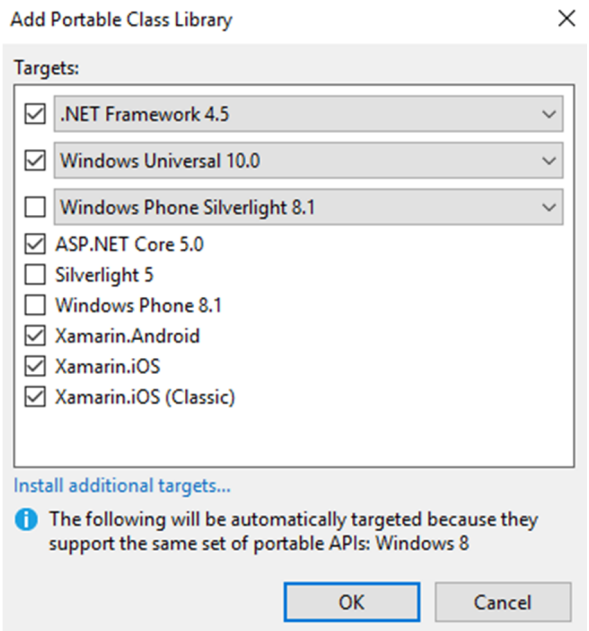

Abbildung 44: Methodik - PCL Setup

Im Anschluss können der Solution ein Xamarin.Forms und UWP Projekt hinzugefügt werden. Diese Projekte liegen, wie im Abschnitt 5.4.2 zu sehen, architektonisch nebeneinander. Daher ist es auch egal, in welcher Reihenfolge man diese anlegt. Außerdem ist der Einfluss auf die Generierung dieser Projekte gering.

Neben der PCL-Projekte und den plattformspezifischen Projekten wird zusätzlich noch ein Shared-Project erstellt. Hier ist es möglich Compilerdirektiven einzusetzen. Dies kann nützlich sein, wenn sich die Implementierungen der Plattformen nur geringfügig unterscheiden. Diese Unterscheidungen sind meist einige Assemblies oder etwas andere Methodenaufrufe.

Das finale Projektsetup ist in Abbildung 45 zu sehen. Eine weitere Unterteilung der Projekte ist bei steigender Komplexität ratsam.

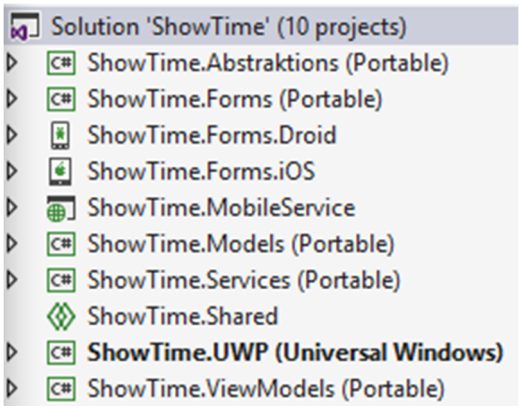

Abbildung 45: Methodik – Projektsetup

6.2 Dependency Injection

Um das DI-Pattern anzuwenden, bietet es sich an, einen der zahlreichen DI-Container zu verwenden. Diese können einfach mit einem NuGet Package hinzugefügt werden. Im Rahmen dieser Arbeit wurde der Autofac-Container verwendet. Dieser wird häufig im Kontext mit Xamarin.Forms eingesetzt. Außerdem ist er einer der am meisten benutzten DI-Container und war einer der besten bei den Tests von AppHarbor[26].

[26] [26]

Um Services, Views, oder ViewModels injizieren zu können müssen diese im DI-Container registriert werden. Dabei wird ein Service, View, oder ViewModel in dem Projekt registriert, in dem sich die Implementation befindet. Das zugehörige Interface kann sich hingegen in einem hierarchisch höheren Projekt befinden. Registriert wird ein Service, View oder ViewModel über Module.

```
namespace ShowTimeUniversal.Servicees.Bootstrapping
{
	public class CoreModule : Module
	{
		protected override void Load(ContainerBuilder builder)
		{
			Builder.RegisterType<ApiServiceClient>()
			  .As<IApiServiceClient>()
			  .SingleInstance();
		}
	}
}

namespace ShowTimeUniversal.UWP.Bootstrapping
{
	public class UWPModule : Module
	{
		protected override void Load(ContainerBuilder builder)
		{
			builder.RegisterType<ViewFactory>()
			  .As<IViewFactory>()
			  .SingleInstance();
		}
	}
}
```

Die Module werden wiederum in dem Container registriert.

```
protected virtual void ConfigureContainer
  (ContainerBuilder builder)
{
	builder.RegisterModule<UWPModule>();
	builder.RegisterModule<CoreModule>();
}
```

Die in den folgenden Abschnitten beschriebenen Dienste (Services) werden mit Hilfe des abstrahierenden Interfaces in den Konstruktor der Service Fassade oder ViewModels wie folgt injiziert:

```
public class SampleViewModel
{
      public SampleViewModel(ISampleService service)
      {
           _service = service;
      }
}
```

6.3 Services

Die Services (Businesslogik und andere Dienste) sind der Kern einer Anwendung und stellen in der Regel einen großen Teil des Aufwands dar. Bei den Services liegen die größten Chancen auf ein maximales Code-Sharing zugrunde, denn hier gibt es viele bekannte NuGet Packages, welche auf dem .NET Framework basieren, auf das Xamarin.Forms und UWP aufbauen. Hat eine Anwendung also eine aufwändige Businesslogik, wird durch Code-Sharing zwischen Xamarin.Forms und UWP viel Redundanz eingespart. Wie dies konkret erfolgt, zeigen die folgenden Abschnitte.

6.3.1 Authentifizierung

Eine häufige Notwendigkeit bei Anwendungen ist die Authentifizierung mit einem REST-Service, welcher die Daten verwaltet und bereitstellt. Um die Authentifizierung immer einheitlich implementieren zu können, stellt Microsoft den Dienst Azure Active Directory zur Verfügung. Mit diesem kann die Authentifizierung der Nutzer für einen eigenen Service oder Microsoft Dienste erfolgen. Da Azure AD für viele Dienste verwendbar ist und anderen Authentifizierungsprozessen sehr ähnelt, wird die Implementierung von Azure AD als Beispiel herangezogen.

Um einen Nutzer bei Azure AD oder anderen Diensten über OAuth oder OAuth 2.0 zu authentifizieren, ist es ratsam ein NuGet Package zu verwenden, welches den Workflow übernimmt. In dem Beispiel wurde das NuGet Package “Microsoft.Identity.Clients.ActiveDirectory” genutzt. Bei der Implementation einer Authentifizierung mit Azure AD zeigt sich direkt, dass nicht alles innerhalb der Services komplett einheitlich gelöst werden kann. Da bei der Authentifizierung häufig ein Web-View aufgerufen werden muss, ist die Implementierung der NuGet Packages plattformabhängig. Durch die plattformabhängigen NuGet Packages werden bei gleicher Implementation verschiedene Assemblies benötigt. Um verschiedene, je nach Plattform unterschiedliche, Assemblies laden zu können, muss der Service in das Shared-Project ausgelagert werden. Da dieses unter allen Plattformen denselben Namespace

besitzt, müssen die Usings (laden der Assemblies) nicht mit Compilerdirektiven versehen werden. Allerdings unterscheiden sich zwischen den Plattformen einige Eigenschaften und Methodenaufrufe, daher müssen diese mit Compilerdirektiven versehen werden. Um den Service später in den DI-Container zu registrieren, muss ein Interface erstellt werden, welches wie folgt aussehen kann:

```
public interface IAuthenticationService
{
	Task<int> GetTokenAsync();
	Task<int> RefreshTokenAsync();
}
```

Die *GetTokenAsync()* Methode fordert nach Eingabe der Nutzerdaten einen Token, zugehörig zu diesen Daten, an. Wenn der Token abgelaufen ist (je nach Anbieter unterschiedlich) fordert die *RefreshTokenAsync()* Methode mit Hilfe eines Refresh Tokens einen neuen Token an. So kann der Nutzer angemeldet bleiben und muss seine Daten nicht jedes Mal neu angeben.

Die wichtigen Eigenschaften, Aufrufe und Compilerdirektiven innerhalb der *GetTokenAsync()* Methode sind folgende:

```
AuthenticationContext context = new AuthenticationContext(CommonAuthority);
#if DROID
PlatformParameters platformParams = new PlatformParameters(
		(Activity)Xamarin.Forms.Forms.Context
);
#elif IOS
PlatformParameters platformParams = new PlatformParameters(
		UIApplication.SharedApplication.KeyWindow.RootViewController
);
#elif WINDOWS_UWP
AuthenticationResult result = await context.AcquireTokenAsync(
		_discoveryResourceId,
		_clientId,
		WebAuthenticationBroker.GetCurrentApplicationCallbackUri()
);
#endif
```

In dem *result* steckt dann unter anderem der Token, RefreshToken und die Dauer, die der Token gültig ist. Die *CommonAuthority* ist der Endpoint, gegen den die Authentifikation abläuft, im Fall Azure AD ist das „https://login.windows.net/Common“. Die *DiscoveryResourceId* ist die API, welche benutzt werden soll, bei Outlook „https://outlook.office.com“. Um eine Anwendung zu identifizieren, gibt es die *_clientId*, welche aus einer Guid besteht „22868fae-abac-4223-9f35-29a599513h40“. Die *_appUrl*

kann so aussehen „https://exampleapp.azure-mobile.net/signin-aad“. Unter UWP ist es ein automatisch generierter Link. Die *platformParams* geben an, in welchem Kontext der WebView geöffnet werden soll, unter UWP entfällt die Angabe.

```
AuthenticationResult result = await context.AcquireTokenByRefreshTokenAsync(
      User.Instance.ApiToken.RefreshToken,
      ClientID
);
```

Die *RefreshTokenAsync()* Methode sieht fast identisch aus, hier lautet der Aufruf allerdings nicht *AcquireTokenAsync(),* sondern *AcquireTokenByRefreshTokenAsync()*. Außerdem entfällt die *_discoveryResourceId* und *_appURL*, hinzu kommt der R*efreshToken*. Auch entfällt hier eine plattformspezifische Implementation, da hier kein WebView aufgerufen werden muss.

6.3.2 Kommunikation mit einer REST-API

Die Kommunikation mit einer REST-API ist bei vielen Anwendungen nötig. Hierfür wird die Kommunikation mit der Office 365 API als Beispiel herangezogen. Diese unterscheidet sich nicht wesentlich von der Kommunikation mit den meisten anderen REST-API’s.

Für die Kommunikation kann der komplette Code zwischen den Plattformen geteilt werden, sodass dieser im PCL-Projekt zu finden ist. Es muss aber darauf geachtet werden, dass ein für alle Plattformen kompatibles NuGet Package verwendet wird. In dem Beispiel wurde das NuGet Package “Microsoft.Net.Http” verwendet. Im Beispiel ist ein GET-Request zu sehen. Ein POST-Request läuft fast identisch ab, zusätzlich wird in der Regel aber noch ein Content mitgegeben.

```
using (var client = new HttpClient() { BaseAddress = new Uri(BASE_URI) })
{
      client.DefaultRequestHeaders.Add(key, auth);
      response = await client.GetAsync(endpoint);
}
```

Es wird ein *HttpClient* mit einer Basisadresse erzeugt, welchem ein Header mitgegeben wird und dann über den *HttpClient* ein Aufruf an die API gesendet, mit dem Endpoint innerhalb der API. Der *key* ist wichtig, damit die API den jeweiligen *auth* (Authentifikationskontext) findet. Dabei wird in der Regel der Schlüssel „Authorization“ verwendet und als Kontext wird meist „Bearer {Token}“ angegeben.

6.3.3 Serialisieren und Deserialisieren von Daten

Auch bei der De- und Serialisierung von Daten kann der komplette Code geteilt werden, sodass dieser im PCL-Projekt zu finden ist. Dies wurde mit XML und JSON getestet. In dem unten folgenden Beispiel wird das De- und Serialisieren von JSON gezeigt. Dies erfolgt mit dem NuGet Package "Newtonsoft.Json".

```
var sampleDTOList = JsonConvert.DeserializeObject<IEnumerable<SampleDTO>>(jsonString);
string jsonString = JsonConvert.SerializeObject(sampleDTOList);
```

In dem Beispiel wird ein JSON String in eine Liste von SampleDTO's deserialisiert und diese wieder in ein JSON String serialisiert. Außerdem wird die Nützlichkeit des DTO-Patterns bei der De- und Serialisierung ersichtlich. Wurde das Objekt einmal erstellt, vereinfacht es die De- und Serialisierung enorm. Ohne dieses müssten das JSON und seine Elemente einzeln angesprochen werden, wodurch die Logik umfangreicher, weniger erweiterbar und schwerer zu pflegen wäre.

6.3.4 Lokale Datenhaltung mit SQLite

Zugriffe auf eine SQLite Datenbank lassen sich fast komplett vereinheitlichen. Hier muss lediglich eine Eigenschaft plattformspezifisch gesetzt werden. Dabei wird in einem PCL-Projekt eine abstrakte SQL-Connection-Factory implementiert, welche dann plattformspezifisch vererbt wird und dort die Eigenschaft gesetzt wird. Wie dies konkret geht, zeigt das Beispiel. Es wurden hierbei die NuGet Packages SQLite.Net-PCL, SQLiteNetExtensions und PCLStorage verwendet.

Zunächst einmal muss eine Verbindung zu einer SQLite Datenbank aufgebaut werden. Hierfür wird, in einem PCL-Projekt, eine Factory implementiert und für diese Factory zunächst ein Interface:

```
public interface ISqliteConnectionFactory
{
	ISQLitePlatform CurrentPlattform { get; }

	string GetPlattformDatabasePath(string databaseName);

	Task<bool> CheckDatabaseExist(string databaseName);

	SQLiteConnection GetConnection(string databaseName);
}
```

Die Factory ist für das einfache Verständnis möglichst simpel gehalten. Möglich wäre noch eine GetConnectionWithLock() Methode oder eine GetAsyncConnection() Methode, wie beim Plugin von MvvmCross[27] zu finden, von dem dieses Codebeispiel adaptiert wurde.

Die plattformübergreifende Factory sieht dann wie folgt aus:

```
public abstract class SqliteConnectionFactoryBase :
  ISqliteConnectionFactory
{
      private IFolder Folder
      {
        get
        {
           return FileSystem.Current.LocalStorage;
        }
      }
      //Plattformabhängige abstrakte Eigenschaft
      public abstract ISQLitePlatform CurrentPlattform
      {
           get;
      }
      //Mit Hilfe von PCLStorage wird ein valider Pfad ermittelt
      public string GetPlattformDatabasePath(string databaseName)
      {
           return PortablePath.Combine(Folder.Path, databaseName);
      }
      //Prüft ob die Datenbank bereits erzeugt wurde
      public async Task<bool> CheckDatabaseExist(string databaseName)
      {
           return await Folder.CheckExistsAsync(databaseName) ==
             ExistenceCheckResult.FileExists ?
             true : false;
      }
      //Stellt eine Verbindung zur SQLite Datenbank her
      public SQLiteConnection GetConnection(string databaseName)
      {
           return GetConnection(new SqLiteConfig(databaseName));
      }
}
```

[27] [27]

Mit Hilfe der Factory ist es möglich eine einfache Verbindung aufzubauen. Das NuGet Package PCLStorage kommt zum Einsatz um einen allgemeingültigen Dateipfad zur Datenbank zu ermitteln. Außerdem ist alles bis auf eine Eigenschaft bereits implementiert, einzig die *ISQLitePlatform* ist abstrakt. Um die jeweilige Plattformeigenschaft zu erhalten, wird die Factory pro Plattform geerbt und dort einzig die *ISQLitePlatform* Eigenschaft gesetzt.

UWP-Projekt:

```
public class UWPSqliteConnectionFactory : SqliteConnectionFactoryBase
{
	public override ISQLitePlatform CurrentPlattform => new SQLitePlatformWinRT();
}
```

Android-Projekt:

```
public class AndroidSqliteConnectionFactory : SqliteConnectionFactoryBase
{
	public override ISQLitePlatform CurrentPlattform => new SQLitePlatformAndroid ();
}
```

iOS-Projekt:

```
public class IOSSqliteConnectionFactory : SqliteConnectionFactoryBase
{
	public override ISQLitePlatform CurrentPlattform => new SQLitePlatformIOS();
}
```

Die eigentlichen schreibenden und lesenden SQLite Zugriffe können plattformübergreifend realisiert werden, befinden sich also in einem PCL-Projekt. Hier gibt es durch die NuGet Packages verschiedenste Zugriffsmöglichkeiten. Es können einzelne Objekte, Objekt Listen oder Objekt Listen und deren Kinder abgerufen, hinzugefügt oder verändert werden. Die Zugriffe können über LINQ oder normale SQL-Abfragen erfolgen. Um eine SQLite Datenbank und deren Tabellen erzeugen zu können, müssen zunächst die Models angepasst werden. Hier können alle möglichen Beziehungen dargestellt werden. Anhand einer Serie und deren Staffeln kann dies dann beispielsweise folgendermaßen aussehen:

```
public class Series
{
	[PrimaryKey, AutoIncrement]
	public int SeriesID { get; set; }

	[ForeignKey(typeof(Provider))]
	public int UserID { get; set; }

	[ManyToOne]
	public Provider User { get; set; }

	public string SeriesName { get; set; }

	[OneToMany]
	public List<Season> Seasons { get; set; }
}

public class Season
{
	[PrimaryKey, AutoIncrement]
	public int SeasonID { get; set; }

	[ForeignKey(typeof(Series))]
	public int SeriesID { get; set; }

	[ManyToOne]
	public Series Series { get; set; }
}
```

Der folgende Beispielcode zeigt einige mögliche Zugriffsoperationen:

```
using (var c = _sqliteConnectionFactory.
  GetConnection(databaseName))
{
       //Neue Tabellen erstellen
       c.CreateTable<Series>();
       c.CreateTable<Season>();

       //Füge eine Liste von Serien und deren Staffeln hinzu
       c.InsertAllWithChildren(seriesList);

       //Finde eine Serie oder eine Liste von Serien mit LINQ
       var seriesUpdate = c.Find<Series>(
         s => s.SeriesName == "SampleName"
       );

       //Update eine einzelne Serie
       seriesUpdate.SeriesName = "NewSampleName";
       c.Update(seriesUpdate);

       //Selektiere alle Serien
       var fullSeriesList = c.Query<Series>(
         "SELECT * FROM Series"
       );

       //Lösche eine Serie
       c.Delete(seriesUpdate);
}
```

6.4 ViewModels

ViewModels steuern und bauen einen View auf, außerdem können sie auch zur Navigation zwischen den Views genutzt werden. Durch ViewModels ist im optimalen Fall keinerlei verbliebene Logik im View, der View stellt einzig die Elemente welche vom ViewModel über Bindings gefüllt und gesteuert werden. Können auch die ViewModels einmalig implementiert werden, spart dies einen enormen Aufwand.

In Xamarin.Forms und UWA ist das Binding in Views vorgesehen und beide basieren auf dem *INotifyPropertyChanged* Interface, wodurch es generell möglich ist, auch die ViewModels zu teilen. Hierbei gab es dennoch einige Herausforderungen und Probleme während der Implementationsphase, welche im Folgenden beschrieben werden.

6.4.1 Command

Um auf Events, wie bspw. Button-Tapped, in einem ViewModel reagieren zu können, muss ein Command in einem View gebunden werden. Das Command benachrichtigt dann das ViewModel sobald ein Event ausgelöst wurde, woraufhin das ViewModel agieren kann.

Xamarin.Forms und UWA benutzen für Commands das gleiche Interface ICommand. Daher kann ein Command für beide Plattformen einfach implementiert werden.

```
public class UniversalCommand : ICommand
{
	public event EventHandler CanExecuteChanged;

	private Action _action;

	public Command(Action action)
	{
		this._action = action;
	}

	public bool CanExecute(object parameter)
	{
		return true;
	}

	public void Execute(object parameter)
	{
		this._action();
	}
}
```

Commands werden von Controls angeboten, die Aktionen unabhängig vom tatsächlichen Event (Button.Click, Menu.Xy, etc.) auslösen sollen. Gelegentlich muss auch eine Aktion ausgelöst werden, für die das betroffene Control kein Command anbietet (z.B. Tap, Selected). In diesen Fällen ist es aber möglich, ein Event in ein Command umzuwandeln, dies muss aber plattformspezifisch implementiert werden.

Ein möglicher Ansatz unter UWP, am Tapped-Event eines StackPanel:

```
public static class StackPanelTappedCommand
{
       public static readonly DependencyProperty CommandProperty =
         DependencyProperty.RegisterAttached(
              "Command", typeof(ICommand),
              typeof(StackPanelTappedCommand),
              new PropertyMetadata(null, OnCommandPropertyChanged)
         );

       public static void SetCommand(DependencyObject d, ICommand value)
       {
            d.SetValue(CommandProperty, value);
       }

       public static ICommand GetCommand(DependencyObject d)
       {
            return (ICommand)d.GetValue(CommandProperty);
       }

       private static void OnCommandPropertyChanged(
         DependencyObject d,
         DependencyPropertyChangedEventArgs e)
       {
            var control = d as StackPanel;
            if (control != null)
                 control.Tapped += OnItemClick;
       }

       private static void OnItemClick(object sender, TappedRoutedEventArgs e)
       {
            var control = sender as StackPanel;
            var command = GetCommand(control);
            if (command != null && command.CanExecute(e))
            {
                 command.Execute(e);
            }
       }
}
```

Als erstes wird ein neues Dependency Property (*CommandProperty*) deklariert und registriert. Die angegebenen Parameter, von links nach rechts aufgeführt, sind ein Key, mit dem das Dependency Property angesprochen werden kann, der Typ der Eigenschaft und der Typ der besitzenden Klasse sowie die *PropertyMetadaten*. Letztgenannten wird die Callback-Methode mitgegeben, welche wiederum aufgerufen wird, wenn sich das Dependency Property ändert. In der Callback-Methode *OnCommandPropertyChanged()* wird auf das Tapped-Event des Dependency Object (StackPanel) der *OnItemClick()* Listener gesetzt. Beim Click bzw. Tap auf das StackPanel wird der Listener ausgelöst und das im StackPanel registrierte Command abgerufen und ausgeführt. Das Command wird im unteren Codeabschnitt im StackPanel gebunden und im ViewModel gesetzt.

View

```
<Page xmlns:common="using:SampleApp.UWP.Common">
  <StackPanel
    common:StackPanelTappedCommand.Command="{Binding StackPanelTappedCommand}">
  </StackPanel>
</Page>
```

ViewModel

```
public ICommand StackPanelTappedCommand { get; set; }
StackPanelTappedCommand = new UniversalCommand(ShowInformation);
```

Unter Xamarin.Forms wurde eine Community Lösung[28] eingesetzt, welche es ermöglicht, Events allgemeingültig in ein Command umzuwandeln. Hierfür müssen lediglich die Klassen in das eigene Projekt kopiert werden und diese dann in XAML verwendet werden. (Beispiel siehe Quelle)

```
<ContentPage xmlns:common="using:SampleApp.Forms.Common">
  <ListView ItemsSource="{Binding People}">
    <ListView.Behaviors>
      <common:EventToCommandBehavior Command="{Binding SayHelloCommand}"
      EventName="ItemTapped" EventArgsConverter="{StaticResource ItemTappedConverter}" />
    </ListView.Behaviors>
    <ListView.ItemTemplate>
      <DataTemplate>
        <TextCell Text="{Binding Name}"/>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</ContentPage>
```

[28] [28]

Neben der Möglichkeit ein Event zu einem Command umzuwandeln, gibt es für Elemente ohne passendes Event die sogenannten Gesture Recognizer, welche es ermöglichen, auf jedes XAML Element ein Command zu setzen.

```
<ContentPage>
  <StackLayout>
    <StackLayout.GestureRecognizers>
      <TapGestureRecognizer Command="{Binding StackPanelTappedCommand}"/>
    </StackLayout.GestureRecognizers>
  </StackLayout>
</ContentPage>
```

6.4.2 DataBinding

Mit Hilfe des DataBindings werden Daten und verschiedenste Eigenschaften in die Views injiziert. So kann das ViewModel unterschiedlichsten Einfluss auf die Gestaltung nehmen, wie Sichtbarkeit, Farben, Abstände und mehr. Außerdem werden so die Views mit wichtigen Informationen angereichert.

Das Binden der Eigenschaften ist in UWP- und Xamarin.Forms-Views in der Regel plattformübergreifend möglich. Hier gibt es allerdings einige Ausnahmen, wie bspw. die Eigenschaft Visibility. Innerhalb eines UWP Views ist die Eigenschaft Visbility als Enum deklariert, bei einem Xamarin.Forms View hingegen als Boolean. Um die Eigenschaft nun einheitlich an das ViewModel binden zu können, muss ein ValueConverter für die UWP oder Xamarin.Forms Eigenschaft geschrieben werden. Wird ein ValueConverter für beide Plattformen benötigt, so ist es möglich, diesen in ein Shared-Projekt auszulagern, in dem dann mit Compilerdirektiven auf die geringen Unterschiede eingegangen werden kann. Im Beispielcode ist ein ValueConverter für den UWP View zu sehen, dieser unterscheidet sich nur minimal von dem des Xamarin.Forms Converters.

```
public class IsVisibleConverter : IValueConverter
{
      public object Convert(object value, Type targetType,
        object parameter, string language)
      {
           if ((bool)value)
                return 0;
           else
                return 1;
      }
      public object ConvertBack(...) { throw new NotImplementedException(); }
}
```

6.5 Userinterface

Wie zuvor erwähnt, wäre es zu zeitaufwändig und komplex, die UI-Elemente zu abstrahieren. Daher und wegen der Vorteile der UWP-UI wird für Xamarin.Forms und UWP je ein View gesondert implementiert. Da beide mit XAML aufgebaut werden und sich auch allgemein sehr ähneln, kann hier aber viel übertragen werden. Ist ein View bei UWP oder Xamarin.Forms fertig aufgebaut, kann der Aufbau fast identisch übernommen werden. Lediglich die UI-Elemente müssen ausgetauscht werden, häufig auch nur namentlich. Dies funktioniert aber nur mit einem Formfaktor so gut. Sobald weitere hinzukommen, wird es schwieriger, was an der unterschiedlichen Art und Weise liegt, wie mit Formfaktoren umgegangen wird und wie gut auf die einzelnen eingegangen werden kann. Ein gutes Beispiel für die Ähnlichkeit ist der Aufbau eines ListViews.

UWP

```
<ListView Grid.Row="2"
ItemsSource="{Binding TimeEntries}" Margin="0,10,0,0">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto" />
          <ColumnDefinition Width="*" />
          <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto"/>
          <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" Grid.Column="0" Text="{Binding WorkDate, Mode=TwoWay}"
        HorizontalAlignment="Left"/>
        <TextBlock Grid.Row="1" Grid.Column="0" Text="{Binding WorkStartEndTime}"
        HorizontalAlignment="Left"/>
        <TextBlock Grid.Row="0" Grid.RowSpan="2" Grid.Column="1"
        Text="{Binding Activity }" Margin="10,0,10,0"  VerticalAlignment="Center" />
        <TextBlock Grid.Row="0" Grid.RowSpan="2" Grid.Column="2"
        Text="{Binding WorkDuration}" HorizontalAlignment="Right"
        VerticalAlignment="Center" Margin="0,0,10,0"/></TextBlock>
      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Xamarin.Forms

```
<ListView Grid.Row="1" ItemsSource="{Binding TimeEntries}"
IsPullToRefreshEnabled="True" RefreshCommand="{Binding RefreshCommand}"
IsRefreshing="{Binding ProgressRingIsActiv, Mode=OneWay}">
  <ListView.ItemTemplate>
    <DataTemplate>
      <ViewCell>
        <Grid Padding="10,0,10,0">
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
          </Grid.RowDefinitions>
          <Label Grid.Row="0" Grid.Column="0" Text="{Binding WorkDate}"/>
          <Label Grid.Row="1" Grid.Column="0" Text="{Binding WorkStartEndTime}"/>
          <Label Grid.Row="0" Grid.RowSpan="2 Grid.Column="1" VerticalOptions="Center"
          Text="{Binding Activity}" LineBreakMode="TailTruncation"/>
          <Label Grid.Row="0" Grid.RowSpan="2" Grid.Column="2"
          Text="{Binding WorkDuration}" VerticalOptions="Center"/>
        </Grid>
      </ViewCell>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Die große Ähnlichkeit ist offensichtlich, doch ist dies nicht immer so. Außerdem zeigen einige Elemente ein anderes Verhalten bei vermeintlich gleichen Eigenschaften. Auch hat Xamarin.Forms deutlich weniger Elemente als UWP.

6.5.1 Formfaktoren

Xamarin.Forms und UWP gehen unterschiedlich mit der Anpassung von Formfaktoren um. Innerhalb der UWP ist es möglich, wesentlich präziser auf unterschiedlichste Formfaktoren einzugehen.

UWP ermöglicht es mit sogenannter Adaptive Trigger eine UI je nach Bildschirmgröße anzupassen. Mit ihnen können alle möglichen Manipulationen an Elementen ausgeführt werden. Es können also Elemente ab einer angegebenen Bildschirmgröße bspw. anders angeordnet oder aus- und eingeblendet werden. Dabei können Adaptive Trigger im XAML definiert werden, einfacher ist es jedoch über die IDE Blend. Über Blend können die Anpassungen ohne Code erfolgen.

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="VisualStateGroup">
    <VisualState x:Name="VisualStateMin800">
      <VisualState.Setters>
        <Setter Target="AppBar.(UIElement.Visibility)" Value="Collapsed"/>
        <Setter Target="Button.(UIElement.Visibility)" Value="Visible"/>
        <Setter Target="Title.(UIElement.Visibility)" Value="Visible"/>
      </VisualState.Setters>
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="800"/>
      </VisualState.StateTriggers>
    </VisualState>
    <VisualState x:Name="VisualStateMin0">
      <VisualState.Setters>
        <Setter Target="AppBar.(UIElement.Visibility)" Value="Visible"/>
        <Setter Target="Button.(UIElement.Visibility)" Value="Collapsed"/>
        <Setter Target="Title.(UIElement.Visibility)" Value="Collapsed"/>
      </VisualState.Setters>
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="1"/>
      </VisualState.StateTriggers>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

In diesem Beispiel wird bei großen Bildschirmen statt eines AppBar-Buttons ein normaler Button angezeigt sowie der Titel ausgeblendet.

Xamarin.Forms hingegen ermöglicht nur ein reagieren auf den Gerätetyp. Es ist möglich über Idiom abzufragen, ob das Gerät ein Smartphone oder Tablet (Xamarin.Forms UWP auch Desktop) ist und dann dementsprechend das UI anzupassen.

```
<ContentPage.Title.IsVisible>
  <OnIdiom>
    <OnIdiom.Phone>False</OnIdiom.Phone>
    <OnIdiom.Tablet>True</OnIdiom.Tablet>
  </OnIdiom>
</ContentPage.Title.IsVisible>
<Button>
  <Button.IsVisible>
    <OnIdiom>
      <OnIdiom.Phone>False</OnIdiom.Phone>
      <OnIdiom.Tablet>True</OnIdiom.Tablet>
    </OnIdiom>
  </Button.IsVisible>
</Button>
```

6.5.2 Custom Renderer

Wie bereits gelesen, ist Xamarin.Forms eine Abstraktion der nativen UI-Elemente von Android, iOS und Windows Phone. Die Abstraktion umfasst aber lediglich die wichtigsten und meist genutzten Elemente und von diesen auch nur die nötigsten Eigenschaften. Daher ist es hin und wieder notwendig, besondere UI-Elemente oder Eigenschaften, welche in Xamarin.Forms nicht vorhanden sind, plattformspezifisch anzupassen. Dies kann über die sogenannten Custom Renderer erreicht werden.

In einem Custom Renderer gibt es immer mindestens eine *OnElementChanged()* Methode. In dieser Methode kann dann das jeweilige native Android oder iOS Element über die Eigenschaft *Control* angesprochen werden. Ein einfaches Beispiel hierfür ist der *ListViewRenderer*.

```
public class NormalListViewRenderer : ListViewRenderer
{
      protected override void OnElementChanged(
        ElementChangedEventArgs <Xamarin.Forms.ListView> e)
      {
           base.OnElementChanged(e);

           var listView = Control as Android.Widget.ListView;
           listView.Divider = new ColorDrawable(
             new Android.Graphics.Color(255, 255, 255)
           );
           listView.DividerHeight = 5;
           listView.SetHeaderDividersEnabled(false);
      }
}
```

Hier werden die Trennlinien des ListViews unter Android verändert. Sie werden weiß eingefärbt, breiter gemacht und die der Kopfzeile weggelassen.

Neben der *OnElementChanged()* Methode hat ein Renderer meist noch unterschiedlichste andere Eigenschaften und Methoden, je nach Bedarf.

6.6 Navigationskonzept

Der View und das ViewModel wurden bereits beschrieben, nun muss noch zwischen den einzelnen Ansichten navigiert werden. Dafür gibt es bei Verwendung des MVVM-Patterns zwei mögliche Varianten, die View-First und die ViewModel-First Navigation. Wie die Namen schon andeuten, unterscheiden sich die beiden darin, dass bei der View-First Navigation von einem View ein anderer View aufgerufen wird, auf den navigiert werden soll. Bei der ViewModel-First Navigation wird hingegen vom ViewModel ein anderes ViewModel aufgerufen, durch welches ein View Instanziiert wird. Beide haben folgende Vorteile gegenüber der anderen Variante:

View-First

- im View klar ersichtlich welches ViewModel benutzt wird
- ein ViewModel kann einfacher durch mehrere Views genutzt werden

ViewModel-First

- der View muss das ViewModel nicht kennen und nicht direkt mit diesem kommunizieren
- bessere Testbarkeit
- View und ViewModel sind noch stärker entkoppelt
- bessere Übergabe von Parametern

View-First und ViewModel-First haben beide ihre berechtigten Verwendungszwecke, aber in diesem Kontext ist die ViewModel-First Navigation überlegen. Die Überlegenheit entsteht durch mehrere Faktoren. Bei der Cross-Plattform-Entwicklung müssen View und ViewModel möglichst stark voneinander entkoppelt sein. Hinzu kommt, dass im Vorfeld festgelegt wurde, dass die ViewModels zwischen den Plattformen geteilt werden sollen, die Views aber plattformspezifisch implementiert werden. Würde also eine View-First Navigation verwendet werden, müsste der Navigationsworkflow mehrfach implementiert werden, bei einer ViewModel-First Navigation lediglich einmal.

Um eine ViewModel-First Navigation umzusetzen, gibt es verschiedene Frameworks. Damit Probleme im Gesamtkontext besser aufgezeigt werden können, wird im Laufe dieser Arbeit eine unabhängige Implementation der ViewModel-Navigation vorgenommen. Außerdem müssten die vorhandenen Frameworks ohnehin erweitert werden.

6.6.1 Rudimentäre Navigation

Eine ViewModel-First Navigation stellt beispielsweise Jonathan Yates in einem Xamarin-Tutorial[29] vor. Diese kann in adaptierter Form auch in diesem Kontext verwendet werden.

Zunächst einmal muss ein Interface implementiert werden, welches die wichtigsten Navigationsfunktionen beinhaltet.

[29] [25]

```
public interface IUniversalNavigator
{
      Task<IViewModel> PopAsync();
      Task<IViewModel> PopModalAsync();
      Task PopToRootAsync();

      Task<TViewModel> PushAsync<TViewModel>(Action<TViewModel> setStateAction = null)
           where TViewModel : class, IViewModel;

      Task<TViewModel> PushModalAsync<TViewModel>(
        Action<TViewModel> setStateAction = null)
           where TViewModel : class, IViewModel;
}
```

Dieses Interface muss dann an einen für Xamarin.Forms und UWP geschriebenen Navigation-Service vererbt werden. Der Navigation Service muss gesondert für Xamarin.Forms und UWP implementiert werden, unterscheidet sich aber nicht in der Logik.

Navigation-Service-PopAsync und –PushAsync-Xamarin.Forms:

```
public async Task<IViewModel> PopAsync()
{
      var page = await Navigation.PopAsync();
      return page.BindingContext as IViewModel;
}

public async Task<TViewModel> PushAsync<TViewModel>()
  where TViewModel : class, IViewModel
{
      TViewModel viewModel;
      var page = _viewFactory.Resolve<TViewModel>(out viewModel);
      await _navigation.PushAsync(page);
      page.BindingContext = viewModel;
      return viewModel;
}
```

Die wichtigsten Funktionen sind dabei im oberen Codeabschnitt zu sehen. *PopAsync()* navigiert zurück, indem der oberste View aus dem Navigationsstapel entfernt wird. *PushAsnyc()* navigiert zu einem neuen View. Hierfür wird mit Hilfe des ViewModel-Type und der ViewFactory das ViewModel und der zugewiesene View instanziiert. Anschließend wird mit den Navigationsfunktionen von Xamarin bzw. UWP darauf navigiert und schlussendlich das ViewModel als *BindingContext* (unter UWP *DataContext* genannt) gesetzt. Hier kann auch der Einsatz des DI-Patterns beobachtet werden.

```
        var view = _viewFactory.Resolve<TViewModel>(out viewModel);
```

In dieser Zeile wird das zuvor registrierte ViewModel mit Hilfe des DI-Patterns instanziiert.

Die nötige ViewFactory unterscheidet sich ausschließlich in den verwendeten Assemblies. Da sich die Implementierung einzig in Form der Assemblies unterscheidet, kann die ViewFactory in das Shared-Projekt ausgelagert werden. Dort werden dann mit Compilerdirektiven die für die Plattform nötigen Assemblies geladen.

```
//Assembliy Compilerdirektiven über dem Namespace
#if WINDOWS_UWP
using Windows.UI.Xaml.Controls;
#elif FORMS
using Xamarin.Forms;
#endif

//Register und Resolve Methoden
public void RegisterPage<TViewModel, TView>()
        where TViewModel : class, IViewModel
        where TView : Page
{
        _map[typeof(TViewModel)] = typeof(TView);
}

public Page ResolvePage<TViewModel>
  (out TViewModel viewModel)
        where TViewModel : class, IViewModel
{
        var pageType = _map[typeof(TViewModel)];
        var page = _componentContext.Resolve(pageType) as Page;
        viewModel = _componentContext.Resolve<TViewModel>();
        return page;
}
```

Die ViewFactory hält ein Dictionary (*_map*), welches einen Typ als Key und Value besitzt. Bei der Registrierung eines ViewModels und Views wird der Typ des ViewModels als Key und der des Views als Value gesetzt. Wird mit Hilfe eines ViewModel-Typs ein View angefragt, dann werden der View und das ViewModel durch Autofac (DI-Pattern) instanziiert.

Die ViewModels und Views können jetzt in der ViewFactory registriert werden.

```
protected void RegisterViews(IViewFactory viewFactory)
{
      viewFactory.Register<WorkTasksListViewModel, WorkTaskListView>();
}
```

6.6.2 Navigation mit verschiedensten Formfaktoren

Im Abschnitt 5.3.2 - Verschiedene Gerätefamilien - wurde bereits darauf hingewiesen, dass durch unterschiedliche Formfaktoren nicht nur designspezifische Herausforderungen zu lösen sind, sondern auch Navigationsprobleme auftreten können. Dies ist der Fall, in dem schon beschriebenen Szenario, wenn eine Seite zu mehreren wird bzw. mehrere zu einer.

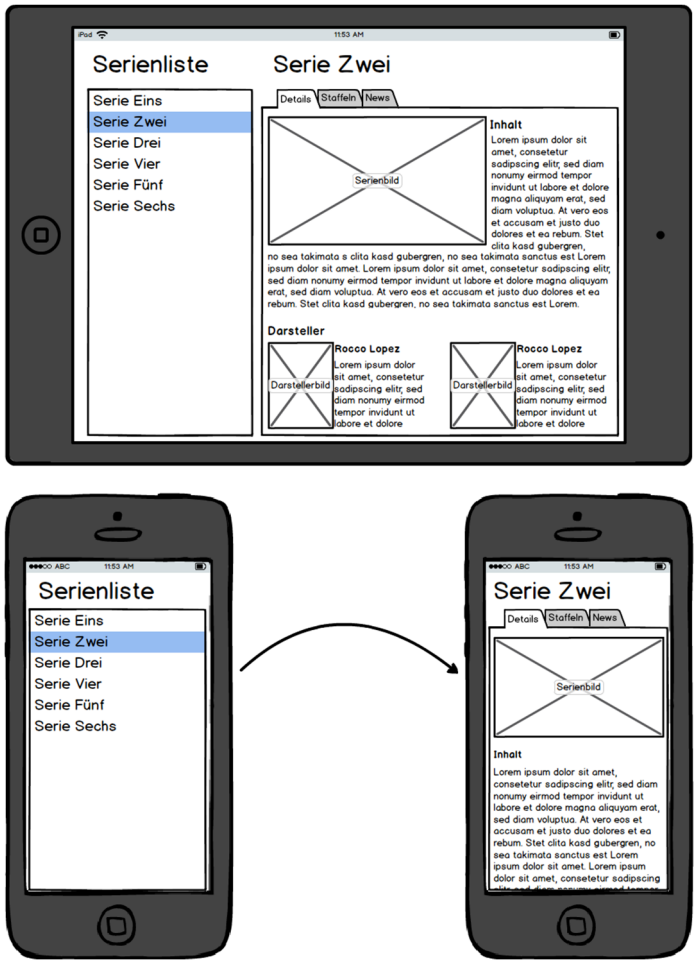

Abbildung 46: Ändern des Navigtionsablaufs

Am Serien-Beispiel würde das bedeuten, dass mit einem Smartphone von der Serienliste beim Klick auf eine Serie auf die Seriendetails gewechselt wird. Auf einem Tablet hingegen ist genug Platz um diese nebeneinander darzustellen. Daher wird dort nicht auf eine neue Seite gewechselt, sondern in dem Seitenabschnitt der Seriendetails werden neue Informationen bereitgestellt.

Um diesem Problem gerecht zu werden, muss zunächst der View angepasst werden. Zum einem werden die einzelnen Seiten in Controls ausgelagert. Also bei einer Serien-App würde das bedeuten, dass die Seiten Serien-Liste und Serien-Detail je ein Control sind. Dann werden zwei Seiten erstellt, eine Seite Serien-Liste und eine Serien-Detail. Diese Seiten sind aber nur Hüllen, in denen auf die beiden Controls verwiesen wird. Sie unterscheiden sich lediglich dadurch, auf welchem der Fokus liegt. Dies ist nötig, da der Nutzer auf einem Desktop die Fenstergröße dynamisch ändern kann. Es müssen also beide Controls verfügbar sein, um sie bei größer oder kleiner werdendem Fenster ein- oder auszublenden. Außerdem müssen einzelne Seiten vorhanden sein, die in dem Navigationstack liegen. Diese Seitenhülle sieht dann unter UWP bspw. so aus:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="VisualStateGroup">
      <VisualState x:Name="VisualStateMin800">
        <VisualState.StateTriggers>
          <AdaptiveTrigger MinWindowWidth="800"/>
        </VisualState.StateTriggers>
      </VisualState>
      <VisualState x:Name="VisualStateMin0">
        <VisualState.Setters>
          <Setter Target="SeriesListControl.(UIElement.Visibility)" Value="Collapsed"/>
          <Setter Target="SeriesDetailControl.(Grid.Column)" Value="0"/>
          <Setter Target="SeriesDetailControl.(Grid.ColumnSpan)" Value="2"/>
        </VisualState.Setters>
        <VisualState.StateTriggers>
          <AdaptiveTrigger MinWindowWidth="1"/>
        </VisualState.StateTriggers>
      </VisualState>
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>
  <Grid.RowDefinitions>
    <RowDefinition Height="*"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <controls:SerienListControl x:Name="SeriesListControl" Margin="0,0,40,0"/>
  <controls:SerienDetailControl x:Name="SeriesDetailControl"
  Grid.Column="1" VerticalAlignment="Stretch"/>
</Grid>
```

Durch die Änderung am View passen sich das Design und die dargestellten Informationen je nach Formfaktor an. Allerdings würde der View mit dem bisherigen Navigation-Service keine Daten erhalten. Daher muss der Service angepasst werden.

Damit der View wieder alle nötigen Daten erhält, muss zunächst die ViewFactory angepasst werden. Hier wird zu jedem ViewModel ein Key registriert und dann zu jedem View neben dem ViewModel auch die ViewModel Keys. So kann weiterhin auf ein ViewModel navigiert werden, aber dennoch im View der BindingContext für die einzelnen Controls gesetzt werden. Der ViewModel Key ist dabei gleich dem *x:Name*.

```
public void RegisterPage<TViewModel, TView>(List<string> regionList)
       where TViewModel : class, IViewModel
       where TView : Page
{
       _mapMultiPageUserControls.Add(typeof(TView), regionList);
       _map[typeof(TViewModel)] = typeof(TView);
}

public List<string> ResolvePageRegions(Type pageType)
{
       List<string> regionList = new List<string>();
       if(_mapMultiPageUserControls.ContainsKey(pageType))
       {
            regionList = _mapMultiPageUserControls[pageType];
       }
       return regionList;
}
```

Der Quellcodeausschnitt zeigt die geänderte Register-Methode und eine neue Methode, welche die für den View nötigen ViewModel Keys zurückliefert. Zusätzlich werden in der ViewFactory die ViewModels in einem Dictionary gehalten.

Die Views und ViewModels müssen dadurch auch etwas anders registriert werden.

```
protected void RegisterViews(IViewFactory viewFactory)
{
       viewFactory.RegisterPage<WorkTasksListViewModel, WorkTaskListView>(
         new List<string>()
         { "WorkTasksListViewUserControl", "TimeEntriesListViewUserControl" }
       );
}
```

Neben der ViewFactory muss auch der Navigationserivce angepasst werden.

Navigation-Service-PopAsync-UWP:

```
public async Task<IViewModel> PopAsync()
{
       _frame.GoBack();
       var pageRegions = _viewFactory.ResolvePageRegions(page.GetType());
       SetViewModelsToPageRegions(pageRegions);
       return page.DataContext as IViewModel;
}

private void SetViewModelsToPageRegions (List<string> pageRegions)
{
       var page = (Page)_frame.Content;
       var mainRegion = pageRegions.ElementAt(0);
       var regionsWithoutMain = pageRegions.Where(r => r != mainRegion);

       ((UserControl)page.FindName(mainRegion)).DataContext =
         _viewFactory.ViewModels[mainRegion];

       foreach(string subRegion in regionsWithoutMain)
       {
            if(_viewModelCache.ViewModels.ContainsKey(subRegion))
            {
                 ((UserControl)page.FindName(subRegion)).DataContext =
                   _viewFactory.ViewModels[subRegion];
            }
       }
}
```

In der *PopAsync()* Methode wird wie zuvor der oberste View aus dem Navigationsstapel entfernt. Im Gegensatz zur vorherigen Implementation werden jetzt aber zusätzlich die Keys der Seitenbereiche abgefragt. Diese Keys sind identisch mit denen der ViewModels. Mit den Keys wird der *DataContext* der Seitenbereiche auf die passenden ViewModels gesetzt.

Navigation-Service-PushAsync-UWP:

```
public async Task<TViewModel> PushAsync<TViewModel>(
  Action<TViewModel> setStateAction = null)
  where TViewModel : class, IViewModel
{
      TViewModel viewModel;
      var newPage = _viewFactory.ResolvePage<TViewModel>(
          out viewModel,
          setStateAction
        );

      var pageRegions = _viewFactory.ResolvePageRegions(newPage.GetType());
      _viewFactory.ViewModels[regions.ElementAt(0)] = viewModel;

      if (!newPage.GetType().Equals(_frame.Content.GetType()))
      {
           _frame.Navigate(pageType);
           SetViewModelsToPageRegions(pageRegions);
      }
      return viewModel;
}
```

Innerhalb der *PushAsync()* Methode wird ebenfalls der *DataContext* der Seitenbereiche auf die passenden ViewModels gesetzt. Zusätzlich wird das ViewModel des Views, auf welchen gewechselt wird, registriert und in der ViewFactory gehalten. Außerdem muss mit *if(!pageType.Equals(_frame.Content.GetType()))* geprüft werden, ob auf einen neuen View gewechselt werden muss oder nur das ViewModel in der Main-Region geändert wurde. Dies ist im Serienbeispiel auf dem Tablet der Fall. Dort wird die Serienliste in der linken Sub-Region angezeigt (Siehe Abschnitt 5.3.2 - Verschiedene Gerätefamilien). Beim Klick des Nutzers muss dieser nicht auf eine neue Seite gelangen, sondern es wird schlicht das ViewModel der Seriendetailansicht mit neuen Daten angereichert.

6.6.3 Navigation mit einem Split-View bzw. Master-Detail-View

Ein weiteres Problem bringt ein UI-Element mit, das unter UWP Split-View und unter Xamarin.Forms Master-Detail-View bezeichnet wird. Mit einem Master-Detail-View bzw. Split-View kann bspw. eine Hamburgernavigation implementiert werden, was im Kontext dieser Arbeit gemacht wurde. Kommt ein solches UI-Element zum Einsatz, ändert sich das Frame, auf dem navigiert wird. Standardmäßig wird auf dem Root-Frame navigiert, welches unter UWP so erreicht wird:

```
(Frame)Window.Current.Content
```

und unter Xamarin.Forms so:

```
App.Current.MainPage.Navigation
```

Wird allerdings ein Split-View bzw. ein Master-Detail-View genutzt, befindet sich dieser auf dem Root-Frame und besitzt einen eigenen neuen Frame, auf welchen bspw. bei einer Hamburgernavigation navigiert wird. Dieser Frame wird so unter UWP erreicht:

```
var frame = (Frame)Window.Current.Content;
var splitView = (SplitView)frame.Content;
(Frame)splitView.Content;
```

und so unter Xamarin.Forms

```
((MasterDetailPage)App.Current.MainPage).Detail.Navigation;
```

Damit immer auf den richtigen Frame navigiert wird, kann dieser unter UWP so deklarieren:

```
private Frame _frame
{
      get
      {
            var frame = (Frame)Window.Current.Content;
            var splitView = frame.Content != null ?
              ((Page)frame.Content).Content as SplitView :
              null;

            return splitView != null ? (Frame)splitView.Content : frame;
      }
}
```

Und unter Xamarin.Forms so:

```
private INavigation _frame
{
      get
      {
            var mainPage = App.Current.MainPage;
            var masterDetailPage = mainPage as MasterDetailPage;
            return masterDetailPage != null ?
              masterDetailPage.Detail.Navigation :
              mainPage.Navigation;
      }
}
```

6.7 Mobile Service

Viele moderne Anwendungen brauchen eine REST-API, diese kann in den unterschiedlichsten Programmiersprachen und mit den unterschiedlichsten Frameworks umgesetzt werden. Eine Möglichkeit ist Azure Mobile Services. Mit einem Azure Mobile Services Projekt wird ein Rumpf für eine REST-API erstellt, das daraus resultierende Projekt sieht dann so aus:

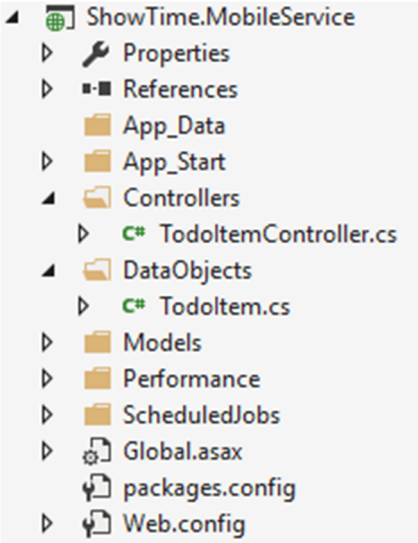

Abbildung 47: Mobile Service Projekt

Im Ordner Controllers befinden sich die REST-Schnittstellen, im Ordner DataObjects das Datenmodell.

```
public class TodoItemController : TableController<TodoItem>
{
	protected override void Initialize(HttpControllerContext controllerContext)
	{
		base.Initialize(controllerContext);
		MobileServiceContext context = new MobileServiceContext();
		DomainManager = new EntityDomainManager<TodoItem>(context, Request, Services);
	}
	// GET Gibt ein JSON mit allen TodoItems zurück
	public IQueryable<TodoItem> GetAllTodoItems()
	{
		return Query();
	}
	// GET Gibt ein TodoItem mit der id als JSON
	// zurück
	public SingleResult<TodoItem> GetTodoItem(string id)
	{
		return Lookup(id);
	}
	// PATCH Aktualisiert ein TodoItem und gibt
	// dieses dann als JSON zurück
	public Task<TodoItem> PatchTodoItem(string id, Delta<TodoItem> patch)
	{
		return UpdateAsync(id, patch);
	}
	// POST Fügt ein neues TodoItem hinzu
	public async Task<IHttpActionResult>PostTodoItem(TodoItem item)
	{
		TodoItem current = await InsertAsync(item);
		return CreatedAtRoute(
		  "Tables",
		  new { id = current.Id },
		  current
		);
	}
	// DELETE Löscht ein TodoItem anhand der id
	public Task DeleteTodoItem(string id)
	{
		return DeleteAsync(id);
	}
}
```

Das fertige Grundgerüst kann nun einfach erweitert werden. So kann nicht nur verhältnismäßig schnell und einfach eine REST-API implementiert werden, es hat des Weiteren auch einige gute Synergieeffekte, wodurch noch mehr Code-Sharing betrieben werden kann. Dies liegt daran, dass Azure Mobile Services in C# und mit .NET entwickelt werden.

Als einfaches, aber gutes Beispiel dient dabei ein DTO, welches zwischen Client und Server (REST-API) geteilt werden kann. Dafür muss nur die PCL, welche das DTO enthält, von beiden referenziert werden.

```
public class TodoItemDTO
{
       public string Id { get; set; }
       public string Text { get; set; }
       public bool Complete { get; set; }
}
```

So kann das DTO von Server und Client genutzt werden.

Server

```
return Query().Select(
  td => new TodoItemDTO()
    {
      Id = td.Id,
      Text = td.Text,
      Complete = td.Complete
    }
);
```

Client

```
return JsonConvert.DeserializeObject<IEnumerable<TodoItemDTO>>(jsonString);
```

Bei mehreren und komplexeren DTO‘s kann dieses Vorgehen viel Zeit einsparen und der Code wird besser wart- und erweiterbar.

Kapitel 7

7 Ergebnisse

In diesem Kapitel wird bewertet, wie gut die architektonischen Überlegungen und die Code-Sharing-Strategien während der Implementierungsphase funktioniert haben. Es werden auch Code-Metriken ausgewertet und die daraus resultierenden quantitativen Daten analysiert und bewertet.

7.1 Architektur

Während der Implementierungsphase haben sich die architektonischen Überlegungen aus dem Abschnitt 5.3 - Geplante Architektur - gefestigt. Die dort zu sehende Architektur konnte auch so umgesetzt werden und hat enorm zum Erfolg beigetragen. Dies zeigt auch eine Wartbarkeitsanalyse[30] der Visual Studio Code-Metriken, hier wurde trotz Shared-Project ein sehr guter Wert von 86 (0-100) erreicht. Außerdem haben sich die einzelnen Design-Patterns als unverzichtbar herausgestellt, nur durch diese konnte das Code-Sharing in dem hohen Maße umgesetzt werden. Allgemein muss gesagt werden, dass eine optimale Architektur bei der Cross-Plattform-Entwicklung besonders wichtig ist, da nicht nur die ohnehin positiven Effekte der besseren Testbarkeit, Wartbarkeit und Erweiterbarkeit erreicht werden, sondern zusätzlich auch redundante Implementierung reduziert wird. Im Endeffekt bedeutet jede Funktion, die zwischen den Plattformen geteilt werden kann, nicht nur einen reduzierten Implementierungsaufwand, sondern auch, dass für Android, iOS und Windows nicht einzeln getestet, gewartet oder erweitert werden muss. Im Umkehrschluss ist es noch fataler, wenn innerhalb der Anwendungsentwicklung eine schlechte Architektur gewählt wird und falsche oder keine Design-Pattern eingesetzt werden. Daher steigen bei der Entwicklung einer Cross-Plattform-Anwendung die Ansprüche an den Entwickler und dessen architektonischen Wissensstand.

7.2 Code-Sharing-Strategien

Die im Abschnitt 4.5 vorgestellten Code-Sharing-Strategien haben sich beide als wichtig und nützlich herausgestellt. Der Einsatz der richtigen Strategie für das jeweilige Problem ist wichtig. Dabei hat sich bewiesen, dass das Shared-Project die bessere Wahl ist, wenn sich einzig die referenzierten Assemblies unterscheiden. Außerdem kann das Shared-Project auch gewählt werden, wenn sich ausschließlich einige wenige Eigenschaften und Aufrufe unterscheiden. Wird der Unterschied allerdings größer, oder

[30] Mehr Infos [37]

ist die Testbarkeit sehr wichtig, ist es von Vorteil eine PCL zu verwenden, die die Gemeinsamkeiten (Basisklassen, Interfaces, …) abdeckt, da bei größeren Unterschieden die im Abschnitt 4.5 genannten Nachteile eines Shared-Projects überwiegen. In der PCL können bspw. in Basisklassen die plattformübergreifenden Funktionen implementiert und/oder allgemeingültige Interfaces definiert werden. Plattformspezifisch kann dann das Interface oder die Basisklasse vererbt und die plattformspezifischen Funktionen und Eigenschaften implementiert werden. Die PCL ist insgesamt wichtiger, da es hierfür eine große Zahl an NuGet Packages gibt, die zahlreiche allgemeingültige Funktionen bereits abbilden, wodurch vieles nicht selbst implementiert werden muss. Allerdings sollten auch die NuGet Packages mit Bedacht eingesetzt werden, da jedes NuGet Package die Abhängigkeit von Dritten erhöht. Bei stark genutzten Packages sind die Risiken aber minimal.

7.3 Shared-Code-Anteil

Im Kapitel 6 - Implementierung - wurde schon deutlich, dass sich weite Teile der Implementierung vereinheitlichen lassen. Allerdings wurden dort einzig qualitative Einschätzungen abgegeben. Um valide quantitative Daten zu erhalten wurde die Meeting-App mit Hilfe der Visual Studio Code-Metriken analysiert. Es wurde die Meeting-App analysiert, da diese schon mehr als ein Prototyp ist und somit die validesten Ergebnisse liefern kann.

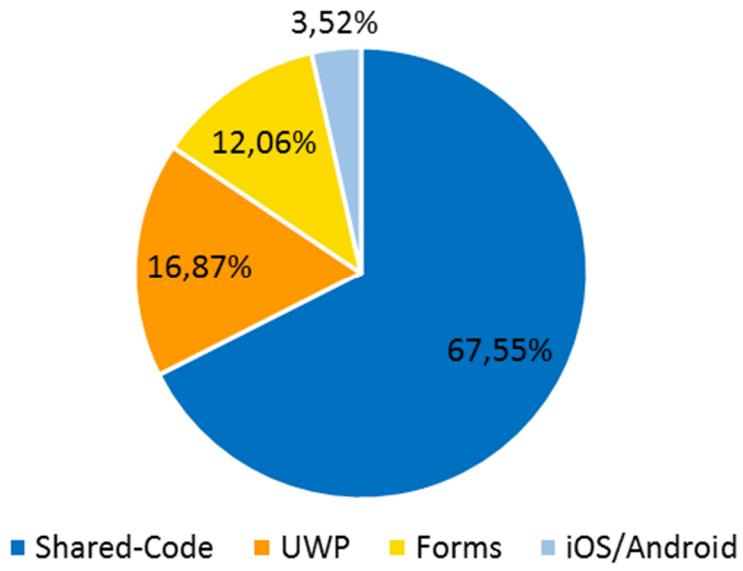

Abbildung 48: Shared-Code-Anteil

Die Daten aus der Code-Analyse zeigen das 67,55% des Quellcodes Shared-Code ist. Weitere 28,93% verteilen sich auf UWP und Xamarin.Forms. Lediglich bei 3,52% des Quellcodes musste unter Xamarin.iOS und Xamarin.Android implementiert werden.

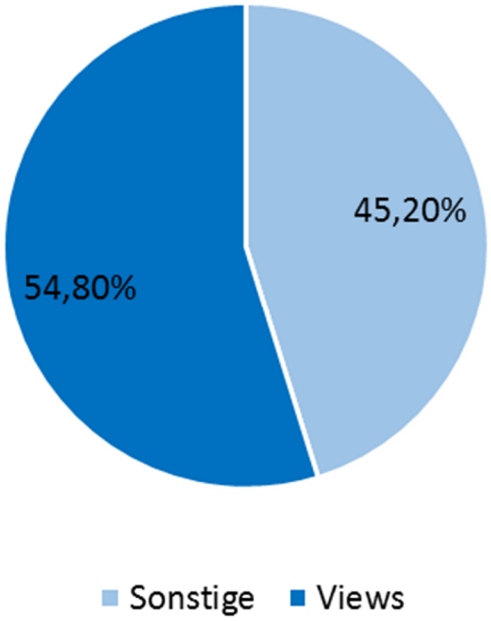

Abbildung 49: Anteil Views an nicht Shared-Code

Von den 28,93%, die sich auf UWP und Xamarin.Forms verteilen, machen die Views 54,8% (15,85% vom gesamten Quellcode) aus. Dieser Anteil ist, wie im Abschnitt 6.5 - Userinterface - ersichtlich geworden, zum Teil übertragbar, wodurch sich erneut viel Aufwand einsparen lässt.

Abschließend lässt sich sagen, dass die qualitativen Aussagen auch durch quantitative Zahlen gestützt werden. Dennoch kommt es natürlich auf die Anwendung an. Eine Anwendung mit geringer Businesslogik, wenigen Services und vielen aufwändigen Views würde sicherlich deutlich schlechtere Ergebnisse erzielen. Auf der anderen Seite gibt es sicherlich auch unzählige komplexere Anwendungen, bei denen der Anteil an Shared-Code deutlich höher ausfallen würde.

Kapitel 8

8 Fazit und Ausblick

In diesem Kapitel wird ein kurzes Fazit zu dem untersuchten Ansatz gezogen. Es wird auch ein Ausblick gegeben, wie dieser noch erweitert werden kann. Zusätzlich wird kurz auf die Preview Version von Xamarin.Forms UWP sowie auf die Übernahme von Xamarin durch Microsoft eingegangen.

8.1 Fazit

Xamarin.Forms und UWP können gut miteinander kombiniert werden. Es kann dabei viel redundante Entwicklung eingespart werden. Eine Cross-Plattform-Anwendung für Android, iOS und Windows, sowie Smartphone, Tablet, Desktop und weitere Geräte ist durch die Kombination dieser beiden Frameworks möglich. Dabei kann ein sehr hoher Anteil an Shared-Code erreicht werden. Es muss aber auch gesagt werden, dass die Kombination von Xamarin.Forms und UWP nicht in allen Szenarien die beste Wahl ist, teilweise ist Xamarin.Forms ausreichend. Außerdem ist der Entwickler bei der Auswahl der NuGets deutlich eingeschränkter, da diese nach Möglichkeit für die einzelnen Plattformen verfügbar sein sollte.

8.2 Ausblick

8.2.1 Erweiterung auf WPF (Windows 7)

Für das Enterprise Development wäre außerdem noch ein Code-Sharing mit Windows Presentation Foundation (WPF) zu prüfen. Durch das Framework WPF ist es möglich, auch klassische Desktop-Anwendungen, bspw. für Windows 7, zu entwickeln. Gerade größere Unternehmen werden noch einige Jahre benötigen, bis sie auf Windows 10 migrieren können, bei einigen ist sogar die Migration auf Windows 7 noch nicht abgeschlossen. Auch kann mit WPF auf einige Features zugegriffen werden, welche unter UWP nicht erreichbar sind. Die Gemeinsamkeiten der Frameworks UWP und WPF sind allerdings sehr groß, daher kann davon ausgegangen werden, dass auch noch eine WPF UI hinzugenommen werden kann, ohne dass dies zu einem wesentlichen Mehraufwand führt.

8.2.2 Xamarin.Forms UWP im Vergleich zum untersuchten Ansatz

Während dieser Arbeit wurde am 17.11.2015, in der Version Xamarin.Forms 2.0, eine Preview für die Erweiterung auf die Universal Windows Platform vorgestellt. Diese ermöglicht es, eine Xamarin.Forms Anwendung auch für die UWP zu veröffentlichen. Die Preview weist allerdings noch einige Bugs auf, als bekannte Bugs äußert dabei Xamarin[31]:

- das Aussehen einiger Elemente ist noch nicht final
- es gibt noch Abstürze bei der Navigation
- Textausrichtungen sind teilweise suboptimal
- es gibt keinen Map Support

Neben diesen Bugs gibt es aber noch weitere. So funktioniert der ActivityIndicator während der Testphase beispielsweise nicht, bzw. wurde dieser dauerhaft angezeigt. Solche Bugs treten auch noch bei einigen anderen Elementen auf. Aufgrund der Anzahl der Bugs wird es wohl noch einige Monate dauern, bis der Preview Status aufgehoben wird und Xamarin.Forms UWP produktiv eingesetzt werden kann.

Sobald Xamarin.Forms UWP den Preview Status verlassen hat, stellt sich die Frage, wann und warum der Mehraufwand für eine zweite UWP UI in Kauf genommen werden sollte. Eine separate UWP UI macht immer Sinn, sobald die Anwendung auch auf einem Desktop laufen soll, denn hier soll der Nutzer die Anwendungsgröße dynamisch verändern können. Wie in dem vorhergehenden Kapitel beschrieben, ist dies in Xamarin.Forms nicht ohne weiteres möglich. Sie macht meist auch dann Sinn, wenn eine Anwendung auf irgendeiner Windows Platform laufen soll, da es hier im Gegensatz zu Android und iOS immer dynamischer Größenanpassungen bedarf. Dies liegt daran, dass es bei Tablets den Desktop Modus gibt, wodurch eine dynamische Änderung genauso wichtig wird und bei Smartphones das Continuum, sodass auch hier die Anwendung auf großen, wie kleinen Bildschirmen dynamisch angepasst werden muss. Neben besseren, feineren und einfacheren Anpassungen an Bildschirmgrößen hat eine UWP UI auch den Vorteil, dass komplexe Designs einfacher umgesetzt werden können. Gerade bei einem Szenario, bei dem eine komplexe Desktopanwendung entwickelt wird und eine reduzierte mobile Variante, ist es wesentlich einfacher, eine getrennte UI zu entwickeln. Soll aber eine sehr simple Anwendung oder nur eine mobile Anwendung entwickelt werden, kann der Gebrauch von Xamarin.Forms ausreichend sein.

[31] Vgl. [29]

8.2.3 Xamarin Übernahme durch Microsoft

In Zukunft wird dieser Ansatz hoffentlich überflüssig werden. Ein erster Schritt ist die Übernahme von Xamarin, am 24.02.2016, durch Microsoft.[32] Diese lässt viel Raum für Spekulationen über die Zukunft von Xamarin, Xamarin.Forms und UWP. Hierdurch kann es z.B. eine immer stärkere Annäherung von Xamarin.Forms und UWP geben, bis diese komplett vereinheitlicht werden. Ähnlich wie es bereits mit der Windows Entwicklung und Plattform geschehen ist (im Abschnitt 4.3 beschrieben), könnte dies also auch mit Xamarin.Forms und UWP passieren. Ob es allerdings so kommen wird, bleibt abzuwarten. Daher wird diese Kombination der beiden Frameworks, auf die nächsten Jahre gesehen, gerade im Enterprise Bereich eine Rolle spielen. Außerdem würde eine Kombination dieser zwei Frameworks durch weitere Annäherungen kurz- und mittelfristig immer weiter erleichtert und langfristig abgelöst werden.

[32] [30]

Abkürzungsverzeichnis

UI	Userinterface
UWP	Universal Windows Platform
UWA	Uiniversal Windows Platform
WPF	Windows Presentation Foundation
API	Application Programming Interface
REST	Representational State Transfer
IoT	Internet of Things
BYOD	Bring Your Own Device
MVVM	Model View ViewModel
XAML	Extensible Application Markup Language
MVC	Model View Controller
DAO	Data Access Object
DTO	Data Transfer Object
DI	Dependency Injection
JIT	Just In Time
IL	Intermediate Language
AOT	Ahead Of Time
HTML	Hypertext Markup Language
JS	JavaScript
OS	Operating System
PC	Personal Computer
PCL	Portable Class Libraries
DLL	Dynamic Link Library

AAD	Azure Active Directory
AD	Active Directory
SQL	Server Query Language
TV	Television
GPS	Global Positioning System
JSON	JavaScript Object Notation
XML	Extensible Markup Language

Abbildungsverzeichnis

Abbildung 1: Layer Überblick [1] 9
Abbildung 2:Layer Detail [1] 10
Abbildung 3: Onion Abhängigkeiten [1] 10
Abbildung 4: Onion Überblick [19] 10
Abbildung 5: The MVVM-Pattern [2] 12
Abbildung 6: Dependency Injection [3] 13
Abbildung 7: Fassade [4] 14
Abbildung 8: Data Transfer Object 15
Abbildung 9: Data Access Object [5] 15
Abbildung 10: Xamarin Architektur [6] 17
Abbildung 11: Xamarin.Forms Architektur [7] 18
Abbildung 12: DatePicker [8] 19
Abbildung 13: Schritte zu Windows 10 [9] 20
Abbildung 14: Windows Universal Platform Architektur [9] 21
Abbildung 15: Gerätefamilien 22
Abbildung 16: Formfaktor - Position ändern [10] 22
Abbildung 17: Formfaktor - Größe ändern [10] 23
Abbildung 18: Formfaktor – Neuanordnen [10] 23
Abbildung 19: Formfaktor – Einblenden [10] 23
Abbildung 20: Formfaktor – Ersetzen [10] 24
Abbildung 21: Formfaktor - Architektur ändern [10] 24
Abbildung 22: Effektive Pixel [10] 25
Abbildung 23: ShowTime UWP Desktop 30
Abbildung 24: ShowTime UWP Mobile 30
Abbildung 25: ShowTime Android 31
Abbildung 26: Serien-App UWP 32
Abbildung 27: Serien-App Android 32
Abbildung 28: Design-Guidelines 1-3 34
Abbildung 29: Design-Guidelines 4 35
Abbildung 30: Design-Guidelines 5 36
Abbildung 31: UI-Elemente Picker 36
Abbildung 32: UI-Elemente DatePicke 37

Abbildung 33: UI-Elemente TimePicker 37
Abbildung 34: Continuum [11] 38
Abbildung 35: Gerätefamilien Matrix 38
Abbildung 36: Responsive Design Mockup 39
Abbildung 37: Adaptive Design Mockup 40
Abbildung 38: Tailored Design Mockup 41
Abbildung 39: Methodik - MVVM 42
Abbildung 40: Methodik - Service Fassade & DI 43
Abbildung 41: Methodik - SF & DI Sequenzdiagramm 43
Abbildung 42: Methodik - DAO & DTO 44
Abbildung 43: Methodik – Gesamtarchitektur 44
Abbildung 44: Methodik - PCL Setup 47
Abbildung 45: Methodik – Projektsetup 48
Abbildung 46: Ändern des Navigtionsablaufs 70
Abbildung 47: Mobile Service Projekt 76
Abbildung 48: Shared-Code-Anteil 80
Abbildung 49: Anteil Views an nicht Shared-Code 81

Referenzen

[1] D. Marbach, "Methodsandtools," [Online]. Verfügbar: http://www.methodsandtools.com/archive/onionsoftwarearchitecture.php. [Zugegriffen 07 12 2015].

[2] "Microsoft MSDN - The MVVM Pattern Image," Microsoft, [Online]. Verfügbar: https://i-msdn.sec.s-msft.com/dynimg/IC564167.png. [Zugegriffen 30 11 2015].

[3] "Microsoft MSDN - Dependency Injection Image," Microsoft, [Online]. Verfügbar: https://i-msdn.sec.s-msft.com/dynimg/IC245731.png. [Zugegriffen 03 12 2015].

[4] "Dofactory - Facade," Dofactory, [Online]. Verfügbar: http://www.dofactory.com/net/facade-design-pattern. [Zugegriffen 03 12 2015].

[5] "Wikipedia - Data Access Object," [Online]. Verfügbar: https://gl.wikipedia.org/wiki/Data_access_object#/media/File:IdeaDAOpng.png. [Zugegriffen 03 12 2015].

[6] "Xamarin," Xamarin Inc., 05 2015. [Online]. Verfügbar: https://blog.xamarin.com/wp-content/uploads/2015/05/code-sharing1.png. [Zugegriffen 07 12 2015].

[7] "Xamarin," Xamarin Inc., [Online]. Verfügbar: https://blog.xamarin.com/wp-content/uploads/2014/06/XamarinForms1.png. [Zugegriffen 07 12 2015].

[8] "Xamarin - DatePicker," Xamarin Inc., [Online]. Verfügbar: http://developer.xamarin.com/api/type/Xamarin.Forms.DatePicker/. [Zugegriffen 07 12 2015].

[9] "Microsoft Virtual Academy - A Developer's Guide to Windows 10 Part 1," Microsoft, 12 08 2015. [Online]. Verfügbar: https://mva.microsoft.com/en-US/training-courses/a-developer-s-guide-to-windows-10-12618?l=w00laTrRB_405095281. [Accessed 07 12 2015].

[10] Microsoft, "MSDN - Responsive Design," Microsoft, [Online]. Verfügbar: https://msdn.microsoft.com/en-US/library/windows/apps/dn958435.aspx. [Zugegriffen 10 11 2015].

[11] „Chip - Display Dock," CHIP Digital GmbH, [Online]. Verfügbar: http://www.chip.de/ii/3/2/7/3/9/7/8/9/7089fdcb-e269-4be4-985c-9f822e505766_72-6bb4ae3c292e6177.jpg. [Zugegriffen 24 03 2016].

[12] „Statista - Mobile Engeräte Weltweit," 30 11 2015. [Online]. Verfügbar: http://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/.

[13] „Statista - Mobile Endgeräte Deutschland,“ 30 11 2015. [Online]. Verfügbar: http://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenutzer-in-deutschland-seit-2010/.

[14] "Microsoft MSDN - Dependency Injection," Microsoft, [Online]. Verfügbar: https://msdn.microsoft.com/en-us/library/ff921152.aspx. [Zugegriffen 03 12 2015].

[15] "Microsoft MSDN - The MVVM Pattern," Microsoft, 10 02 2012. [Online]. Verfügbar: https://msdn.microsoft.com/en-us/library/hh848246.aspx. [Zugegriffen 30 11 2015].

[16] "Microsoft MSDN - Data Transfer Object," Microsoft, [Online]. Verfügbar: https://msdn.microsoft.com/en-us/library/ff649585.aspx. [Zugegriffen 03 12 2015].

[17] D. G. Starke, „Effektive Softwarearchitekturen - Ein praktischer Leitfaden,“ München, Carl Hanser Verlag, 2014, p. 203.

[18] D. G. Starke, „Effektive Softwarearchitekturen - Ein praktischer Leitfaden,“ München, Carl Hanser Verlag, 2014, pp. 182,183.

[19] D. Marbach, „Jaxenter,“ Software & Support Media GmbH, 22 04 2015. [Online]. Verfügbar: https://jaxenter.de/die-zwiebelarchitektur-und-ihre-vorzuege-19474. [Zugegriffen 07 12 2015].

[20] "Xamarin - Understanding the Xamarin mobile platform," Xamarin Inc., [Online]. Verfügbar: https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_1_-_understanding_the_xamarin_mobile_platform/. [Zugegriffen 07 12 2015].

[21] "Xamarin - Sharing code options," Xamarin Inc., [Online]. Verfügbar: https://docs.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/. [Accessed 07 12 2015].

[22] "Xamarin - Xamarin.Forms," Xamarin Inc., [Online]. Verfügbar: https://developer.xamarin.com/guides/cross-platform/xamarin-forms/. [Zugegriffen 07 12 2015].

[23] "Microsoft MSDN - Einführung UWP," Microsoft, [Online]. Verfügbar: https://msdn.microsoft.com/en-us/library/windows/apps/dn894631.aspx. [Zugegriffen 07 12 2015].

[24] „Microsoft MSDN - Gerätefamilien,“ Microsoft, [Online]. Verfügbar: https://msdn.microsoft.com/en-us/library/windows/apps/dn706137.aspx. [Zugegriffen 07 12 2015].

[25] J. Yates, "adventuresinxamarinforms," 21 11 2014. [Online]. Verfügbar: http://adventuresinxamarinforms.com/2014/11/21/creating-a-xamarin-forms-app-part-6-view-model-first-navigation/. [Zugegriffen 16 11 2016].

[26] AppHarbor, "AppHarbor.com," AppHarbor Inc., [Online]. Verfügbar: http://featuretests.apphb.com/DependencyInjection.html. [Zugegriffen 2015 10 20].

[27] MvvmCross, "GitHub," [Online]. Verfügbar: https://github.com/MvvmCross/MvvmCross-Plugins/tree/master/SQLite-PCL. [Zugegriffen 12 12 2015].

[28] A. Simmon, „anthonysimmon.com,“ [Online]. Verfügbar: https://anthonysimmon.com/eventtocommand-in-xamarin-forms-apps/. [Zugriff am 22 10 2015].

[29] Xamarin, "Xamarin - UWP Preview," Xamarin Inc., [Online]. Verfügbar: https://developer.xamarin.com/guides/xamarin-forms/platform-features/windows/installation/universal/. [Accessed 18 01 2016].

[30] S. Guthrie, "Microsoft Blog - Übernahme von Xamarin," Microsoft, 24 02 2016. [Online]. Verfügbar: http://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/. [Zugegriffen 24 02 2016].

[31] „Statista - Smartphones nach Betriebssystem Deutschland,“ 30 11 2015. [Online]. Verfügbar: http://de.statista.com/statistik/daten/studie/198435/umfrage/marktanteile-der-smartphone-betriebssysteme-am-absatz-in-deutschland/.

[32] „Statista - Tablets nach Betriebssystem Welt,“ 30 11 2015. [Online]. Verfügbar: http://de.statista.com/statistik/daten/studie/242259/umfrage/prognose-zu-marktanteilen-bei-tablets-nach-betriebssystem/.

[33] „Statista - Desktop Betriebssysteme,“ 30 11 2015. [Online]. Verfügbar: http://de.statista.com/statistik/daten/studie/158102/umfrage/marktanteile-von-betriebssystemen-in-deutschland-seit-2009/.

[34] E. M. Edward Moemeka, Real World Windows 10 Development, Apress, 2015.

[35] C. Petzold, Creating Mobile Apps with Xamarin.Forms, Microsoft Press, 2014.

[36] M. Dobrev, „Untersuchung hybrider Ansätze zur plattformübergreifenden App-Entwicklung,“ TU Dresden, Dresden, 2014.

[37] "MSDN - Blog," Microsoft, 20 11 2007. [Online]. Verfügbar: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/. [Zugegriffen 17 03 2016].

[38] D. G. Starke, „Effektive Softwarearchitekturen - Ein praktischer Leitfaden,“ München, Carl Hanser Verlag, 2014, p. 176.

Anhang CD

- Projekte mit Quellcode
- Masterarbeit in digitaler Form