TU Graz

Thomas Loidolt

Evaluierung zur Plattformunabhängigen Entwicklung mit Mono

Masterarbeit

Technische Universität Graz

Institut für Softwaretechnologie
Betreuer: Univ.-Prof. Dipl-Ing. Dr.techn.Wolfgang Slany

Lannach, Juli 2014

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen, als solche kenntlich gemacht habe.

Graz, am ______________________ ______________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Danksagung

Der Dank an dieser Stelle richtet sich vor allem an Herrn Dipl.-Ing. Dr.techn. Wolfgang Slany, der es mir ermöglicht hat diese Arbeit auszuwählen, da diese genau meinen Vorstellungen, dem technischen Bereich und meinem absoluten Interesse entspricht. Des Weiteren gilt mein Dank der Firma Comm-Unity, im Besonderen Herrn DI Michael Stark, der sich für diese Thematik sofort begeistern konnte und Potential in der Idee, wie auch in der Umsetzung erkannte. Der Ursprung dieser Idee setzt sich aus mehreren Gründen bzw. Faktoren zusammen. Auf der einen Seite der immer stetig steigende Entwicklungsaufwand für die Portierung von Software mit identen Systematiken auf diverse mobile Betriebssysteme, auf der anderen Seite die Qualität, im speziellen jener der Funktionalität und Zuverlässigkeit, zu steigern. Abschließend möchte ich mich noch bei all meinen Studienkollegen sowie Freunden und hauptsächlich bei meiner Familie bedanken, die mich im Laufe des Studiums tatkräftig unterstützt haben.

Kurzfassung

Da sich gerade in der heutigen Zeit der Prozess der kontinuierlichen Veränderung in der mobilen Landschaft immer schneller entwickelt, stellt genau dies für die Unternehmen eine Herausforderung dar. Die Wahl der "richtigen"mobilen Strategie ist für Erfolg oder Misserfolg in diesem Unternehmenssektor verantwortlich. Um eine Entscheidung zu treffen, müssen bestimmte vorgegebene Rahmenbedingungen stimmen und gesetzt werden. Der Wunschgedanke bei der Entwicklung von Software ist immer noch: "Write once, run everywhere". Genau aus diesem Paradigma ergeben sich komplexe Problemstellungen in Verbindung mit der plattformunabhängigen Entwicklung. Es gibt bereits Ansätze diese Problemstellungen zu lösen, diese bewegen sich meist in die Entwicklung von HTML5 Applikation oder auch Hybridlösungen. Je nach Anforderungen an das Projekt oder die Applikation gerät man auch bei dieser Variante an die Grenzen. Die beste Performance und Flexibilität betreffend der optimalen Ressourcennutzung erhält man immer noch durch native Applikationen. Generell ist dabei das Wichtigste den Aufbau des Betriebssystems zu kennen, wie auch die Entwicklungsmöglichkeiten. In dieser Arbeit wird, aufgrund der Vorgaben der Firma Comm-Unity, im Speziellen auf die Plattformen Android und Windows Phone eingegangen. Die Frage stellt sich nur, ob plattformunabhängige Entwicklung auf nativer Basis möglich ist und wenn ja, mit welchen Einschränkungen ist dabei zu rechnen. Das Mono-Projekt, mit der auf C# basierenden Programmiersprache, bietet in diesem Bereich die besten Vorraussetzungen. Mit der Mono-Umgebung können Applikationen geschrieben werden, die eine gemeinsame Codebasis besitzen und trotz dessen nativ auf der Plattform laufen. Für Android gibt es noch eine Erweiterung mit der auf die Java-Application Progamming Interface (API) des Betriebssystems zugegriffen werden kann. Als Ergänzung zur Entwicklung mit Mono gibt es noch das MvvmCross-Framework, das den Schwerpunkt auf Databinding und der Model View ViewModel (MVVM)-Architektur setzt. Ein weiterer Aspekt ist das Testen der Applikation. Um Fehlerquellen zu elimi-

nieren ist das Testen und auch die agile Vorgehensweise ein wichtiger Punkt dieser Arbeit. Bei der Umsetzung wurde auf die definierten Ziele der Firma Comm-Unity und deren Anforderungen an die Applikation eingegangen.

Abstract

Nowadays the process of continuous change in the mobile landscape develops faster and faster and there are new challenges to tackle for companies. The selection of the ”right”mobile strategy is responsible for success or failure in the enterprise sector. In order to make a decision, the right conditions have to be in place. A main goal in software development is still: ”Write once, run everywhere”. This paradigm triggers complex problems arising in the context of platform-independent development. There are already approaches for solving some of these problems with HTML5 applications or hybrid solutions. Depending on the project or application requirements several limitations come along by using web technologies. The best performance and flexibility concerning the optimal use of resources are still obtained through developing native applications. Core knowledge of the operating system, is as important as the knowledgeof the development possibilities on each device. In this work, because of Comm-Unity's specifications, the Android and Windows Phone platforms are specifically addressed. The question remains, however, whether platform-independent development is possible native, and if yes, which restrictions must be considered. The Mono project based on the C# programming language, offers the best conditions in this area. With the Mono environment, applications can be written, which have a common code base and are able to run on the platform native. For Android there is another extension to access the operating system commands via the Java API. Complementary to development with Mono, there is also the MvvmCross framework, which sets the focal point on data binding and the MVVM architecture. A further aspect is application testing. In order to eliminate sources of errors, testing and the agile approach are important aspects of this work. The implementation addressed the defined goals of the Comm-Unity company and their application requirements.

Inhaltsverzeichnis

Abstract **iv**

1. Einleitung **1**
1.1. Problemumfeld . . . 2
1.1.1. Plattformunterschiede . . . 3
1.1.2. HTML5 Limitierung . . . 3
1.1.3. Hybride Annäherung . . . 4
1.2. Ziele . . . 4

2. Plattformgrundlagen **6**
2.1. Android . . . 6
2.1.1. Architektur . . . 7
2.1.2. Dalvik Virtuelle Maschine (VM) . . . 9
2.1.3. Komponenten . . . 10
2.1.4. Oberflächen Design . . . 11
2.1.5. Verknüpfung Layouts mit Activities . . . 11
2.2. Windows Phone . . . 13
2.2.1. Architektur . . . 13
2.2.2. Modern-UI . . . 15
2.2.3. XAML und Pages . . . 16

3. Mono **18**
3.1. Laufzeitumgebung . . . 18
3.2. .net Base Class Library . . . 19
3.3. Mono für Android . . . 20
3.3.1. Android Callable Wrappers . . . 21
3.3.2. Managed Callable Wrappers . . . 21
3.4. Xamarin . . . 21
3.4.1. Xamarin Platform . . . 21

3.5. Code-Sharing . . . 22
3.5.1. Shared Projects . . . 23
3.5.2. Portable Class Library . . . 23
3.6. MvvmCross . . . 24
3.6.1. Model-View-ViewModel . . . 25
3.6.2. Service Location & Inversion of Control . . . 26
3.6.3. Databinding . . . 30
3.6.4. Testen . . . 34

4. Testen 37
4.1. Konventioneller Softwareentwicklungsprozess . . . 37
4.1.1. Wasserfallmodell . . . 37
4.2. Agile Methode . . . 39
4.3. Entwickler vs. Tester . . . 40
4.4. Test Driven Development . . . 41
4.4.1. 3 Phasen des Test Driven Development (TDD) . . . 41
4.4.2. Unit-Test . . . 42
4.4.3. Mocking . . . 44
4.4.4. Testtypen . . . 45
4.5. Testmetrik . . . 46
4.5.1. Testmetrik nach Hetzel . . . 47

5. Umsetzung 48
5.1. Ziele . . . 48
5.2. Durchführung . . . 49
5.2.1. Visual Studio (VS)-Solution erstellen . . . 49
5.2.2. Datenbankzugriff . . . 50
5.2.3. Workflow - MVVM . . . 52
5.2.4. Testbarkeit . . . 53
5.2.5. Generischer Hardwarezugriff . . . 54
5.2.6. Webservice-Zugriff . . . 55

6. Zusammenfassung und Ausblick 57

A. Codebeispiel MVVM 60
A.1. ViewModel - PCL . . . 60
A.2. View - Android . . . 61

A.3. View - Windows Phone 61

B. Mocking - Beispiel **63**

C. Webservice **67**

C.1. GeoCoding 67

Abkürzungsverzeichnis **70**

Literatur **72**

Abbildungsverzeichnis

1.1. Quelle: *Mobile OS Statistik* 2014 2

2.1. Softwarestack Android, Quelle:vgl. Meier, 2012, S.16 7
2.2. Dalvik VM, Quelle: Oechsle, 2013, S.270 9
2.3. Android Layout 12
2.4. Softwarestack Windows Phone, Quelle:vgl. Andrew Whitechapel, 2012, S6. 13
2.5. Windows Phone Seite 17

3.1. Base Class Library, Quelle:*Base Class Library* 2014 19
3.2. Mono for Droid, Quelle: *Mono for Droid* 2014 20
3.3. Xamarin Platform, Quelle: Xamarin, 2013 22
3.4. Shared Project, Quelle: Shackles, 2012 23
3.5. PCL, Quelle: *Code-Sharing* 2014 24
3.6. MVC - MVP 25
3.7. Quelle:*MVVM-Entwurfsmuster* 2014 26

4.1. Wasserfallmodell, Quelle: *Wasserfallmodell* 2014 38
4.2. Agiles Entwickeln, Quelle:*Test-Driven-Development* 2014 40
4.3. Entwickler vs. Tester, Quelle: Brader, Hilliker und Wills, 2013, S.17 41
4.4. TDD, Quelle:*Test-Driven-Development* 2014 42

5.1. Anwendungsfälle 50
5.2. VS-Solution 50
5.3. PCL-SQLite, Quelle:*PCL-Code-Sharing* 2014 51
5.4. MenuAuswahlViewModel 52
5.5. MVVM-Test, Quelle:*Mvvm ViewModel-Test* 2014 53

1. Einleitung

Im Zuge der technischen Errungenschaften in vielen Bereichen und Sektoren der Technik, haben sich unter anderem im Bereich von Handys, oder auch Smartphones, welche als Weiterentwicklung der früheren Mobiltelefone gesehen werden können, immense Fortschritte entwickelt. Früher noch als simples Kommunikationsmittel genutzt, sind Smartphones heutzutage nahezu Alleskönner, von der Übermittlung von Bildern bis hin zu komplexen Anwendungen für Customer Relationship Management (CRM) oder auch Enterprise Resource Planning (ERP)-Systemen.

Das Schlagwort in Bezug auf Smartphones ist „App“. Unter „App“ versteht man eine Applikation, welche genau für das sich am Smartphone befindliche Betriebssystem - wie Android, iOS und Windows Phone - um die gängigsten Betriebssysteme zu nennen, geschrieben worden ist. „Apps“ werden meistens in dem für das Betriebssystem vorhergesehenen nativen Code entwickelt.

Der Markt und die Variationen an mobilen Geräten und Betriebssystemen sind kontinuierlich dem Prozess des Wachstums unterzogen und die Komplexität der Entwicklung dieser „Apps“ wird sukzessive schwieriger. Die Aufgabenstellung ist es, ein möglichst großes Ausmaß an Plattformen mit adäquatem Aufwand zu bedienen. Hinsichtlich der ökonomischen Betrachtungsweise wäre es optimal, ein Maximum an gängigen, mobilen Geräten - wie auch Betriebssystemen - mit einem Minimum an Entwicklungsaufwand abzudecken. Dies wäre dann der Fall, wenn jene „Apps“ plattformunabhängig entwickelt werden könnten. Betrachtet man den aktuellen Smartphone-Markt, erkennt man, dass die drei oben erwähnten Smartphone-Betriebssysteme im Moment 96% des gesamten Marktes abdecken. Laut Zukunftsprognose werden die Marktanteile zwar leicht varieren, nichtsdestotrotz werden diese drei Global-Player auch in Zukunft den Markt dominieren.[1]

[1] *Mobile OS Statistik* 2014.

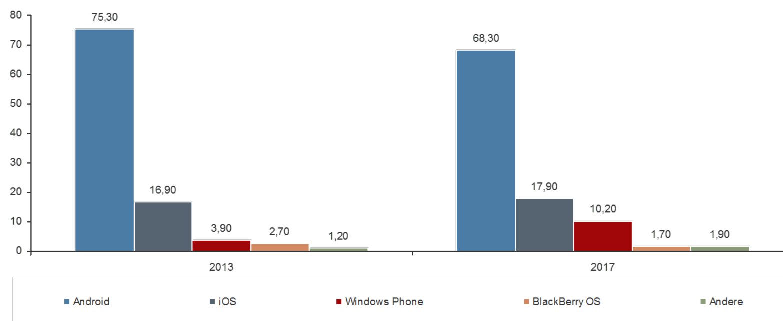

Abbildung 1.1.: Quelle: *Mobile OS Statistik* 2014

Das Testen von Software nahm im Laufe der Zeit und angesichts der steigenden Komplexität von Programmen einen immer höheren Stellenwert ein und ist aus dem heutigen Softwareentwicklungsprozess nicht mehr wegzudenken. Im Zuge der Innovationen innerhalb des Softwareentwicklungsprozesses, wie z.B. der agilen Vorgehensweise, kristallisierte sich die Methodik der testgetriebenen Entwicklung - im englischen auch unter TDD bekannt - heraus. Das Paradigma dieser Methode kennzeichnet sich dadurch, dass zuerst Tests geschrieben werden und im Anschluss der Programm- oder auch Sourcecode.

Durch die Kombination dieser 2 Thematiken, der plattformunabhängigen Entwicklung und dem Paradigma der testgetriebenen Entwicklung, ergeben sich neue Synergien im Bereich der Qualitätssicherung und dem Entwicklungsaufwand von plattformunabhängigen Applikationen.

1.1. Problemumfeld

Durch den zuvor erwähnten Innovationsfortschritt verschiedenster mobiler Betriebssysteme ergeben sich neue Anforderungen an die Applikationsentwicklung und an die Entwickler selbst. Dadurch entstehen auch mehrere Problemdomänen.

1.1.1. Plattformunterschiede

Im Unternehmensbereich stellt sich generell die Frage, auf welcher Plattform hauseigene Lösungen entwickelt werden. In den meisten Fällen wird speziell in Bezug auf die Entwicklungssprache der jeweiligen Entwickler auf unternehmenseigene Personalressourcen zurückgegriffen. Bei Java-orientierten Softwareentwicklungsunternehmen fällt daher die Entscheidung meist auf Android, bei Mac auf iOS mit objective C und bei Windowsentwicklern auf Windows Phone mit C#.

Das Wissen der Entwickler muss dementsprechend um die APIs erweitert werden, wie auch um die Designkonzepte des mobilen Betriebssystems Der Vorteil ergibt sich aus den Sprachkonventionen und der Technologie, die dadurch sich die Einarbeitungszeit verkürzen.

Für die plattformunabhängige Entwicklung stellt diese Unternehmensspezialiserung jedoch einen Nachteil dar. Unternehmen entscheiden sich für eine Technologie und versuchen in diesem Bereich Kernkompetenzen aufzubauen, Softwareentwickler zu akquirieren, eine Nische zu finden und möglichst homogene Systeme zu entwickeln.

Mit der Einführung von Apples iOS und etwas später Googles Android Betriebssystem musste unter den Unternehmen eine Entscheidung getroffen werden, welche Plattform in Zukunft unterstützt wird. Somit löste sich die bereits entstandene Homogenität im System auf und es wurden zusätzliche Entwickler mit anderer Basis- und Sprachkompetenz eingestellt.

Die Unterstützung von mehreren Plattformen wurde somit zu einer kostspieligen Angelegenheit, da sich auch die mobilen Betriebssysteme und die zugehörigen Software Development Kits (SDKs) kontinuierlich weiterentwickeln.[2]

1.1.2. HTML5 Limitierung

Die ersten Schritte hinsichtlich portablen, plattformunabhängigen Applikationen ging in Richtung Hyper Text Markup Language 5 (HTML5). Mitt-

[2] vgl. Olson u. a., 2012, S.63.

lerweile unterstützen alle aktuellen Browser Applikationen der mobilen Betriebssysteme HTML5, wodurch sich eine enorme Verbesserung in Bezug auf die Entwicklung von Applikationen herausstellt, vor allem durch die Unterstützung diverser Frameworks wie JQuery und SenchaTouch, die auf mobile Plattformen abgestimmt sind. Der Einsatzbereich solcher Frameworks bzw. HTML5 ist hinsichtlich der mobilen Umgebungen begrenzt. Der HTML5 Cache unterstützt zwar das Arbeiten im offline-Modus, erfordert allerdings entsprechenden Aufwand dies effizient und handhabbar umzusetzen. Um Hardwarekomponenten anzusprechen - wie z.B.: Kamera, Global Positioning System (GPS) oder Kompass - müssen Frameworks wie PhoneGap verwendet werden, welche einen nativen Zugriff auf die Hardware ermöglichen.

1.1.3. Hybride Annäherung

Eine weitere Möglichkeit, wie schon zuvor erwähnt, ist die Entwicklung einer hybriden Applikation. Um hier die gerätespezifischen Funktionalitäten über den Browser anzusprechen, wird meist JavaScript verwendet. Dies ist für Applikationen, welche generelle Anforderungen in Bezug auf die Hardware benötigen, eine gute Lösung. Im industriellen Bereich, beispielsweise in der Verwendung eines Barcodescanners, kann es in der Interaktion mit entsprechender Hardware zu Schwierigkeiten kommen. Hierfür muss das Plug-In in nativer Sprache entwickelt werden und Wissen über die SDK der Plattform vorhanden sein. Dies kann speziell bei plattformunabhängigen Applikationen zu Schwierigkeiten führen, da hier für jede Plattform das Modul nachimplementiert werden muss.

1.2. Ziele

Das Ziel der vorliegenden Masterarbeit ist es zu verifizieren, ob es möglich ist gegebene der Firma Comm-Unity Softwareanforderungen für die Smartphoneplattformen Android und Windows Phone mit Mono so umzusetzen, dass der Kern dieser Applikation eine gemeinsame Basis bildet und ausschließlich die plattformspezifischen Ansichten - pro Plattform - angepasst werden müssen. Desweiteren ist zu beachten, dass diese Applikation der Leistung einer nativen

Applikation nahe kommt. Ein Augenmerk ist auch auf die Speicherung und Verwaltung von Daten gerichtet, sodass sich daraus eine konsistente und ausfallssichere Applikation ergibt. Um den oben erwähnten Applikationskern fehlerfrei zu halten, wird versucht diesen testgetrieben entwickelt und Änderung der Anforderungen oder der Software anhand von Regressionstests erweitert.

Die Zielsetzung führt zur Formulierung folgender konkreter Vorgaben und Evaluierungszielen:

- Abbildung des Workflows der Applikation unabhängig von der Graphical User Interface (GUI)
- Effiziente Datenpersistierung
- Webbasierte Datentransaktionen
- Generischer Hardwarezugriff (z.B.: Kamera, GPS)
- Testbarkeit

2. Plattformgrundlagen

2.1. Android

Android ist mit Abstand das Betriebssystem, welches auf mobilen Geräten am häufigsten zum Einsatz kommt. Der Marktanteil liegt in Europa bei knapp 70%, weitabgeschlagen iOS mit 18% und dahinter Windows Phone mit 10%.[1]

Das Unternehmen Android wurde 2005 von Google Inc. gekauft und gab 2007 bekannt, ein neues Smartphonebetriebssystem zu entwickeln.[2] Die aktuelle Version des Android ist 4.4.X (KitKat).

Die erste Auslieferung von Androidgeräten im Jahr 2008 war nicht von Erfolgen geprägt, da Medien und die ersten Nutzer der Geräte, Android stark kritisierten. Dabei hat Android einige Vorteile gegenüber den anderen Plattformen, wie z.B. dass Android eine offene Plattform ist und somit frei und für jeden verfügbar. Dies ermöglicht verschiedensten Herstellern Android für ihre Geräte speziell zu adaptieren und zu portieren. Das wiederum führt dazu, dass es einen enormen Mitbewerb unter den verschiedensten Herstellern gibt und dies die Optimierung der Hardware signifikant antreibt.[3]

Nach vorangegangen Startschwierigkeiten wuchs Android und die Community kontinuierlich. Nach einer großen Anzahl an Beta-Releases 2007 und 2008 wurde die erste offizielle SDK 1.0 im September 2008 vorgestellt. Basierend auf dieser kamen eine Menge an weiteren SDK-Versionen dazu.[4]

Im Herbst 2009 wurde Android 2.0 (Eclair) als Betriebssystem released.[5]

[1] vgl. *Mobile OS Statistik* 2014, S.23.

[2] vgl. Shackles, 2012, S.2.

[3] vgl Meier, 2012, S.39.

[4] vgl. Shackles, 2012, S.2.

[5] vgl. Shackles, 2012, S.2.

2.1.1. Architektur

Android basiert auf einem Linux Kernel und einer Reihe von C/C++ Bibliotheken und stehen dem Applikationsframework zur Verfügung welches Services für das Managen der Laufzeitumgebung und der Applikationen bereitstellt.

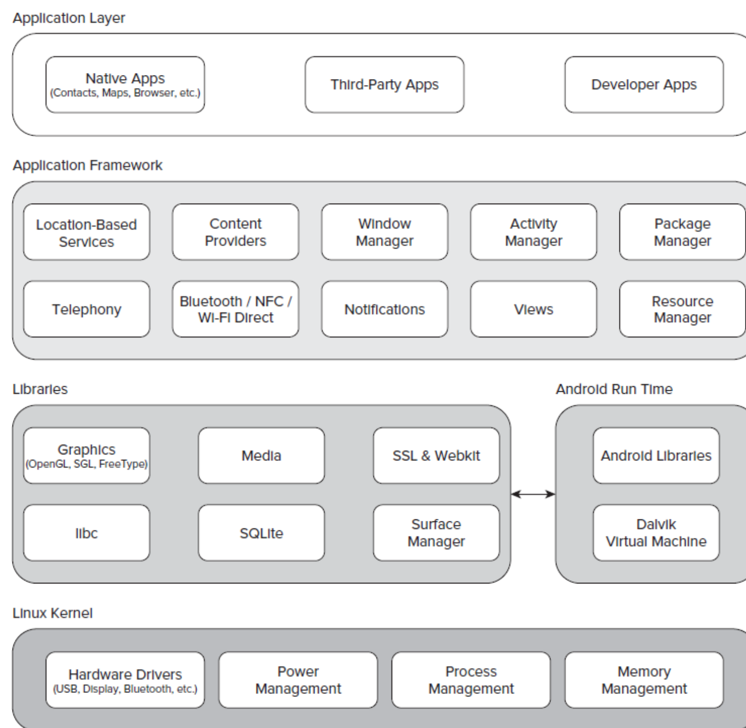

Abbildung 2.1.: Softwarestack Android, Quelle:vgl. Meier, 2012, S.16

Linux Kernel
Die Kernarchitektur basiert auf dem Linux Kernel 2.6. Somit werden die Kernservices (Memory Management, Security, Network und Power Management) von dieser Basis gehandhabt. Der Kernel bietet auch noch eine weitere Schicht (Layer) zwischen der Hardware und dem Rest des Software Stacks.[6]

[6]vgl. Meier, 2012, S.15.

Bibliotheken
Diese laufen an der Spitze des Kernels, Android inkludiert dabei C/C++ Kernbibliotheken wie libc, SSL und folgende:

- Eine Medienbibliothek zur Wiedergabe von Audio- und Videodaten
- Einen Oberflächenmanager zur Verwaltung der Displays
- Diverse Grafikbibliotheken wie SGL and OpenGL für 2D und 3D Grafiken
- SQLite zur Datenbankunterstützung
- Eine weitere Bibliothek zur Unterstüzung von Websicherheit und zur Darstellung von Websites[7]

Android Laufzeitumgebung
Die Android Laufzeitumgebung (Runtime) ist das Herzstück des Android Betriebssystems, dieses enthält die Bibliotheken für die Dalvik VM und die Dalvik VM selbst. Die Dalvik VM zeichnet Android aus, da dies eine eigene VM ist und als Hauptaufgabe die Prozessverwaltung und das softwareseitige Memorymanagement über hat.[8]

Applikationsframework
Auf dieser Ebene werden alle nötigen Bibliotheken zur Android-Applikationsentwicklung zur Verfügung gestellt. Unter anderem auch jene zur Kommunikation mit der Hardware.[9]

Applikationen
Dies ist die Schicht in dem sich die Android-Anwendungen selbst befinden. Dabei kann man noch die folgenden drei Kategorien unterscheiden:

- Mitgelieferte Anwendungen von Google
- Anwendungen von Drittanbietern
- Selbstentwickelte Anwendungen[10]

[7] vgl. Meier, 2012, S.15.
[8] vgl. Meier, 2012, S.15.
[9] vgl. Meier, 2012, S.15.
[10] vgl. Meier, 2012, S.15.

2.1.2. Dalvik VM

Eines der wichtigsten Elemente in Android ist die Dalvik VM. Anstatt die konventionelle Java VM für mobile Geräte zu verwenden, nutzt Android die Dalvik VM, welche im Gegensatz zur normalen stackbasierten VM, eine registerbasierende ist. Die Dalvik VM nutzt den darunterliegenden Linux-Kernel um auf "low-level"Funktionalitäten wie Sicherheit, Threading und Prozess- und Speicherverwaltung zuzugreifen.

Der gesamte Hardware- und Systemzugriff einer App erfolgt über die Dalvik VM. Die Dalvik VM führt spezielle Dalvik Dateien (.dex) aus, deren Format speicheroptimiert ist.

Jede App läuft in einem eigenen Linux-Prozess, in welchem die .dex-Datei von der Dalvik VM ausgeführt wird.[11]

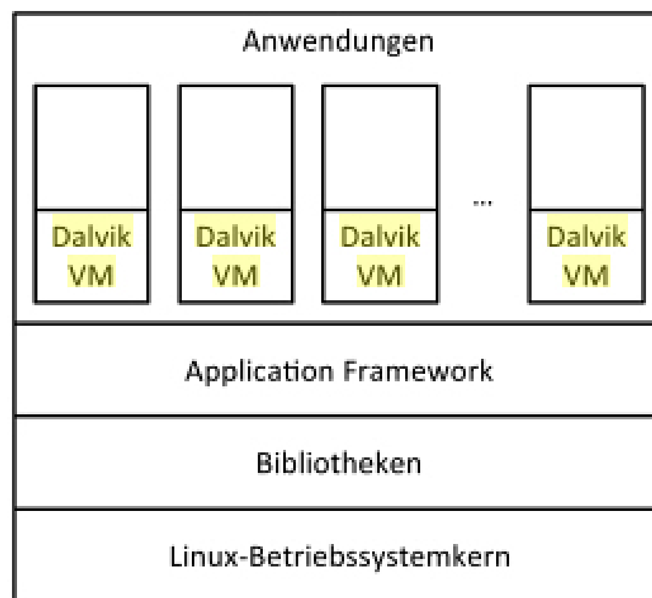

Abbildung 2.2.: Dalvik VM, Quelle: Oechsle, 2013, S.270

[11] vgl. Oechsle, 2013, S.270.

Es ist zwar möglich, effizientere Applikationen in C/C++ zu schreiben, Google empfiehlt dies nicht zu machen, da bestimme Sicherheitsmechanismen nicht mehr zur Verfügung stehen. Sollte dennoch die Anforderung bestehen, Apps in C/C++ zu schreiben, stellt Android ein Native Development Kit (NDK) zur Verfügung. Dieses kann verwendet werden um C++ Bibliotheken zu nutzen.[12]

2.1.3. Komponenten

Komponenten sind ein essentieller Teil von Android und jede einzelne Komponente hat ihren speziellen Einstiegspunkt in der Applikation. Nicht jede Komponente ist ein Einstiegspunkt für den Nutzer, hat aber ihre eigene Idendität und spielt eine wichtige Rolle im Gesamten.[13]

Activity

Eine Activity representiert eine einzelne Seite mit der zugehörigen Oberfläche. Sie könnten miteinander kommunizieren, agieren aber eigenständig und unabhängig von einander.[14]

Service

Die Komponente Service läuft im Hintergrund und ist dazu gedacht über einen längeren Zeitraum Aufgaben zu erfüllen, wie z.B. das sequentielle Abrufen von Daten. Die Absicht dahinter besteht darin, den Nutzer so wenig wie möglich zu beeinträchtigen. Falls sich die Applikation im Hintergrund befindet kann der Service auch weiterlaufen.[15]

---

[12] vgl. Meier, 2012, S.16.
[13] vgl. *Android Komponenten* 2014.
[14] vgl. *Android Komponenten* 2014.
[15] vgl. *Android Komponenten* 2014.

Content Provider

Ein Content Provider verwaltet geteilte applikationsspezifische Daten. Daten können in diversen Strukturen, wie Datenbanken, Dateien oder auch im Web gespeichert werden. Android selbst bietet als Beispiel zum bearbeiten der Kontaktinformationen einen Content Provider an.[16]

Broadcast receiver

Der Broadcast receiver dient zum systemweiten Empfang von Ankündigungen. Viele dieser Broadcast receiver entspringen aus dem System wie z.B. die Ankündigung des Akkustands. Eine Applikation kann auch einen Broadcast absetzen um andere über einen bestimmten Status zu informieren.[17]

2.1.4. Oberflächen Design

Die Oberfläche orientiert sich an dem Android-Standard des User Interface (UI)-Designs. Dieses kann in drei Elemente unterteilt werden:

Views
: Views sind Anzeigeklassen mit denen der spezifische Inhalt von Daten in mehreren Varianten angezeigt werden kann.

ViewGroups
: ViewGroups dienen zur Gruppierung von Views bzw. als Container.

Activities
: Activities sind jene Klassen die das Bindeglied zwischen den Views und den Daten bilden. In ihnen kann die Logik für die Anzeige abgebildet werden. Man kann sie als Pendant der Page-Klassen in Windows Phone Applikationen sehen.

2.1.5. Verknüpfung Layouts mit Activities

Mit View-Klassen kann man auf das in XML definierte Layout der Oberfläche zugreifen. Der Hintergrund - die Oberfläche in XML zu definieren - kommt aus dem Softwaredesignprinzip Geschäftslogik und Oberfläche so gut

[16] vgl. *Android Komponenten* 2014.
[17] vgl. *Android Komponenten* 2014.

als möglich zu separieren. Die oben erwähnte ViewGroup-Klasse dient als Container für die darin enthaltenen Views, welche im unteren Beispiel als LinearLayout-Element definiert ist. Diese Layouts werden im Projektverzeichnis im Unterordner "layout" in dem Ressourcen-Ordner erstellt. Das Layout wird in der Activity über den Methodenaufruf der Activity SetContentView, in der OnCreate Methode generiert.

```
protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            this.SetContentView(Resource.Layout.
            mainpage);
            Init();
        }
```

Die Oberfläche und das XML hierzu sehen folgendermaßen aus:

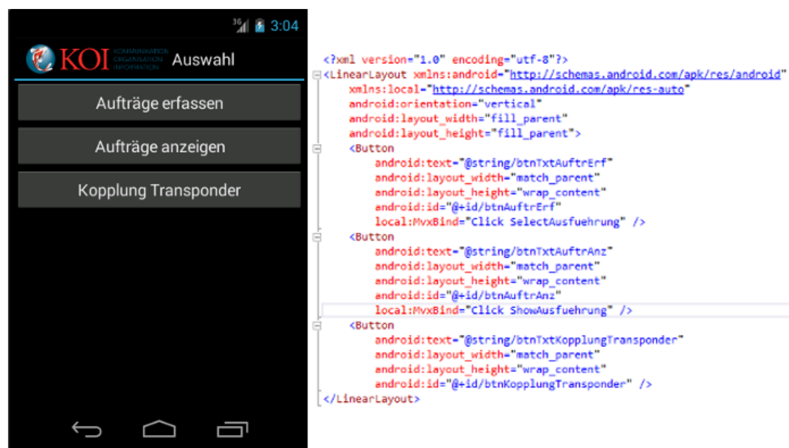

Abbildung 2.3.: Android Layout

2.2. Windows Phone

Windows Phone gilt als Nachfolger des Betriebssystems Windows Mobile, welches mit der Version 6.5.3 als letzte Version emittiert wurde und basierte im Gegensatz zu Windows Phone 8 auf dem Windows CE Kern. Windows Phone 7 wurde später als vorhergesehen auf den Markt gebracht und basierte, wie auch Windows Mobile, auf dem Windows CE Kern. Der Unterschied bestand jedoch in der API, welche sich nur in den Basisklassen von der .net Runtime überschnitten. Hinter Windows Phone 8 verbirgt sich der Windows NT Kern, genauso wie bei Windows 8 und Windows RT.

Windows Phone gilt als Antwort für die von Apple und Google auf den Markt gebrachten Smartphones iOS und Android. Nach den ersten Kinderkrankheiten, mit denen Windows Phone 7 und Windows Phone 7.5 ausgeliefert wurden, erfreute sich Windows Phone 8 immer größere Beliebtheit bei Anwendern wie auch Entwicklern, da Performance und Benutzerfreundlichkeit von Microsoft an die erste Stelle gerückt sind.

2.2.1. Architektur

Die Architektur von Windows Phone 8 erschließt sich aus mehreren logischen Ebenen.

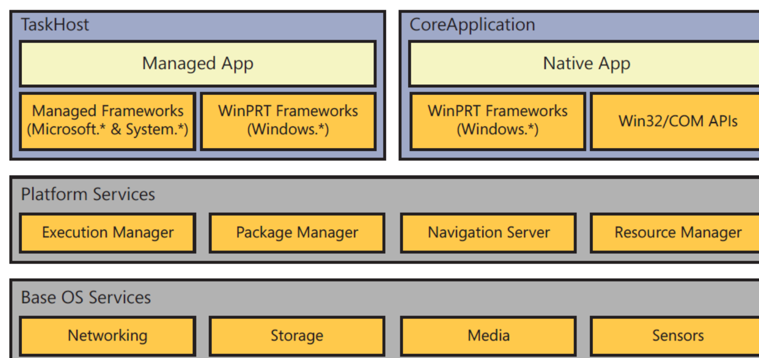

Abbildung 2.4.: Softwarestack Windows Phone, Quelle:vgl. Andrew Whitechapel, 2012, S6.

Die oberste Ebene untergliedert sich in zwei Elemente, welche auch als App-Modell bezeichnet werden. Auf der einen Seite befindet sich das TaskHost-Modell das mit dem Start von Windows Phone 7 eingeführt wurde. Auf der anderen Seite befindet sich das CoreApplication-Modell welches dem Entwicklungs-Modell von Windows 8 Applikationen angeglichen wurde und ausschließlich für die Entwicklung von Direct3D-Apps gedacht ist.[18]

Beide Modelle basieren auf verschiedensten Plattformservices. Für Managed-Code Apps sind diese nicht immer ersichtlich, da sie von anderen Bibliotheken gekapselt und somit indirekt genutzt werden. Allerdings sind sie für diese Apps ein essentieller Bestandteil um einen stabilen und flüssigen Verlauf zu gewährleisten.[19]

Execution Manager

Der Execution Manager verwaltet die Laufzeitumgebung der einzelnen Apps und kümmert sich um das Event-Handling in Bezug auf startup, shutdown und deactivation der App. Eine zusätzliche Aufgabe ist noch die Verwaltung von Hintergrundprozessen und das damit verbundene Scheduling.[20]

Package Manager

Die Durchführung von Installation/Deinstallation übernimmt der Package Manager, wie auch die Verwaltung der zugehörigen Metadaten über den gesamten Lebenszyklus der App. Falls die App noch zusätzlich Kacheln (Tiles) verwendet, die am UI angezeigt werden, fällt dies ebenso in den Aufgabenbereich dieses Managers.[21]

Navigation Server

Der Navigation Server kümmert sich um die Navigation jener Apps die sich im Vordergrund befinden bzw. auch um die Reihenfolge, in der sie gestartet worden sind. Das heißt, wird eine App gestartet bereitet der Navigation Server alles für den Execution Manager, für die Ausführung der App vor.

[18] vgl. Andrew Whitechapel, 2012, S.6.
[19] vgl. Andrew Whitechapel, 2012, S.7.
[20] vgl. Andrew Whitechapel, 2012, S.7.
[21] vgl. Andrew Whitechapel, 2012, S.7.

Dies ist auch der Fall, falls eine bereits laufende App durch das Drücken der Back-Taste ausgewählt wird.[22]

Resource Manager

Der Resource Manager sorgt dafür, dass auf die Systemressourcen möglichst effektiv zugegriffen werden kann, indem er speziell die CPU und das Speicherverhalten beobachtet und eingreift.[23]

2.2.2. Modern-UI

Die Oberfläche von Windows Phone unterliegt einigen fundamentalen Design-Prinzipien, welche für den Entwickler, wie auch den Anwender von Vorteil sind.

Leicht und Einfach

Das Design von Windows Phone 8 soll ein Durcheinander verhindern und dem User helfen, sich auf seine Absichten und Aufgaben zu fokusieren. Als Grundlage dieses Designs dient ein bekanntes Beispiel aus dem öffentlichen Nahverkehr und zwar die U-Bahn. [24] Der U-Bahnnahverkehr fertigt täglich unmengen an Personen ab und bringt diese in kürzerster Zeit zum Ziel. Auf Basis dieser Strukturen und Methodiken entwickelte sich das Design. Windows Phone filtert auschließlich die wichtigsten Informationen aus tausenden von Daten heraus und hält die Oberfläche so simpel wie möglich um den User so effizient wie möglich zu seinem Ziel zu führen.[25]

Typographie

Ein wichtiges und unumgängliches Element im Oberflächen Design ist der Text selbst. Nur all zu oft wird Text auf eine sehr uninteressante Art und Weise repräsentiert, da man hier eher auf den Inhalt der Information blickt, als auf die Aufbereitung dieser. Windows Phone nutz einen signifikanten Font

[22]vgl. Andrew Whitechapel, 2012, S.7.

[23]vgl. Andrew Whitechapel, 2012, S.7.

[24]Ursprünglich nannte sich das Design von Windows Phone auch Metro-Design, auf Grund rechtlicher Differenzen mit einer gleichnamigen Großhandelskette verlor Microsoft das Namensrecht und blieb bei dem Begriff Modern-UI für das Design.

[25]vgl. Andrew Whitechapel, 2012, S.3.

namens WP Segoe für das gesamte UI. Für die Entwicklung werden mehrere Style-Standards zur Verfügung gestellt.[26]

Bewegung

Die Bewegungen flüssig und natürlich zu gestalten ist Hauptaugenmerk bei Windows Phone. Das wechseln zwischen den einzelnen Seiten der Apps, wie auch zwischen den Apps selbst.[27]

Inhalt

Ein wesentlicher Punkt unter Windows Phone ist der Inhalt verknüpft mit einem Microsoft-Account. Ohne verbundenem Account ist die Nutzung des Windows Phone nur eingeschränkt möglich. Bei Verbindung mit einem gültigen Microsoft-Account können Kontakte, Kalender, e-Mails, Dokumente synchronisiert und bearbeitet werden.[28]

2.2.3. XAML und Pages

Extensible Markup Language oder kurz auch Extensible Application Markup Language (XAML) genannte ist die Oberflächen-Beschreibungssprache von Microsoft. Mit XAML ist es möglich, Objekte zu instanziieren, wie auch Eigenschaften von Objekten zu setzen und dies in Form einer hierachisch aufgebauten Struktursprache anzuzeigen. Einer der größten Vorteile welche die XAML-technologie mit sich bringt ist die Separierung von Oberfläche(XAML) und Code. Daraus ergibt sich die Option Entwicklung und Design nahezu vollständig zu trennen.[29]

Hier als Beispiel eine Windows Phone Page in XAML definiert:

[26]vgl. Andrew Whitechapel, 2012, S.3.
[27]vgl. Andrew Whitechapel, 2012, S.3.
[28]vgl. Andrew Whitechapel, 2012, S.3.
[29]vgl. *XAML for Windows Phone 8* 2014.

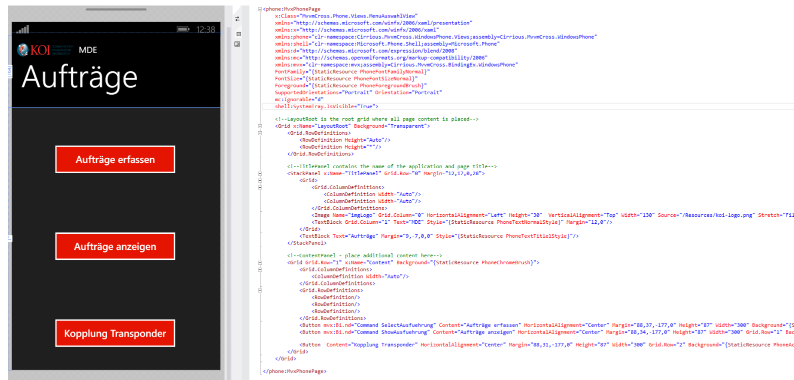

Abbildung 2.5.: Windows Phone Seite

3. Mono

Der Grundgedanke von Mono ist die plattformunabhängige Entwicklung auf Basis der Programmiersprache C# und der Klassenbibliothek von .net. Es ist ein OpenSource Projekt, das auf dem Ecma International (ECMA) Standard für C# und der Common Language Runtime (CLR)) aufgebaut ist. Den Startschuss der Entwicklung dieses Projekts setzte Novell, um C# auch für Linux verwenden zukönnen. Auf Basis dessen ergaben sich viele Erweiterungen wie Mono für Android, MonoTouch und Mono für Spielkonsolen.[1]

3.1. Laufzeitumgebung

Die Laufzeitumgebung ist in C/C++ entwickelt worden und implementiert die Common Language Infrastructure (CLI) als virtuelle Machine.

Mono beinhaltet folgende Services:

- Code Ausführung
- Garbage Collection
- Code Generierung
- Exception Handling
- Betriebssystemschnittstellen
- Programmisolierung unter Nutzung von Applikationsbereichen
- Thread Mangement
- Konsolen Zugriff
- Sicherheitssystem

[1] vgl. Peppers, 2014, S.6.

3.2. .net Base Class Library

Die Base Class Library (BCL) bildet die Basis der Klassenbibliotheken für die Mono Laufzeitumgebung. Sie beinhaltet die wichtigsten Funktionalitäten unter .net - Dateizugriff, Sicherheitsattribute, string-Manipulierung, Streams, Collections und einige mehr - wie auf folgender Abbildung sichtbar ist.

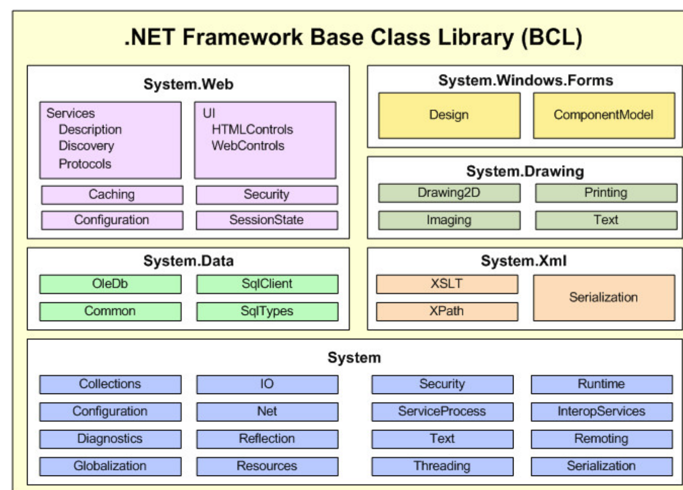

Abbildung 3.1.: Base Class Library, Quelle:*Base Class Library* 2014

Das Codebeispiel zeigt den Zugriff auf eingebettete Ressourcen in Windows Phone und Android, welcher ein Stream-Objekt zurückliefert.

Windows Phone

```
...
using System.IO;
using System.Reflection;
...
Stream content = Assembly.GetExecutingAssembly().
GetManifestResourceStream("MyProject.Phone.Assets
.content.txt")
...
```

Android

```
...
using System.IO;
using System.Reflection;
...
 Stream content = Assembly.GetExecutingAssembly().
 GetManifestResourceStream("MyProject.Android.Assets
 .content.txt")
...
```

3.3. Mono für Android

Mono für Android stellt die spezifische Erweiterung für die Android API in .net zur Verfügung. Damit ist es möglich native Android-Applikationen zu schreiben, die selben UI-Elemente wie in Java zu verwenden und den gesamten Umfang der .net Base Class Library zu verwenden.

Die Mono-Laufzeitumgebung, wie auch die Dalvik VM laufen auf der selben Ebene und bieten über die gegebenen APIs Zugriff auf den darunter liegenden Linux Kernel. Der Zugriff auf Android-Funktionalitäten findet über die Dalvik VM und die entsprechenden Namespaces Android.* bzw. Java.* statt. Auf die .net-Funktionalitäten wie System, System.IO, usw. kann über die Mono-Laufzeitumgebung zugegriffen werden. Die Bindeglieder zwischen den beiden Laufzeitumgebungen bilden die Managed Callable Wrappers (MCWs) und die Android Callable Wrappers (ACWs).[2]

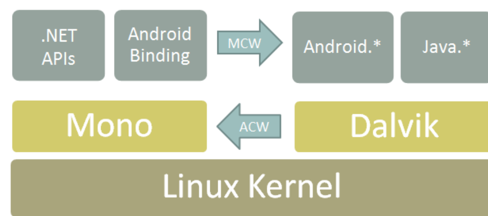

Abbildung 3.2.: Mono for Droid, Quelle: *Mono for Droid* 2014

[2] vgl. Olson u. a., 2012, S.13.

3.3.1. Android Callable Wrappers

Als ACW werden die Schnittstellen bezeichnet, mit denen die Android Laufzeitumgebung auf Managed Code über Java Native Interface (JNI)) zugreift.[3]

3.3.2. Managed Callable Wrappers

MCWs fungieren als JNI-Schnittstelle um vom Managed-Code Android-Funktionalitäten aufzurufen und stellt das Implementieren von Java-Klassen, wie auch das Überschreiben von virtuellen Methoden, zur Verfügung. Die gesamte Android-API, welche über den Namespace "Android.*" angesprochen werden kann, sind MCWs.

Jeder der MCWs enthält eine globale Java-Referenz, die über die Android.Runtime.IJavaObject.Handle-Eigenschaft zugreifbar ist. Mit diesen globalen Referenzen ist das zuordnen von Java-Instanzen zu Managed-Instanzen erst möglich.

3.4. Xamarin

Xamarin ist ein amerikanisches IT-Unternehmen und wurde im Mai 2011 gegründet. Xamarin gilt als kommerzielle, treibende Kraft bei der Entwicklung der Mono-Plattform, wie auch der plattformspezifischen API in .net von Android und iOS. Xamarin setzt sich zum Hauptziel, den Entwickler zu entlasten und effizientere mobile Applikationen zu entwickeln, ganz nach dem Motto:"Write-Once-Run-Anywhere".[4]

3.4.1. Xamarin Platform

Xamarin stellt unter dem Namen "Xamarin Platform" eine Reihe von Entwicklungstools, APIs und third-party Bibliotheken zur Verfügung.

---

[3] *Mono for Droid* 2014.
[4] vgl. Xamarin, 2013, S.17.

Abbildung 3.3.: Xamarin Platform, Quelle: Xamarin, 2013

Basis dieser Plattform bildet die Mono-Laufzeitumgebung mit der Implementierung von .net. Es wird ein eigenes Entwicklungsstudio (Xamarin Studio) und ein Plug-In für Visual Studio (ab 2012) zur Verfügung gestellt. Der "Component Store"von Xamarin gibt die Möglichkeit weitere Extensions bzw. auch Bibliotheken, wie z.B. den SAP-Client (nur Enterprise-Edition) herunterzuladen und zu nutzen.

3.5. Code-Sharing

Code-Sharing bezeichnet das gemeinsame Verwenden von Code-Stücken außerhalb des ursprünglichen Projekts. Das Teilen von Code-Stücken ermöglicht die Mehrfachnutzung von implementierten Methodiken, Modulen und Klassen und erhöht somit den Grad der Portabiliät der Applikation. Die Portabiliät ist eine wichtige Messgröße bei der Entwicklung von plattformunabhängignen Applikation und die Wahl der Architektur ist dabei maßgebend.[5]

[5] Shackles, 2012.

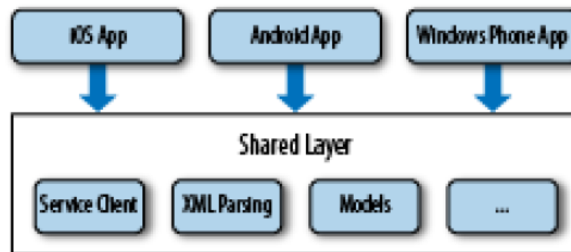

Abbildung 3.4.: Shared Project, Quelle: Shackles, 2012

Zu Unterscheiden ist hier ob Sourcecode im ursprünglichen Sinn oder bereits vorkompilierte Bibliotheken(dll) geteilt werden.

Xamarin bietet zur Unterstützung 2 Möglichkeiten an:

- Shared Projects
- Portable Class Library

3.5.1. Shared Projects

Die Variante des "Shared Projects" ist die simpelste Variante Sourcecode, bzw. deren Source-Dateien zu teilen. Hierbei werden die Source-Dateien in das plattformspezifische Projekt gelinkt und erst beim Kompilieren der Applikation mitintegriert. Bei dieser Form von von Code-Sharing können auch Compiler-Anweisungen eingefügt werden, um plattformspezifische Modifikationen vorzunehmen. Als Output wird die zu kompilierende App generiert.

3.5.2. Portable Class Library

Eine Portable Class Library (PCL) ist eine vorkompilierte DLL die in plattformspezifischen Projekten integriert werden kann. Der Unterschied zu den "Shared Projects" besteht darin, dass die PCL auf die Zielplattformen abgestimmt sein muss, da jede Plattform Differenzen im Subset der BCL hat.

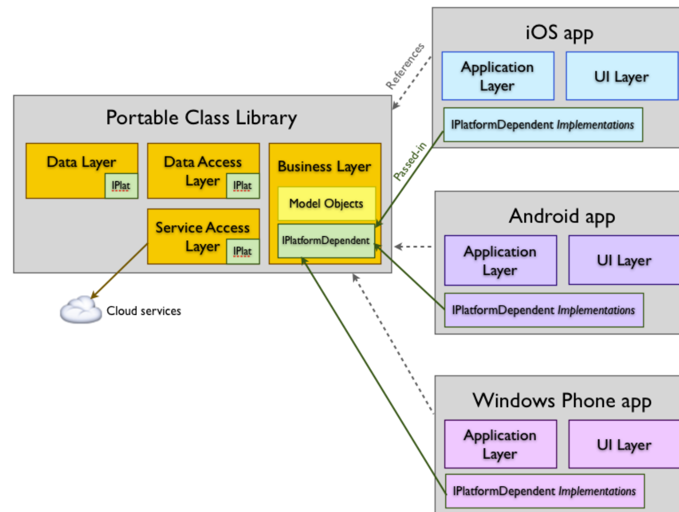

Abbildung 3.5.: PCL, Quelle: *Code-Sharing* 2014

3.6. MvvmCross

MvvmCross ist ein Framework zur Unterstützung der plattformunabhängigen Entwicklung mit C#. Das aus einer globalen Gemeinschaft aufgebaute und entwickelte Framework hat strikte Kodierungs- und Designprinzipien.

- Portablilität - Das Nutzen von Portable Class Librarys (PCLs) steht im Vordergrund, sowie auch die Separierung von Model, ViewModel und auch View
- Interface Driven Development - Um möglichst plattformunabhängige und robuste Apps zu entwickeln soll nach dem Prinzip Inversion of Control und Dependency Injection vorgegangen werden.
- Kodieren zum Testen - Es soll möglichst unabhängig, gekapselt und modular entwickelt werden. Nach dem Prinzip offen für Erweiterungen geschlossen für Veränderung.
- Mvvm - Es sollen Entwurfsmuster verwendet werden, im speziellen das Model-View-ViewModel gekoppelt mit dem Databinding. Dadurch wird die App strukturiert und übersichtlicher.

- Nativ UIs - Dem Benutzer steht die ihm bekannte native Oberfläche zur Verfügung
- Deine Meinung zählt - Es soll dem Entwickler möglich sein, alle Teile des MvvmCross Frameworks zu überschreiben.
- Die App ist König - Auslieferung ist alles.[6]

3.6.1. Model-View-ViewModel

Das Model View Controller (MVC) Entwurfsmuster wurde in den 80iger Jahren eingeführt und war ein Meilenstein im Design von Software. Der Grundgedanke dahinter ist das Splitten von Daten(Model), der Logik (Controller) und der Oberfläche(View). Die Oberfläche nimmt die gesamten Benutzereingaben entgegen und benachrichtigt den Controller, welcher widerum gegebenenfalls die Daten modifiziert. Mit der Zeit änderte sich die Architektur des MVC Entwurfsmuster in Richtung des Model View Presenter (MVP). Die Idee hinter dem MVP Entwurfsmuster ist die selbe wie bei dem des MVC, jedoch besteht hier der Unterschied, dass das Datenmodell auschließlich mit dem Presenter (Logik) kommuniziert.[7]

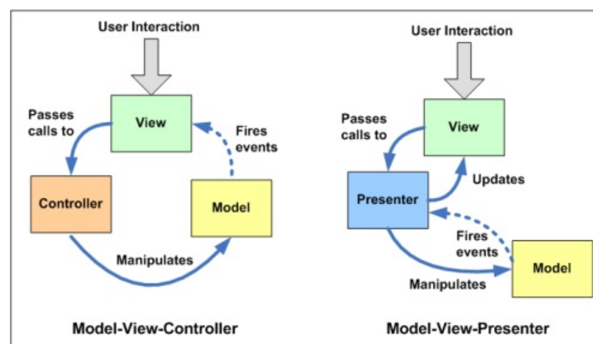

Abbildung 3.6.: MVC - MVP

[6] *MvvmCross Prinzipien* 2014, vgl.
[7] vgl. Esposito, 2012, S. 348.

Das MVP-Entwurfsmuster ist sehr generisch aufgebaut und kann nahezu auf jede Applikation adaptiert werden, ob Desktop, Mobile, oder Web. Microsoft hat sich das Entwurfsmuster zu nutzen gemacht und zusätzlich noch die XAML-Technologie integriert und dieses Entwurfsmuster MVVM getauft.

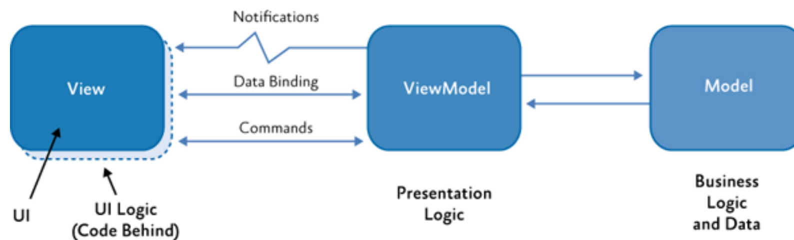

Abbildung 3.7.: Quelle:*MVVM-Entwurfsmuster* 2014

3.6.2. Service Location & Inversion of Control

2 wichtige Prinzipien des MvvmCorss sind:

- Service Location
- Inversion of Control

Registrierung und Auflösung

Die grundlegend Idee von Service Location ist das unabhängige schreiben von Klassen und Interfaces. In der statischen Klasse Mvx können alle Interfaces registriert und aufgelöst werden. Unter Auflösung versteht man hier das Abrufen des konkreten Objekts, mit dem zugehörig implementierten Interface.

```
public interface IDoThis
{
    void DoThis();
}

public class DoThisConcrete : IDoThis
```

```
{
 public void DoThis()
 {
     //Do what you want
 }
}
```

Um dieses Interface zu registrieren gibt es unter anderem die *Singleton Registration*.

Um das Objekt der oben erwähnte Klasse zu registrieren kann folgende Methode verwendet werden:

```
Mvx.RegistrationSingleton<IDoThis>(new
   DoThisConcrete());
```

Um das Objekt an einer beliebigen Stelle des Programms abzurufen kann folgende Methode verwendet werden:

```
var dothis = Mvx.Resolve<IDoThis>();
```

Mit dieser Variante der Registration und Auflösung wird das Objekt einmal erzeugt und immer wieder die selbe Referenz beim Aufruf der Resolve-Methode zurückgegeben.

Eine andere Möglichkeit der Registrierung mit dem selben Effekt ist folgender:

```
Mvx.ConstructAndRegisterSingleton<IDoThis,
   DoThisConcrete>();
```

Falls das Objekt einer Klasse erst bei dem ersten Aufruf der Resove-Methode generiert werden soll, gibt es die Möglichkeit der *Lazy-Singleton* Registrierung.

```
Mvx.RegistrationSingleton<IDoThis>(() => new
   DoThisConcrete());
```

Wie zuvor wird nach dem ersten Aufruf der Resolve-Methode jedes Mal das selbe Objekt zurückgegeben.

Für den Fall das nicht immer die selben Implementierungen eines Interfaces benötigt werden kann über *Last-registered* Abhilfe geleistet werden.

```
Mvx.RegisterType<IDoThis, DoThisConcrete>();
Mvx.RegisterType<IDoThis>(new DoThisConcreteTo())
Mvx.RegsiterType<IDoThis, DoThisConcrete3>();
```

Bei dem Aufruf der Resolve-Methode wird das zuletzt angemeldete Objekt retourniert. Dies kann unter Umständen dann benötigt werden, falls eine Standard-Implementierung überschrieben werden muss, wie Eingangs bei den Grundprinzipien des MvvmCross erwähnt wurde (Deine Meinung zählt).

Eine zusätzliche Funktionalität des Frameworks ist die *Bulk Registration by Convetion*. Mit dieser ist es möglich über Reflection alle generierbaren Klassen, welche einen Standard-Konstruktur besitzen und nicht abstrakt sind über folgendes Codestück automatisch zu registrieren.

```
CreatableTypes()
            .EndingWith("Service")
            .AsInterfaces()
            .RegisterAsLazySingleton();
```

In diesem Aufruf werden alle Klassen registriert, welche mit Service enden und ein gegebenes Interface implementieren.

Constructor Injection

Die statische Klasse Mvx verfügt über einen weiteren hilfreichen Mechanismus und zwar einen Reflection basierenden. Dieser wird zur automatischen Auflösung von Parametern während der Objektgenerierung benutzt.

Als Beispiel die Klasse Worker:

```
public class Worker
{
   public Worker(IDoThis dothis)
   {
     //do work
   }
```

```
}
```

Um dieses Objekt zu erzeugen, kann diese Anweisung ausgeführt werden:

```
Mvx.IocConstruct<Worker>();
```

Diese Funktionalität wird intern von MvvmCross genutzt, wenn der Konstruktor eines ViewModels aufgerufen wird.

```
 private readonly IGeoCodingService _geocodingService;

public GeoLocationViewModel(IGeoCodingService service)
            : base()
        {
            geocodingService = service;
        }
```

Die Mvx Klasse löst das IGeoCodingService-Objekt auf, wenn das ViewModel erzeugt wird.

Diese Auflösung des Interfaces über den Konstruktor ist ein Grundbaustein des Konzepts für Inversion of Control (IoC) da,

- ... es damit möglich ist verschiedenste Klassenimplementierungen des Interfaces IGeoCodingService zu registrieren und damit plattformunabhängig zu gestalten.
- ... eine Mock-Klasse für Unit-Tests leicht implementiert werden kann.
- ... Standardimplementierungen leicht überschrieben werden können.

IoC und Plattformspezifische Implementierung

Soll eine plattformspezifische Implementierung eingebunden werden, kann in der Setup-Klasse die konkrete Implementierung registriert werden. Zur Erläuterung der plattformspezifischen Implementierung von IoC ein kurzes Beispiel:

Die Implementierung des Interfaces `IFileContentProvider` in Android.

```
namespace MvvmCross.Android
{
    public class FileContentProvider :
       IFileContentProvider
    {
        public Stream GetFileContent(string path)
        {
            return
               Assembly.GetExecutingAssembly().GetManifestResourceStrea
        }
    }
}
```

Die Registrierung des Interfaces in der Setup-Klasse des Android-Projekts.

```
 protected override void InitializeLastChance()
{
  Mvx.RegisterSingleton<IFileContentProvider>(new
     FileContentProvider());
  base.InitializeLastChance();
}
```

3.6.3. Databinding

Databinding wird genutzt um die Separierung von Logik und Benutzeroberfläche, im Sinne des Mvvm-Entwurfsmusters, optimal umzusetzen. Das Databinding in MvvmCross wurde ursprünglich eingeführt um sich Microsofts XAML anzunähern und sich dieser Struktur der Oberflächenbeschreibung anzupassen. Sukzessive wurde diese Oberflächenbeschreibungssprache syntaktisch und um einige Ausdrücke erweitert.[8]

[8] *MvvmCross Databinding* 2014, vgl.

MvvmCross Databinding

Das Databinding in MvvmCross ist eines der Herzstücke des gesamten Projekts und im Laufe der Zeit haben sich mehrere Varianten gebildet und wurden weiterentwickelt.

JSON Die erste Version dieser Beschreibungssprache in MvvmCross basierte auf JSON. In der neuesten Version dieses Frameworks v3 steht die Möglichkeit der Definition der Oberfläche über JSON allerdings nicht mehr zur Verfügung.

SWISS Swiss war der direkte Nachfolger von JSON als Beschreibungssprache und stellte die selbe Funktionalität zur Verfügung, ist syntaktisch jedoch klarer und lesbarer definiert.

Als Beispiel der Vergleich zwischen JSON und Swiss:

```
JSON-Style

 {'DrawableId' { 'Path':'State', 'Converter':'StateConverter'}}

Swiss-Style

DrawableId State, Converter=StateConverter
```

FLUENT

Um Text-Formatierungsvarianten in C# zur Verfügung zustellen, welche über XML bereits möglich sind, wurde als Erweiterung das Fluent-Binding eingeführt. Mit dieser Binding-Option ist es möglich Eigenschaften von Steuerelementen (Button, TextView, TextBox, etc...) mit dem ViewModel zu verbinden.

Als Beispiel der Vergleich zwischen XML-Binding und Binding in C#-Code:

```
XML-Style

Text State, Converter=StateConverter

C#-Code
```

```
this.CreatingBinding(label) .For(l => l.Text) .To(vm => vm.Name)
      .WithConversion("StateConverter");
```

TIBET

Das Tibet-Binding beinhaltet einige neue Ideen in Bezug auf Databinding und ist eine Erweiterung des Swiss-Binding.

Die Aspekte des Tibet-Binding untergliedern sich wie folgt:

- Multi-Binding
- Wertkombinierung
- Literales-Binding
- Macro-Binding
- Funktionale Erweiterung für ValueConverters
- Eingebettete Wertkonvertierungen

Als Beispiel für ein Literal-Binding:

```
Value Format('My Name is {1} and I am {0} years old', 10,
FirstName)
```

RIO

Bei der Verwendung von ViewModels, ganz nach dem Mvvm-Entwurfsmuster, steht in C# vorallem das `INotifyPropertyChanged` Interface im Mittelpunkt.

Eine Standardimplementierung dieses Interfaces sieht folgendermaßen aus:

```
private string _state;
public string State
{
    get { return _state; }
    set
    {
        if (value != _state)
        {
            _state = value;
            RaisePropertyChanged("State");
```

```
            }
        }
    }
```

Mit dem Rio-Binding kann die selbe Funktionalität mittels FieldBinding um einiges kürzer implementiert werden.

```
public readonly INC<string> States = new
   NC<string>();
```

Der Zugriff erfolgt mit dem Aufruf der Value-Eigenschaft, in diesem Fall mit `States.Value`.

Der zweite imense Vorteil des Rio-Binding ist das MethodBinding. Der generelle Fall des Bindings von Methoden an Events eines Steuerelements im GUI erfolgt über ein `ICommand` Interface. Als Beispiel der konventionelle Fall des Bindings von Methoden.

```
private ICommand _selectAusfuehrungCmd;
public ICommand SelctAusfuehrung
{
    get
    {
    if(_selectAusfuehrungCmd == null)
        _selectAusfuehrungCmd
    = new MvxCommand(SelectAusfuehrung);
         return _selectAusfuehrungCmd;
    }
}
```

Mit dem MethodBinding ist der Aufruf ohne Deklaration eines `ICommand` Interfaces möglich indem auschließlich im XML der Event des Steuerelements mit der Methode verknüpft wird.

Als Beispiel die Methode:

```
public void SelectAusfuehrung()
{
  ShowViewModel<SelectAusfuehrungViewModel>();
}
```

Wird im XML auf diese Weise verknüpft:

```
Click ShowAusfuehrung
```

3.6.4. Testen

Um Klassen zu testen ist der MvvmCross-Aufbau optimal ausgelegt mit den Prinzipien der ViewModel und dem IoC-Container.

Zum Testen kann NUnit als Testframework, wie auch das Microsoft (MS)-Testframework verwendet werden. Dies untersützt das Testen von .net 4.5 PCL-Projekten. Um Objekte zu mocken, kann das Moq-Framework genutzt werden.

Setup

Bei der Verwendung von MvvmCross gibt es in jedem plattformspezifischen Projekt eine Setup-Klasse, um spezifischen Implementierungen von Interfaces, wie bei dem IoC-Prinzip erwähnt, zu registrieren. Ebenso muss auch das Test-Projekt initialisiert werden, das geschieht über folgendem Weg:

```
using Cirrious.MvvmCross.Test.Core;
using Moq;
using NUnit.Framework;

[TestFixture]
public class SpecificTestClass : MvxIoCSupportingTest
{
    [Test]
    public void ViewModelTest()
    {
```

```
        base.Setup();

    }
}
```

Ein weiterer wichtiger Punkt ist das Registrieren von den zusätzlichen Objekten, welche als Interfaces in den ViewModels, oder speziell von der Methode Resolve, der Klasse Mvx, aufgerufen werden.

```
protected override void AdditionalSetup()
{
    Ioc.RegisterSingleton<SQLiteConnection>(new
       Connection("test.sqlite"));
    Ioc.RegisterSingleton<AppContainer>(new
       AppContainer(Ioc.Resolve<SQLiteConnection>()));
    Ioc.RegisterSingleton<IPopUpDialog>(new
       PopUpMock());

    MockDispatcher = new Test.MockDispatcher();
    Ioc.RegisterSingleton<IMvxViewDispatcher>
      (MockDispatcher);
    Ioc.RegisterSingleton<IMvxMainThreadDispatcher>
      (MockDispatcher);

    // for navigation parsing
    Ioc.RegisterSingleton<IMvxStringToTypeParser>(new
       MvxStringToTypeParser());

}
```

Wie in dem oben ersichtlichen Sorucecode muss noch der Dispatcher für das Verwalten der ViewModels überschrieben werden.

```
 public class MockDispatcher : MvxMainThreadDispatcher
          , IMvxViewDispatcher
    {
        public readonly List<MvxViewModelRequest>
           Requests = new List<MvxViewModelRequest>();
```

```
    public bool RequestMainThreadAction(Action
       action)
    {
        action();
        return true;
    }

    public bool ShowViewModel(MvxViewModelRequest
       request)
    {
        Requests.Add(request);
        Current = request;
        return true;
    }

    public bool
       ChangePresentation(MvxPresentationHint
       hint)
    {
        throw new NotImplementedException();
    }

    public MvxViewModelRequest Current { get;
       private set; }

}
```

4. Testen

Der Begriff des Testens selbst kommt ursprünglich aus dem Bereich der Qualitätssicherung. Das Testen in der Softwareentwicklung hat den Ursprung vor ca. 50 Jahren und hat sich im Laufe der Zeit auf Grund der Anforderungen der Wirtschaft und der Unternehmen, wie auch der Veränderung der Softwaretechnologie selbst einem kontinuierlichem Wandel unterzogen.

4.1. Konventioneller Softwareentwicklungsprozess

Die Zeit, als die Softwareentwicklung für die Industrie interessant wurde, war jene zu der die Mainframes aufkamen. Jeder Hardwareanbieter hatte sein eigenes System und sein eigenes Schema Software zu entwickeln. Hin und wieder waren sich diese Systeme sehr ähnlich, wie auch die Jobs, die darauf liefen. Somit konnten sich auch die Entwickler ohne horrenden Aufwand auf diese Systeme einstellen und mit diesen entwickeln. Allerdings war es sehr mühsam mit solchen Systemen einfach Programme zum Laufen zu bringen, da man unzählige Zeilen von Code benötigte. Sehr oft kam es vor, dass dieselbe implementierte Logik in einer anderen Sprache oder Plattform nicht auf diese Weise funktionierte, wie es die ursprüngliche Version tut. Da solche Mainframes ein großer Kostenfaktor in der Anschaffung und Wartung waren, konnte sich nicht jedes Unternehmen dieses Stück Technologie leisten. Dies war die Geburtsstunde von service-orientierten Dienstleistungen.

4.1.1. Wasserfallmodell

Das Prinzip, das hinter dem Wasserfallmodell steckt ist, dass jedes Softwareprojekt in verschiedenste Phasen eingeteilt werden sollte. Von der Analysephase bis hin zur Wartung wird das Projekt vom Start weg unterteilt.

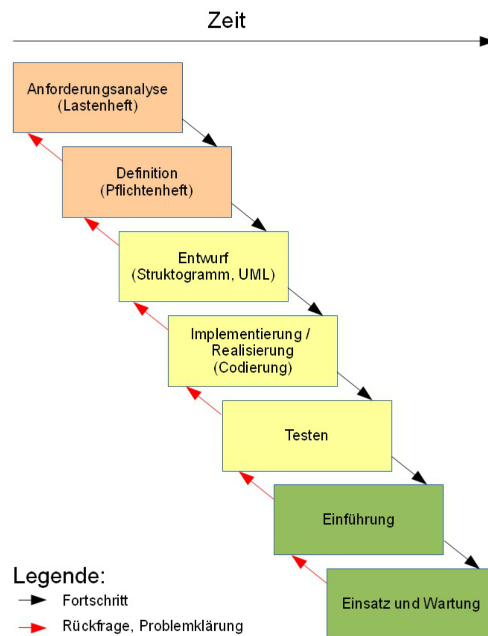

Abbildung 4.1.: Wasserfallmodell, Quelle: *Wasserfallmodell* 2014

In der ersten Periode werden die Anforderungen definiert. Diese werden meist in einem Lastenheft und zur genaueren Spezifikation in einem Pflichtenheft festgehalten. Diese Definitionen werden der Softwarearchitekturabteilung übergeben, falls diese existiert, andernfalls direkt der Entwicklungsabteilung. Sobald die Architektur steht wird das Design der Entwicklungsabteilung überreicht, die alle Klassen und Module ausimplementieren. Nachdem die Entwicklung fertig ist kommt die Qualitäts-Sicherung (QS) ins Spiel, welche für das Testen der Programme verantwortlich ist. Die letzten zwei Schritte sind dann die Einführung des Systems und der Einsatz, sowie Wartung und Weiterentwicklung. Der gesamte Zyklus des Testens im Wasserfallmodell ist eine sehr kostspielige, komplexe und lang andauernde Angelegenheit. Die QS prüft die Software bzw. das System anhand unzähliger Testskripts die beschreiben welche Funktionalität getestet wird und welche Resultate erwartet werden. Sollten sich auf Grund von neuen Anforderungen Funktionen oder Module im Programm ändern, kann es mehrere Wochen andauern diese

Skripts zu adaptieren um die Regressions-Tests zu integrieren. Eine weitere Fehlerquelle ist, dass die Testskripts oft von den Entwicklern geschrieben wurden, die das System implementiert haben, was dazu führt, dass das System darauf getestet wird wie es funktionieren würde und nicht wie es funktionieren sollte.[1]

4.2. Agile Methode

Es ist unabhängig davon ob ein Softwareprojekt 18 Monate dauert um eine Enterprise-Applikation zu entwickeln oder ob es eine Webseite für den privaten Gebrauch ist - man benutzt eine bestimmte Vorgehensmethodik. Es gibt Anforderungen, es wird geplant, umgesetzt, getestet und danach im besten Fall abgenommen.

Das größte Problem bei dem Wasserfallmodell ist, dass bereits zu Beginn des Projekts versucht wird alle möglichen Anforderungen zu definieren, da sich diese im Laufe des Projekts ändern können. Dies kann mehrere Gründe haben, beispielsweise dass sich das Unternehmen neu ausrichtet, der Kunde verschiedenste Funktionen nur eingeschränkt, oder überhaupt nicht benötigt, usw... Damit einhergehend muss der gesamte Prozess, von der Analysephase bis hin zur QS, erneut durchlaufen werden. In den meisten Fällen wird dadurch der gesamte Projektverlauf verschoben. Um diese Probleme zu lösen, bzw. eine Alternative zu dem Wasserfallmodell zu finden, haben sich verschiedenste Entwickler mehrerer Spaten und Bereiche zusammengesetzt und das Konzept der iterativen und inkrementellen Entwicklung konzeptioniert. Aus diesem Konzept entwickelte sich die agile Vorgehensweise bei Softwareentwicklungsprojekten.[2]

Konkrete Ansätze und Methodiken die zur heutigen Zeit Anwendung finden sind:

- Scrum
- Extreme Programming (XP)
- Feature Driven Development

[1] vgl. Bender und McWherter, 2011, S. 4.
[2] vgl. Bender und McWherter, 2011, S. 5.

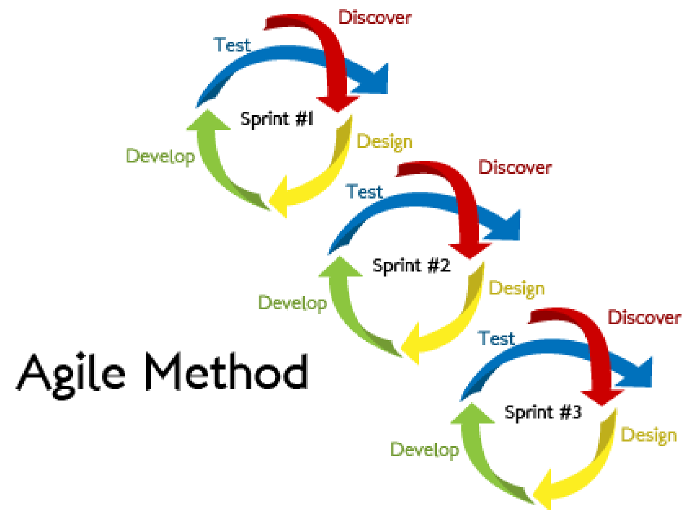

Abbildung 4.2.: Agiles Entwickeln, Quelle:*Test-Driven-Development* 2014

- Clear Case
- Adaptive Software Development[3]

4.3. Entwickler vs. Tester

Viele Softwareentwicklungsfirmen trennen heutzutage dezidiert zwischen dem Entwickeln und dem Testen. Es gibt genügend Gründe dies zu machen. Zum Beispiel bei der Entwicklung von Flugsicherungssystemen für Flugzeuge zeigt es von guter Hygiene die Entwicklung und die Tests zu trennen, um absolut unabhängig voneinander zu bleiben. Es gibt jedoch noch einige agile Entwicklungsteams, die diese Separierung nicht umsetzen, sowie auch Entwicklungsteams, bei denen es auf Grund der History nicht geändert wurde, bzw. die Entwickler zugleich die Tester sind.

[3]vgl. Bender und McWherter, 2011, S. 5.

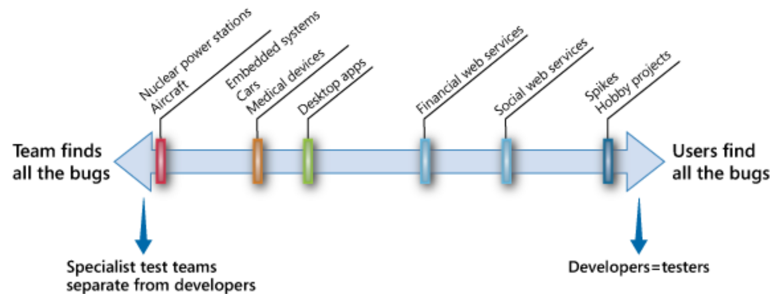

Abbildung 4.3.: Entwickler vs. Tester, Quelle: Brader, Hilliker und Wills, 2013, S.17

Wo Entwickler und Tester getrennt agieren, ist Unit-testing das Gebiet des Entwicklers und das Testen des gesamten Systems übernimmt das Testteam. Wie in Abbildung 4.3 ersichtlich, ist das Auffinden von Bugs von 2 stark getrennten Teams deutlich effektiver.

4.4. Test Driven Development

Im Jahr 1999 hat eine Gruppe von passionierten Entwicklern ein Konzept namens Extreme Programming (XP) entwickelt. Dieses Konzept basiert auf best-practice Beispielen und hebt die Vorteile in diversen Methodiken in Form von definierten Prinzipien hervor. Ein Schlüsselkomponente dieser Prinzipien ist die der test-first Programmierung. Als eine Applikations-Design-Variante wirkt TDD am besten wenn der User aktiv die Gestaltung und die Logik der Applikation mitgestaltet. Für den Entwickler ist das Verständnis für den Anwendungsfall und den Grundgedanken des Features sehr wichtig.[4]

4.4.1. 3 Phasen des TDD

TDD ist in einen 3-Phasen-Zyklus gegliedert. Die Vorgehensweise bei TDD ist, dass zu Beginn die gegebene Anforderung als Test formuliert und implemen-

[4]vgl. Bender und McWherter, 2011, S. 5.

tiert wird. Dieser Testfall muss beim ersten Durchlauf fehlschlagen und solang er das tut, entspricht dieser Teil der Software nicht der Anforderung. Der nächste Schritt ist den Programmcode so zu modifizieren, dass der Testfall durchläuft. Gebot dabei ist, dies auf eine möglichst simple Weise umzusetzen, sodass im Progammcode nur das Nötigste getan werden muss um den Test zum Laufen zu bringen.[5]

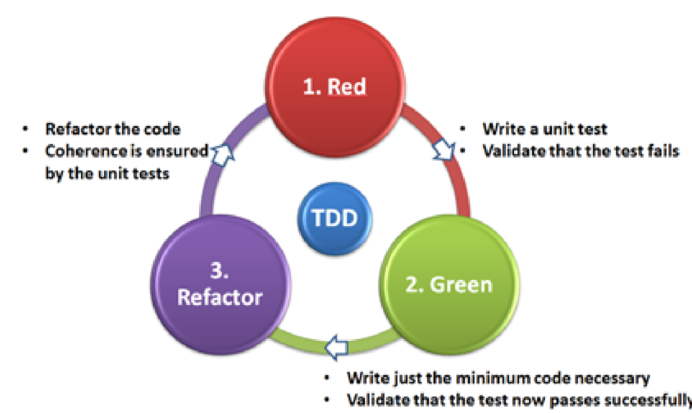

Abbildung 4.4.: TDD, Quelle:*Test-Driven-Development* 2014

4.4.2. Unit-Test

Die Unit Tests (UTs) ist ein Eckpfeiler des TDD. Wenn diese in korrekter Weise umgesetzt worden sind und die Anforderungen an das System wiederspiegeln, reflektieren sich diese Spezifikationen in der Qualität eines Pflichtenhefts wieder. UTs sind an und für sich nicht schwierig umzusetzen und verändern den Entwicklungsprozess nur marginal. Das Hauptaugenmerk bei UTs ist die Abgeschlossenheit jedes Tests, sodass jeder Test ohne Abhängigkeiten von anderen Tests ausgeführt werden kann. Es gibt diverse Charakteristiken, die ein UT aufweist und diese sind:

- Isolation von anderem Code
- Isolation von anderen Entwicklern

[5] vgl. Bender und McWherter, 2011, S. 20.

- Eindeutiges Ziel
- Wiederholbar
- Vorhersehbar[6]

Um UTs umzusetzen wird meist ein Framework benötigt. Um den Aufbau eines UTs zu zeigen wird hier das MS Testframework verwendet. Eine Testklasse hat mehrere Elemente, die über MS-Test-Attribute in C# definiert werden.

- TestClass - Dieses Attribut definiert die Test-Klasse, dass der Test direkt im Visual Studio ausgeführt werden kann.
- TestInitialize - Wird eine Methode mit diesem Attribut versehen, wird diese vor jeder Testmethode aufgerufen.
- TestMethod - Die Methoden mit diesem Attribut werden als Testmethoden ausgeführt.
- TestCleanUp - Beim Aufruf der Methode mit diesem Attribut kann der Ausgangszustand vor dem Test wiederhergestellt.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class CalculatorTest
{
private Calculator _calc;

  [TestInitialize]
  public void Init()
  {
    _calc = new Calculator();
  }

  [TestMethod]
  public void AddTest()
  {
    int result = _calc.Add(2,3);
    Assert.AreEqual(5, result);
  }
```

[6] vgl. Bender und McWherter, 2011, S. 20.

```
    [TestCleanUp]
    public void Clear()
    {
        _calc = null;
    }
}
```

4.4.3. Mocking

Gut durchdachte Software sollte so wenig Abhängigkeiten wie möglich beinhalten. Es kann allerdings Gründe geben, dass die Separierung einzelner Klassen oder Komponenten nicht mehr sinnvoll durchgeführt werden kann. Mit der Zeit entsteht dann eine Vernetzung oder Verschachtelung von diversen Modulen, wie Datenbank, Webservices, Systemtransaktionen und anderen. Mock-Objekte sind dazu gedacht, diese enge Kopplung der Komponenten aufzuheben. Der Grund dafür ist, dass im UT ausschließlich ein bestimmtes Stück an Code getestet werden soll.[7] UTs sollen schnell ausgeführt werden können. Auch wenn es nur eine kleine Applikation ist, können hunderte von UTs existieren. Das Ziel ist die Tests sequentiell zu durchlaufen um zu prüfen, dass die Änderung an Klassen die Applikation nicht negativ beeinflussen.

In diesem Code-Ausschnitt wird gezeigt, wie ein Mock-Objekt definiert und in einem UT verwendet werden kann:

```
public class DataRepoTest
{
        private AppContainerMock _mock;
        private DataRepository _repo;

        [TestInitialize]
        public void Init()
        {
            _mock = new AppContainerMock();
            _repo = _mock.Repository;
```

[7] vgl. Beck, 2003, S.147.

```
        }

        [TestCleanup]
        public void CleanUp()
        {
            _mock.Dispose();
        }

         [TestMethod]
        public void InsertNewAusfuehrung()
        {
            AusfuehrungsKopf kopf = (new
               AusfuehrungsKopf() { Ident =
               Guid.NewGuid().ToString(),
               CreationDate = DateTime.Now, Source =
               "Device" });
            int result =
               _repo.SaveAusfuehrungsKopf(kopf);
            Assert.AreEqual(1, result);
            Assert.AreNotEqual(0, kopf.ID);
        }
 }
```

[8]

4.4.4. Testtypen

STRESS-TESTS

Stress-Tests werden meist angewendet, sobald die Applikation auf einer QS-Plattform läuft. Es ist jedoch sehr schwierig eine geeignete Testmetrik für diese Art von Test zu finden, solange die Applikation nicht auf der Zielhardware läuft. Ein Stress-Test nimmt die Performance einer Applikation ins Auge und untersucht ob unter gegebenen Zugriffen auf die Applikation noch alles wie

[8]Das gesamte Codebeispiel ist im Anhang B zu finden

vorhergesehen läuft. Der simulierte Zugriff von mehreren Usern auf eine Webseite ist ein Beispiel eines Stress-Tests.[9]

User-Acceptance-Tests

Die meisten Applikationen unterliegen bei der Abnahme User Acceptance Tests (UATs). Die Tests werden im Wasserfallmodell meist am Ende, vor der Abnahme im Kreise der Nutzer durchgeführt. Dies ist der Zeitpunkt, zu dem die Nutzer die Applikation allgemein das erste mal sehen. In Organisationen in denen agile Methoden eingesetzt werden, wird die Applikation dem Nutzer so oft als möglich vorgeführt, sodass auf Veränderungen sofort reagiert werden kann. Das soll unter anderem auch sicherstellen, dass die Software absolut den Geschäftsanforderungen des Kunden oder Nutzers entspricht und die Kosten auf Grund der Änderungswünsche nicht ins unermessliche wachsen.[10]

Regression-Tests

Falls trotz der Vorgehensweise des TDD im späteren Verlauf Fehler auftreten sollten, werden Regression-Tests angewandt. Man schreibt den kleinst möglichen Test, der fehlschlägt und sobald dieser ohne Fehler durchläuft gilt der Fehler eingangs als behoben.[11]

4.5. Testmetrik

Der Begriff Testmetrik etstand aus der Welt der Testtechnologie. Schon früh erkannte man, dass alleine die Erhebung der gefundenen Fehler für die Fehlerstatistik ein zu unaussagekräftiges Maß ist. In den 1970igern begann man damit die Testüberdeckung zu messen und auf die Programmzuverlässigkeit zu schließen. Einer der ersten Pioniere war Bill Hetzel, der schon früh das Potential der Testmetriken feststellte und in seinem Buch die ersten Testmessungsansätze festhielt.[12]

[9]vgl. Bender und McWherter, 2011, S. 23.
[10]vgl. Bender und McWherter, 2011, S. 23.
[11]vgl. Beck, 2003, S.140.
[12]vgl. Sneed, Seidl und Baumgartner, 2010, S.197.

4.5.1. Testmetrik nach Hetzel

Hetzel begründete die Schwierigkeit und Herausforderung des Testens damit, dass das Finden von geeignete Maßzahlen und Größen so kompliziert ist. Er definierte 4 grundlegende Maße:

- Testaufwand
- Testüberdeckung
- Testeffizienz
- Testeffektivität

Der Testaufwand wird dabei in Tagen gemessen, die Testeffektivität in gefundenen Mängel und die Testeffizienz ergibt sich aus der Relation zwischen der Anzahl der gefundenen Mängel zum Testaufwand.[13] Die Testüberdeckung untergliedert Hetzel in 4 Unterkategorien:

- Anforderungsüberdeckung
- Entwurfsüberdeckung
- Modulüberdeckung
- Codeüberdeckung

"Anforderungsüberdeckung ist der Prozentsatz getesteter Anforderungen von allen Anforderungen. Entwurfsüberdeckung ist der Prozentsatz getesteter Architektureigenschaften bzw. technischer Features. Modulüberdeckung ist der Prozentsatz ausgeführter Module. Sofern man Methoden mit Prozeduren gleichsetzt, würde man dies heute als Methodenüberdeckung bezeichnen. "[14]

[13] vgl. Sneed, Seidl und Baumgartner, 2010, S.201.
[14] vgl. Sneed, Seidl und Baumgartner, 2010, S.202.

5. Umsetzung

Die Umsetzung erfolgt anhand eines bereits existierenden Projekts. Dieses Projekt ist eine Android-App in Java geschrieben und dient zur Instandhaltung. In dieser sind alle Funktionalitäten integriert, die auch plattformunabhängig umgesetzt werden müssen. Das Projekt dient zur Auftragsabwicklung bei der Instandhaltung von Leuchten. Es besteht dabei die Möglichkeit Aufträge direkt am Smartphone anzulegen, die Tätigkeiten abzuarbeiten und den Auftrag abzuschließen. Weiters können noch nicht abgeschlossene Aufträge angezeigt werden und abgearbeitet werden. Die fertig abgeschlossenen Aufträge werden ebenso angezeigt. Jeder dieser Aufträge ist in mehrere Tätigkeiten untergliedert, basierend auf der Auswahl des Service, welches die Informationen zu den Tätigkeiten beinhaltet. Die Tätigkeiten lassen sich in 2 Kategorien untergliedern und zwar in Anleitungen, die verpflichtend abgearbeitet werden müssen um den Auftrag abzuschließen und in Maßnahmen, die optional sind.

5.1. Ziele

Die Ziele wie Eingangs erwähnt sind Kernpunkte, die mit Mono umgesetzt werden müssen können. Hier nochmals die Auflistung mit einer detailierten Beschreibung:

- Workflows der Applikation unabhängig von der GUI - Es muss eine Software-Architektur gefunden werden, mit der es möglich ist, die Daten, bzw. Logik von der Oberfläche zu trennen. Beim Erstellen einzelner Displays, soll hier eine lose Bindung zwischen der Oberfläche, des plattformspezifischen Layouts, und den anzuzeigenden Daten gefunden werden.

- Effiziente Datenpersistierung - Um die Daten des Nutzers zu speichern, bzw. seine Eingaben, muss eine Methodik gefunden werden, die möglichst effizient und sicher Daten zu speichern ermöglicht. Mit Sicherheit in diesem Bezug ist einerseits das fehlerfreie Speichern der Daten auf dem Smartphone und andererseits das Speichern bezüglich des Datenschutzes gemeint.
- Webbasierte Datentransaktionen - Die Kommunikation zwischen Smartphone und Webservices ist zur heutigen Zeit eine unumgängliche Funktionalität. Diese werden oft im Vordergrund über Nutzeraktionen ausgelöst, aber meistens befinden sich diese im Hintergrund ohne dass der Nutzer aktiv in seinem Arbeitsprozess am Smartphone beeinflusst wird.
- Generischer Hardwarezugriff - Da die Logik von der Oberfläche separiert werden soll, so muss auch der Zugriff auf die Hardware so gestaltet werden, dass dieser über den Kern der Applikation möglich ist.
- Testbarkeit - Um eine bestimmte Stabilität der Applikation zu gewährleisten, wie auch das Entwickeln nach der agilen Methode sind Tests ein Eckpfeiler der gesamten Arbeit. Eine gut geteste Applikation erspart Kosten, somit auch Zeit und erhöht das Ansehen beim Abnehmer, da im Normalfall mehr Fehler und Bugs bei der Entwicklung gefunden werden und nicht bei, oder nach der Abnahme der Applikation.

5.2. Durchführung

Am Anfang der Durchführung wurde die bereits bestehende Applikation analysiert und die ausschlagebenden Anwendungsfälle herauskristallisiert. Dabei ergaben sich 3 Fälle:

1. Auftrag erfassen
2. Auftrag anzeigen
3. Auftrag abarbeiten

5.2.1. VS-Solution erstellen

Als Initialschritt wurde eine VS-Solution 2013 auf Basis eines PCL- Kernprojektes (UWP.Core), wie in Punkt 3.5.2 erwähnt, angelegt. Weiters wurde noch ein

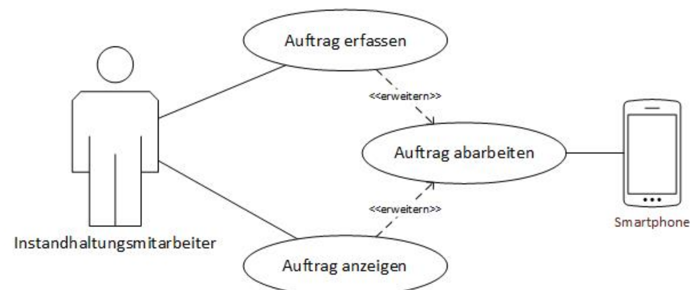

Abbildung 5.1.: Anwendungsfälle

Windows Phone Projekt (UWP.WP) und ein Android Projekt (UWP.Android), über das Xamarin Plugin zur Untersützung der Entwicklung von Android in C#, erstellt. Der letzte Schritt war das hinzufügen des Test-Projekts (UWP.Core.Test), basierend auf dem MS-Testframework wie in Punkt 4.4.2 angeführt. Die Wahl ein PCL, anstatt eines Shared Projects zu erstellen viel auf Grund der Vorteile dieses Projekttyps.[1]

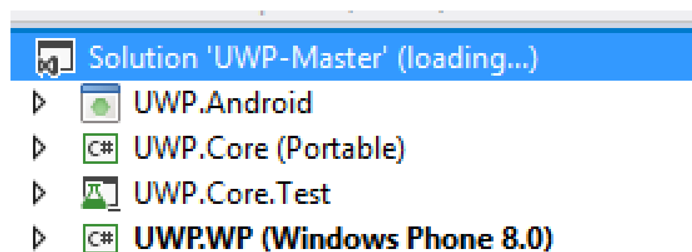

Abbildung 5.2.: VS-Solution

5.2.2. Datenbankzugriff

Zur Datenpersistierung wird von Xamarin empfohlen SQLite zu nehmen, da es auf beiden Plattformen(Android, Windows Phone) zur Verfügung steht.[2] Die Aufgabe hierbei war es eine generische Klasse für die PCL zu entwicklen,

[1] *Portable Class Library* 2014.
[2] *Xamarin-SQLite* 2014.

mit der im Kern die Logik zur Datenspeicherung integriert werden kann. Xamarin stellt dabei eine Datei zur Verfügung, mit der es möglich ist auf SQLite zuzugreifen, allerdings ist hier die Einschränkung auf Shared Projects beschränkt und somit ungeeignet. Es existiert allerdings eine Lösung[3] für dieses Problem und dabei wurden die benötigen Klassen generisch im Kern-Projekt definiert und im jeweiligen plattformspezifischen Projekt implementiert. Mit diesem Object Relational Mapping (ORM)-Framework ist es möglich Tabellen und Spalten über die jeweiligen Attribute in C# zu definieren und über die Klassen dieser Komponenten die Daten zu verwalten.

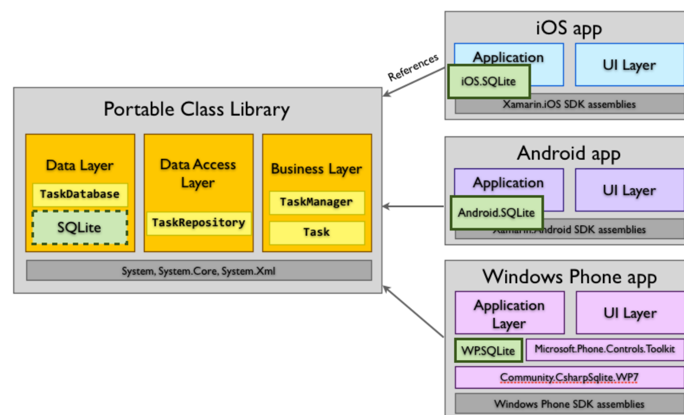

Abbildung 5.3.: PCL-SQLite, Quelle:*PCL-Code-Sharing* 2014

Im folgenden Codeausschnitt sieht man die Verwendung des Frameworks im Kernprojekt:

```
...
private
   MvvmCross.Core.Data.SQLiteBase.SQLiteConnection
   _connection;
...
public AusfuehrungsKopf GetAusfuehrungsKopf(long id)
  {
        var q = from u in
           _connection.Table<AusfuehrungsKopf>()
```

[3]**SQLite2014**.

```
                where u.ID == id
                select u;

    if (q.Count<AusfuehrungsKopf>() > 0)
        return q.ElementAt<AusfuehrungsKopf>(0);
    else
        return null;
}
```

5.2.3. Workflow - MVVM

Für die Abbildung des Workflows wird ganz nach den Vorgaben von Microsoft das MVVM-Entwurfsmuster gewählt. Mit diesem Design kann das ViewModel und das Model im Kernprojekt definiert werden und die View im plattformspezifischen Projekt. Im ersten Schritt wurde für die Umsetzung des MVVM-Entwurfsmuster das MVVMLight-Framework verwendet und im Schritt darauf das MvvmCross-Framework, da es simpler ist, einen größeren Umfang an Funktionalität mitbringt und speziell für diese Art der Anwendung entwickelt wurde.

Als Beispiel das MenuAuswahlViewModel das weiter auf das ViewModel zur Anzeige der Aufträge verweist.

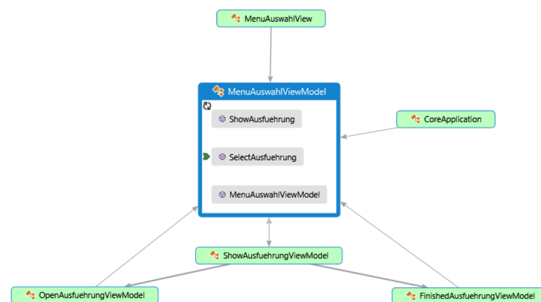

Abbildung 5.4.: MenuAuswahlViewModel

Im Anhang A ist ein komplettes Beispiel einer Smartphone-Seite angelehnt an das MVVM-Entwurfsmuster zu finden.

5.2.4. Testbarkeit

Da das Testen wie in der Einleitung einen wesentlichen Teil der erfolgreichen Abwicklung von IT-Projekten spielt und auch im Vordergund bei der Verwendung der agilen Vorgehensweise ist, spielt es eine wesentliche Rolle in dieser Arbeit.

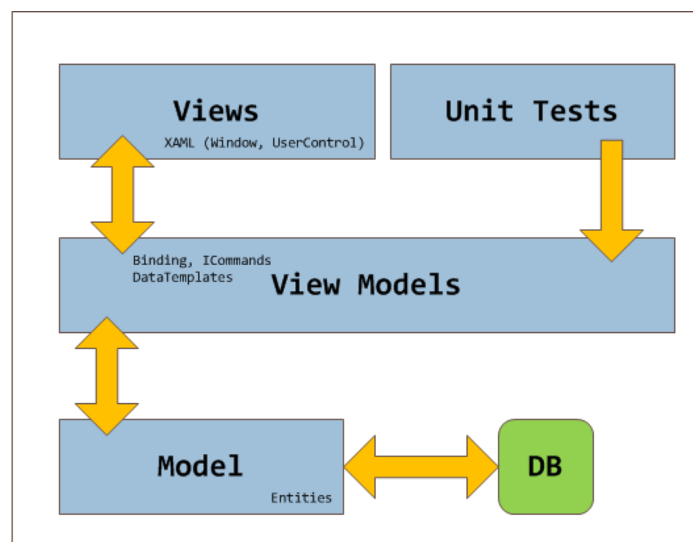

Abbildung 5.5.: MVVM-Test, Quelle:*Mvvm ViewModel-Test* 2014

Bei einer architektonischen Teilung der Oberfläche mit der Logik, bzw. den Daten, sowie es mit dem MVVMEntwurfsmuster der Fall ist, können die einzelnen Komponenten sehr gut entkoppelt und getestet werden, wie auf Abbildung 5.5 erkennbar ist.

5.2.5. Generischer Hardwarezugriff

Ein weiterer wichtiger Punkt war der Zugriff auf die Hardware unabhängig von Gerät und Hersteller. Über das in Punkt 3.6.2 erwähnte Prinzip Service Location und IoC ist es möglich in dem Kernprojekt eine generische Struktur für die Kamera, GPS und andere integrierte Sensoren zu definieren und im plattformspezifischen Projekt zu implementieren. Dass MvvmCross-Framework bietet bereits Plug-Ins um die Kamera und GPS zu nutzen. Mit diesem Plug-In für die Kamera ist es möglich ein Foto zu schießen, oder aus dem Album auszuwählen und in einem vordefinierten Ordner in der Applikationsstruktur zu speichern. Ein kurzes Beispiel wie diese Funktion in einem ViewModel genutzt wird.

```
private readonly IMvxPictureChooserTask
   _pictureChooserTask;
private readonly IMvxFileStore _fileStore;

 public ImageCapturingViewModel(
            IMvxPictureChooserTask
               pictureChooserTask, IMvxFileStore
               fileStore)
        {
            _pictureChooserTask = pictureChooserTask;
            _fileStore = fileStore;
        }

private void DoAddPicture()
{
  _pictureChooserTask.TakePicture(400, 95, OnPicture,
     () => { });
}

 private void OnPicture(Stream stream)
{
  var memoryStream = new MemoryStream();
  stream.CopyTo(memoryStream);
  PictureBytes = memoryStream.ToArray();
```

```
}
```

5.2.6. Webservice-Zugriff

Ein wesentlicher Teil der Applikation ist die Kommunikation zwischen der Applikation und diversen Webservices, ob Representational State Transfer (REST), oder Simple Object Access Protocol (SOAP). Da Mono ein Subset der BCL zur Verfügung stellt, kann auch die Klasse `WebRequest`, oder auch `HttpWebRequest`, wie gewohnt in .net, zum Aufruf von diversen Webservices verwendet werden.

GeoLocation

Das GeoLocationService von Google wurde (GeoCoding)[4] benutzt um eine Addressauflösung durchzuführen. Es wurde dem Service die Addresse übergeben und als Response die GPS-Daten ausgewertet.

Ein kurzer Codeausschnitt als Beispiel:

```
private void FinishRequestAusfuehrung(IAsyncResult
    result)
        {
            HttpWebResponse response =
                (HttpWebResponse)_request
                    .EndGetResponse(result);
            using (StreamReader reader = new
                StreamReader(response
                    .GetResponseStream()))
            {
                webResult = reader.ReadToEnd();
            }
            JObject jsonContent =
                JObject.Parse(webResult);
```

[4] *Google GeoCoding* 2014.

```
        Location loc =
           _parser.GetLocation(jsonContent);
        AusfuehrungsLocation lo =
           (AusfuehrungsLocation)
           result.AsyncState;
        lo.Location = loc;
        _ausfLocationCallback(lo);
    }
```

Die gesamte Klasse dieses Beispiels befindet sich im Anhang C.1

6. Zusammenfassung und Ausblick

Gesamtziel dieser Masterarbeit war es zu evaluieren, ob die gegebenen Anforderungen zur Entwicklung von mobilen Applikationen mit Mono erfüllt werden können. Für die Firma Comm-Unity war es wichtig, dass gegebene Anforderungen und Funktionalitäten bei der Entwicklung der Software umsetzbar bzw. abrufbar sind. Hierbei handelte es sich einerseits um die Flexibilität der Software in Bezug auf die Hardware bzw. des Betriebssystems und andererseits auf die Einschränkung des Aufwands bei der Entwicklung selbst. Das erste Teilergebnis war, dass die Funktionalitäten der BCL auf allen Zielplattformen zur Verfügung stehen. Damit konnten Basismethoden implementiert werden, wie Daten aus Dateien auszulesen und Webrequests abzusetzen. Ein weiteres Ziel war das Definieren des Workflows der Applikation unabhängig von dem Betriebssystem bzw. der GUI. Über die Implementierung des MVVM-Entwurfmusters und unter Zuhilfenahme des MvvmCross-Frameworks konnte dies auch umgesetzt werden. Da gerade bei der Entwicklung von plattformunabhängigen Applikationen ein enormes Fehlerpotential herrscht, war die Testbarkeit dieser wichtig. Durch die Wahl der MVVM-Architektur ist auch das Testen in Form von Unit-Tests möglich und somit auch die agile Vorgehensweise. Um die Zielplattformen Windows Phone und Android und später auch iOS bedienen zu können wurde für die Datenpersistierung SQLite verwendet, da dies auf diesen Plattformen nativ zur Verfügung steht. Die Hardware lässt sich dank der durch Xamarin implementierten API von Android in C# und den MvvmCross-PlugIns ohne großen Aufwand nutzen. Am Ende meiner Arbeit wurde von Xamarin ein weiteres, zukunftsweisendes Feature (Xamarin.Forms) veröffentlicht, das einen Nachteil bei der bisherigen Implementierung der aktuellen Version wettmacht. Bis jetzt musste jede GUI pro Betriebssystem separat entwickelt werden, obwohl sich teilweise das Layout und die Controls vom Design her überschneideten. Mit dem neuen Feature soll es möglich sein die GUI nur einmal zu definieren und beim Kompilieren auf die plattformspezifischen Elemente zu wrappen.

Aus dem Ganzen lässt sich der Schluss ziehen, dass sich alle vordefinierten Ziele anhand der Ergebnisse erfüllen lassen. Die Kombination von Mono, Xamarin und dem MvvmCross-Framework bietet ein großes Potential für die Entwicklung von plattformunabhängigen Applikationen und lässt für die Zukunft viel erwarten.

Anhang

Anhang A.

Codebeispiel MVVM

Als Codebeispiel wird hier das Hauptmenü der Applikation gezeigt, das mittels MVVM aufgeteilt ist.

A.1. ViewModel - PCL

```
using System;
using Cirrious.MvvmCross.ViewModels;
using MvvmCross.Core.Services;
using Cirrious.CrossCore;

namespace MvvmCross.Core
{
    public class MenuAuswahlViewModel : BaseViewModel
    {
        public MenuAuswahlViewModel()
            : base()
        {
            Mvx.Resolve<IAusfuehrungsService>()
            .StartSynchAusfuehrung();
        }

        public void SelectAusfuehrung()
        {
            ShowViewModel
```

```
            <SelectAusfuehrungViewModel>();
        }

        public void ShowAusfuehrung()
        {
            ShowViewModel
            <ShowAusfuehrungViewModel>();
        }
    }
}
```

A.2. View - Android

```
using Android.App;
using Android.OS;
using Cirrious.MvvmCross.Droid.Views;
using MvvmCross.Core;

namespace MvvmCross.Android.Views
{
    [Activity(Label = "Auswahl")]
    public class MenuAuswahl : MvxActivity
    {
        protected override void OnCreate(Bundle
           bundle)
        {
            base.OnCreate(bundle);
            SetContentView(Resource.Layout
            .MenuAuswahl);
        }
    }
}
```

A.3. View - Windows Phone

```
 using System.Collections.Generic;
```

```
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Shell;
using Cirrious.MvvmCross.WindowsPhone.Views;
using MvvmCross.Core;
using Cirrious.CrossCore;

namespace MvvmCross.Phone.Views
{
    public partial class MenuAuswahlView :
       MvxPhonePage
    {
        public new MenuAuswahlViewModel ViewModel
        {
            get { return
               (MenuAuswahlViewModel)base.ViewModel; }
            set { base.ViewModel = value; }
        }
        public MenuAuswahlView()
        {
            InitializeComponent();

        }
    }
}
```

Anhang B.

Mocking - Beispiel

```
public class AppContainerMock : AppContainer,
   IDisposable

{

       private bool _isDisposed = false;
       public AppContainerMock()
           :base()
       {
       }

       protected override void InitDB()
       {
            _connection = new
               Connection("test.sqlite");
           _connection.CreateTable
               <AusfuehrungsKopf>();
           _connection.CreateTable
               <AusfuehrungsPos>();
       }
}

public class DataRepoTest
{
       private AppContainerMock _mock;
```

```
private DataRepository _repo;

#region Zus tzliche Testattribute

[TestInitialize]
public void Init()
{

    _mock = new AppContainerMock();
    _repo = _mock.Repository;

}

[TestCleanup]
public void CleanUp()
{

    _mock.Dispose();

}
#endregion

 [TestMethod]
public void InsertNewAusfuehrung()
{
    AusfuehrungsKopf kopf = (new
       AusfuehrungsKopf() { Ident =
       Guid.NewGuid().ToString(),
       CreationDate = DateTime.Now, Source =
       "Device" });
    int result =
       _repo.SaveAusfuehrungsKopf(kopf);
    Assert.AreEqual(1, result);
    Assert.AreNotEqual(0, kopf.ID);
}

         [TestMethod]
```

```
public void GetAusfuehrungen()
{
    AusfuehrungsKopf head = new
        AusfuehrungsKopf() { Ident =
        Guid.NewGuid().ToString(),
        CreationDate = DateTime.Now, Source =
        Resource.RT.DbAkSource, ExState =
        Resource.RT.DbAkStateInWork };
    int result =
        _repo.SaveAusfuehrungsKopf(head);

    Assert.AreNotEqual(0, head.ID);

    long id = head.ID;
    AusfuehrungsKopf testHead =
        _repo.GetAusfuehrungsKopf(id);
    Assert.AreEqual(testHead.ID, id);

    result = _repo.DeleteExecutionHeader(id);
    Assert.AreEqual(1, result);

    testHead = _repo.GetAusfuehrungsKopf(id);
    Assert.IsNull(testHead);

}

[TestMethod]
public void UpdateAusfuehrung()
{
    AusfuehrungsKopf head = new
        AusfuehrungsKopf() { Ident =
        Guid.NewGuid().ToString(),
        CreationDate = DateTime.Now, Source =
        Resource.RT.DbAkSource, ExState =
        Resource.RT.DbAkStateInWork };
    int result =
        _repo.SaveAusfuehrungsKopf(head);
```

```
            Assert.AreNotEqual(0, head.ID);

            long id = head.ID;
            AusfuehrungsKopf testHead =
               _repo.GetAusfuehrungsKopf(id);
            Assert.AreEqual(testHead.ID, id);

        }

}
```

Anhang C.

Webservice

C.1. GeoCoding

```
 namespace MvvmCross.Core.Services.Location
{
    public class GeoCodingService : IGeoCodingService
    {
        private readonly string _url;
        private HttpWebRequest _request;
        string webResult = string.Empty;
        Action<Location> _locationCallback;
        Action<AusfuehrungsLocation>
           _ausfLocationCallback;
        AusfuehrungsLocation _currentAusfuehrung;

        LocationParser _parser;
        public GeoCodingService()
        {
            _url =
               "http://maps.googleapis.com/maps/api
            /geocode/json?
                  address={0},sensor=true";
            _parser = new LocationParser();
        }
```

```
public void
   GetGPSfromAusfuehrung(AusfuehrungsLocation
   ausfLocation, Action<AusfuehrungsLocation>
   callback)
{
     _request = WebRequest.CreateHttp(
     string.Format(_url,
         ausfLocation.Kopf.Address));
     _request.BeginGetResponse(
       new AsyncCallback
       (FinishRequestAusfuehrung),
          ausfLocation);
     _currentAusfuehrung = ausfLocation;
     _ausfLocationCallback = callback;

}

private void
   FinishRequestAusfuehrung(IAsyncResult
   result)
{
     HttpWebResponse response =
         (HttpWebResponse)_request
         .EndGetResponse(result);
     using (StreamReader reader =
         new StreamReader(response
         .GetResponseStream()))
     {
         webResult = reader.ReadToEnd();
     }
     JObject jsonContent =
        JObject.Parse(webResult);
     Location loc =
        _parser.GetLocation(jsonContent);
     AusfuehrungsLocation lo =
        (AusfuehrungsLocation)
        result.AsyncState;
```

```
            lo.Location = loc;
            _ausfLocationCallback(lo);
        }

    }

    public class LocationParser
    {
        public Location GetLocation(JObject
           jsonContent)
        {
            Location loc = null;
            JArray token =
               (JArray)jsonContent["results"];
            if (token.Count > 0)
                {
                JObject element = (JObject)token[0];
                JToken geoDataLat =
                   (JToken)element["geometry"]
                  ["location"]["lat"];
                JToken geoDataLng =
                   (JToken)element["geometry"]
                  ["location"]["lng"];
                loc = new Location() { Lat =
                   (double)geoDataLat, Lng =
                   (double)geoDataLng };
                }
            return loc;
        }
    }
}
```

Abkürzungsverzeichnis

ACW Android Callable Wrapper

API Application Progamming Interface

BCL Base Class Library

CLI Common Language Infrastructure

CLR Common Language Runtime

CRM Customer Relationship Management

ECMA Ecma International

ERP Enterprise Resource Planning

GPS Global Positioning System

GUI Graphical User Interface

HTML Hyper Text Markup Language

HTML5 Hyper Text Markup Language 5

IoC Inversion of Control

JNI Java Native Interface

MCW Managed Callable Wrapper

MS Microsoft

MVC Model View Controller

MVVM Model View ViewModel

MVP Model View Presenter

NDK Native Development Kit

ORM Object Relational Mapping

PCL Portable Class Library

QS Qualitäts-Sicherung

REST Representational State Transfer

SDK Software Development Kit

SOAP Simple Object Access Protocol

TDD Test Driven Development

UAT User Acceptance Test

UI User Interface

UT Unit Test

VM Virtuelle Maschine

VS Visual Studio

XAML Extensible Application Markup Language

XML Extensible Markup Language

XP Extreme Programming

Literatur

Andrew Whitechapel, Sean McKenna (2012). *Windows Phone8 Development Internals*. 1. Aufl. Microsoft Press Corp., S. 800. ISBN: 1118157702 (siehe S. 13–16).

Android Komponenten (2014). URL: http://developer.android.com/guide/components/fundamentals.html (siehe S. 10, 11).

Base Class Library (2014). URL: http://devreminder.wordpress.com/net/net-framework-fundamentals/ (siehe S. 19).

Beck, Kent (2003). *Test-driven Development - By Example*. Boston: Addison-Wesley Professional, S. 360. ISBN: 978-0-321-14653-3 (siehe S. 44, 46).

Bender, James und Jeff McWherter (2011). *Professional Test Driven Development with C# - Developing Real World Applications with TDD*. New York: John Wiley & Sons. ISBN: 978-1-118-10210-7 (siehe S. 39–43, 46).

Brader, Larry, Howie Hilliker und Alan Cameron Wills (2013). *Testing for Continuous Delivery with Visual Studio 2012*. 1. Aufl. United States: Microsoft Developer Guidance. ISBN: 978-1-621-14018-4 (siehe S. 41).

Code-Sharing (2014). URL: http://developer.xamarin.com%20/guides/cross-platform/application_fundamentals/shared_projects/ (siehe S. 24).

Esposito, Dino (2012). *Architecting Mobile Solutions for the Enterprise*. 12. Aufl. Microsoft Press Corp., S. 445. ISBN: 0735663025 (siehe S. 25).

Google GeoCoding (2014). URL: https://developers.google.com/maps/documentation/geocoding/?hl=de (siehe S. 55).

Meier, Reto (2012). *Professional Android 4 Application Development*. 1. Aufl. John Wiley & Sons, S. 864 (siehe S. 6–8, 10).

Mobile OS Statistik (2014). URL: http://de.statista.com/statistik/studie/id/12549/dokument/mobile-betriebssysteme-statista-dossier/ (siehe S. 1, 2, 6).

Mono for Droid (2014). URL: http://developer.xamarin.com/guides/android/under_the_hood/architecture/ (siehe S. 20, 21).

Mvvm ViewModel-Test (2014). URL: `http://www.codeproject.com/Articles/768427/The-big-MVVM-Template` (siehe S. 53).

MvvmCross Databinding (2014). URL: `https://github.com/MvvmCross%20/MvvmCross/wiki/Databinding` (siehe S. 30).

MvvmCross Prinzipien (2014). URL: `https://github.com%20/MvvmCross/MvvmCross/wiki/The-MvvmCross-Manifesto` (siehe S. 25).

MVVM-Entwurfsmuster (2014). URL: `http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx` (siehe S. 26).

Oechsle, Rainer (2013). *Java-Komponenten : Grundlagen, prototypische Realisierung und Beispiele für Komponentensysteme*. 1. Aufl. Hanser, S. 320. ISBN: 3446431764 (siehe S. 9).

Olson, Scott u. a. (2012). *Professional Cross-Platform Mobile Development in C#*. 1. Aufl. John Wiley & Sons, S. 384. ISBN: 1118157702 (siehe S. 3, 20).

PCL-Code-Sharing (2014). URL: `http://developer.xamarin.com%20/guides/cross-platform/application_fundamentals/building_cross%20_platform_applications/sharing_code_options/` (siehe S. 51).

Peppers, Jonathan (2014). *Xamarin Cross-platform Application Development -*. Birmingham: Packt Publishing Ltd. ISBN: 978-1-849-69847-4 (siehe S. 18).

Portable Class Library (2014). URL: `http://developer.xamarin.com/guides/cross-platform/application_fundamentals/pcl/introduction_to_portable_class_libraries/` (siehe S. 50).

Shackles, Greg (2012). *Mobile Development with C# - Building Native iOS, Android, and Windows Phone Applications*. Sebastopol: Ö'Reilly Media, Inc." ISBN: 978-1-449-33830-5 (siehe S. 6, 22, 23).

Sneed, Harry M., Richard Seidl und Manfred Baumgartner (2010). *Software in Zahlen - Die Vermessung von Applikationen*. München: Hanser Fachbuchverlag. ISBN: 978-3-446-42175-2 (siehe S. 46, 47).

Test-Driven-Development (2014). URL: `http://www.sciabarra.com/agile-%20sites.php` (siehe S. 40, 42).

Wasserfallmodell (2014). URL: `http://wiki.lazarus.freepascal.org%20/images/0/03/Waterfall-model.jpg` (siehe S. 38).

Xamarin (2013). *Power, Speed and Quality Key Strategies for Mobile Excellence*. Whitepaper. URL: `http://resources.xamarin.com/rs/xamarin/images/Xamarin_Whitepaper-Key_Strategies_for_Mobile_Excellence.pdf` (siehe S. 21, 22).

Xamarin-SQLite (2014). URL: http://developer.xamarin.com/guides/cross-platform/application_fundamentals/data/part_3_using_sqlite_orm/ (siehe S. 50).

XAML for Windows Phone 8 (2014). URL: http://msdn.microsoft.com/en-us/library/windowsphone/develop/cc189036(v=vs.105).aspx (siehe S. 16).