FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Wirtschaftsinformatik

Evaluierung und Umsetzung einer wiederverwendbaren effizienten nativen mobilen Cross-Plattform-Entwicklung

Christoph Mahlert

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Wirtschaftsinformatik

Evaluierung und Umsetzung einer wiederverwendbaren effizienten nativen mobilen Cross-Plattform-Entwicklung

Evaluation and implementation of a reusable efficient native mobile cross-platform development

Bearbeiter:	Christoph Mahlert
Aufgabensteller:	Univ.-Prof. Dr. Uwe Baumgarten
Betreuer:	Nils T. Kannengießer, M.Sc.
Abgabedatum:	14.04.2014

Erklärung

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Starnberg, den 14.04.2014 _____

1 Inhaltsverzeichnis

2 Motivation

2012 wurden erstmals weltweit doppelt so viele Smartphones wie PCs verkauft [Bal12]. Tatsächlich verbreiten sich Smartphones zehnmal schneller als der PC in den 80er Jahren, zweimal schneller als das Internet in seinen Boom-Zeiten in den 90er Jahren und dreimal schneller als der erst in den letzten Jahren eingesetzte Trend zu sozialen Netzwerken wie Facebook, Twitter & Co [Pet12].

Diese Zahlen belegen eindrucksvoll, dass der Smartphone- und Tablet-Boom zweifelsohne die Post-PC Ära eingeleitet hat. Die große Verbreitung von mobilen Endgeräten bietet Unternehmen eine Plattform, mobile Anwendungen komfortabel an die Nutzer zu bringen. Auch unternehmensintern lassen sich mit Hilfe mobiler Anwendungen viele Prozesse und Kommunikationswege vereinfachen und beschleunigen. Da der Markt für Smartphones und Tablets schneller wächst als der PC-Markt, bieten diese Geräte nicht nur eine Chance für neue Ideen und Anwendungen. Sie setzen Unternehmen gewissermaßen auch insofern unter Druck, als dass der Fokus in der Vergangenheit stark vom PC auf mobile Anwendungen umgeschwenkt wurde und eine Strategie für die Entwicklung sogenannter Apps gefunden werden muss. So können mobile Anwendungen nicht nur die Nutzung bestehender Prozesse vereinfachen, sondern ermöglichen neue Anwendungsfälle womöglich erst.

Da Unternehmen in erster Linie kosteneffizient und qualitativ arbeiten möchten, stellt sich die Frage, wie genau diese mobilen Anwendungen umgesetzt bzw. entwickelt werden sollten. Diese Frage wird insbesondere dann interessant, wenn man mehr als nur ein mobiles Betriebssystem unterstützen möchte. Nach Gartner werden 2017 90% der Unternehmen zwei oder mehr Betriebssysteme unterstützen, was dazu führt, dass sich diese Unternehmen eine Strategie überlegen müssen, um diese Komplexität meistern zu können [Gar12].

Diese Arbeit wird verschiedene Möglichkeiten zur mobilen Entwicklung aufzeigen. Im Kern wird sich die Arbeit jedoch auf eine native Cross-Plattform-Entwicklung fokussieren, wenngleich andere Möglichkeiten zumindest angedeutet werden sollen. Es wird die aktuelle Situation eines großen Telekommunikationsunternehmens dargestellt. Ausgehend davon soll eine mobile Plattform entstehen, auf der ein Prototyp implementiert wird und die für zukünftige Projekte genutzt werden kann und soll. Nicht zuletzt soll diese Arbeit als eine Entscheidungshilfe für andere Unternehmen dienen.

3 Einleitung

Das Thema „*Evaluierung und Umsetzung einer wiederverwendbaren effizienten nativen mobilen Cross-Plattform-Entwicklung*“ ist eine Problematik in einer Zeit, in der mobile Endgeräte zunehmend an Bedeutung gewinnen. Aus Entwicklersicht laufen auf all diesen Endgeräten die unterschiedlichsten Betriebssysteme wie z.B. Android, iOS, Bada, Symbian, Windows Phone oder Blackberry OS. Wenn bis 2017 90% der Unternehmen zwei oder mehr Betriebssysteme unterstützen [Gar12] und möglichst viele Nutzer erreichen wollen, welche Betriebssysteme sollen dann unterstützt werden? Schaut man sich nur die Verkaufszahlen von Smartphones aus dem dritten Quartal 2013 an, liegen Android und iOS weit vor der Konkurrenz und Android wiederum weit vor iOS [Gar13]. Analysiert man das Verhalten der Nutzer, so stellt man fest, dass iOS-Nutzer signifikant mehr Zeit mit ihrem Gerät verbringen als Android-Nutzer [Sim13]. Je nach dem welche Statistik für aussagekräftiger gehalten wird, können unterschiedliche Prioritäten gesetzt werden.

Doch wie wichtig sind diese Statistiken überhaupt? Geht man nicht allzu weit in die Vergangenheit zurück, 2007 erweist sich schon als ausreichend, stellt man fest, dass Nokia, welches weder Android noch iOS verwandte, einen Smartphone Marktanteil von über 50% hatte. In den vergangenen Jahren rutschte dieser Marktanteil auf 3% ab, weshalb Nokia in eine existenzbedrohende Situation geriet, welche letztendlich dazu führte, dass Nokia seine mobile Sparte aus finanzieller Not heraus an Microsoft verkaufte [Sta13] [Lap13].

Dieses Beispiel zeigt, wie wichtig eine Cross-Plattform-Strategie ist. Sicherlich kann man nicht damit rechnen, dass solche gravierenden Änderungen in den Machtverhältnissen bzw. in den Marktanteilen öfter innerhalb so kurzer Zeit stattfinden. Man kann es jedoch auch nicht vollkommen ausschließen, denn eine solche Entwicklung war vermutlich auch 2007 für die meisten Experten nicht hervorzusehen.

Doch wie geht man eine mobile Cross-Plattform-Entwicklung am besten an? Welche unterschiedlichen Ansätze gibt es und welcher ist je nach Schwerpunkt der geeignetste?

3.1 Forschungsstand und Themenabgrenzung

Mobile Cross-Plattform-Entwicklung ist ein Thema, welches aus unterschiedlichen Blickwinkeln beleuchtet werden kann.

Die einfachste Lösung, eine Applikation für mehrere Betriebssysteme anzubieten, wäre eine separate Entwicklung für jede Zielplattform. Einfach aus dem Grunde, da sich in einem solchen Fall keine Gedanken über eine Cross-Plattform-Entwicklung und eventuelle Abhängigkeiten gemacht

werden muss. Denn für eine plattformübergreifende Entwicklung müssen oft Kompromisse eingegangen werden, da bestimmte Features von Betriebssystem A möglicherweise nicht auf Betriebssystem B verfügbar sind. Diese Möglichkeit ist aber nur auf den ersten Blick einfach und interessant. Tatsächlich erhöht dieses Szenario die Komplexität der Entwicklung enorm, da man mehrere Entwicklungsprojekte für nur eine Applikation managen muss. Nicht nur, dass man Betriebssystem-spezifische Kenntnisse jeder einzelnen Zielplattform mitbringen müsste, wäre es zusätzlich auch zwingend erforderlich, einzelne Change Requests oder Tickets mehrfach für jede Codebasis vorzunehmen.

Da die mobilen Endgeräte über immer leistungsfähigere Hardware verfügen, kommen zunehmend auch webbasierte Techniken für die Entwicklung von Anwendungen in Betracht, die entweder nur mit geringem Aufwand portiert oder im Idealfall völlig plattformunabhängig agieren und gar nicht erst portiert werden müssen, um auf den Zielplattformen zu laufen.

Die Vor- und Nachteile zwischen einer nativen und einer webbasierten Entwicklung wurden in der Bachelorarbeit „*Evaluierung der Umsetzung nativer mobiler Anwendungen im Vergleich zu webbasierten Technologien*“ im Jahr 2011 evaluiert [Mah11]. Das Thema „Cross-Plattform“ steht zwar nicht im absoluten Mittelpunkt dieser Arbeit, hat aber trotzdem eine exponierte Stellung, da der Faktor Portierung von und auf unterschiedliche Betriebssysteme als ein Entscheidungskriterium erkannt wurde und bei den Handlungsempfehlungen je nach Anwendungsfall eine wichtige Rolle spielt. Die Plattformunabhängigkeit wurde in dieser Arbeit jedoch nur der webbasierten Entwicklung zugestanden, da davon ausgegangen wird, dass native Programme separat im jeweiligem Software Development Kit (SDK) oder Native Development Kit (NDK) entwickelt werden.

In der Ausarbeitung „*Comparing cross-platform development approaches for mobile applications*“ aus dem Jahre 2012 von H. Heitkötter, S. Hanschke und T. Majchrzak ist der Fokus auf eine plattformübergreifende Entwicklung gelegt, jedoch wird dabei nicht auf die vielfältigen Möglichkeiten einer solchen Entwicklung eingegangen. Es werden ausschließlich Frameworks betrachtet, die auf Web-Techniken aufsetzen [Hen12].

Auch die Ausarbeitung von I. Singh und M. Palmieri aus dem Jahr 2011 mit dem Titel „*Comparison of cross-platform mobile development tools*“ betrachtet lediglich Frameworks, die Web-Techniken wie z.B. HTML5 und JavaScript nutzen [Sin11].

Des Weiteren soll an dieser Stelle auch die Bachelorarbeit von T. Paananen „*Smartphone cross-platform frameworks*“ aus dem Jahr 2011 erwähnt werden [Paa13]. Das Hauptaugenmerk dieser Arbeit liegt zwar auf eine Cross-Plattform-Entwicklung, jedoch unterscheidet der Autor nicht zwi-

schen den Entwicklungsmöglichkeiten, was dazu führt, dass in seiner Arbeit, ebenso wie in den anderen Arbeiten, eine Cross-Plattform-Entwicklung mit reinen webbasierten oder hybriden Frameworks durchgeführt wird. Frameworks, mit denen man hingegen eine native Applikation implementieren kann bzw. unterstützende Technologien und Grundlagen, die für eine native Cross-Plattform-Entwicklung notwendig sind, werden nicht in Betracht gezogen.

In dieser Arbeit wird ein Überblick über die vorhandenen Entwicklungsmöglichkeiten mobiler Applikationen gegeben. Um die Arbeit einzuschränken, sind damit auch tatsächlich die Möglichkeiten mobiler Applikationen und nicht die der mobilen Spiele gemeint. Viele der vorgestellten Möglichkeiten sind selbstverständlich auch für eine Spieleentwicklung geeignet. Allerdings gibt es auch Technologien, die ganz speziell für eine Spieleentwicklung ausgelegt sind und daher nicht näher betrachtet werden. Auch werden die webbasierten Möglichkeiten, anders als in den anderen Arbeiten, nur angedeutet. Das Hauptkriterium liegt auf einer *nativen* Cross-Plattform-Entwicklung mobiler Business Anwendungen. Die Entwicklungsmöglichkeiten webbasierter Frameworks sind zwar relativ groß, können aber aus den verschiedensten Gründen, die im Verlaufe dieser Arbeit erwähnt werden, nicht an die Nutzererfahrung nativer Applikationen heranreichen. *Nativ* bedeutet in dem Zusammenhang daher, dass auf sämtliche webbasierte oder hybride Frameworks verzichtet wird und eine vollumfänglich native App entstehen soll. Weiterhin sind in den letzten ein bis zwei Jahren bestehende Technologien rasant fortgeschritten bzw. einige neue Technologien wie *RoboVM, Hyperloop* oder *Codename One* erschienen*,* die für eine Cross-Plattform-Entwicklung eine interessante Alternative darstellen. Diese und andere Lösungen werden im Verlaufe dieser Arbeit beleuchtet und dahingehend evaluiert, ob sie den hohen Erwartungen einer nativen Entwicklung nachkommen können.

3.2 Vorgehen

Zwar soll der Fokus dieser Arbeit auf einer nativen Cross-Plattform-Entwicklung liegen, dennoch werden im folgenden Kapitel zunächst verschiedenste Entwicklungsmöglichkeiten aufgezeigt, um die native Cross-Plattform-Entwicklung in einem bestehenden Kontext einbetten zu können. Anschließend werden die Möglichkeiten und die verschiedenen Ansätze erläutert, welche die native Cross-Plattform-Entwicklung bietet. Diese Ansätze sind insbesondere deswegen wichtig, als dass sie zugleich die Grundlagen der Cross-Plattform-Frameworks, die in dieser Arbeit kategorisiert werden, bilden. Dabei werden auch Definitionen für *nativ* erarbeitet, so dass das Feld der nativen Cross-Plattform-Entwicklung eindeutig abgesteckt ist. Am Ende des Kapitels sind die Möglichkeiten der nativen Cross-Plattform-Entwicklung deutlich dargelegt und man steht vor der Entscheidung, auf welche Strategie man bei der Umsetzung einer nativen plattformunabhängigen Applikation setzen möchte.

Um diese Entscheidung treffen zu können, werden im fünften Kapitel die Anforderungen und die Methodologie benannt. Es wird die aktuelle Situation eines großen Telekommunikationsunternehmens skizziert. Ausgehend vom Ist-Zustand wird ein Soll-Zustand definiert, welcher bestimmten Anforderungen gerecht werden muss. Damit die Arbeit allgemeingültig bzw. für andere Leser und Unternehmen als Handlungsempfehlung dienen kann, wird eine Methode vorgestellt, die auf Scores basiert. Je nach Schwerpunkt können die Scores anders vergeben oder gewichtet werden, so dass man mit anderen Anwendungsfällen ggf. eine andere Wahl treffen kann.

Im darauffolgenden Kapitel werden zwei Frameworks beleuchtet, die mit besagten Scores bewertet werden sollen. Auch werden Frameworks benannt, die schon im Vorhinein ausgeschlossen wurden.

Mit dem Framework, welches im fünften Kapitel die höchsten Scores erhielt, wurde ein Prototyp entwickelt, der einen realen Anwendungsfall des beschriebenen Telekommunikationsunternehmens implementiert. Das sechste Kapitel dokumentiert den Projektverlauf dieses Prototyps, dessen Anforderungen, sowie die besonderen Herausforderungen, welche während der Implementierungsphase aufgetreten sind, so dass am Ende des Kapitels ersichtlich ist, welche neuen Erkenntnisse bzgl. einer Cross-Plattform-Entwicklung in der Praxis gewonnen wurden.

Im letzten Kapitel werden die gewonnenen Informationen dieser Arbeit kurz zusammengefasst. Es wird aus technischer und finanzieller hinterfragt, ob eine native Cross-Plattform-Entwicklung den hohen Anforderungen und Erwartungen gerecht wird. Dabei werden Handlungsempfehlungen ausgesprochen und über erste Erfahrungen mit externen Entwicklungsfirmen berichtet.

4 Mobile Entwicklung

In diesem Kapitel wird eine Übersicht über die Möglichkeiten der mobilen Entwicklung gegeben. Dabei werden zunächst die unterschiedlichen Arten der Entwicklung erwähnt, um anschließend die native Cross-Plattform-Entwicklung einordnen und den Fokus der Arbeit darauf legen zu können.

4.1 Arten der mobilen Entwicklung

Wie bereits angedeutet, ist die mobile Entwicklung ein Thema, welches aus unterschiedlichen Blickwinkeln beleuchtet werden kann. Zuallererst wird dabei auf die native Entwicklung eingegangen, gefolgt von der webbasierten und hybriden Entwicklung. Zuletzt werden noch sonstige Lösungen benannt, welche in keine der besagten Kategorien eingeordnet werden können.

4.1.1 Native Entwicklung

Wenn man von einer nativen Entwicklung spricht, muss die Frage geklärt werden, was man überhaupt unter nativ versteht. Dass diese Frage nicht so trivial ist, wie sie auf den ersten Blick wirkt, kann man anhand von Android zeigen. Das zugrunde liegende Betriebssystem von Android ist eine Version von Linux, daher wäre, wenn man nativ streng definiert, die eigentliche native Programmiersprache C / C++. Google selbst warnt allerdings vor der Nutzung von C / C++ und argumentiert damit, dass dies die Komplexität der Apps enorm steigern würde, während die meisten Apps davon nicht profitieren würden [Goo13]. Google empfiehlt daher eine Entwicklung in Java. Zwar stellt Google ein Native Development Kit (NDK) bereit, mit welchem man in C / C++ programmieren kann, allerdings ist die Programmierung mit dem NDK eher als Ausnahme zu verstehen und sollte nur in Ergänzung zum Software Development Kit (SDK) genutzt werden. Das SDK und die Architektur von Android und insbesondere die Android Laufzeitumgebung zielen auf Java ab [Mah11]. Daher definiert man nativ in einem etwas größeren Rahmen und sagt, dass man von einer nativen Entwicklung sprechen kann, wenn eine Applikation in einer plattformspezifischen Sprache entwickelt wird, welche dann von einem Compiler für diese Plattform optimiert wird. Da die Architektur von Android auf Java ausgerichtet ist, kann man deshalb auch Java in diesem Zusammenhang als eine native Sprache unter Android bezeichnen.

Hilft diese etwas weiter gefasste Definition von nativ, wenn man dabei mehrere Zielplattformen im Hinterkopf behält? Eher nicht, wenn die folgende Abbildung betrachtet wird:

Betriebssystem	Im SDK genutzte Programmiersprache
Android	Java (JavaSE)
iOS	Objective-C
Windows Phone 8	.NET (C#,VB.NET)
Blackberry OS	Java (J2ME)

Abbildung 1: native Programmiersprachen

Bei den vier unterschiedlichen Systemen hat man bereits mit drei unterschiedlichen Programmiersprachen zu tun, bei denen eine einfache Portierung mit Codeübernahme quasi schon im Vorhinein ausgeschlossen ist. Unter Android und Blackberry OS könnte man zumindest einiges an Code portieren. Einfach dürfte aber auch diese Portierung nicht werden, da in der Android-Architektur Komponenten wie Content Provider und Broadcast Receiver vorkommen, die in dieser Art und Weise nicht auf Blackberry OS vorhanden sind. D.h. obwohl Android und Blackberry OS die gleiche Programmiersprache im SDK teilen, wäre die Portierung einer Applikation mit einem hohen Aufwand verbunden.

Nun könnte man natürlich damit argumentieren, dass auf all diesen Systemen für einen Entwickler prinzipiell auch C / C++ benutzbar wäre. Dies ist im Kern richtig, doch macht es eine Entwicklung enorm komplex und kaum mehr managebar, da einige Abstraktionsschichten wegfallen würden und man für jedes System Expertenwissen bräuchte. Ein Programmierer müsste z.B. wissen, wie man aus C / C++ heraus auf jedem System eine Benutzeroberfläche erzeugt und wie man mit dieser interagieren kann. Selbst die grundlegendsten Dinge wie der Aufbau einer Entwicklungsumgebung würden sich als komplex erweisen. Man müsste in der Entwicklungsumgebung alle SDKs/NDKs so einbinden, dass auf Knopfdruck eine installierbare Applikation für alle gewünschten Zielplattformen kompiliert werden würde. Wenn man nun noch die in der Einleitung erwähnte Entwicklung der Marktanteile und den Untergang Nokias im Hinterkopf hat, lässt sich erahnen, dass ein Erwerb des benötigten Expertenwissens für alle aktuellen Plattformen äußerst schwierig wäre. Mit *Marmalade* gibt es zwar ein natives Cross-Plattform-Framework [Mar14], in welchem man in C / C++ programmiert, aber trotz Hinzufügen neuer Abstraktionsschichten wird das Bauen einfachster GUI-Elemente wie z.B. einer ListView auch in diesem Framework nicht direkt unterstützt und muss selbst implementiert werden. Daher eignen sich Frameworks wie *Marmalade* nicht besonders gut für Business-Anwendungen. Ihre Stärken liegen in der Entwicklung von Spielen.

Diese Probleme hat man bei einer webbasierten Entwicklung (siehe Punkt 4.1.2) nicht. Der Grund, warum man trotzdem nativ, unabhängig davon welche der genannten Definitionen man verwen-

det, entwickeln möchte, liegt auf der Hand: Ein Entwickler muss so gut wie keine Kompromisse eingehen, was Performanz und das Ausschöpfen der Möglichkeiten angeht. Ein nicht zu unterschätzender Punkt ist zudem das Look & Feel einer Applikation. Native UI-Elemente sehen und „fühlen“ sich anders an als eine webbasierte Benutzeroberfläche. Wenn die App nicht in das Ökosystem des jeweiligen Betriebssystems passt, fühlen sich bestimmte Benutzergruppen möglicherweise nicht angesprochen und nehmen die App allein aus diesem Grund nicht an. Punkt 4.2 („*Möglichkeiten der nativen Cross-Plattform-Entwicklung*“) stellt Lösungsmöglichkeiten vor, wie man die in diesem Kapitel genannten Probleme überwinden kann, um so die Vorteile einer nativen Entwicklung nutzen zu können.

4.1.2 Webbasierte Entwicklung

Zunächst soll an dieser Stelle der Unterschied zwischen *mobile Webseite* und *Web Applikation (Web App)* erläutert werden, da in diesem Zusammenhang immer wieder Missverständnisse auftreten. Diese Missverständnisse treten vor allem deswegen auf, da die Literatur keine eindeutige Definition der genannten Begrifflichkeiten hergibt. Daher wird an dieser Stelle auf die Definition zurückgegriffen, welche in der Bachelorarbeit „*Evaluierung der Umsetzung nativer mobiler Anwendungen im Vergleich zu webbasierten Technologien*", in der webbasierte Entwicklung ein Schwerpunkt war, erarbeitet wurde [Mah11]. Zur besseren Verständlichkeit dient die folgende Abbildung:

Kriterium	mobile Webseite	Web Applikation
Skalierung / Viewport Meta Tag	✓	✓
Look & Feel einer nativen App	✓	✓
native Kapselung / eigenes Icon	✗	✓
Einstellung in App Store	✗	✓
vollständig offline ausführbar	✓	✓

Abbildung 2: mobile Webseite vs. Web Applikation

Eine Web App soll ausgehend davon anhand von fünf Kriterien definiert werden [Mah11]:

- Die Skalierung ist mit Hilfe des V*iewport Meta Tags* auf mobile Endgeräte ausgelegt.
- Das Look & Feel, insbesondere in Hinblick auf die Benutzeroberfläche und der Navigation, entspricht nahezu der einer nativen Applikation. Dieses Kriterium ist eher als Idealvorstellung zu verstehen, da eine webbasierte GUI nur schwer an eine native heranreichen kann. Dennoch soll dies zumindest der Anspruch sein, auch wenn man Abstriche in Kauf nehmen muss.

- Die Anwendung ist innerhalb einer nativen Anwendung gekapselt und kann daher installiert werden und besitzt ein eigenes Icon.
- Die Anwendung kann in den jeweiligen App Store / Play Store etc. gestellt werden.
- Die Anwendung kann vollständig offline ausgeführt werden.

Eine native Kapselung findet bei einer mobilen Webseite hingegen nicht statt. Aus diesem Grund ist es nicht möglich, eine mobile Webseite in einen App Store zu stellen. Ob die Anwendung offline ausgeführt werden kann, soll bei einer mobilen Anwendung im Vergleich zu einer Web App nur optional sein.

Insbesondere auf Grund des Kriteriums der nativen Kapselung ist in diesem Kapitel („*Webbasierte Entwicklung*“) von mobilen Webseiten die Rede. Das Fehlen einer nativen Kapselung hat unmittelbar eine große Schwachstelle mobiler Webseiten zur Folge. Sie sind nicht in einem App Store einstellbar und so von der breiten Masse der Nutzer in eben solchen nicht auffindbar und installierbar. Das nächste Kapitel („*Hybride Entwicklung*“) meint hingegen die Web Apps. Da Web Apps aber auch die Web-Techniken nutzen, ist dieses Kapitel gleichermaßen für die hybride Entwicklung interessant.

Eine webbasierte Entwicklung nutzt, wie der Name bereits verrät, Techniken, welche ihren Ursprung in der Web-Entwicklung haben und daher plattformunabhängig und ideal für eine Cross-Plattform-Entwicklung sind. Dazu gehören vor allem HTML5, JavaScript und CSS3. Warum diese Techniken für die webbasierte Entwicklung wichtig sind und welche Möglichkeiten sie bieten, wird in den folgenden Unterkapiteln beschrieben.

4.1.2.1 HTML5

Die Hypertext Markup Language (HTML) in der Version 5 ist noch kein Standard, sondern ein so genannter Arbeitsentwurf. Allerdings wurde bereits im Mai 2011 der Status vom World Wide Web Consortium (W3C) für HTML5 auf „Last Call“ gesetzt, was bedeutet, dass die Spezifikation einen Zustand angenommen hat, der ganz nahe am finalen Release sein wird. Laut dem Zeitplan des W3Cs soll HTML5 noch in diesem Jahr (2014) offiziell verabschiedet werden [W3C13]. HTML5 ist für die mobile Entwicklung interessant, da die Spezifikationen u.a. um Funktionen im Bereich Multimedia, lokale Datenspeicherung, Offline-Funktionalität, Ortungsfunktionen und Canvas erweitert wurde. Im Folgenden werden die genannten Funktionen kurz vorgestellt.

Die bedeutendste Neuerung im Bereich Multimedia ist die Fähigkeit eines Browsers Inhalte wie Video und Musik nativ abspielen zu können. In älteren HTML-Versionen existierte zwar ein Tag namens *embed*, welcher aber nie zum Standard wurde, weshalb Plug-Ins genutzt werden mussten, um sicher zu stellen, dass der multimediale Inhalt browserunabhängig die Nutzer erreicht.

Dabei hatten viele Firmen eine unterschiedliche Strategie, weshalb u.a. der Realplayer, Windows Media, QuickTime oder Adobe Flash existiert(en). Zum einen ist es unkomfortabel, wenn ein Benutzer so viele Plug-Ins installiert haben muss, zum anderen unterstützen bis heute nicht alle Browser diese Standards. Beispielsweise ist es bis zum heutigen Tage nicht möglich im mobilen Safari Flashinhalte auszuführen. Die Einführung der HTML-Tags *video* und *audio* setzt dieser für einen Benutzer unübersichtlichen Plug-In-Vielfalt ein Ende und ermöglicht das native Abspielen innerhalb des Browsers [Hog101].

Das Speichern und die Persistenz von Daten sind für bestimmte Anwendungsfälle unumgänglich. Vor HTML5 gab es im Bereich der Web-Techniken zwei Varianten, die beide für mobile Anwendungen nicht als ideal zu bezeichnen sind. Entweder mussten Benutzerdaten auf einem Server gespeichert werden, auf dem sich beispielsweise ein Benutzer einloggen muss, um die gesicherten Daten zuordnen zu können. In dieser Variante wären die Daten dann nicht in der Kontrolle des Benutzers und nur zugänglich, wenn dieser auch Zugriff auf das Internet hat. Oder die Daten wurden auf dem Client in Cookies gespeichert, die wiederum bei den Entwicklern keine große Beliebtheit genießen [Law11]. In HTML5 wurden gleich mehrere neue Konzepte unter dem Punkt *Data Storage* aufgenommen. Nennenswert wären dabei die Funktionen *Local Storage*, *Session Storage*, *Web SQL Database* und *Indexed Database API*. Local Storage und Session Storage werden in der Literatur auch oft als *Web Storage* bezeichnet, da sich beide dasselbe *Storage()* Objekt Interface in JavaScript teilen [För11]. Ohne in dieser Arbeit genauer auf die Data Storage Funktionen in HTML5 eingehen zu wollen, sei angemerkt, dass diese die eingangs erwähnte Problematik in Bezug auf Persistenz lösen und daher im Grunde genommen erst ernstzunehmende mobile Applikationen ermöglichen.

Klassischerweise sind mobile Webseiten im Vergleich zu nativen Anwendungen nur über das Internet zu erreichen. Jedoch gibt es selbst in der heutigen Zeit nicht immer und überall einen Zugriff auf das Internet, was einen enormen Nachteil für mobile Webseiten darstellen würde. Um diesen Nachteil auszugleichen, können in HTML5 webbasierte Anwendungen mit Hilfe der *Offline Application Cache API* offline verfügbar gemacht werden. Die Vorteile der API sind im Wesentlichen die Folgenden:

- *Offline Browsing*: Ein Anwender kann die Seite nutzen und innerhalb dieser navigieren, selbst wenn er offline ist.
- *Geschwindigkeit*: Alle Ressourcen können lokal gecached werden. Der Zugriff auf den Cache ist schneller als das Laden der Ressourcen über ein Netzwerk.

- *Reduzierter Traffic*: Die Applikation bzw. der Browser lädt nur Ressourcen vom Server, die sich verändert haben, wodurch sich der Traffic und die Serverauslastung verringert.

Ein Entwickler kann mit Hilfe der *Offline Application Cache API* präzise spezifizieren, welche Dateien gecached werden sollen. Die Applikation läuft selbst dann korrekt, wenn der Anwender den Refresh-Button betätigt, während er offline ist.

GPS und Ortungsfunktionen sind heutzutage in fast jedem mobilen Endgerät vorhanden. Damit auch Webanwendungen sich diese Funktionen zu Nutze machen können, wurde in HTML5 die *Geolocation API* spezifiziert. Diese API erlaubt den Zugriff auf die aktuelle Position des Gerätes eines Benutzers. Dabei ist es via HTML5 möglich einige Parameter einzustellen, um bei der Positionsbestimmung oder anderen Faktoren wie Stromverbrauch Einfluss zu nehmen. Beispielsweise verbraucht ein Endgerät mit aktivierten Ortungsdiensten in der Regel deutlich mehr Strom. Die Geolocation API bietet deshalb unterschiedliche Konfigurationsmöglichkeiten hinsichtlich Genauigkeit, Timeout und Cachedauer an, wodurch der Strombedarf z.B. durch eine längere Cachedauer verringert werden kann [Hol11].

Damit man in HTML nicht nur auf starre Inhalte festgelegt ist, wurde die Spezifikation in Version 5 um *Canvas* erweitert. Mit Canvas werden Leinwände erzeugt, auf denen man mit Hilfe von JavaScript dynamisch Bitmap-Grafiken zeichnen kann. Die Möglichkeiten von Canvas sind so weit gefasst, dass im Verbund mit JavaScript ganze Spiele programmiert werden können [Ful11]. Solche Möglichkeiten eröffneten vor Version 5 nur Plug-Ins wie Adobe Flash oder Microsoft Silverlight, welche nun nicht mehr nötig sind.

4.1.2.2 JavaScript

JavaScript stellt eine Grundlage für die webbasierte Entwicklung dar. Ursächlich dafür ist die Tatsache, dass ein Benutzer mit der mobilen Applikation in vielen Fällen interagieren muss. Viele der neuen Möglichkeiten, darunter die *Geolocation API, Web Storage* oder *Canvas* benötigen den Gebrauch von JavaScript bzw. sind ohne JavaScript de facto nicht nutzbar.

4.1.2.3 CSS3

Cascading Style Sheets (CSS) ist eine deklarative Sprache für Stilvorlagen von strukturierten Dokumenten, weshalb es hervorragend mit HTML zusammen arbeitet. CSS ermöglicht es, den HTML-Inhalt und dessen Darstellung voneinander zu trennen. Anhand ihres Namens, ihrer ID oder ihrer Position können einzelne Elemente in einem HTML-Dokument identifiziert werden. Die Regeln für die konkrete Darstellung dieser Elemente können direkt im Dokument, aber auch in einer separa-

ten Datei, definiert werden. Letzteres erhöht die Wiederverwendbarkeit für andere Dokumente. Durch die Trennung von Inhalt und dessen Darstellung ist es möglich, dass für verschiedene Ausgabemedien unterschiedliche Darstellungen festgelegt werden. Beispielsweise hat ein Smartphone in der Regel eine geringere Auflösung als ein Tablet. Möchte man ein HTML-Dokument sowohl auf einem Smartphone, als auch auf einem Tablet anbieten, kann man mit Hilfe von CSS auf die unterschiedlichen Eigenheiten und Bedürfnisse eingehen, ohne dass der Inhalt im Dokument verändert werden muss.

Wenn man als Programmierer eine hohe Wiederverwendbarkeit anstrebt, eignet sich besonders das Model-View-Controller (MVC) Konzept, bei dem Modell, Präsentation und Steuerung voneinander getrennt werden. Dieses Konzept ist eher aus der objektorientierten Programmierung bekannt, lässt sich aber bei einer Verwendung von HTML5, JavaScript und CSS leicht für die webbasierte Entwicklung adaptieren. Ein einfaches Beispiel verdeutlicht dies:

HTML (Model)

```
<div id="beispiel"></div>
```

CSS (View)

```
.empty {
   background: red;
 }
```

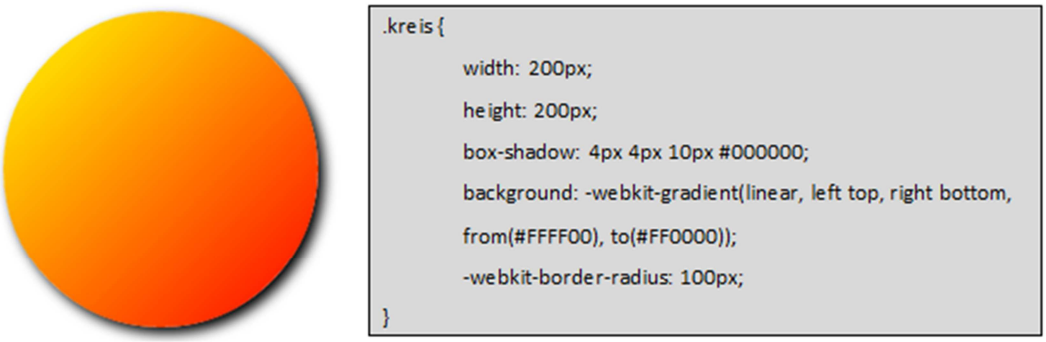
JavaScript (Controller)

```
var x = document.getElementById('beispiel'); // Zugriff auf das Modell
if(x.hasChildNodes) {
      x.className = 'empty'; // ändert die View
}
```

In diesem Beispiel ist die Logik in JavaScript implementiert. Es wird auf das HTML-Modell und auf das Element mit der ID *beispiel* zugegriffen. Falls besagtes Element keine Kinderknoten besitzt (hier ist das der Fall), ändert der Controller die Hintergrundfarbe, so wie es in der View definiert ist. Bei mehreren mobilen Webseiten sind so eine hohe Wiederverwendbarkeit und eine leichte Portierung gegeben. Außerdem ist bei einer komplexen Anwendung der Code einfacher zu lesen.

CSS hat neben der besseren Strukturierung noch eine weitere nützliche Funktion. Die GUI einer mobilen Anwendung ist enorm wichtig, da der Benutzer mit dieser interagieren muss. Daher stellt CSS bei den webbasierten Techniken eine wesentliche Lösung dar, denn de facto wird damit die Benutzeroberfläche definiert. Diese sollte möglichst so sein, dass keine Unterschiede zu einer nativen GUI merkbar sind. In CSS3 gibt es eine Menge neuer Eigenschaften, die es erlauben visuelle Effekte zu erzeugen, die man bisher nur mit Bildern darstellen konnte. Zu diesen Effekten zählen

u.a. abgerundete Ecken, Schlagschatten, semitransparente Hintergründe und Farbverläufe. Abbildung 3 zeigt einen in CSS3 erzeugten Kreis, welcher neben abgerundeten Kanten auch einen Schlagschatten und einen Farbverlauf besitzt.



```
.kreis {
        width: 200px;
        height: 200px;
        box-shadow: 4px 4px 10px #000000;
        background: -webkit-gradient(linear, left top, right bottom,
        from(#FFFF00), to(#FF0000));
        -webkit-border-radius: 100px;
}
```

Abbildung 3: CSS3 Kreis mit Effekten inkl. dazugehöriger CSS3 Code

Zusätzlich zu den genannten Effekten ist es möglich, Elemente mit CSS3 Transformationen schnell und einfach zu drehen, zu neigen und zu verzerren, um so dreidimensionale Körper zu simulieren.

Weitere für die mobile Entwicklung nützliche neue Features in CSS3 sind Transitions und Animationen. Transitions ermöglichen ganz ohne JavaScript o.Ä. fließend animierte Übergänge, so dass innerhalb einer mobilen Webseite die View mit einem speziellem Effekt wie einem Slide-Effekt gewechselt werden kann – ähnlich wie man es z.B. aus PowerPoint-Präsentationen kennt.

All diese Features führen dazu, dass man mit CSS3 anspruchsvolle Benutzeroberflächen erzeugen kann. Der Vorteil der webbasierten GUIs liegt darin, dass man diese im Vergleich zu nativen GUIs einfacher und schneller umsetzen kann. Die webbasierten Benutzeroberflächen sehen durchaus ansehnlich aus, jedoch ist es nicht so einfach möglich ein Look & Feel herzustellen, der dem einer nativen GUI entspricht. Webbasierte Benutzeroberflächen führen, gerade wenn man anspruchsvolle Transitions implementiert, zu Rucklern auf den Endgeräten. Bei einer nativen GUI kann man sich auf ein absolut flüssiges Verhalten verlassen. Auch weiß man bei einer nativen Benutzeroberfläche, dass alle GUI-Elemente in das Ökosystem passen und sich die Nutzer an deren Bedienung schon gewöhnt haben, wenn man davon ausgeht, dass sie ebenso andere auf dem Gerät befindliche Apps nutzen.

4.1.2.4 Web-Frameworks

Web-Frameworks für mobile Webseiten lassen sich im Wesentlichen in die zwei Kategorien *Markup-basiert* und *deklarativ* einordnen. Ein bekannter Vertreter der Markup-basierten Frameworks ist *jQuery Mobile* [The14], während *Sencha Touch* [Sen14] ein weit verbreitetes deklaratives Framework ist. Deklarative Frameworks setzen auf einen programmatischen Ansatz, d.h. die

Elemente einer Benutzeroberfläche werden durch JavaScript Objekte beschrieben, woraus das entsprechende Framework zur Laufzeit den HTML-Code generiert. In Markup-basierten Frameworks hingegen wird die GUI in HTML definiert, woraus das Framework zur Laufzeit beispielsweise den CSS3 Code generiert, der für Transitions und somit für flüssige View-Übergänge sorgt. Unterschiedliche Web-Frameworks führen daher zu unterschiedlichen Anforderungen an einen Entwickler, obwohl beide Framework-Arten Web-Techniken einsetzen. So muss man bei einem deklarativen Framework wie *Sencha Touch* zwingend JavaScript, bei *jQuery Mobile* hauptsächlich HTML beherrschen. Des Weiteren wird es einem Entwickler bei einem Markup-basiertem Framework frei gestellt, in welcher Sprache er die Logik programmiert – denkbar wären prinzipiell sämtliche Web-Technologien wie z.B. Python oder PHP.

Alle Frameworks eint jedoch die gleiche Schwäche: Das Resultat ist weiterhin eine mobile Webseite. D.h. es ist nicht möglich der App ein Icon zu geben und diese in einen App Store zu stellen. Die App Stores der jeweiligen Betriebssysteme sind insbesondere wegen der einfachen Verbreitung wichtig. Eine mobile Webseite schränkt zudem die Möglichkeiten einer potenziellen App massiv ein. Man kann grundsätzlich nur das entwickeln, wozu ein Browser in der Lage ist – schließlich läuft eine mobile Webseite innerhalb eines Browsers. Das W3C arbeitet jedoch ständig an weiteren Spezifikationen, die einem Browser immer mehr Zugriff auf native Features bzw. auf die Hardware gibt. So wurde am 9. Mai 2013 die *HTML Media Capture Spezifikation* fertiggestellt, welche einem Browser via HTML Zugriff auf Kamera und das Mikrofon gewährt [W3C131]. Der Status des Arbeitsentwurfs für die Spezifikation von *Web Notifications* befindet sich zurzeit auf „Last Call“. Durch die Web Notification API wird es einer mobilen Webseite möglich sein, Benachrichtigungen außerhalb des Kontextes einer Webseite zu verschicken [W3C132].

Um jedoch nicht auf die Funktionalitäten eines Browsers angewiesen zu sein, gibt es noch eine dritte Art von Web-Frameworks: die Bridge-Frameworks.

4.1.3 Hybride Entwicklung

Bridge-Frameworks bilden eine Brücke zwischen den Web-Techniken und den nativen Elementen. Bei der Verwendung eines Bridge-Frameworks wird auch von einer *hybriden Entwicklung* gesprochen.

Das bekannteste Framework für eine hybride Entwicklung ist *Apache Cordova* [The13], welches z.B. auch von *PhoneGap* [Ado13] genutzt wird. Ursprünglich war *PhoneGap* ein eigenständiges Framework, ehe es Ende 2011 von Adobe gekauft und der Apache Software Foundation gespendet wurde. Seitdem wird *PhoneGap* zwar parallel weiter entwickelt, nutzt aber *Apache Cordova*

als eine Art Engine und erweitert diese um eigene Funktionen [Pho13]. Bei Apps, die mit *Adobe Cordova* oder *PhoneGap* entwickelt wurden, identifizierte das Technologie-Zentrum Informatik und Informationstechnik (TZI) Sicherheitslücken und gab sieben Sicherheitsregeln für die Arbeit mit *Cordova* vor, die prinzipiell aber auch für andere Entwicklungsarten gelten [Hei14]:

1. Eine aktuelle *Apache Cordova/Phonegap*-Version verwenden
2. Auswahl der Android-Berechtigungen und Plug-Ins genau prüfen
3. Testhilfsmittel nur für Entwicklungszwecke verwenden
4. API-Level definieren. Empfehlung: mindestens API-Level 17
5. Externe Kommunikation nur über gesicherte Verbindungen durchführen
6. Externe Websites im Standard-Browser öffnen
7. Externe Kommunikation, wenn möglich, vermeiden

Bridge-Frameworks lassen sich oft mit anderen Web-Frameworks kombinieren, so dass zum Beispiel *Sencha Touch* standardmäßig *Apache Cordova* und *PhoneGap* unterstützt. Der Applikationscode wird bei Nutzung eines Bridge-Frameworks in einem Wrapper um die native WebView-Komponente ausgeführt, welcher mit einer JavaScript-zu-Native-Brücke verbunden ist.

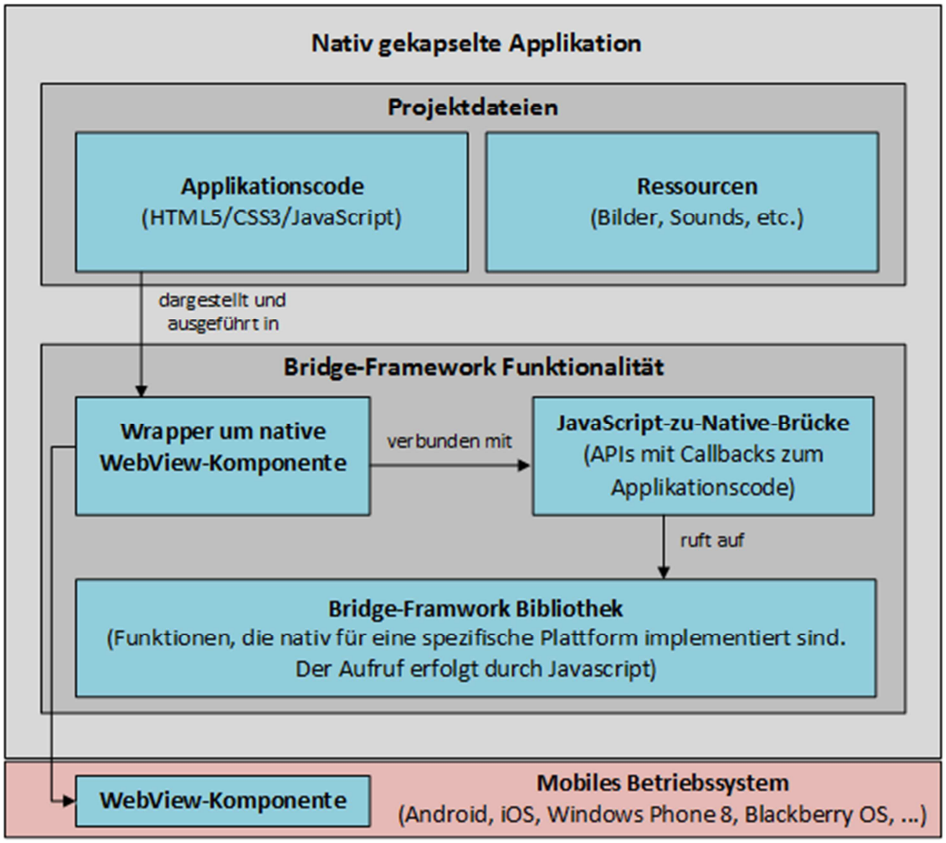


Abbildung 4: Architektur eines Bridge-Frameworks

Über diese Brücke kann via JavaScript auf native Elemente zugegriffen werden, was mit einer mobilen Webseite sonst nicht möglich wäre. Welche nativen Funktionen überbrückt werden können,

bestimmt die Library des Frameworks, welche allerdings – je nach Framework – selbständig erweitert werden kann. Die Architektur eines Bridge-Frameworks ist in Abbildung 4 noch einmal bildlich dargestellt.

Mit Hilfe eines Bridge-Frameworks lassen sich daher einige Schwächen der mobilen Webseite lösen, indem eben solche in eine Web Applikation transformiert wird. Mit dieser ist es nun möglich, Zugriff auf native Funktionen zu erhalten und die App in einen App Store zu stellen.

Bridge-Frameworks bringen jedoch neue, eigene Probleme mit sich. Zwar hat man Zugriff auf eine Reihe von nativen Funktionen, allerdings wird die Performanz durch den Wrapper und der Brücke negativ beeinflusst. Dabei sind Web-Techniken generell schon langsamer als eine native Entwicklung. Ein weiteres großes Problem ist das Debuggen. Lassen sich mobile Webseiten noch innerhalb der Entwicklungstools normaler Desktop-Browser komfortabel debuggen, wird dies nun bei Funktionen, die auf native Elemente zugreifen, äußerst schwer. Größere Projekte, bei denen man auf ein Debugging angewiesen ist, sind somit nur schwer realisierbar bzw. nicht umsetzbar. Das Look & Feel ist identisch zu mobilen Webseiten, so dass auch Web Apps nicht an eine native Benutzeroberfläche heranreichen.

Zusammenfassend lässt sich sagen, dass das Potenzial der Web-Frameworks groß und insbesondere für eine Cross-Plattform-Entwicklung interessant und verlockend ist. Die genannten Probleme führen jedoch dazu, dass weder mobile Webseiten noch Web Apps an eine native Benutzererfahrung heranreichen können.

4.1.4 Sonstige Arten der Entwicklung

Neben einer nativen Entwicklung und einer, die auf Web-Techniken basiert, gibt es noch weitere Entwicklungsmöglichkeiten, die sich weder in die eine noch in die andere Kategorie einordnen lassen. Oft nutzen diese Frameworks ebenfalls Web-Techniken, haben aber einen anderen Ansatz als die bereits vorgestellten, weshalb Entwickler zwar auch auf native Funktionen Zugriff erhalten, die Architektur aber nicht der eines typischen Bridge-Frameworks entspricht. In der Regel setzen solche Frameworks auf eine eigene Laufzeitumgebung. Die *Adobe Integrated Runtime* soll stellvertretend für diese Art der Entwicklung kurz vorgestellt werden.

4.1.4.1 Adobe Integrated Runtime

Die *Adobe Integrated Runtime (AIR)* ist eine proprietäre plattformunabhängige Laufzeitumgebung, mit der so genannte Rich-Internet-Anwendungen (RIAs) erstellt werden können [Ado14]. Plattformunabhängig bedeutet in diesem Fall, dass Android, iOS, Blackberry OS, Mac OSX und

Windows, nicht zu verwechseln mit Windows Phone, unterstützt werden. Es sind Web-Techniken wie HTML und JavaScript möglich, der Zugriff auf native Funktionen erfolgt jedoch durch ActionScript. Dabei implementiert und abstrahiert die *AIR Runtime* durch ActionScript-Erweiterungen einige native Funktionen, wie z.B. den Zugriff auf den Beschleunigungssensor, wodurch ein Entwickler durch ein von *AIR* bereitgestelltes Interface diese Funktionen, unabhängig vom Betriebssystem, mit Hilfe der ActionScript API aufrufen kann [Ado141]. Die folgende Abbildung illustriert die Interaktionen zwischen einer nativen Erweiterung, der *AIR Runtime* und dem zugrunde liegendem Betriebssystem.

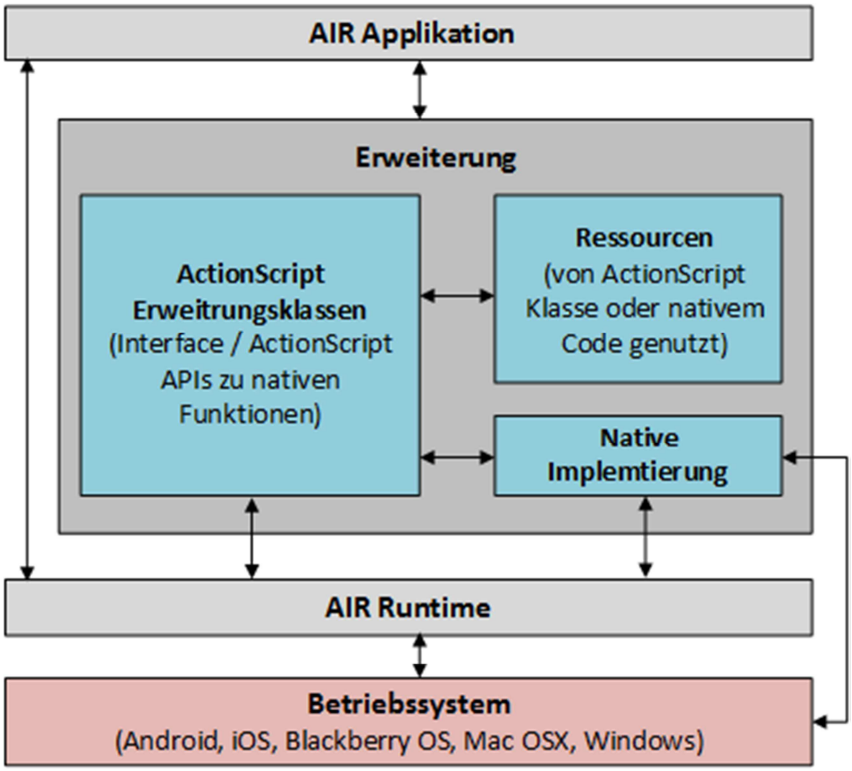


Abbildung 5: AIR Architektur einer nativen Erweiterung

Die Nutzung einer Laufzeitumgebung bringt mehrere Nachteile mit sich. Ohne die *AIR Runtime* ist eine mit *Adobe AIR* entwickelte Applikation nicht lauffähig, d.h. es muss sichergestellt werden, dass ein potenzieller Nutzer der App auch die dazugehörige Laufzeitumgebung installiert hat. Dazu bietet Adobe prinzipiell zwei Optionen an.

Die erste Option ist eine shared Runtime. Bei diesem Konzept muss ein Benutzer der App die Laufzeitumgebung selbstständig installieren. Die *AIR Runtime* ist beispielsweise in Androids Play Store als eigenständige App download- und installierbar [Goo14]. Die Größe der Runtime ist momentan (Stand Februar 2014) 11,68 MB groß – eine Größe, die viele native Applikationen, die keine Laufzeitumgebung benötigen, nicht erreichen werden. Der Vorteil einer shared Runtime, die als separate App vom Hersteller selbst angeboten wird, liegt darin, dass Updates vom Hersteller

direkt den Benutzer erreichen. Das Problem liegt jedoch an der Distribution der Laufzeitumgebung. Einem Laien ist möglicherweise nicht verständlich, warum er zu einer App noch eine weitere herunterladen muss.

Die zweite Möglichkeit, eine Laufzeitumgebung an einen Benutzer zu bringen, bietet das Konzept einer captive Runtime. Hinter diesem Konzept verbirgt sich die Idee, die Laufzeitumgebung mit der Applikation zu bündeln. Der Vorteil gegenüber einer shared Runtime liegt auf der Hand: Die Distribution der Laufzeitumgebung fällt deutlich einfacher, da die App nach der Installation sofort lauffähig ist und der Nutzer die Runtime nicht als separate App suchen und installieren muss. Captive Runtimes haben allerdings auch Nachteile. Da die Laufzeitumgebung inkludiert ist, wird die App deutlich aufgebläht. Der größte Nachteil liegt jedoch darin, dass die Nutzer einem erhöhten Sicherheitsrisiko ausgesetzt sind. Werden Sicherheitslücken in der Laufzeitumgebung entdeckt und vom Hersteller ausgemerzt, kann ein Nutzer nicht selbstständig die Runtime updaten. Er ist so lange einer Sicherheitslücke ausgesetzt, bis die ganze App mit aktualisierter Laufzeitumgebung angeboten wird. Dies könnte zudem auch zu Problemen in der Versionierung der App führen, da bei einem möglichen Update der Laufzeitumgebung nichts an der App selbst geändert wurde.

Bei Verwendung von *Adobe AIR* oder ähnlichen Frameworks mit eigener Laufzeitumgebung müssen vorher die Vor- und Nachteil von shared und captive Runtimes abgewogen werden. Allerdings wird einem Entwickler die Wahl auf manchen Betriebssystemen abgenommen. Apple z.B. fordert im „*iOS Developer Program License Agreement*“ unter dem Punkt 3.3.2, dass interpretierter Code, der nicht in Apple’s Webkit-Framework läuft, auf iOS nur erlaubt ist, sofern die Runtime in der Applikation mitgeliefert wird, was das Konzept einer Captive Runtime entspricht [App14].

4.2 Möglichkeiten der nativen Cross-Plattform-Entwicklung

Von den vorgestellten Entwicklungsmethoden im Kapitel 4.1 („*Arten der mobilen Entwicklung*“) bietet die native Entwicklung die beste Nutzererfahrung. Daher ist eine native Applikation für mehrere mobile Betriebssysteme das Ziel dieser Arbeit. Im Punkt 4.1.1 („*Native Entwicklung*“) wurde jedoch auch ausführlich beschrieben, dass eine native Cross-Plattform-Entwicklung auf Grund der unterschiedlichen Systemarchitekturen und der verwendeten Programmiersprachen nur schwer möglich ist. Um eine Cross-Plattform-Strategie zu finden, kann man verschiedene Ansätze verfolgen, welche in diesem Kapitel beschrieben werden sollen.

Weiterhin wurden im Punkt 4.1.1 („*Native Entwicklung*“) zwei Definitionen für *nativ* genannt. Die erweiterte Definition war, dass man von einer nativen Entwicklung sprechen kann, wenn eine Applikation in einer plattformspezifischen Sprache entwickelt wird, welche dann von einem Compi-

ler für diese Plattform optimiert wird. Dabei wurde herausgearbeitet, warum eine Cross-Plattform-Entwicklung in purem C / C++ für Business-Anwendungen als zu komplex anzusehen ist. Für eine native und gleichzeitig plattformübergreifende Entwicklung muss daher noch eine weitere Definition gefunden werden, da die plattformspezifischen Sprachen verschiedener Systeme, von C / C++ abgesehen, zu unterschiedlich sind. Um dieses Problem zu lösen und gleichzeitig von einer nativen Entwicklung sprechen zu können, muss es auch erlaubt sein, eine Programmiersprache zu verwenden, die nicht plattformspezifisch ist. Der Fokus liegt aus diesem Grunde nicht mehr auf die Sprache, in der programmiert wird, sondern vielmehr auf das Ergebnis. Wenn das Ergebnis eine plattformspezifische Sprache ist oder ausschließlich native Interfaces genutzt werden, so dass z.B. eine native Benutzeroberfläche entsteht, kann man von einer nativen Cross-Plattform-Entwicklung sprechen. Anders ausgedrückt ist eine Entwicklung auch als nativ zu bezeichnen, wenn der native Code zwar nicht selbst geschrieben, dafür aber generiert wird. Die Frage, die dann zu klären wäre, ist: Bleiben bei dieser erweiterten Definition die Vorteile und die hohe Nutzererfahrung einer nativen Entwicklung bestehen oder führt es zu gravierenden Nachteilen? Eine Antwort auf diese Frage soll im Verlaufe dieser Arbeit gefunden werden.

Grob lassen sich die Technologien und die Möglichkeiten der nativen Cross-Plattform-Entwicklung in die drei Kategorien *unterstützende*, *halbautomatisierte* und *vollständige* Lösungen einordnen. Unterstützende Lösungen helfen dabei nativ entwickelte Applikationen für eine bestimmte Plattform auf ein anderes mobiles Betriebssystem zu portieren. Die Vorrausetzung ist also eine bereits nativ entwickelte App. Das Ergebnis ist meistens keinesfalls eine lauffähige Applikation für ein anderes System, sondern vielmehr eine Grundlage für eine Portierung mit Codeübernahme. Dazu wird der zu portierende native Code von der Ausgangssprache in die native Sprache der Zielplattform übersetzt. Bei der Entwicklung eines Frameworks für eine native Cross-Plattform-Entwicklung, müssen Methoden wie die hier vorgestellten genutzt werden. Da diese Technologien auch die Grundlagen halbautomatisierter und vollständiger Lösungen darstellen, werden sie als erstes behandelt, um so die Frameworks, die darauf aufsetzen, verstehen zu können. Was genau unter einer halbautomatisierten und einer vollständigen Lösung zu verstehen ist, wird anschließend erläutert.

4.2.1 J2ObjC

J2ObjC ist ein zwischen Alpha- und Betastatus befindliches Open-Source Kommandozeilen-Programm von Google, welches Java-Code zu Objective-C für iOS übersetzt. Es wird explizit von Google erwähnt, dass *J2ObjC* keine plattformunabhängige Benutzeroberfläche unterstützt und es auch nicht geplant ist, diese Funktion zukünftig zu implementieren [Goo141]. Die Idee von *J2ObjC* liegt vielmehr darin, Java-Code, der keine UI-Module enthält - interessant wäre dabei insbesonde-

re die Logik einer Applikation - zu Objective-C zu übersetzen. Dabei wird nur Client-seitiger Code unterstützt, so dass z.B. Code, welcher auf das java.net Paket zugreift, nicht übersetzt werden kann [Bal14].

Damit *J2ObjC* funktioniert, muss Xcode mit dem iOS SDK auf MacOS installiert sein. Ein einfaches Beispiel würde wie folgt aussehen. Gegeben sei folgende in Java geschriebene „Hallo Welt“ - Anwendung:

```
package de.mahlert.j2objc;

public class Test {
	public static void main(String[] args) {
		System.out.println("Hallo Welt!");
	}
}
```

Um diese Anwendung in Objective-C zu übersetzen, ist es am einfachsten, *J2ObjC* zunächst zum PATH hinzuzufügen:

```
export PATH=${PATH}:/Users/Christoph/Documents/j2objc-0.9
```

Anschließend wird *J2ObjC* auf das entsprechende Java-File angewandt:

```
$ j2objc Test.java
translating Test.java
Translated 1 file: 0 errors, 0 warnings
```

J2ObjC generiert aus der Test.java eine Test.m und den dazugehörigen Header Test.h. Ohne weitere Argumente wird als Ziel dieser Dateien der Pfad *de/mahlert/j2objc* genutzt, was dem Paketnamen der Java-Klasse entspricht. Würde die Test.java Android-spezifischen Code, UI-Module oder sonstigen nicht unterstützten Java-Code enthalten, würde an dieser Stelle ein Fehler ausgeworfen werden. Da in diesem Beispiel jedoch keine Fehler aufgetreten sind, lässt sich der übersetzte Objective-C Code betrachten:

Test.m

```
#include "IOSObjectArray.h"
#include "de/mahlert/j2objc/Test.h"
#include "java/io/PrintStream.h"
#include "java/lang/System.h"
@implementation DeMahlertJ2objcTest
+ (void)mainWithNSStringArray:(IOSObjectArray *)args {
	[((JavaIoPrintStream *) nil_chk([JavaLangSystem out])) printlnWithNSString:@"Hallo Welt!"];
}
- (id)init {
	return [super init];
}
+ (J2ObjcClassInfo *)__metadata {
  static J2ObjcMethodInfo methods[] = {
	{ "mainWithNSStringArray:", "main", "V", 0x9, NULL },
	{ "init", NULL, NULL, 0x1, NULL },
   };
```

```
    static J2ObjcClassInfo _DeMahlertJ2objcTest = { "Test", "de.mahlert.j2objc", NULL, 0x1, 2, methods, 0,
    NULL, 0, NULL};
    return &_DeMahlertJ2objcTest;
}
@end
```

Test.h

```
#ifndef _DeMahlertJ2objcTest_H_
#define _DeMahlertJ2objcTest_H_
@class IOSObjectArray;
#import "JreEmulation.h"
@interface DeMahlertJ2objcTest : NSObject {
}
+ (void)mainWithNSStringArray:(IOSObjectArray *)args;
- (id)init;
@end
#endif
```

Der generierte Code ist deutlich komplizierter, als wenn er direkt in Objective-C geschrieben worden wäre. *J2ObjC* inkludiert einen JRE Emulator, daher kann das Tool nur den Java-Code konvertieren, der in der JRE Emulation enthalten ist. Es werden die meisten Java-Runtime-Konstrukte wie Exceptions, anonyme Klassen, Generics, Threads und Reflection unterstützt. Das Changelog von *J2ObjC* verrät, dass dem JRE Emulator ständig neue Java-Methoden hinzugefügt werden und ein regelmäßiges updaten lohnenswert sein kann [Bal141].

Um zu überprüfen, ob der übersetzte Code lauffähig ist, muss dieser zunächst mit *j2objcc* kompiliert werden. *j2objcc* ist ein Wrapper-Script um den Objective-C Compiler *clang*, welches die JRE Emulation hinzufügt. Es lassen sich alle Argumente benutzen, die auch mit *clang* benutzbar sind, so dass z.B. mit dem Flag –g Debugsymbole zur Binary hinzugefügt werden würden.

```
$ j2objcc -c Test.m
$ j2objcc -o test Test.o
$ ./test Test
Hallo Welt!
```

Das Beispiel funktioniert ohne Probleme. Dies war bei allen Tests der Fall, bei denen kein Android-spezifischer Code enthalten war. *J2ObjC* eignet sich daher gut, wenn man Programmlogik vom Rest des Codes separiert und diesen anschließend in Objective-C übersetzen lassen möchte. Da sich viele Konstrukte aus Java nicht so einfach zu Objective-C übersetzen lassen, sieht der Code oft komplex aus. Da er aber problemlos funktioniert, stellt *J2ObjC* eine interessante Möglichkeit dar, Teile des Codes einfach und schnell zu übersetzen.

Um den ganzen Konvertierungsprozess zu beschleunigen und nicht im dem Terminal arbeiten zu müssen, empfiehlt sich eine Integration in eine Entwicklungsumgebung. Der generierte Code lässt sich zur weiteren Verarbeitung und zur Erstellung einer Benutzeroberfläche in Xcode einfügen,

wofür *J2ObjC* Bibliotheken zur Java-Kompatibilität bereitstellt, gegen die beim Kompilieren gelinkt werden muss. Nutzt man z.B. JUnit-Tests, muss gegen die *libjunit.a* gelinkt werden:

```
j2objcc –o test -ljunit Test.m
```

Der Ablauf einer Entwicklung in Java mit Übersetzung zu Objective-C durch *J2ObjC* wird zur besseren Verständlichkeit in der unten abgebildeten Grafik dargestellt. Die *J2ObjC* JAR-Dateien enthalten beispielsweise Annotation für das Speichermanagement von Java-Objekten.

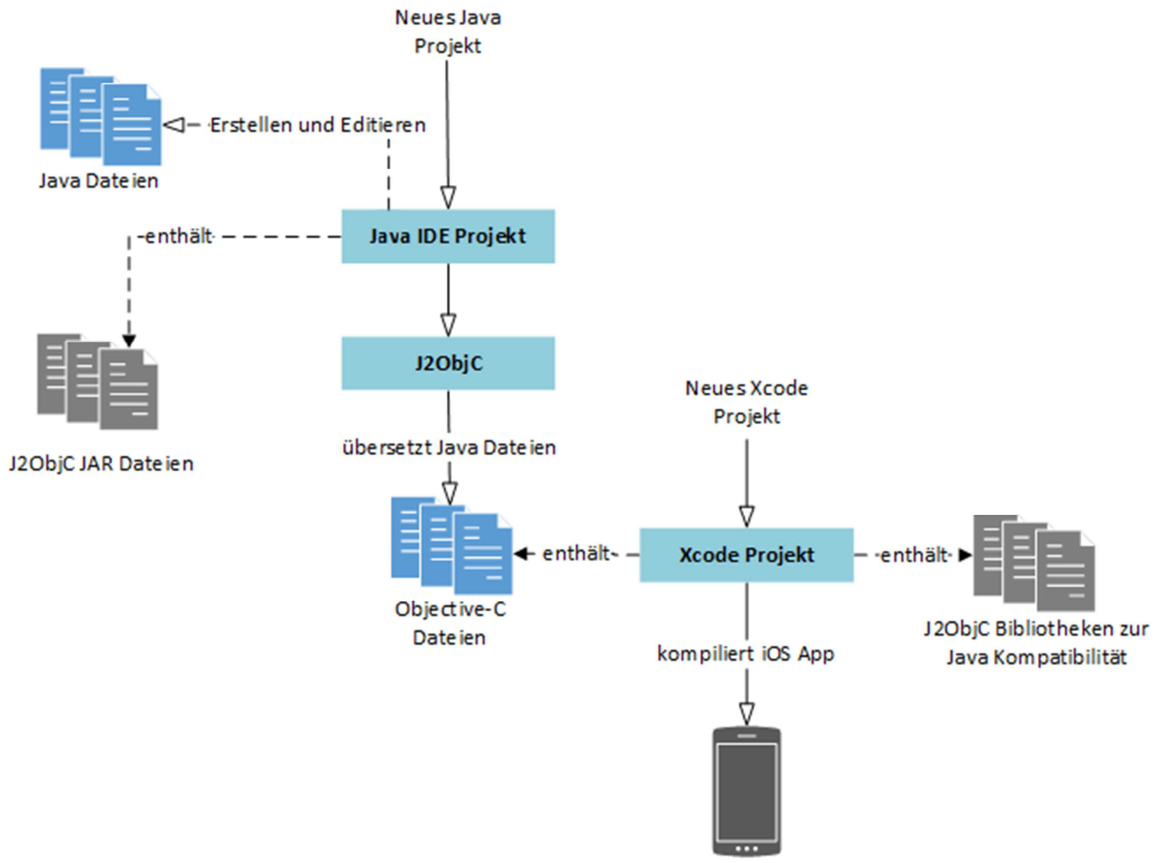


Abbildung 6: J2ObjC Entwicklungsablauf

J2ObjC unternimmt während der Übersetzung zwar schon einige Optimierungsschritte, allerdings lohnt es sich, die optionalen Parameter anzuschauen. Einer davon ist z.B. *--dead-code-report* und gehört zum Feature *Dead Code Elimination. J2ObjC* entfernt, wenn besagte Option aktiviert wird, nicht genutzten Code, um den Objective-C Code nicht unnötig aufzublähen. Dabei ist *J2ObjC* auf *ProGuard* angewiesen. *ProGuard* ist ein weit verbreiteter Code Obfuscater für Java, nimmt aber auch Code-Optimierungen wie das Entfernen unbenutzten Codes vor [Laf14]. Für *J2ObjC* ist nur Letzteres wichtig, weshalb alle anderen Optimierungen, sowie das Obfuscating, deaktiviert werden sollten. Eine mögliche Konfigurationsdatei für *ProGuard* könnte beispielsweise wie folgt aussehen:

```
-injars Applikation.jar
-libraryjars <java.home>/lib/rt.jar

-dontoptimize
-dontobfuscate
-dontpreverify
-printusage
-dontnote

-keep public class de.mahlert.app.Main {
   public static void main(java.lang.String[]);
}

-keepclassmembers class * {
   static final % *;
   static final java.lang.String *;
}
```

Applikation.jar ist dabei der Java-Bytecode, der übersetzt werden soll. Wenn es sich um eine Library handelt, kann die Main-Methode entfernt werden. Des Weiteren sollte der Paketname angepasst werden. Eine solche Konfiguration ließe sich beispielsweise als *proguard.conf* abspeichern und folgendermaßen aufrufen:

```
java -jar proguard.jar @proguard.conf > unbenutzt.log
```

ProGuard erzeugt anschließend ein Log, welches den unbenutzten Code enthält. Dieses Log wiederum dient als Eingabe für die Optimierungen von *J2ObjC*, wozu folgender Parameter hinzugefügt werden muss:

```
--dead-code-report unbenutzt.log
```

Eine weitere Optimierungsmöglichkeit bietet sich auch insofern, als dass man in seinem Java-Code nativen Objective-C Code integrieren kann. Dies würde so aussehen:

```
/*-[ #if JAVA ]-*/
<< Java-Code >>
}
/*-[ #else
<< Objective-C Code>>
}
#endif ]-*/
```

Eine solche Methode eignet sich insbesondere für Low-Level Optimierungen.

Ein Problem, welches noch nicht näher betrachtet wurde, ist das des Speichermanagements. Um dieses muss sich ein Java-Entwickler dank automatisch funktionierendem Garbage Collector nicht kümmern. In Objective-C gibt es standardmäßig ein solches automatisiertes Speichermanagement nicht. *J2ObjC* bietet drei Optionen an, um dieses Problem zu lösen: Reference Counting, Automatic Reference Counting (ARC) und Garbage Collection (GC). Letzteres wird allerdings nur für Ma-

cOS und nicht für iOS unterstützt, was darin begründet ist, dass Apple selbst keine GC-Bibliothek auf iOS anbietet und Google es daher als eine Art Best Practice ansieht [Goo131]. ARC ist prinzipiell die beste der zwei verbliebenen Möglichkeiten, da bei dieser Methode die Verantwortung auf den Compiler abgeschoben wird, welcher zur Kompilierungszeit die benötigten Methoden automatisch hinzufügt. Der Standard in *J2ObjC* ist jedoch Reference Counting, da Google ARC als noch nicht ausgereift genug bezeichnet. Mit der Option *--use-gc* lässt sich, falls gewünscht, jedoch manuell auf ARC umstellen. Beim Reference Counting liegt die Verantwortung beim Entwickler, der dazu Sorge tragen muss, dass die Referenzen eines Objektes korrekt gezählt werden, anderenfalls könnte es zu Nullpointer-Exceptions oder Memory Leaks führen. Um dem entgegenzuwirken, enthält *J2ObjC* Annotation für das Speichermanagement, so dass man schwache Referenzen auch als solche bezeichnen kann. Des Weiteren stellt Google ein Tool namens *MemDebug* bereit, welches die Speicherallokationen und die Objektreferenzen überwacht, wodurch sich leicht Probleme wie Speicherlecks finden lassen.

4.2.2 XMLVM

XMLVM ist ein Cross-Compiler, der den Code nicht auf Quell-, sondern auf Bytecode-Ebene übersetzt. Das Projekt ist Open-Source und kann Bytecode Instruktionen einer Java Virtuellen Maschine (JVM) und einer Microsoft Common Language Runtime (CLR) bzw. dessen Zwischensprache Common Intermediate Language (CIL) zu JavaScript, Python, Objective-C, C++, Java Bytecode oder .NET CIL konvertieren [XML14]. Der Quellcode komplexer Hochsprachen kann, anders als z.B. in *J2ObjC*, weiterhin von einem üblichen Compiler geparsed werden. *XMLVM* repräsentiert den dann kompilierten Bytecode als eine Menge von XML-Dokumenten, die mit XML-Technologien wie XQuery und XPath manipuliert und in die gewünschte Zielsprache mit Hilfe von XLST transformiert werden können.

Es werden keine fertigen Binaries zum Download angeboten. Stattdessen muss *XMLVM* aus dem Repository ausgecheckt und selbst kompiliert werden:

```
Buildfile: /Users/Christoph/Documents/workspace/xmlvm/build.xml
init:
    […]
build-xmlvm:
    [echo] Compiling XMLVM
    […]
build-objc-compat-lib:
    […]
build-android-ios-compat-lib:
    […]
cc-android-ios-compat-lib:
cc-android-ios-compat-lib.impl:
    […]
    [java] [02/26/14 15:00:07.532]    DEBUG: Forcing --enable-ref-counting for target OBJC
```

```
build-csharp-compat-lib:
    […]
build-sdl-compat-lib:
    […]
build-android-sdl-compat-lib:
    […]
xmlvmjar:
    […]
    [jar] Building jar: /Users/Christoph/Documents/workspace/xmlvm/dist/xmlvm.jar
classpath:
classpath:
jar:
BUILD SUCCESSFUL
Total time: 36 seconds
```

In dem Output sieht man, dass während des Build-Prozesses erst die Kompatibilitäts-Bibliotheken zu Android, iOS, Objective-C, C# etc., anschließend dann die *xmlvm.jar* kompiliert wird, mit der man letztendlich weiter arbeitet. Interessant in diesem Output ist auch die Meldung *--enable-ref-counting for target OBJC* beim Kompilieren der Android-iOS Kompatibilitätsbibliothek. Die Meldung ist ein Hinweis darauf, wie das Speichermanagement in Objective-C realisiert wird, wenn der Ausgangscode Java ist. Wie *J2ObjC* nutzt *XMLVM* für Objective-C standardmäßig Reference Counting.

Die Transformation von Java zu Objective-C soll nun anhand eines einfachen Beispiels erläutert werden. Es wurde in Java eine Klasse namens *Berechnung* mit einer Methode implementiert, die das Modulo zweier Zahlen berechnet:

```
public class Berechnungen {
        static int modulo(int a, int b) {
                return a % b;
        }
}
```

Folgt man der Dokumentation auf der *XMLVM*-Projektseite, kann eine Cross-Kompilierung von Java zu Objective-C nur funktionieren, wenn in Objektive-C eine Stack-Maschine nachgeahmt wird [XML141]. Dies klingt nachvollziehbar, da die Java Virtuelle Maschine eine Stack-Maschine ist, während C mit Registern arbeitet. Schaut man sich die XLST-Templates aus dem *XMLVM* Quellcode an, versteht man, wie *XMLVM* den Transformationsprozess umsetzt. Die Stack-Operationen der VM werden über XSLT in der Zielsprache abgebildet, wobei die Stack-Operationen an sich beibehalten werden. Eine Modulo-Operation, die von Java zu Objective-C konvertiert werden soll, sieht im entsprechenden XSLT-Template, in diesem Fall *xmlvm2objc.xsl*, beispielsweise wie folgt aus:

```
<xsl:template match=“vm:irem”>
        <xsl:text>
          _op2.i = _stack[--_sp].i;
          _op1.i = _stack[--_sp].i;
          _stack[_sp++].i = _op1.i % _op2.i;
        </xsl:text>
</xsl:template>
```

Die Variable *stack* steht dabei für den Stack, *sp* für den Pointer auf das erste Element eines solchen. Es werden zwei Operanden aus dem Stack geholt (pop) und anschließend das Ergebnis zurück auf den Stack gelegt (push). Da die Runtime, gegen die generiert wird, keine Stack-Maschine ist, muss es eine Möglichkeit geben, die entsprechenden Slots im Stack durch ein Objekt o.Ä. zu repräsentieren bzw. zu mappen. Dies ist in C einfach umsetzbar, in dem die Speicherbereiche durch ein *union* überlappt werden:

```
typedef union {
        id o;
        int i;
        float f;
        double d;
        JAVA_LONG l;
} XMLVMElem;
```

Dieser Beschreibung folgend müsste sowohl die XML-Repräsentation des Bytecodes als auch der generierte Objective-C Code der Modulo-Methode eine Stack-Maschine darstellen. Allerdings sieht die XML-Repräsentation wie folgt aus:

```
<vm:method name="modulo" signature="(II)I" isStatic="true">
  <vm:signature>
    <vm:parameter type="int" />
    <vm:parameter type="int" />
    <vm:return type="int" />
  </vm:signature>
  <dex:code register-size="3">
    <dex:var name="var-register-1" register="1" param-index="0" type="int" />
    <dex:var name="var-register-2" register="2" param-index="1" type="int" />
    <vm:source-position file="Berechnungen.java" line="5" />
    <dex:rem-int vx="0" vx-type="int" vy="1" vy-type="int" vz="2" vz-type="int" />
    <dex:return vx="0" vx-type="int" class-type="int" />
  </dex:code>
</vm:method>
```

Die XML-Repräsentation des Bytecodes zeigt Register und keine Stack-Maschine. Ein Hinweis darauf, weshalb die Ausgabe anders als erwartet ist, liefert das Schlüsselwort *dex*, welches für *Dalvik Executable* steht und den Bytecode einer Dalvik Virtuellen Maschine (DVM) enthält. Nach der Analyse des *XMLVM* Quellcodes bestätigt sich dieser Verdacht und lässt sich so erklären, dass in neueren Versionen von *XMLVM* der Android Cross-Compiler *dx* enthalten ist. *XMLVM* nutzt für Zielplattformen, die keine Stack-Maschinen implementieren nicht mehr den Java Bytecode, sondern transformiert diesen zunächst durch den *dx* Cross-Compiler zu Dalvik Bytecode, der von ei-

ner DVM ausgeführt werden kann. Während eine JVM auf einen Stack basiert, ist eine DVM Register-basiert und somit deutlich näher an den Zielplattformen, die ebenso Register-basiert sind. Der *dx* Cross-Compiler optimiert den Code für Registermaschinen, wovon *XMLVM* profitiert, sofern die Zielplattform ebenso eine Registermaschine ist.

Mit diesem Wissen lässt sich die XML-Repräsentation interpretieren. Die Methode heißt demnach *modulo* und ist statisch. Die Signatur wird so beschrieben, dass zwei Integer-Werte als Input benötigt werden und ein Integer zurückgeliefert wird. Die Integer-Werte werden nun aus den Registern, statt – wie auf der *XMLVM*-Projektseite beschrieben - vom Stack, gelesen und das Ergebnis wird ebenfalls in ein Register geschrieben. Hinter *dex:rem-int* verbirgt sich die Funktion, die das Ergebnis berechnet. Um aus dieser XML-Repräsentation den gewünschten Objective-C Code generieren zu können, muss erneut in das XSLT-Template *xmlvm2objc.xsl* nachgesehen werden:

```
<xsl:template match="dex:rem-int|dex:rem-int-2addr">
  <xsl:text> _r</xsl:text>
  <xsl:value-of select="@vx"/>
  <xsl:text>.i = _r</xsl:text>
  <xsl:value-of select="@vy"/>
  <xsl:text>.i % _r</xsl:text>
  <xsl:value-of select="@vz"/>
  <xsl:text>.i;
  </xsl:text>
</xsl:template
```

dex:rem-int kann demnach so berechnet werden:

```
_r0.i = _r1.i % _r2.i;
```

Die jeweiligen Typen werden weiterhin durch das erwähntes *union* in C gemappt, so dass der generierte Objective-C Code der Modulo Methode letztendlich wie folgt ausschaut:

```
+ (int) modulo___int_int :(int)n1 :(int)n2
{
        XMLVMElem _r0;
        XMLVMElem _r1;
        XMLVMElem _r2;
        _r1.i = n1;
        _r2.i = n2;
        _r0.i = _r1.i % _r2.i;
        return _r0.i;
}
```

Um den in Java geschriebenen Hallo-Welt Code, welcher schon für *J2ObjC* genutzt wurde, in *XMLVM* als ein nur leicht komplexeres Beispiel zu Cross-Kompilieren, erfolgt folgender Aufruf:

```
java -Xmx700m -jar xmlvm.jar --in=bin --out=out --target=objc --app-name=test
```

In /bin liegt die in Bytecode vorliegende Test.class. *XMLVM* erstellt zunächst wieder ein XML-Model, in dem die Klasse repräsentiert wird. Dieses Model wird mit den bereits erwähnten XLST-Templates in den Code der gewünschten Zielplattform transformiert, wozu die Option *--target* dient. Das Ergebnis ist eine Datei namens *de_mahlert_xmlvm_test.m* mit dazugehörigem Header. *de.mahlert.xmlvm* ist das in Java erstelle Package und *test* die Klasse mit dem bekannten Hallo-Welt Code. *XMLVM* erstellt für jede Klasse eines Paketes eine eigene Datei. Ein Auszug aus der *de_mahlert_xmlvm_test.m*:

```
+ (void) main___java_lang_String_ARRAYTYPE :(XMLVMArray*)n1
{
        XMLVMElem _r0;
        XMLVMElem _r1;
        XMLVMElem _r2;
        _r2.o = n1;
        _r0.o = JAVA_NULL;
        _r1.o = JAVA_NULL;
        [_r2.o retain];
        _r0.o = [java_lang_System _GET_out];
        [_r0.o retain];
        _r1.o = @"Hallo Welt!";
        [((java_io_PrintStream*) _r0.o) println___java_lang_String:_r1.o];
        [_r0.o release];
        [_r1.o release];
        [_r2.o release];
        return;
}
@end
int main(int argc, char* argv[])
{
        NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
        xmlvm_init();
        XMLVMArray * args;
        if (argc==0) {
          args=nil;
        } else {
          args= [XMLVMArray createSingleDimensionWithType:0 andSize:argc];
          for (int i = 0; i < argc; i++) {
            [args replaceObjectAtIndex:i withObject:[NSString
            stringWithUTF8String:argv[i]]];
          }
        }
        [de_mahlert_xmlvm_test main___java_lang_String_ARRAYTYPE: args];
        [pool drain];
        [args release];
        return 0;
}
```

Das Ergebnis ist von einem Menschen fast nicht mehr lesbar. Man kann davon ausgehen, dass mit diesem Code kein Entwickler mehr arbeiten kann. D.h. der Code funktioniert zwar wie gewünscht, wenn er kompiliert wird, allerdings ist es quasi unmöglich einen solchen Code zu warten oder zu

erweitern. Wenn darin aber kein Interesse besteht und nur eine funktionierende und lauffähige Binary das Ziel ist, wäre *XMLVM* hervorragend dazu geeignet.

Die target-Option, welche das Ziel definiert, unterstützt folgende Eingaben:

```
--target=[xmlvm|jvm|clr|dfa|class|exe|js|c|posix|python|qooxdoo|iphone]
```

Neben *iphone* gibt es in den aktuellen Versionen von *XMLVM* auch das Target *iphonec*. Während *iphone* den Ausgangscode zu Objective-C Code transformiert, generiert *iphonec* C Code. Der generierte C Code hat gegenüber Objective-C den Vorteil, dass er einen integrierten Garbage Collector (GC) enthält, welcher weniger anfällig für Memory Leaks als das in Objective-C verwendete Reference Counting ist.

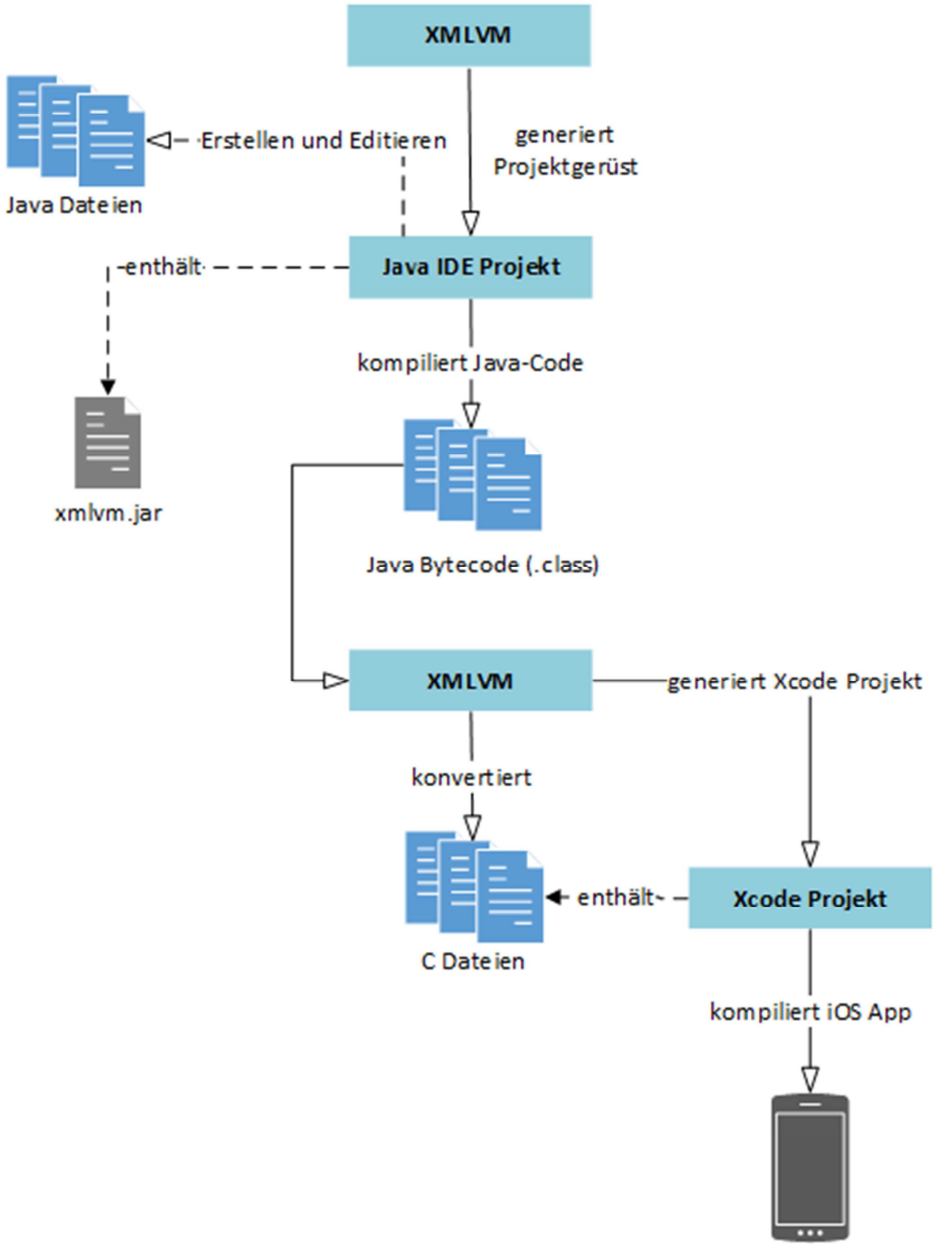


Abbildung 7: XMLVM Entwicklungsablauf

Wird vom Entwickler die Option --*skeleton* angegeben, erzeugt *XMLVM* das Startprojekt, welches in einer Java IDE geöffnet werden kann. In der Entwicklungsumgebung muss die *xmlvm.jar* im Klassenpfad eingetragen sein. Wird *iphone* oder *iphonec* als Target genutzt, generiert *XMLVM* nicht nur den entsprechenden Code, sondern inkludiert auch sämtliche Abhängigkeiten und ein Xcode-Projekt, mit dem die installierbare Anwendung kompiliert werden kann. Der Ablauf mit dem Target *iphonec* ist in Abb. 7 dargestellt. Diese Prozedur sieht auf den ersten Blick relativ einfach und kurz aus, kostet aber in der Praxis einige Minuten Zeit, weshalb eine Entwicklung, bei der man z.B. oft Testen und Cross-Kompilieren muss, als nicht praktikabel einzustufen ist. Mit *Codename One*, welches später in dieser Arbeit vorgestellt wird, existiert jedoch ein Framework, welches diesen Vorgang automatisiert und enorm beschleunigt (siehe Kapitel 6.3 „*Codename One*“).

In Objective-C entwickelte Apps sind in der Regel langsamer als in C geschriebene Apps, da Objective-C *Dynamic Method Binding* implementiert, bei dem Methodenaufrufe, im Gegensatz zu C, erst zur Laufzeit an konkrete Methodenimplementierungen gebunden werden, d.h. konkret vergeht beim Aufruf einer Funktion in Objetictive-C etwa zwei bis drei Mal so viel Zeit als in C [Cat14] [Buc14]. Da der von *XMLVM* generierte Code zwar komplex und nahezu unleserlich, dafür aber funktional ist, stellt sich automatisch die Frage, inwiefern er performant ist bzw. ob er mit direkt in C geschrieben Apps mithalten kann. Um dies zu überprüfen, hat der Entwickler Steve Hannah das mathematische Problem der Türme von Hanoi [Köl14] in Java, in Objective-C und in C implementiert, wobei der Java Code durch *XMLVM* in C konvertiert wurde. Die Laufzeiten zum Lösen des Problems sind wie folgt [Ste14]:

C (selbst geschrieben):	31.7 Sekunden
C (*XMLVM*):	40.2 Sekunden
Objective-C (selbst geschrieben):	154.5 Sekunden

Der durch *XMLVM* generierte Code ist demnach zwar etwas langsamer als der selbstgeschriebene C Code, dafür aber deutlich schneller als der Objectice-C Code. Dies bedeutet, dass eine generierte iPhone Applikation schneller sein kann als eine solche, die eigenhändig in Objective-C entwickelt wurde. Dieser Umstand macht *XMLVM* zu einem äußerst interessanten Projekt.

4.2.3 RoboVM

RoboVM ist ein neues Open-Source Projekt, welches momentan (Stand: Februar 2014) in der Version 0.0.10 vorliegt [Tri14]. Das Ziel von *RoboVM* ist es, native iOS Applikationen in Java zu schreiben, um so große Teile des Java-Codes mit anderen Projekten, z.B. einer Android Applikation, zu teilen. Die Dokumentation ist momentan noch äußerst lückenhaft, so dass man zumindest zum

jetzigen Zeitpunkt noch nicht produktiv arbeiten kann. Das Potenzial lässt sich jedoch schon erkennen. Der Ablauf einer Entwicklung in *RoboVM* ist in der folgenden Abbildung dargestellt:

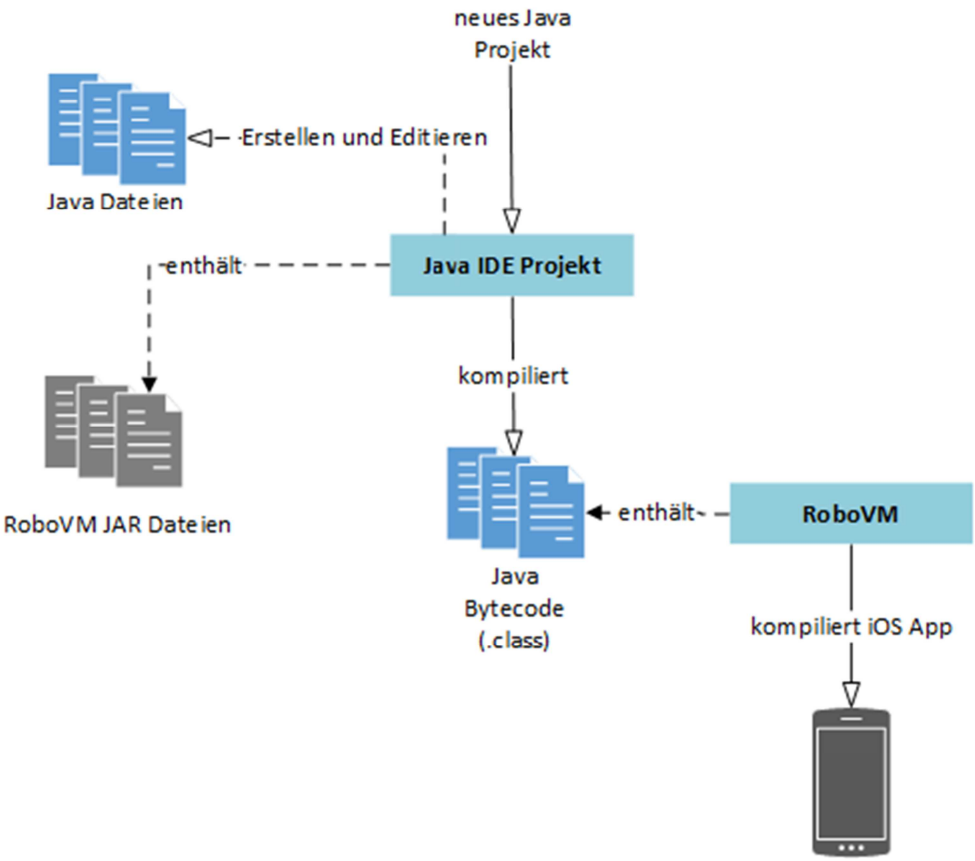


Abbildung 8: RoboVM Entwicklungsablauf

Im Java-Code lassen sich vom Namen her an Objective-C angelehnte Methoden aufrufen, um so programmatisch eine Benutzeroberfläche zu erzeugen. In den *RoboVM* JAR-Dateien sind die Cocoa Bindings enthalten, wodurch ein Zugriff auf die nativen Funktionen von iOS gewährleistet wird. Es ist daher zwingend erforderlich, dass diese JARs auch im Klassenpfad enthalten sind. Man kann die JAR-Dateien von *RoboVM* als eine Art Brücke zwischen Java und Cocoa / Objective-C verstehen, wobei die Überbrückung momentan noch viele Lücken aufweist. Ein Vorteil ist jedoch, dass man direkt aus der IDE heraus, also z.B. Eclipse, einen iOS Simulator starten kann und keinen Umweg über Xcode gehen muss. MacOS mit installiertem Xcode sind aber weiterhin Pflichtvoraussetzungen. Damit der Java Code letztendlich funktioniert, werden die Java Dateien zu Java Bytecode kompliliert. *RoboVM* kommt mit einem Ahead-of-Time Compiler, welcher auf *LLVM* [Lat14] basiert und den Java Bytecode in nativen ARM Maschinencode übersetzt.

4.2.4 Avian

Avian ist eine abgespeckte Java Virtuelle Maschine in der Version 0.7, die ein Subset der Java Features unterstützt [Rea14]. Das Projekt ist Open-Source und wurde nicht speziell für die mobile

Entwicklung erschaffen, eignet sich aber trotzdem dafür, sofern man Java-Code auf andere Plattformen nutzen möchte. *Avian* wurde in C++ implementiert, benötigt jedoch keine C++ Standardbibliotheken. Der Fokus der VM lag dementsprechend darauf, dass sie klein und schnell ist, wodurch es prädestiniert für eingebettete Systeme ist. Da mobile Entwicklung nicht der primäre Zweck ist, gibt es kaum eine Dokumentation diesbezüglich. Anders als bei *RoboVM* gibt es keine Brücke zwischen Java und Cocoa / Objective-C. Man kann sich einen Einsatz von *Avian* im Kontext einer mobilen Cross-Entwicklung allerdings trotzdem vorstellen, wenn man nur die Business Logik teilen möchte oder eine Benutzeroberfläche benötigt. Eine Entwicklung, die auf *Avian* aufsetzt, würde wie folgt aussehen:

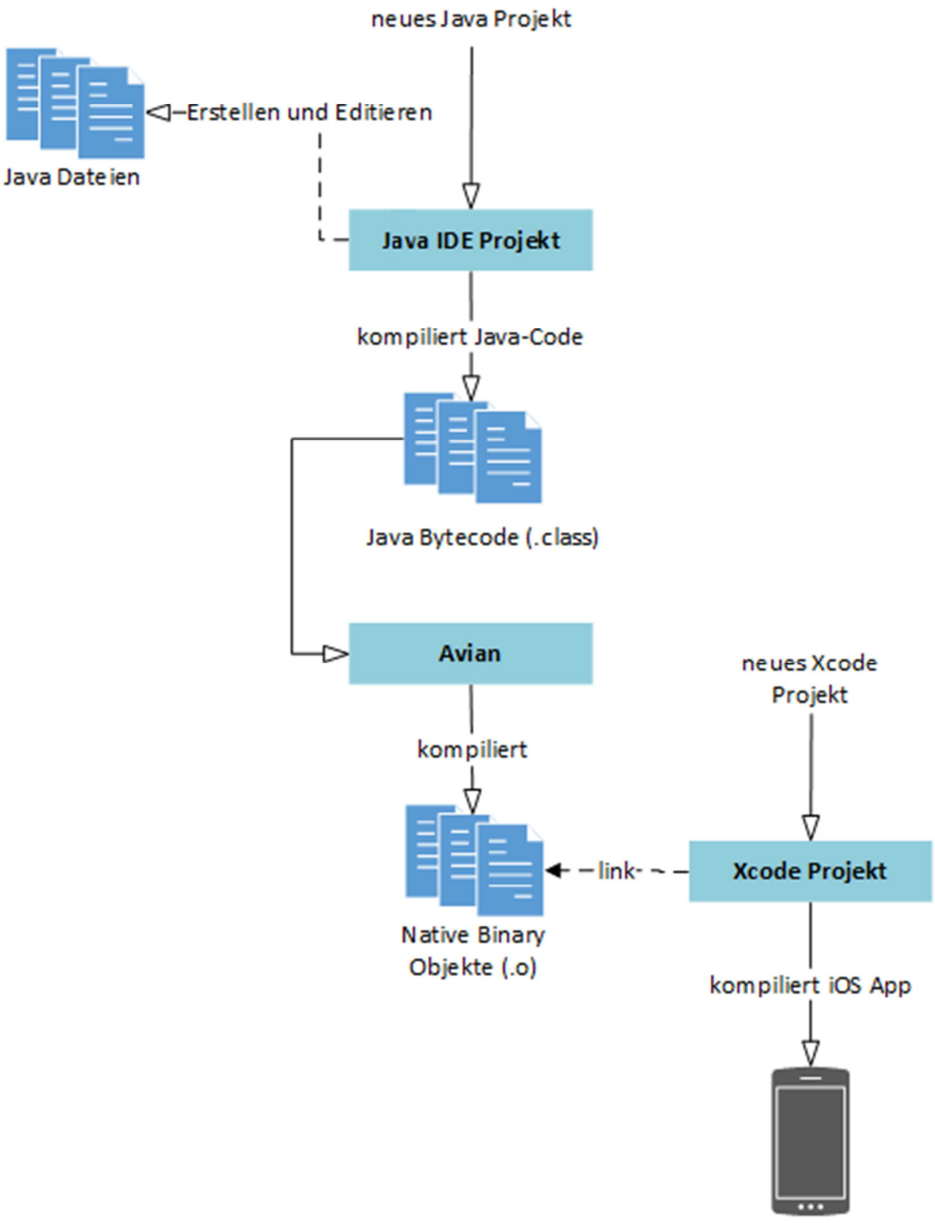


Abbildung 9: Avian Entwicklungsablauf

Man entwickelt in Java die Business Logik, welche direkt in ein Android Projekt importierbar ist. Da *Avian* die in den iOS verwendeten ARM-CPUs unterstützt, wird der Java Bytecode zu einem nativen Binary Objekt kompiliert, wogegen man in einem Xcode Projekt linken muss, um so mittels eines Java-Native-Interfaces (JNI) auf die Funktionen Zugriff zu erhalten.

4.2.5 Hyperloop

Hyperloop ist ein neues Open-Source Projekt, welches im Vorlauf dieser Arbeit noch nicht existierte. Das Projekt wurde Mitte Oktober 2013 vorgestellt und konvertiert in JavaScript geschriebenen Code zu einem nativen Code [Hay14]. Momentan werden iOS und WinRT unterstützt, allerdings hat *Hyperloop* noch keinen stabilen, sondern nur einen experimentellen Status erreicht und es wird im GIT ausdrücklich von einem produktiven Einsatz abgeraten [App141]. Ein Support für Android ist angedacht, zum jetzigen Zeitpunkt aber noch nicht enthalten. Auch die Dokumentation ist bestenfalls als rudimentär zu bezeichnen.

Ein Beispiel deutet aber schon das Potenzial an:

```
@import("UIKit");
@import("CoreGraphics");
var view = new UIView();
view.frame = CGRectMake(0,0,100,100);
```

Hyperloops Compiler importiert im obigen Code das UIKit Framework und löst automatisch Abhängigkeiten auf, wodurch man mittels JavaScript Code auf das UIView Interface zugreifen kann, welches im UIKit Framework in Objective-C definiert ist. Auch GCRectMake ist eine normale C Funktion, die nach dem Import der Bibliothek durch bekannte JavaScript Syntax aufrufbar ist. Der Compiler transformiert diesen Code zu nativem Code, im Falle von iOS zu Objective-C, weshalb man von einer nativen Entwicklung sprechen kann, obwohl mit JavaScript gearbeitet wird.

Für JavaScript-Entwickler stellt diese Methode eine interessante Lösung dar. Dennoch sollte man, sobald das Projekt einen stabilen Status erreicht und zur Produktion freigegeben wird, einen wesentlichen Punkt im Auge behalten. Dabei sind die Debugging-Möglichkeiten von *Hyperloop* gemeint. Zumindest momentan lässt sich in *Hyperloop* geschriebener JavaScript nicht als solcher debuggen. Stattdessen muss man die Applikation kompilieren, um den nativen Code, im Falle von iOS wäre das Objective-C, zu erhalten. Der Objective-C Code muss dann in Xcode importiert werden, wo er dann debugged werden kann. Das Vorgehen ist daher umständlich und zudem ist es fragwürdig, inwiefern ein solches Debugging noch hilfreich ist, da ein Entwickler aus der JavaScript-Welt sich möglicherweise nicht in der Objectice-C Welt auskennt. Des Weiteren muss man den Konvertierungsprozess von JavaScript zu Objective-C verstehen, um eventuell in Objective-C gefundene Bugs in JavaScript eliminieren zu können. Ob und wie dieses Problem in *Hyper-*

loop noch gelöst wird, sollte unbedingt im Auge behalten werden, bevor man seine Entwicklung darauf aufbaut.

4.2.6 Mono

Mono ist ein Open-Source Projekt in der Version 3.2.7 (Stand: Februar 2014) mit dem Ziel einer plattformunabhängigen Softwareentwicklung [Xam14]. Es implementiert Microsofts .NET Framework auf Basis der ECMA Standards für C# und der Common Language Runtime (CLR). Das Äquivalent der CLR aus der Java-Welt ist die Java Virtuelle Maschine (JVM). Ebenso wie die JVM mit dem Java Bytecode ihre eigene Sprache besitzt, führt die CLR die Common Intermediate Language (CIL) aus. CIL ist dafür ausgelegt auf der CLR zu laufen, ist ansonsten aber völlig plattformunabhängig.

Die *Mono Runtime*, d.h. die CLR, ist überwiegend in C geschrieben und eine native Anwendung, die spezifisch für jede Zielplattform entwickelt sein muss. Momentan wird eine Vielfalt von Systemen unterstützt [Xam14]:

- Linux
- Mac OS X, iPhone OS
- Sun Solaris
- BSD - OpenBSD, FreeBSD, NetBSD
- Microsoft Windows
- Nintendo Wii
- Sony PlayStation 3

Da Android eine Version von Linux ist, wird auch dieses unterstützt. Die Android Version von Mono enthält zusätzlich zu den Kernbibliotheken Interfaces zu den Android APIs, wodurch eine native Entwicklung für Android möglich ist.

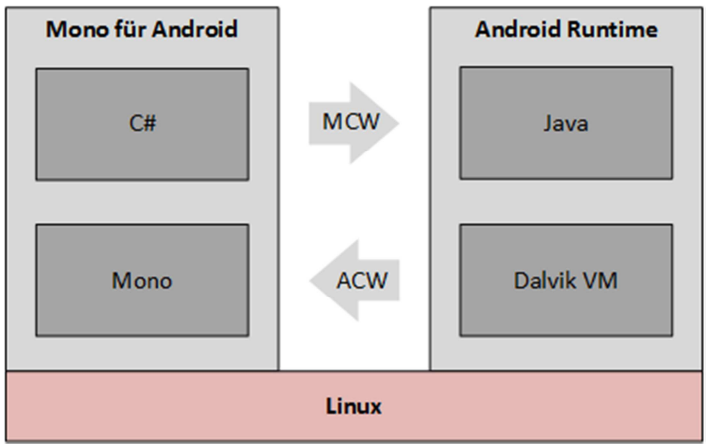


Abbildung 10: Die Mono für Android Runtime existiert neben der Android Runtime

Die Android Schnittstellen sind durch den Namespace *Mono.Android* erreichbar. Die Mono Runtime existiert auf Android Seite an Seite mit der Android Runtime. Wann immer eine Kommunikation zwischen der Mono für Android Runtime und der Dalvik Runtime notwendig ist, nutzt Mono Wrapper, welche in Abb. 10 durch Pfeile repräsentiert sind. Mono für Android generiert Managed Callable Wrappers (MCWs) und nutzt diese, sobald die Mono Runtime Zugriff auf nativen Android Code benötigt [Ols12]. Alle Schnittstellen, die via *Mono.Android* erreichbar sind, nutzen MCWs. Umgekehrt, wenn die Android Runtime Zugriff benötigt, werden Android Callable Wrappers (ACWs) als native Java-Brücken genutzt. ACWs sind nötig, da die Dalvik VM zur Laufzeit keine Klassen via die JNI-Funktion *DefineClass()* registrieren kann, da diese Funktion von Dalvik nicht unterstützt wird. Die ACWs sind daher in Java implementiert und leiten die Nachrichten der Android Runtime weiter. Da ACWs zur Kompilierungszeit erstellt werden, kommt es zu ein paar Einschränkungen. Beispielsweise gibt es nur einen limitierten Support für generische Typen, so dass ein Aufruf der Methode *Android.Content.Intent.GetParcelableExtra* nur ein *Java.Lang.Object* zurückliefert, statt des eigentlichen Typs, da dieser zur Kompilierungszeit unbekannt ist [Xam141].

Mono unterstützt sowohl eine Just-in-Time (JIT) als auch eine Ahead-of-Time (AOT) Kompilierung. Für Android wird JIT verwendet, während Apple auf iOS kein JIT erlaubt und daher iOS AOT benutzt, den Code also statisch kompiliert. Dies führt, ähnlich wie auf Android, zu einigen Limitierungen, weshalb z.B. keine generischen Subklassen vom Typ *NSObject* unterstützt werden [Xam13]. Mono für iOS, auch MonoTouch genannt, ermöglicht den Zugriff auf die Cocoa Touch APIs, wodurch native iOS Anwendungen ermöglicht werden. Für eine iOS Entwicklung ist zwingend MacOS erforderlich, um die Applikation kompilieren zu können.

Für Android ist es theoretisch möglich, die eigentliche App und die Mono Runtime zu trennen und beide separat in den Play Store zu stellen. Für iOS ist dies nicht möglich, da Apple shared Runtimes verbietet [App14], weshalb dort keine Wahlfreiheit besteht und die Mono Runtime immer mitgeliefert werden muss. Da eine mobile Anwendung aber wahrscheinlich niemals auf sämtliche Funktionen der Mono Runtime zugreift, wäre es ineffizient, wenn bei jeder Applikation die komplette Runtime integriert wäre. Daher besitzt Mono ein statisches Analysetool, welches zur Kompilierungszeit den ungenutzten Code entfernt, wodurch die Binary nur diese Teile der Runtime enthält, die tatsächlich genutzt werden [Sha12].

Eine Entwicklung für Windows Phone erweist sich als ein wenig einfacher als für Android oder iOS. Dies ist darin begründet, dass das .NET Framework das native Framework von Windows Phone ist. Für eine Entwicklung ist Windows mit Visual Studio und dem Windows Phone SDK erforderlich. Damit der C# Code lauffähig ist, sind keine weiteren Anpassungen seitens Mono notwendig.

Mono unterstützt alle wichtigen Konzepte wie z.B. Threading oder Garbage Collection. Es ist daher eine interessante Lösung für eine Cross-Plattform-Entwicklung und insbesondere attraktiv für Entwickler, die aus der .NET Welt stammen. Mit *Xamarin* wird im weiteren Verlauf in dieser Arbeit ein Framework vorgestellt, welches auf *Mono* aufsetzt (siehe Kapitel 6.2 „*Xamarin*“).

4.2.7 In the box

In the box ist ein bereits eingestelltes Open-Source Projekt, welches für den produktiven Einsatz nicht nutzbar ist [Goo142]. Es wird in dieser Arbeit dennoch erwähnt, da die Idee dahinter einen Ansatz verfolgt, der von allen anderen Lösungen vollkommen abweicht.

In the box hatte das Ziel, die Dalvik VM und die Android API von Android 2.3 („*Gingerbread*“) auf iOS zu portieren, wodurch die entsprechenden Android Applikationen auf iOS ausführbar wären. Aus objektiver Sicht könnte ein solcher Ansatz für zwei Zielgruppen von Interesse sein:

1) Entwickler, die Interesse an den technischen Hintergründen haben (Forschung) oder ein eigenes Framework auf Basis der Dalvik VM implementieren möchten.
2) Unternehmen, die ein mobiles Betriebssystem entwickeln, welches einen sehr geringen Marktanteil besitzt.

Bezüglich des ersten Punktes wären insbesondere die folgenden Fragen von Bedeutung:

a) Ist es möglich, die Dalvik VM auf die Zielplattform zu portieren, ohne darauf root-Rechte zu besitzen?
b) Würde eine solche Lösung im App Store der jeweiligen Plattform den Review-Prozess überstehen?
c) Um die Android Applikationen installieren zu können, müsste wahrscheinlich auch ein eigener Installer für die Zielplattform geschrieben werden. Wie lässt sich dieser in das bestehende Ökosystem integrieren?

Die Fragen zielen also darauf ab, ob eine Portierung der Dalvik VM so umsetzbar wäre, dass möglichst jeder Nutzer der Zielplattform davon profitieren könnte. Sobald root-Rechte benötigt werden, ist ein Standardnutzer von dieser Lösung ausgeschlossen. Eine solche Lösung wäre aus diesem Grunde für Unternehmen, welche lediglich an einer großen Verbreitung der eigenen App(s) Interesse haben, wirtschaftlich gesehen nicht sinnvoll.

Unternehmen, die ein eigenes mobiles Betriebssystem entwickeln, könnten hingegen eine portierte Dalvik VM direkt integrieren und müssten sich keine Gedanken bzgl. root-Rechte und dergleichen machen. Die Motivation dieser Unternehmen könnte darin liegen, dass sie auf Grund ei-

nes geringen Marktanteils zu wenig Entwickler anziehen und durch eine Portierung Zugriff auf die Vielfalt an Applikation eines anderen Systems erhalten. Einen solchen Weg hat beispielsweise Blackberry eingeschlagen [Lum14]. Die Gefahr darin liegt jedoch an einer Kannibalisierung des eigenen Ökosystems. Denn warum sollte man noch nativ für Blackberry entwickeln, wenn man über den Umweg Android ein Vielfaches der Nutzer erreicht?

4.2.8 Halbautomatisierte und vollständige Lösungen

Die bisher erwähnten Lösungen sind größtenteils als unterstützend zu bezeichnen, da sie darauf abzielen, Teile des Codes auf mehreren Systeme laufen zu lassen. Die Rede ist von *größtenteils*, da bei einigen Techniken der Übergang durchaus fließend ist. Jedoch kann muss man sich nicht ausschließlich auf das Entwickeln der eigentlichen App konzentrieren, da man zunächst einmal die eingesetzte Technologie verstehen und beherrschen muss. Auch die Entwicklungsabläufe sind zeitaufwendig, da man die Generierung des nativen Codes manuell anstoßen und den dann erhaltenen Code oft in einer anderen Entwicklungsumgebung importieren muss, um die jeweilige Binary der App für die gewünschte Zielplattform zu erhalten.

Halbautomatisierte und vollständige Lösungen setzen auf die vorgestellten Technologien auf. Allerdings treten diese soweit in den Hintergrund, dass ein Entwickler weder Know-How bzgl. der Funktionsweise mitbringen muss, noch mitbekommt, dass eine solche Technologie im Hintergrund arbeitet. Dieser Umstand spart wertvolle Zeit und ermöglicht es einem Entwickler, sich auf die eigentliche Implementierung einer Applikation konzentrieren zu können.

Der Unterschied zwischen halbautomatisierten Lösungen und vollständigen Lösungen liegt darin, dass ein Entwickler bei ersterer weiterhin Know-How für die unterschiedlichen Zielplattformen mitbringen muss. Typischerweise wird der Code der Business-Logik und die benötigten Daten und deren Zugriff darauf zwischen allen Plattformen geteilt, während die plattformspezifischen Funktionen wie Hardwarezugriffe und die Benutzeroberfläche für jede Plattform separat entwickelt werden.

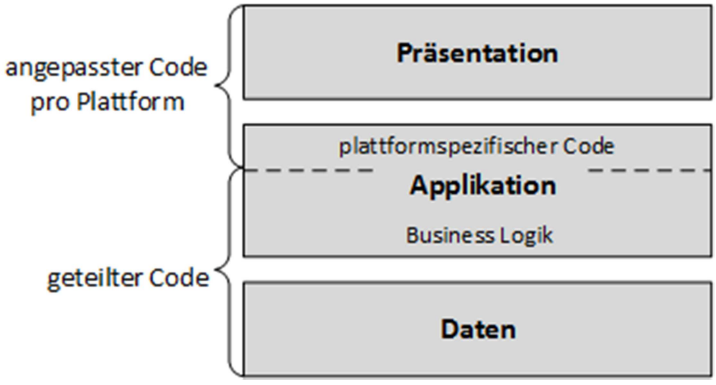


Abbildung 11: Code Separation halbautomatisierter Lösungen

Benötigt die App zum Beispiel einen Zugriff auf die Kamera, gehört der Zugriff auf einer solchen zwar zum Applikationscode, muss aber für jede Zielplattform über dessen native Schnittstellen separat entwickelt werden. Idealerweise ist dieser Teil einer Applikation, wie in Abbildung 11 angedeutet, aber deutlich geringer als der Code, der zwischen allen Plattformen geteilt werden kann, so dass sich eine Cross-Plattform-Entwicklung anbietet. Der große Vorteil der halbautomatisierten gegenüber den unterstützenden Lösungen ist der, dass alles innerhalb einer Entwicklungsumgebung stattfindet und auf Knopfdruck die fertigen Applikationen der jeweiligen Plattform kompiliert werden. Einem Entwickler ist daher nicht bewusst, was genau im Hintergrund nach Klick auf den Kompilierungsbutton geschieht.

Vollständige Lösungen besitzen eine weitere Abstraktionsebene, wodurch keine Separation des Codes mehr stattfindet. Diese Frameworks verfolgen eine "*Write once, run anywhere*"-Philosophie. D.h. selbst Hardwarezugriffe oder die Erstellung von Benutzeroberflächen ist plattformunabhängig. Dies ist dadurch möglich, dass die Frameworks eigene Schnittstellen implementieren, die statt der nativen Schnittstellen aufgerufen werden. Da eine native Applikation entwickelt werden soll, fungieren die selbstgebauten Schnittstellen als eine Art Wrapper hinter denen sich die nativen Aufrufe pro Plattform verbergen. Kompiliert man beispielsweise eine Android Applikation, wird gegen die Android Schnittstellen gebaut. Da unterschiedliche Betriebssysteme auf unterschiedliche Architekturen und unterschiedliche native Sprachen setzen, werden Techniken eingesetzt, die im Verlaufe dieses Kapitels („*Möglichkeiten der nativen Cross-Plattform-Entwicklung*“) vorgestellt wurden. Ein Entwickler muss sich um all diese Prozesse jedoch keine Gedanken machen bzw. benötigt auch kein spezielles Know-How, da ihm diese Arbeiten vom Framework abgenommen werden.

Für eine native Cross-Plattform-Entwicklung stellt sich nun die Frage, auf welche Strategie gesetzt werden soll. Zusammengefasst gibt es die folgenden Möglichkeiten:

1) Unterstützende Lösungen (*J2ObC*, *XMLVM*, …)
2) Implementierung eines eigenen Frameworks auf Basis der unterstützenden Lösungen
3) Halbautomatisierte Lösungen (*Code Separierung*)
4) Vollständige Lösungen (*Write once, run anywhere*)

Die Entwicklungsabläufe bzw. die dafür benötigte Zeit und vor allem das benötigte Know-How, welches man sich für eine Entwicklung aneignen muss, schließen unterstützende Lösungen für eine effiziente Entwicklung aus. Selbst wenn es innerhalb eines Unternehmens einen Experten gibt, kann sich dieser nicht um alle benötigten Entwicklungsschritte aller Entwicklungsprojekte kümmern. Ein Unternehmen würde sich zudem abhängig von einem solchen Angestellten machen.

Wenn er das Unternehmen verlassen würde, entstünde eine große Lücke. Die Technologien sind zwar höchst interessant, da sie die Grundlagen aller anderen Möglichkeiten bilden, sollen aber in dieser Arbeit im Folgenden nicht als alleinige Lösung dienen, da nach einer effizienten und wiederverwendbaren Lösung gesucht wird. Dabei ist wiederverwendbar nicht ausschließlich auf den Code bezogen, sondern auch auf die Entwickler, die damit arbeiten und zu einem späteren Zeitpunkt den vorhandenen Code ggf. warten müssen.

Da alle in dieser Arbeit vorgestellten unterstützenden Lösungen Open-Source sind, könnte man aus ihnen ein eigenes Framework entwickeln, welches auf die beschriebenen Techniken aufsetzt und die Entwicklung effizient gestalten würde. Die Implementierung eines eigenen Frameworks erfordert aber sehr viel Know-How und kostet Zeit, die nicht in die Entwicklung der eigentlichen mobilen Anwendung fließen kann. Weiterhin ist ein eigenes Framework prinzipiell nur notwendig, sofern es keine bestehende halbautomatisierte oder vollständige Lösung gäbe, die den Anforderungen entspräche. Darum geht es im weiteren Verlauf dieser Arbeit um die Evaluation bestehender Lösungen. Die Entwicklung eines eigenen Frameworks wäre aus diesem Grund der letzte Ausweg und wird nicht weiter betrachtet, sofern sich nach der Evaluierung der Frameworks nicht herausstellen sollte, dass kein Framework den Anforderungen genügt.

Write once, run anywhere (WORA) ist eine für Entwickler sehr verführende Philosophie. Doch kann dieser Ansatz auf Grund der weiteren Abstraktionsebene im Vergleich zu halbautomatisierten Lösungen auch Schwachpunkte aufweisen, die sich als nicht akzeptabel herausstellen könnten. WORA kann nur funktionieren, so lange die gewünschten nativen Funktionen vom Framework zu einer Schnittstelle abstrahiert wurden. Doch kann ein Framework überhaupt alle gewünschten Funktionen abstrahieren? Und wie verhält es sich, wenn eine zwingend benötigte Schnittstelle nicht vorhanden ist? Lässt sich die mobile Anwendung dann nicht mehr in diesem Framework implementieren?

Doch auch halbautomatisierte Lösungen werfen Fragen auf. Das Ziel einer wiederverwendbaren effizienten Cross-Plattform-Entwicklung ist das Code Sharing zwischen allen Plattformen. Die in halbautomatisierten Lösungen benötigte Code Separierung wirkt diesem Ziel entgegen. Es bleibt daher die Frage zu klären, wie viel Code tatsächlich geteilt werden kann und wie viel Code pro Plattform angepasst werden muss.

Es gibt auch Fragen, die sowohl halbautomatisierte als auch vollständige Lösungen aufwerfen. Zum Beispiel ist für eine iOS-Entwicklung, zumindest für die Kompilierung der Binary, prinzipiell immer MacOS nötig. Ist eine Cross-Plattform-Entwicklung daher nur auf MacOS möglich oder bieten die Frameworks auch dafür Lösungen an?

Um all diese Fragen beantworten zu können, werden im nächsten Kapitel die Anforderungen und die Methodologie erläutert. Ausgehend davon werden halbautomatisierte und vollständige Lösungen gesucht und evaluiert.

5 Anforderungen und Methodologie

Im vorherigen Kapitel wurden die Entwicklungsmöglichkeiten mobiler Anwendungen beschrieben und dabei insbesondere die Möglichkeiten einer nativen mobilen Cross-Plattform-Entwicklung erläutert, welche in die drei Kategorien unterstützende, halbautomatisierte und vollständige Lösungen eingeordnet wurden. Es wurde festgelegt, dass eine halbautomatisierte oder eine vollständige Lösung für eine wiederverwendbare und effiziente Entwicklung zu bevorzugen ist.

In diesem Kapitel werden die Kriterien festgelegt, anhand derer die möglichen Lösungen evaluiert werden. Es wird die aktuelle Situation eines großen Telekommunikationsunternehmens, im folgenden *TK* genannt, beschrieben, welches nicht nur eine einzige mobile Anwendung, sondern auch eine einheitliche Strategie für die zukünftige Entwicklung mobiler Applikationen und einer mobilen Entwicklungsplattform entwickeln möchte.

5.1 Herausforderungen

Um die aktuellen Herausforderungen der Firma *TK* verstehen zu können, soll zunächst beschrieben werden, welche Erfahrungen das Unternehmen mit bisherigen mobilen Anwendungen gesammelt hat, welche Entscheidungen getroffen wurden und wie aktuell der typische Entstehungsprozess einer App bei *TK* aussieht.

Die Firma *TK* bietet mehrere unterschiedliche mobile Anwendungen für Android und iOS im Play Store, respektive in Apples App Store, an. Andere Plattformen werden derzeit nicht unterstützt. Die genannten Apps unterscheiden sich zwar funktional, implementieren aber teilweise ähnliche Hilfsfunktionen, wie z.B. Login im Kundencenter. Die Apps wurden von *TK* nicht selbst, sondern von externen Firmen entwickelt. Da *TK* kein besonders großes Know-How im Bereich der mobilen Entwicklung besaß, definierte das Produktmanagement bei Beauftragung einer App keine fertigen Anforderungen. Stattdessen bezogen sich diese hauptsächlich auf das Äußere einer App, wobei eine genaue Spezifizierung der Funktionen vernachlässigt wurde. Daher gab es auch keine Präferenzen bzgl. der Art der Entwicklung, so dass prinzipiell eine webbasierte oder eine native Applikation entstehen konnte. Die externen Entwicklungsfirmen entschieden sich für hybride Applikationen, die in *Sencha Touch* entwickelt und mit *PhoneGap* gekapselt wurden. Da die funktionalen Anforderungen im Vorhinein nicht ausreichend definiert wurden, kam es während des Entwicklungsprozesses zu mehreren Änderungswünschen seitens des Produktmanagements.

Man musste seitens *TK* feststellen, dass die Resonanz der Kunden auf die Apps nicht positiv war und die Rezensionen und Kritiken schlecht ausfielen, was sich besonders in den Bewertungen der Apps in den jeweiligen App Stores widerspiegelte. Von fünf möglichen Sternen, wobei fünf verge-

bene Sterne die höchste Zufriedenheit darstellen, wurden überwiegend nur ein oder zwei vergeben. Die Kommentare der Kunden tendierten in die Richtung, dass die Idee der App gefiel, sie jedoch schlecht umgesetzt und nahezu unbenutzbar sei. Damit war vor allem die schlechte Performanz der Anwendung gemeint, welche sich auch stark auf die Funktionalität auswirkte. Es wurde von „Hängern“ berichtet, die teilweise so stark waren, dass sie eine Bedienung der gesamten Anwendung erschwerten oder gar blockierten. Außerdem klagten die Kunden u.a. über lange Ladezeiten und über eine „seltsame“ Benutzeroberfläche bzw. über die Navigierung innerhalb der App. All diese Punkte konnten weitgehend von *TK* nachvollzogen und teilweise reproduziert werden, weshalb man an die externe Entwicklungsfirma herantrat und die Probleme erläuterte. Zudem eignete man sich innerhalb dieser Zeit, d.h. zwischen Auftrag der Applikation und das Auftreten der negativen Rezensionen, Know-How im Bereich der mobilen Entwicklung an, was sich z.B. darin äußerte, dass ein zusätzlicher Mitarbeiter mit entsprechendem Wissen angestellt wurde. Gemeinsam mit der externen Entwicklungsfirma versuchte man die Probleme zu lösen. Einige Dinge konnten teilweise verbessert werden, was in den App Stores mit geringfügig besseren Bewertungen honoriert wurde. Allerdings gab es weiterhin überwiegend schlechte Kritiken und die Unzufriedenheit und der Druck, etwas verbessern zu müssen, wuchs an. Schlussendlich wurde die Erkenntnis erlangt, mit einer webbasierten Entwicklung an eine Grenze gestoßen zu sein, die weder den eigenen Anspruch, noch denen der Nutzer gerecht wird. Ein weiter Hauptgrund für die unbefriedigende Benutzbarkeit waren neben der Art der Entwicklung die permanenten Change Requests seitens des Produktmanagements. Es wurde erkannt, dass für die Entwicklung einer App ein besseres Anforderungsdokument und eine höhere Standardisierung notwendig sind.

Auf Grund dieser Erfahrungen und dem gewonnenen Know-How wurde beschlossen, dass die vorhandenen und zukünftigen mobilen Anwendungen nativ entwickelt werden sollen. Wie der momentane Entstehungsprozess einer mobilen Anwendung innerhalb von *TK* aussieht und welche Erfahrungen mit einer nativen Entwicklung gesammelt wurden, soll im Folgenden erläutert werden.

TK besteht aus mehreren Business-Einheiten, darunter u.a. eine, die für das Produktmanagement zuständig ist. Sobald das Produktmanagement die Idee einer neuen mobilen Anwendung hat, die einen Mehrwert für einen Kunden darstellt, wird diese mit der technischen Abteilung besprochen, um deren Machbarkeit zu evaluieren. Hat die Technologieabteilung keine gravierenden Einwände, die eine Umsetzung als unmöglich erscheinen lässt, werden die genauen Anforderungen gesammelt und festgehalten. Sind diese Prozesse abgeschlossen, wird das Entwicklungsprojekt pro Zielplattform ausgeschrieben. Momentan sind Android und iOS die Zielplattformen, jedoch kristallisiert sich bereits heraus, dass zukünftig ebenfalls eine Anwendung für Windows Phone 8 ge-

wünscht ist. Bzgl. Zielplattformen sollen sowohl Smartphones als auch Tablets unterstützt werden. Die Entwicklung der mobilen Anwendung wird an eine externe Firma vergeben, die pro Entwicklungsprojekt und pro Zielplattform variieren kann. Im Extremfall wird die Smartphone und die Tablet App separat vergeben und entwickelt. Aus diesen Gründen kann eine einzige native mobile Anwendung gleich mehrere Entwicklungsprojekte nach sich ziehen. Momentan sind es mit Android und iOS im Idealfall jeweils zwei Entwicklungsprojekte und im Extremfall, wenn Smartphone und Tablet Anwendung separat entwickelt werden, sogar vier Entwicklungsprojekte. Sofern Windows Phone 8 dazu kommt, könnten es dementsprechend drei bis sechs Entwicklungsprojekte werden, die für eine einzige Anwendung durchgeführt werden. Die jeweilige externe Entwicklungsfirma, welche den Auftrag erhält, kann das Projekt anschließend nur Hand in Hand mit der Technologieabteilung umsetzen, da in der Regel ein Zugriff auf Daten erfolgen muss, die über Schnittstellen im Backend von *TK* abrufbar sind. Es besteht keine vorhandene Bibliothek für die mobilen Betriebssysteme, welche Schnittstellen zum Backend implementiert, so dass die externen Firmen einen direkten Zugriff auf das Backend von *TK* erhalten und den Abruf der Daten ohne irgendeine vorhandene Standardisierung selbst implementieren. Dadurch kann es je nach externer Firma und Zielplattform oder gar Art des Devices (Smartphone/Tablet) vorkommen, dass eine Funktion mit gleichem Zweck auf eine unterschiedliche Art und Weise implementiert wird.

Eine negativ bewertete hybrid entwickelte Anwendung wurde bereits durch eine native Implementierung ersetzt bzw. durch Friendly User Tests (FUT) an ausgewählte Kunden vergeben. Die Zufriedenheit der Kunden hat sich signifikant verbessert, so dass jetzt deutlich höhere Bewertungen vergeben werden. Auf Grund dieser neuen Erfahrungen sieht man sich darin bestätigt, die mobilen Anwendungen weiterhin nativ zu entwickeln.

Allerdings steht eine Umsetzung, wie sie derzeit geschieht, vor einigen Herausforderungen, für die zukünftig eine Lösung gefunden werden muss:

- Wie beschrieben können derzeit bis zu vier Derivate einer Applikation entstehen (Android, iOS, Tablet, Smartphone). Dadurch entstehen, alle Apps und Derivate zusammengerechnet, Entwicklungskosten im Millionenbereich.

- Die Anzahl der Derivate könnte mit Zunahme weiterer Zielplattformen weiter steigen. Es kristallisiert sich bereits heraus, dass neben Android und iOS ebenfalls Windows Phone 8 gewünscht ist.

- Durch die vielen Entwicklungsprojekte, an denen auch Techniker seitens *TK* beteiligt werden, nimmt die Komplexität der Entwicklung einer mobilen Anwendung enorm zu. Bei derzeit möglichen vier Entwicklungsprojekten pro Anwendung ist es schwer die Projekte so aufeinander abzustimmen, dass ein Launch/Release-Termin der App festgelegt werden kann, den alle einhalten können. In der Praxis stellt sich bereits heraus, dass eine App für eine bestimmte Plattform, deren Entwicklung abgeschlossen ist, nicht freigegeben werden kann, da noch auf die Fertigstellung für eine andere Zielplattform gewartet werden muss.

- Durch die vielen Entwicklungsprojekte, die teilweise von verschiedenen externen Firmen durchgeführt werden, entstehen viele unterschiedliche Codebasen, die keine große Überlappung haben.

- Auf Grund der vielen Projekte und der zahlreichen Codebasen entstehen viele Fehlerquellen. Die externen Firmen erhalten zwar ein Anforderungsdokument, doch ein Kontrollieren seitens *TK* wird fortwährend schwerer. Bei nur einem Entwicklungsprojekt stellt es kein Problem dar, Design oder Workflow bis ins kleinste Detail zu überprüfen. Doch bei der weiter steigenden Komplexität wird dies zu einer Herausforderung.

- Die Wartbarkeit der unterschiedlichen Codebasen erweist sich als schwer. Es ist nicht nur die Herausforderung eine gewünschte Anpassung mehrfach zu tätigen, sondern auch die Tatsache, dass man bei eventuell auftretenden Bugs in einer Anwendung für Plattform A nicht weiß, ob der Bug ggf. auch in der Anwendung für Plattform B auftritt. Dieser Umstand muss zunächst einmal überprüft werden, was Zeit in Anspruch nimmt und sich somit auch negativ auf das Budget und andere Faktoren auswirkt.

- Es entstehen nicht nur höhere Kosten bei der Entwicklung und der Wartung der Derivate der mobilen Anwendung, sondern auch beim Testen einer solchen. Jede mobile Anwendung wird durch eigens angestellte Tester geprüft. Außerdem führt *TK* für jede mobile Anwendung Friendly User Tests durch. Es ergibt sich die Frage, wie ein solcher Test für mehrere Derivate durchgeführt, koordiniert und dargestellt wird. Ein Friendly User Test, bei dem die Derivate nicht berücksichtigt werden bzw. bei dem das Ergebnis über alle Derivate konsolidiert wird, macht zum Beispiel wenig Sinn, wenn man ein aussagekräftiges Ergebnis erhalten möchte. Bei einem negativen Ergebnis wäre es unmöglich zu erkennen,

an welcher Stelle man anzusetzen hätte. Aus diesem Grund müssen komplexere Friendly User Tests geplant und durchgeführt werden.

- Um vernünftige bzw. nachvollziehbare Entscheidungen während der Entwicklung treffen oder die Arbeit externer Firmen bewerten zu können, ist ein Know-How seitens *TK* für alle Zielplattformen notwendig.

Darüber hinaus nutzt *TK* unternehmensinterne Apps, welche zukünftig erweitert werden sollen, um bestimmte Abläufe vereinfachen und beschleunigen zu können. Heute werden beispielsweise Techniker für eine Provisionierung, eine Entstörung o.Ä. beim Kunden Vorort mit einem Notebook ausgestattet, um die nötigen Prozesse im Backend auslösen zu können. So muss bei bestimmten Prozessen z.B. die Media-Access-Control-Adresse (MAC-Adresse) des Kundengerätes abgetippt und eingetragen werden, was zum einen fehleranfällig und zum anderen zeitaufwendig ist. Mit einer mobilen Anwendung könnte man solche Probleme lösen, in dem die MAC-Adresse von einem Barcode-Scanner gelesen wird. Solche und andere Anwendungsfälle sollen zukünftig in einer mobilen Anwendung abgebildet sein. Dies stellt insofern eine Herausforderung dar, als dass diese Anwendungsfälle reibungslos und sicher funktionieren müssen. Ebenso wie die für Kunden gedachten mobilen Anwendungen, werden die unternehmensinternen Apps auf das Backend zugreifen. Aus diesem Grund besteht ein noch größerer Wunsch nach einheitlichen Schnittstellen auf das Backend, welche ebenfalls für die öffentlichen Apps genutzt werden sollen.

5.2 Anforderungen

Um all die geschilderten Herausforderungen und Probleme des Telekommunikationsunternehmens *TK* lösen und weiterhin nativ entwickeln zu können, soll eine Strategie für die native mobile Entwicklung gefunden werden, auf die man langfristig setzen möchte. Es sollen insbesondere die folgenden Ziele erreicht werden:

- Es soll ein standardisierter Zugriff auf das Backend von *TK* erfolgen, wobei externe Firmen keinen direkten Zugang mehr auf das Backend von *TK* erlangen sollen.

- Hilfsfunktionen, die von unterschiedlichen mobilen Anwendungen genutzt werden, um auf das Backend von *TK* zugreifen zu können, sollen nur einmalig entwickelt werden und für andere Entwicklungsprojekte nutzbar bzw. wiederverwendbar sein.

- Es sollen Komponenten bzw. Libraries entwickelt werden, die von externen Firmen zu nutzen sind.

- Durch die vielen unterschiedlichen Codebasen steigt die Komplexität für das Entwickeln, Testen und Warten. Wenn möglich, soll es nur eine Codebasis pro Anwendung für alle Plattformen geben.

- Die Entwicklung einer nativen mobilen Anwendung soll kosteneffizienter werden. Momentan gibt es durch die vielen Derivate einer Anwendung eher den entgegengesetzten Trend, der sich durch die Hinzunahme von Windows Phone 8 zu verschärfen droht.

- Die Entwicklungs- und Testzeiten sollen verkürzt werden und besser planbar sein. Die unterschiedlichen Derivate einer App führen zu einer unübersichtlichen Planung, welches einen negativen Einfluss auf die Produkteinführungszeit nimmt.

- Es soll für eine mobile Anwendung einen einheitlichen Release-Termin für alle Plattformen geben, wobei zum Beispiel das Derivat der Plattform A auf Grund möglicher Probleme nicht den Release-Termin von einem Derivat von Plattform B gravierend nach hinten verschieben darf. Als gravierend werden vier oder mehr Wochen angesehen.

- Um die Entwicklung externer Firmen zu standardisieren, soll eine mobile Plattform bzw. ein einheitliches Framework gefunden werden. Auch unternehmensinterne Apps sollen auf einer solchen Plattform entwickelt werden.

- Es sollen Guidelines für externe Entwicklungsfirmen vorgegeben werden, um eine einheitliche Entwicklung zu fördern und somit die Qualität insgesamt steigern zu können.

Aus diesen Zielen lassen sich direkt weitere Anforderungen an die Architektur einer mobilen Anwendung und an ein Framework für eine Cross-Plattform-Entwicklung ableiten.

<u>Anforderungen an ein Framework:</u>

- Um eine höhere Standardisierung zu erreichen, sollen mit dem Framework sowohl die Mitarbeiter von *TK* als auch die externen Entwicklungsfirmen arbeiten. Es ist daher davon auszugehen, dass viele unterschiedliche Personen mit dem Framework in Kontakt treten. Das Framework und dessen Entwicklungsumgebung müssen daher schnell und einfach erlernbar sein. Wünschenswert wäre ein Framework, welches in verschiedenen Entwicklungsumgebungen integrierbar ist, so dass es Auswahlmöglichkeiten gibt.

- Da das Framework für eine langfristige Periode genutzt werden soll, sind nicht alle benötigten Funktionen bereits im Vorhinein erkennbar. Es wäre inakzeptabel, wenn während eines zukünftigen Entwicklungsprojektes festgestellt werden würde, dass eine ganz spezifische Funktion vom Framework nicht unterstützt wird, woran das ganze Projekt scheitern würde. Aus diesem Grund muss die Erweiterbarkeit des Frameworks gewährleistet sein.

- Das Framework muss mindestens Android, iOS und Windows Phone 8 unterstützen. Da nicht abzusehen ist, welche Plattformen noch dazu kommen könnten, wären weitere Plattformen wünschenswert.

- Der Hersteller des Frameworks muss einen sehr guten Support gewährleisten. Service-Level-Agreements (SLAs) sollen möglichst die Qualität sicherstellen, so dass es z.B. fest definierte Reaktionszeiten auf Tickets gibt.

- Die Lizensierung muss dementsprechend gestaltet sein, dass das Framework zwar von *TK* lizensiert wird, aber ebenso externe Firmen im Auftrag von *TK* damit arbeiten dürfen, ohne extra Lizenzen kaufen zu müssen. Alternativ soll die Laufzeit einer Lizensierung möglichst variabel gestaltbar und je nach Bedarf bestell- und benutzbar sein.

- Das Framework muss nicht nur die allgemeinen, sondern auch die Anforderungen an die Architektur einer mobilen Anwendung erfüllen.

- Innerhalb *TK* werden die in der Benutzeroberfläche sichtbaren Texte (z.B. Labels) in einer App vom Produktmanagement vorgegeben. Um diese Vorgaben schnell umsetzen zu können, wäre es wünschenswert, sämtliche Strings so exportieren zu können, dass sie direkt vom Produktmanagement, welches kein technisches Know-How besitzt, angepasst werden können. Die geänderte exportierte Datei muss anschließend wieder importierbar sein.

<u>Anforderungen an die Architektur einer mobilen Anwendung:</u>

- Um eine hohe Wiederverwendbarkeit zu erreichen, sollen Libraries entwickelt werden, die je nach Bedarf von unterschiedlichen Entwicklungsprojekten integrierbar sein müssen.

- Ebenso im Sinne der Wiederverwendbarkeit und einer besseren Wartbarkeit sollen Business-Logik und Benutzeroberfläche voneinander getrennt werden, beispielsweise durch eine Model-View-Controller (MVC) Architektur.

- Performanz ist ein wichtiger Faktor. Die Umstellung von einer hybriden Entwicklung auf eine native Entwicklung war hauptsächlich auf Gründe der Performanz zurückzuführen. Die Architektur darf die Performanz nicht negativ beeinflussen.

5.3 Methodologie

Die im Folgenden beschriebene Methodologie soll dabei helfen, eine geeignete halbautomatisierte oder vollständige Lösung (siehe Punkte 4.2.8 *„Halbautomatisierte und vollständige Lösungen“*) für die Entwicklung einer nativen plattformübergreifenden Entwicklung zu finden, welche die beschriebenen Anforderungen von *TK* erfüllen muss. Da jedoch auch andere Leser dieser Arbeit von den hier vorgestellten Lösungen profitieren sollen, wird eine Methodik gewählt, die auf Scores basiert. Je nach Schwerpunkt können die Scores anders vergeben oder gewichtet werden, so dass man mit anderen Anforderungen ggf. eine andere Wahl treffen kann.

5.3.1 Entscheidungsfaktoren

Man könnte prinzipiell eine beliebige Menge an Faktoren wählen, anhand derer eine Evaluierung stattfinden soll. Um eine etwas allgemeinere Aussage treffen zu können, lehnt sich die Wahl der Kriterien an das FURPS Modell an, welches von Bernd Bruegge und Allen H. Dutoit beschrieben wurde [Bru04]. Das FURPS Modell wiederum ähnelt sehr der Norm ISO/IEC 9126, welche Qualitätskriterien für die Bewertung von Softwareprodukten beschreibt [Nai08]. Die fünf Buchstaben von FURPS stehen dabei für fünf Qualitätsmerkmale:

- Functionality (Funktionalität)
- Usability (Benutzbarkeit)
- Reliability (Zuverlässigkeit)
- Performance (Effizienz)
- Supportability (Wartbarkeit / Änderbarkeit)

Diese Qualitätsmerkmale sind für die Bewertung von Softwareprodukten im Allgemeinen, nicht jedoch für Frameworks im Speziellen, gedacht. Frameworks stellen in dem Sinne einen Sonderfall dar, als dass man sie aus zwei Blickwinkeln betrachten kann. Zum einen ist es das Framework als eigenständige Software und zum anderen die mobile Anwendung, die damit geschrieben wird. Daher werden die Kategorien wie folgt leicht abgeändert und im Einzelnen kurz erklärt.

Funktionalität

Diese Kategorie soll aus Entwicklersicht die Möglichkeiten des Frameworks beleuchten, wozu nicht nur die unterstützten Plattformen gehören. Interessant an dieser Stelle ist beispielsweise ebenso die Abstraktionsebene des Frameworks und wie gut bzw. ob die Schnittstellen zu den nativen Funktionen der Zielplattformen funktionieren. D.h. es soll überprüft werden, ob man z.B. einen Zugriff auf die Hardwaresensoren erhält und ob das Framework einen Entwickler im Vergleich zu einem nativen Software Development Kit (SDK) der jeweiligen Plattform einschränkt. Auf der anderen Seite bietet das Framework unter Umständen auch Möglichkeiten, die einfacher als in einem SDK zu realisieren sind. Auch dies soll in die Bewertung mit einfließen.

Eine spezielle Anwendersicht ist in dieser Kategorie nicht nötig, da ein Benutzer ohnehin lediglich das Nutzen kann, was der Entwickler anhand der ihm zu verfügenden Möglichkeiten, die ein Framework bietet, implementiert.

Benutzbarkeit

Die Benutzbarkeit betrachtet aus Entwicklersicht die Entwicklungsumgebungen und bewertet beispielsweise, wie sich der Build-Prozess für den Entwickler gestaltet. Wie im Kapitel 4.2 *(„Möglichkeiten der mobilen Cross-Plattform-Entwicklung*“) erläutert, kann ein Entwicklungsablauf zeitaufwendig sein, da man für unterschiedliche Zielplattformen unter Umständen Code in andere Entwicklungsumgebungen importieren muss. Ein Punkt ist dabei z.B. auch, wie der Build-Prozess von iOS Anwendungen abläuft, da für diese prinzipiell immer MacOS benötigt wird. Die Benutzbarkeit einer Cross-Plattform-Entwicklung würde aber darunter leiden, wenn ein Entwickler zwingend MacOS nutzen müsste, obwohl er ein anderes System für die Entwicklung präferiert. Da das Framework auch von externen Firmen und somit von einer Vielzahl an Personen benutzt werden wird, ist es ein sehr wahrscheinliches Szenario, dass nicht immer ein System mit MacOS vorhanden ist.

Aus Anwendersicht wird erwartet, dass sich die App wie eine im Software Development Kit entwickelte native Anwendung anfühlt, d.h. die Benutzeroberfläche soll nicht nur wie eine im SDK entwickelte aussehen, sondern auch genauso flüssig funktionieren. An dieser Stelle haben insbesondere hybride Entwicklungsmethoden (siehe Punkt 4.1.3) ihre Probleme, welche in einer nativen Cross-Plattform-Entwicklung nicht vorkommen dürfen.

Zuverlässigkeit und Effizienz

Zuverlässigkeit und Effizienz werden in dieser Arbeit unter einem Punkt zusammengefasst.
Aus Entwicklersicht werden die Zuverlässigkeit und die Stabilität des Frameworks in die Bewertung einfließen. Beispielsweise werden im vorherigen Qualitätsmerkmal („*Benutzbarkeit*“) zwar

u.a. die Möglichkeiten des Debuggings mit einbezogen, aber nicht wie zuverlässig und stabil es läuft.

Aus Benutzersicht wird geprüft, ob und wie stabil die implementierten Funktionen der Applikation in der Praxis funktionieren. Außerdem wird die Effizienz anhand von Ladezeiten überprüft. Erwartet wird z.B., dass sich eine ListView mit mehreren 10000 Elementen befüllen lässt, ohne dass dies zu Hängern oder zu anderen spürbaren negativen Einflüssen in der Benutzeroberfläche führt.

Wartbarkeit und Support

Diese Kategorie bezieht sich ausschließlich auf das Framework und nicht auf die damit erstellte mobile Anwendung.
Es wird untersucht, wie gut Dokumentation und Support des Frameworks sind. Bzgl. des Supports sollen zum einen die Lizenzmodelle und deren Preise und zum anderen freie Quellen wie Foren und Communities betrachtet werden. Es wird zudem darauf geachtet, ob ein kostenpflichtiger Support ggf. Service-Level-Agreements in den Lizenzen beinhaltet.
Weiterhin wird überprüft, inwiefern eine Erweiterung des Frameworks möglich ist. Da das Framework für einen längeren Zeitraum genutzt werden soll, ist noch nicht absehbar, welche Features zukünftig benötigt werden könnten. Eine Erweiterung eventuell fehlender Features ist daher von großem Interesse.
Da innerhalb *TK* die in der Benutzeroberfläche sichtbaren Texte (z.B. Labels) in einer App vom Produktmanagement vorgegeben sind, soll zudem bewertet werden, ob das Framework einen Export / Import von den in der mobilen Anwendung vorhandenen Strings unterstützt.

Kompromissbereitschaft

Die Kompromissbereitschaft ist ein Faktor, der eigens für diese Arbeit genutzt wird. Er soll den Zusammenhang zwischen einem hohen Portierbarkeitsgrad und den damit verbundenen funktionalen Einbußen bewerten. Eine hohe Bewertung dieses Faktors bedeutet, dass mit wenigen Kompromissen zu rechnen ist, während man sich bei einer geringen Bewertung auf erhebliche Einschränkungen gefasst machen sollte.
Kompromissbereitschaft ist kein Bestandteil des FURPS Modells, da man typischerweise bei einer Software keine Kompromisse eingehen will, wenn man diese evaluiert bzw. die Qualität bewertet.
Eine Verwendung des Faktors lässt sich mit den verschiedenen Ansätzen und Möglichkeiten der halbautomatisierten und vollständigen Lösungen (siehe Kapitel 4.2.8) begründen. Vollständige Lösungen verfolgen eine "*Write once, run anywhere*"-Philosophie und setzen somit auf einen sehr hohen Portierbarkeitsgrad. Bei einer halbautomatisierten Lösung müssen bestimmte Teile einer Anwendung wie z.B. die Benutzeroberfläche pro Plattform entwickelt werden, was dem eigentli-

chen Ziel einer Cross-Plattform-Entwicklung, eine hohe Wiederverwendbarkeit und Portierbarkeit, entgegenwirkt. Dafür muss ein Entwickler bei halbautomatisierten Lösungen womöglich weniger Kompromisse als bei vollständigen Lösungen eingehen. Die Kompromissbereitschafft dürfte sich von Entwickler zu Entwickler unterscheiden. Daher ist eine entsprechende Gewichtung bei der Wahl des Frameworks wichtig, weshalb der Faktor bei der Evaluierung berücksichtigt werden soll.

5.3.2 Subjektive Faktoren

Es werden in dieser Arbeit Faktoren erwähnt, die nicht in die Gewichtung einfließen. Warum und welche Kriterien bewusst nicht gewichtet werden, soll kurz beschrieben werden.

Programmierkenntnisse eines Entwicklers

Das Ziel dieser Arbeit ist es, eine Lösung für die Probleme des Unternehmens *TK* zu finden. Wie im Punkt 5.1 („*Herausforderungen*“) und im Punkt 5.2 („*Anforderungen“*) erwähnt, arbeiten eine Vielzahl von Entwicklern an den mobilen Anwendungen. Darunter sind insbesondere externe Entwickler, die von Projekt zu Projekt andere sein können. Die im Framework verwendete Programmiersprache spielt daher insofern keine Rolle, als das man ohnehin nicht hervorsehen kann, welche Programmiersprache externe Entwickler bevorzugen. Des Weiteren könnte man für zukünftige Projekte externe Programmierer engagieren, die genau auf die im Framework verwendete Sprache spezialisiert sind.

Erfahrungen eines Entwicklers im Bereich der mobilen Entwicklung

Es sollen keine Vorkenntnisse der Entwickler berücksichtigt werden, d.h. es ist irrelevant ob ein Entwickler zum Beispiel Android- oder iOS-Erfahrungen besitzt. Der Grund liegt darin, dass auf Cross-Plattformen spezialisierte Frameworks so stark abstrahiert sein können, dass der Entwicklungsprozess weder mit der einen, noch mit der anderen Plattform vergleichbar ist. Bei einer Bewertung dieses Kriteriums würde man sich der Möglichkeiten berauben, die durch eine starke Abstraktion entstehen. Eine Möglichkeit wäre beispielsweise die Beauftragung externer Firmen, die bisher wenig oder im Extremfall gar keine Erfahrungen in der Entwicklung mobiler Anwendungen gesammelt haben. D.h. konkret: Wenn das Framework z.B. auf Java basiert, könnte man auch Firmen für eine Ausschreibung akzeptieren, die zwar kein spezielles Know-How im Bereich der mobilen Entwicklung, dafür jedoch „Java-Ninjas“ in ihren Reihen haben. Die Chance liegt darin, dass für einen neuen Entwicklungsauftrag mehr externe Entwicklungsfirmen in Frage kämen und so möglicherweise der Entwicklungspreis gedrückt werden könnte.

5.3.3 Gewichtete Evaluierung

Die vorgestellten Entscheidungsfaktoren sollen gemäß den Ergebnissen mit Scores zwischen 1 und 5 bewertet werden. Die Scores haben dabei folgende Bedeutung:

Score	Bedeutung
1	Die Erwartungen wurden nicht erfüllt.
2	Die Erwartungen wurden schlecht erfüllt.
3	Die Erwartungen wurden größtenteils erfüllt.
4	Die Erwartungen wurden erfüllt.
5	Die Erwartungen wurden übertroffen.

Zusätzlich sollen die Entscheidungsfaktoren mit Werten zwischen 1 und 5 gewichtet werden:

Entscheidungskriterium	Gewichtung
Funktionalität	4
Benutzbarkeit	4
Zuverlässigkeit und Effizienz	4
Wartbarkeit und Support	5
Kompromissbereitschaft	3

Die Funktionalität, Zuverlässigkeit und Effizienz sind wichtige Kriterien, werden allerdings nicht ganz so hoch wie die Wartbarkeit und der Support eingestuft, da eventuell funktionale Mängel oder Stabilitätsprobleme durch einen sehr guten Support ausgeglichen werden können. Des Weiteren soll das Framework über eine lange Zeit genutzt werden, wodurch der Support und die Wartbarkeit eine exponierte Stellung einnehmen. Die Benutzerbarkeit wurde mit vier statt fünf Punkten gewichtet, da zwar aus Anwendersicht eine hohe Usability erwartet wird, aber aus Entwicklersicht kleinere Mängel im Build-Prozess und dergleichen nicht dem Endprodukt, d.h. der mobilen Anwendung, schaden. Weiterhin sind durch einen guten Support des Frameworks solche Prozesse verbesserbar. Die Kompromissbereitschaft wurde seitens *TK* mit einem Faktor von drei gewichtet. Dies lässt sich so interpretieren, dass eine höhere Portierbarkeit auf Kosten einiger zu erwartender Kompromisse zu akzeptieren ist.

Diese Gewichtung spiegelt die Herausforderungen und Anforderungen des Unternehmens *TK* wider. Leser, die andere Anforderungen an eine mobile Cross-Plattform-Entwicklung stellen, können und sollten daher eine eigene Gewichtung vornehmen.

6 Evaluierung von Frameworks

In diesem Kapitel soll eine Lösung gefunden werden, welche den aktuellen und zukünftigen Herausforderungen und Anforderungen (siehe Kapitel 5.1 und 5.2) gerecht wird. Dabei wurden im Vorhinein sämtliche für eine plattformübergreifende Entwicklung in Frage kommenden Frameworks betrachtet.

6.1 In Frage kommende Frameworks

Im bisherigen Verlauf dieser Arbeit wurde erläutert, dass folgende Möglichkeiten für eine native Cross-Plattform-Entwicklung bestehen:

1) Unterstützende Lösungen (*J2ObC*, *XMLVM*, …)
2) Implementierung eines eigenen Frameworks auf Basis der unterstützenden Lösungen
3) Halbautomatisierte Lösungen (*Code Separierung*)
4) Vollständige Lösungen (*Write once, run anywhere*)

Dabei wurden im Kapitel 4.2.8 („*Halbautomatisierte und vollständige Lösungen*“) die Vorteile der letzten zwei gegenüber den ersten zwei Möglichkeiten erläutert. Daher kamen für eine Evaluierung nur halbautomatisierte und vollständige Lösungen in Frage, die mindestens die drei Plattformen Android, iOS und Windows Phone 8 abdeckten.

Da eine native Entwicklung das Ziel ist, wurden sämtliche Frameworks, die diesem Grundsatz widersprechen, nicht in Betracht gezogen. Ausgeschlossen wurden aus diesem Grund insbesondere Web-Frameworks wie *Sencha Touch, jQuery Mobile, Intel's App Framework* oder *PhoneGap*. Bei anderen Frameworks ist ein solcher Bezug zu Web-Techniken allerdings nicht auf den ersten Blick ersichtlich. Zu diesen Frameworks gehört beispielsweise das *Oracle Application Development Framework* (*Oracle ADF*), welches für die Benutzeroberfläche *PhoneGap* und *HTML5* einsetzt [Ora14].

Ein weiteres ausgeschlossenes Framework ist *Appcelerator Titanium*, in welchem man in JavaScript entwickelt, der Code jedoch zu nativem Code transformiert wird. Das Produkt wurde nicht in Betracht gezogen, weil die Firma *Appcelerator* an einem Nachfolger namens *Ti.Next* arbeitet, welches auf das in Kapitel 4.2.5 vorgestellte *Hyperloop* aufsetzt [Hay14]. Der Ausschluss erfolgte, da ein Framework für einen langen Einsatzzeitraum gesucht wird und nicht auf ein Produkt gesetzt werden soll, welches schon zeitnah ersetzt wird. Des Weiteren wurden im Kapitel 4.2.5 einige Nachteile *Hyperloops* erwähnt, die einen produktiven und effizienten Einsatz zum jetzigen Zeitpunkt nicht ermöglichen, so dass auch der Nachfolger *Ti.Next* ausgeschlossen werden kann.

Abgelehnt wurden ebenso native Cross-Plattform-Frameworks, die auf eine Spieleentwicklung optimiert sind und speziell dafür beworben werden. *Marmalade* ist beispielsweise ein Framework, bei dem ein Entwickler in C / C++ entwickelt und welches die geforderten Plattformen unterstützt. Der Hersteller selbst bezeichnet und bewirbt es als „Game Development SDK" [Mar14], was darin begründet ist, dass selbst grundlegendste UI-Elemente wie z.B. eine List View vom Framework nicht unterstützt werden und daher von einem Entwickler selbst implementiert werden müssten. Für die effiziente Entwicklung einer mobilen Business-Anwendung ist dieser Umstand inakzeptabel. Ähnliches trifft auf das *Corona SDK* zu, welches optimiert für eine Spieleentwicklung ist, wobei dieses zumindest ein paar native UI-Elemente unterstützt [Cor14]. Darüber hinaus ist es beim *Corona SDK* nicht ersichtlich, wie die Cross-Plattform-Abdeckung realisiert wird. Die im SDK verwendete Programmiersprache ist LUA, jedoch ist unklar, ob der geschriebene Code interpretiert oder nativer Code generiert wird.

Nach diesem Ausschlussprinzip blieben die Frameworks *Xamarin* und *Codename One* über. Während *Xamarin* eine halbautomatisierte Lösung darstellt, verfolgt *Codename One* eine *Write once, run anywhere* Philosophie.

6.2 Xamarin

Xamarin ist ein Framework der Firma *Xamarin Inc.*, welches seit Mai 2011 existiert und auf das in Kapitel 4.2.6 vorgestellte *Mono* aufsetzt. Monos Ziel ist es, als ein Open-Source Projekt Microsofts .NET Framework für Linux zu implementieren. Da Android und iOS ebenfalls auf einer Version von Linux basieren, allerdings andere Architekturen als ein „gewöhnliches" Linux wie z.B. Debian besitzen, gab es jeweils für Android und iOS Projekte, die dafür sorgen sollten, dass die Mono Runtime auch auf diesen mobilen Betriebssystemen lauffähig wird.

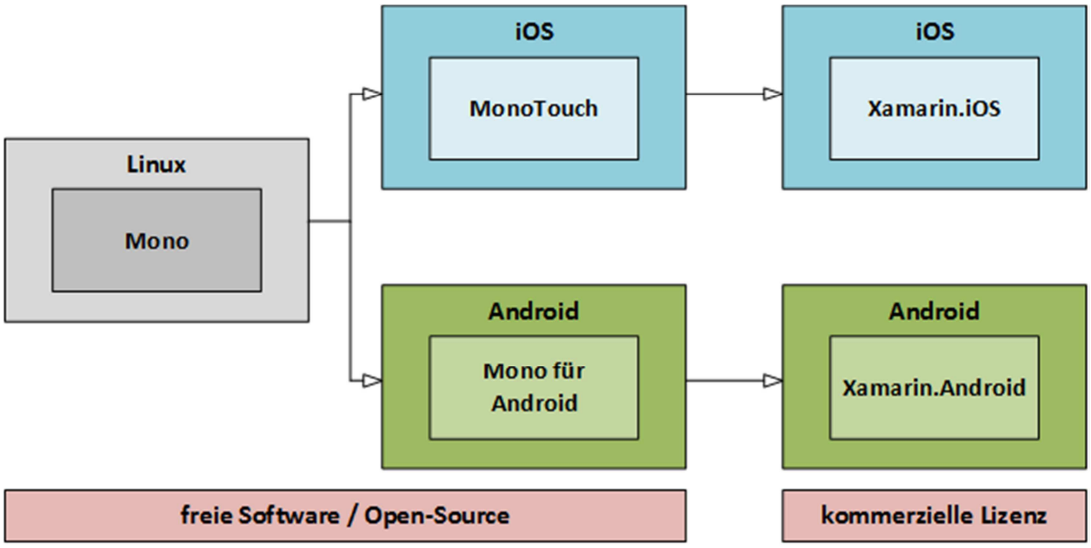


Abbildung 12: Entstehung von Xamarin

Dadurch entstand *Monotouch*, welches die Mono Implementierung für iOS darstellt, und *Mono für Android*. Mono war bis 2011 unter der Führung der Firma Novell, ehe Novell von Attachmate im April 2011 übernommen wurde. Der neue Eigentümer hatte jedoch kein Interesse an *Mono-Toch* und *Mono für Android*, so dass die betroffenen Entwickler einen Monat später entlassen wurden und daraufhin die Firma *Xamarin Inc.* gründeten [Hei141]. *Xamarin Inc.* entwickelt bis zum heutigen Tage die Mono Laufzeitumgebungen weiter, wodurch *Xamarin.iOS* aus *MonoTouch* und *Xamarin.Android* aus *Mono für Android* entstanden. Der wesentliche Unterschied zwischen den ursprünglichen Mono Laufzeitumgebungen und *Xamarin* besteht darin, dass Letzteres unter einer kommerziellen Lizenz steht, während der Mono Quellcode frei verfügbar ist (vgl. Abb. 12).

6.2.1 Voraussetzungen und Möglichkeiten

Wie für *Mono* gilt auch für *Xamarin*, dass Microsofts .NET Framework auf Basis der ECMA Standards für C# und der Common Language Runtime (CLR) implementiert ist. Daher ist es zwingend erforderlich, dass ein Entwickler die Programmiersprache C# beherrscht. C# ist die native Sprache für Windows Phone, weshalb auf diesem Betriebssystem der Code ohne besondere Anpassungen seitens *Xamarin* lauffähig ist. Durch *Xamarin.Android* und *Xamarin.iOS* werden noch zwei weitere mobile Betriebssysteme abgedeckt, so dass insgesamt die drei mobilen Systeme Android, iOS und Windows Phone unterstützt werden. Darüber hinaus ist der in *Xamarin* geschriebene Code auf Windows und MacOS nutzbar. Für Windows wird Microsofts .NET Framework und für MacOS *Xarmarin.Mac*, welches auf *MonoMac* basiert, als Laufzeitumgebung verwendet.

Xamarin ist eine halbautomatisierte Lösung, weshalb nur gewisse Teile wie die Business-Logik, die Serviceschicht oder das Datenmanagement plattformübergreifend implementiert werden können. Aus diesem Grund muss für Android ein Activity-basierter UI Layer und für iOS ein UIKit-basierter UI Layer implementiert werden. Des Weiteren ist pro Plattform die Implementierung eines so genannten Application Layers nötig (siehe Abb. 13), in dem alle plattformspezifischen Funktionen, wie z.B. Hardwarezugriffe, enthalten sind.

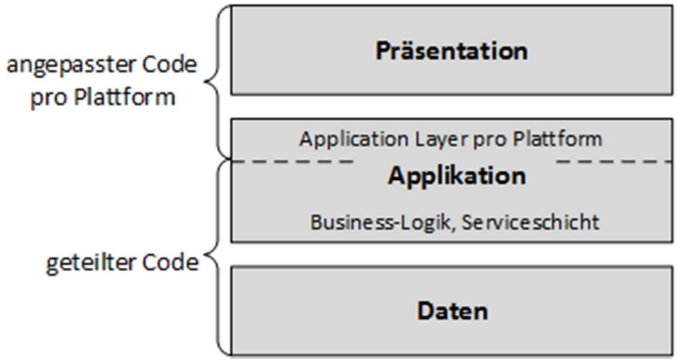


Abbildung 13: Code Separation in Xamarin

Ableitend daraus schreibt das Framework eine Model-View-Controller (MVC) Architektur vor, da eine Code Separierung aus den genannten Gründen für jede Applikation immer zwingend erforderlich ist.

Da *Xamarin* auf *Mono* aufsetzt, bleiben die im Kapitel 4.2.6 („*Mono*“) beschriebenen Einschränkungen grundsätzlich auch bei *Xamarin* bestehen.

Für Android wird eine Just-in-Time (JIT) Kompilierung durchgeführt, wobei *Android Managed Callable Wrappers (MCWs)* generiert werden.

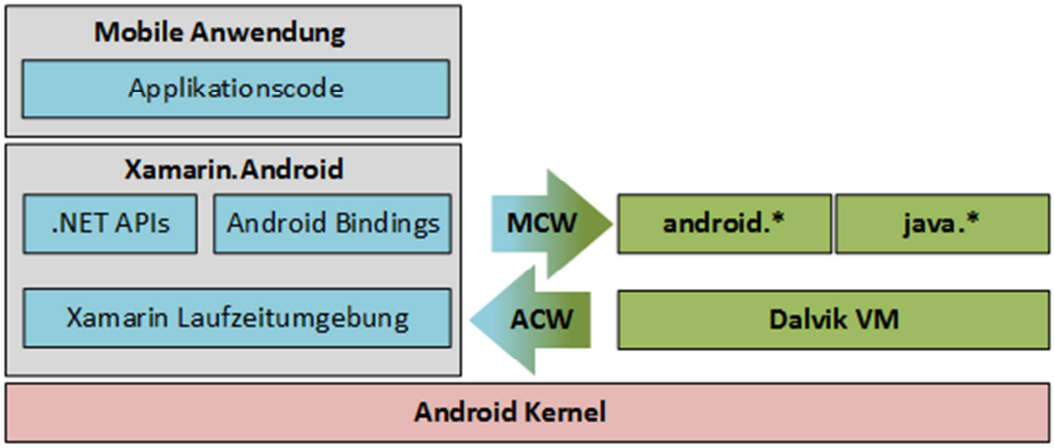


Abbildung 14: Xamarin.Android Aufbau und Kommunikation

Diese werden genutzt, sobald Xamarins Laufzeitumgebung, welche sich auf derselben Ebene wie die Dalvik VM befindet (vgl. Abb. 14), Zugriff auf nativen Android Code benötigt. Da die Dalvik VM zur Laufzeit keine Klassen via die JNI-Funktion *DefineClass()* registrieren kann, werden *Android Callable Wrappers (ACWs)* als native Java-Brücken genutzt. Die in Java implementierten ACWs leiten die Nachrichten der Dalvik VM weiter. Da ACWs zur Kompilierungszeit erstellt werden, kommt es insofern zu Einschränkungen, als dass es kaum eine Möglichkeit für generische Typen gibt, weswegen beispielsweise ein Aufruf der Methode *Android.Content.Intent.GetParcelableExtra* nur ein *Java.Lang.Object* zurückliefert, statt des eigentlichen Typs, da dieser zur Kompilierungszeit unbekannt ist [Xam141].

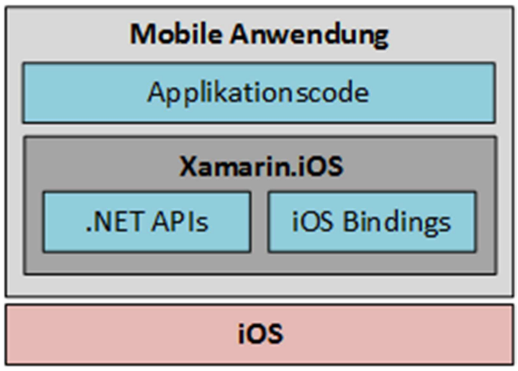


Abbildung 15: Xamarin.iOS Aufbau

Wie in der linksstehenden Abbildung illustriert, fehlt im iOS Aufbau die Laufzeitumgebung. Da Apple auf iOS Runtimes verbietet [App14], wird daher für dieses System eine Ahead-of-Time (AOT) - Kompilierung angewendet, weshalb die Anwendung direkt auf iOS ausführbar ist. Die statische Kompilierung führt jedoch, ähnlich wie auf Android, zu einigen Limitierungen, weshalb z.B. keine generischen Subklassen vom Typ *NSObject* unterstützt werden [Xam13].

Abgesehen von den genannten Einschränkungen gibt es jedoch keine weiteren Limitierungen, da man für jede Plattform mit C# auf die nativen Schnittstellen via Bindings zugreifen kann. In *Xamarin* werden neue APIs bzw. Bindings sofort integriert. So veröffentlichte Apple am 10.03.2014 iOS in der Version 7.1 und noch am selben Tag erschien ein Update von *Xamarin.iOS*, welches sämtliche APIs von iOS 7.1 implementiert hatte [deI14].

Da pro Plattform ein Application Layer implementiert wird, muss ein Entwickler plattformspezifische Kenntnisse mitbringen. Er muss beispielsweise wissen, was eine Android Activity ist und wie sich die Implementierung zu Apples UIKit differenziert. Jede einzelne Funktion, die auf Schnittstellen der Plattformen angewiesen ist, muss pro Plattform implementiert werden. Ein Entwickler sollte die benötigten Rechte je Plattform kennen und verwalten. Dieser Umstand wirkt dem eigentlichen Ziel einer Cross-Plattform-Entwicklung, ein hohe Wiederverwendbarkeit und Portierbarkeit, entgegen. Jedoch hat diese Methodik den Vorteil, dass bei einer Implementierung keine funktionalen Kompromisse eingegangen werden müssen und dennoch zumindest Business-Logik, Serviceschicht und Datenschicht plattformübergreifend in einer *Portable Class Librariy (PCL)* verwendet werden können. Um dennoch mehr portieren zu können, wurde *Xamarin.mobile* erschaffen, welches eine weitere Abstraktionsebene einfügt. Momentan werden in *Xamarin.mobile* einheitliche Schnittstellen für die Kontakte, die Kamera und Geolocation angeboten, die von iOS, Android und Windows Phone gleichermaßen aufgerufen werden.

Es gibt für *Xamarin* fertige Komponenten, welche man im Component Store erwerben kann und wodurch sich eine Entwicklung ggf. beschleunigen lässt. Nicht alle Komponenten sind von *Xamarin Inc.* entwickelt worden, da der Store auch für Fremdentwickler offen steht. Daher gibt es sowohl kostenlose als auch kostenpflichtige Komponenten. Es wird keine bestimmte Zielgruppe bevorzugt, da es auf der einen Seite z.B. Bibliotheken für die Erstellung von Charts gibt, die für jeden Entwickler interessant sein könnten. Auf der anderen Seite gibt es sehr spezifische Bibliotheken wie z.B. das SAP Mobile .NET SDK, welches einem Programmierer den Zugriff auf die SAP-Systeme von Unternehmen erleichtert.

6.2.2 Support und Dokumentation

Für *Xamarin* gibt es vier Lizenztypen: Starter, Indie, Business und Enterprise. Lediglich die Starterversion ist kostenlos und als eine Art Testversion gedacht, um einen kleinen Einblick in *Xamarin* zu erhalten. Es lassen sich zwar prinzipiell mobile Anwendungen entwickeln, die man in einen App Store stellen kann, allerdings ist die Entwicklung stark eingeschränkt, da man nur Anwendungen kompilieren kann, die eine Größe von 64k nicht überschreiten.

Diese Limitierung wird erst mit der Indie-Lizenz aufgehoben, welche pro Plattform und pro Entwickler jährlich $299 kostet. Mit der Plattform sind dabei die Zielplattformen, also Android und iOS gemeint. So wäre es möglich, ausschließlich *Xamarin.Android* oder nur *Xamarin.iOS* zu lizensieren. Da C# bzw. .NET ohnehin auf Windows Phone läuft, wird dieses nicht weiter bei der Lizenzierung erwähnt. Sowohl in der Starter- als auch in der Indie-Version ist eine Entwicklung in Xamarin Studio, jedoch nicht in Visual Studio möglich. Die beiden Entwicklungsumgebungen werden im Punkt 6.2.3 genauer beleuchtet. In genanntem Kapitel wird ebenso herausgearbeitet, warum eine effiziente Cross-Plattform-Entwicklung mit *Xamarin* nur in Visual Studio möglich ist.

Ein besonderer Support ist weder in der Starter-, noch in der Indie-Lizenz enthalten. Man kann auf das öffentlich zugängliche Developer Center zugreifen, in dem es neben der API Dokumentation ein paar wenige Tutorials in Schriftform und als Video gibt, die vor allem für Anfänger und Einsteiger gedacht und ansonsten als rudimentär zu bezeichnen sind. Weiterhin gibt es kleine Codeausschnitte zu oft verwendeten Funktionen wie z.B. Autovervollständigung in Textfeldern. Darüber hinaus gibt es ein öffentliches Forum, in dem ein Entwickler Fragen stellen kann. Stand März 2014 existieren in diesem Forum für Android und iOS ca. 10000 Threads mit etwa 30000 Postings, so dass man das Forum als gut frequentiert einstufen kann [Xam142]. Auf eine Antwort besteht allerdings kein Anspruch, da das Forum überwiegend von den Usern selbst getragen wird und eine Antwort eines *Xamarin* Entwicklers eher selten ist. Abgesehen vom Forum wird auf der *Xamarin*-Homepage zusätzlich auf StackOverflow verlinkt, wo Stand März 2014 9400 Fragen zu *Xamarin* gestellt wurden, wovon 2500 und damit über 25% der Fragen unbeantwortet sind [Sta14].

Für Unternehmen ist ein solcher Support inakzeptabel, da man bei einem Entwicklungsprojekt, welches oft einen straffen Zeitplan unterworfen ist, auf qualitativ hochwertige Antworten zu ggfs. auftretenden Problemen angewiesen ist. Zu diesem Zwecke gibt es die Business- und Enterprise-Lizenzen, die sich jedoch noch einmal deutlich voneinander differenzieren.

Die Business-Lizenz kostet $999 pro Jahr, Plattform und Entwickler und beinhaltet drei weitere Features gegenüber der Indie-Lizenz. Eines dieser Features ist der Email-Support, bei dem man seine Anliegen direkt den *Xamarin* Entwicklern schildern kann. Es gibt jedoch keine festgelegten Fristen, innerhalb derer eine Antwort erwartet werden kann. Ein weiteres Feature in der Business-Lizenz ist die Möglichkeit, *Xamarin* in Visual Studio zu integrieren. Das dritte Feature ist genau genommen ein Bündel aus mehreren Features, die speziell für Unternehmen interessant sein dürften. So ist für iOS beispielsweise ein Ad-Hoc Deployment für ein besseres Testen möglich. Auch ist in diesem Paket Headless-Build lizensiert, wodurch eine Kompilierung automatisiert durch Scripte funktioniert.

Die Enterprise-Lizenz beinhaltet die meisten Features und einen Support mit Service Level Agreements, wonach eine Antwortzeit innerhalb eines Arbeitstages garantiert wird. Darüber hinaus wird eine Kick-Off Session veranstaltet, bei dem man sich über eine Stunde lang direkt mit einem *Xamarin*-Entwickler unterhalten kann. Es gibt einen fest zugeteilten Technical Account Manager, wodurch man nicht nur mit beliebigen *Xamarin*-Mitarbeitern in Kontakt treten kann, sondern eine feste Bezugsperson als Anlaufstelle für Fragen hat. Weiterhin bekommen Enterprise-Kunden einen Zugang zu gepatchten *Xamarin*-Versionen, die Hotfixes zu verifizierten Bugs enthalten. Außerdem sind in der Lizenz fertige Komponenten im Wert von $500 bereits enthalten, die eine Entwicklung beschleunigen können. Nur in der Enterprise-Lizenz besteht die Wahl, Android-Projekte Ahead-of-Time zu kompilieren, ansonsten wird standardmäßig eine Just-in-Time Kompilierung durchgeführt. Die Art der Kompilierung hat einen Einfluss auf das Reverse Engineering (siehe Kapitel 6.2.6). Die Lizenz veranschlagt $1.899 pro Jahr, Plattform und Entwickler und ist auf Grund der deutlich besseren Supportleistungen gegenüber der Business-Lizenz für erfolgreiche Entwicklungsprojekte im Unternehmensbereich zu empfehlen. In der Business-Lizenz ist zwar wie erwähnt der Email-Support inbegriffen, allerdings ohne Service Level Agreements oder dergleichen. Des Weiteren würde man keine Hotfix-Releases für Bugs bekommen, die ggf. das ganze Entwicklungsprojekt verzögern oder unter Umständen sogar ganz blockieren.

Unabhängig von den Lizenzen gibt es die *Xamarin University*, welche $1.995 im Jahr veranschlagt. Der Mehrwert besteht darin, dass Online-Kurse live für viele Zeitzonen passend durchgeführt werden. Man kann in dem lizensierten Jahr so viele Kurse besuchen, wie man möchte, weshalb es möglich ist nur die Kurse auszuwählen, die einen interessieren bzw. einem weiterhelfen. Mitglieder der *Xamarin University* können sich in einem privaten Forum austauschen und durch Prüfungen Zertifikate wie *Xamarin Certified Mobile Developer* erlangen.

Zum Internetauftritt und zum öffentlichen Developer Center lässt sich generell sagen, dass diese teilweise für Verwirrung sorgen. Das Framework wird zwar u.a. mit Android, iOS und Windows Phone beworben, allerdings findet man für Letzteres so gut wie keine Informationen, da fast ausschließlich erwähnt wird, dass .NET ohnehin auf Windows Phone lauffähig und der Code portierbar sei. Diese Äußerungen sind zwar korrekt, doch stellen sie insofern ein Problem dar, als dass vorausgesetzt wird, wie eine Entwicklung für Windows Phone funktioniert. Daher entsteht zwar der Eindruck, *Xamarin* sei die Anlaufstelle für .NET Entwickler, die mit ihrem vorhandenen Know-How plattformübergreifend entwickeln wollen, doch macht es einen Einstieg für Entwickler mit einem anderen Background unnötig schwer, da diese sich Informationen wie API-Referenzen zu Windows Phone von der Microsoft Seite besorgen müssen.

Für weitere Verwirrung sorgen Features, die zwar im Developer Center erwähnt, offiziell aber noch nicht angeboten werden. So wird beispielsweise eine Test Cloud beschrieben, in welcher sich hunderte von unterschiedlichen mobilen Devices befinden sollen. Mit dieser Test Cloud soll ein automatisiertes Testen so ablaufen, dass vordefinierte UI-Operationen ausgeführt und eventuelle Bugs wie fehlerhafte Return-Codes oder Abstürze berichtet werden [Xam143]. In der Dokumentation ist jedoch nur das Feature als solches beschrieben, nicht wie man es nutzen kann oder in welcher Lizenz es enthalten ist. Erst durch eigene Recherche und auf *Xamarin*-fremden Seiten erfährt man, dass sich dieses Feature seit April 2013 im privaten Beta-Status befindet und ein Zugang einzig durch eine Einladung seitens *Xamarin Inc.* möglich ist [Lar14].

6.2.3 Entwicklungsumgebung und Build-Prozess

Wie bereits erwähnt, werden je nach Lizenz die zwei Entwicklungsumgebungen Xamarin Studio und Visual Studio angeboten, wobei es für Letzteres ein Plug-In gibt. Die zwei IDEs integrieren zahlreiche Features wie einen Code Editor mit Syntaxhervorhebung, Code Completion, Code Navigierung, Tooltips, Refactoring, einen Debugger und vieles mehr. Die Entscheidung, welche Entwicklungsumgebung benutzt werden soll, ist von zwei Parametern abhängig.

Auf der einen Seite muss das System berücksichtigt werden, auf dem entwickelt werden soll und auf der anderen Seite muss die Zielplattform betrachtet werden, für welche die mobile Anwendung geschrieben wird. So gibt es Xamarin Studio zwar für MacOS und Windows, jedoch ist in der Windows-Version nur eine Entwicklung für Android und nicht für iOS möglich. Die MacOS-Version von Xamarin Studio unterstützt sowohl Android als auch iOS.

Visual Studio hingegen ist grundsätzlich nur für Windows verfügbar, jedoch ist mit entsprechendem Plug-In sowohl eine Entwicklung für Android als auch für iOS möglich. Eine iOS Entwicklung in Visual Studio funktioniert allerdings nur in Kombination mit einem MacOS-System, da dass iOS SDK und Xcode für eine erfolgreiche Kompilierung Voraussetzung sind. D.h. für eine Cross-Plattform-Entwicklung, die auf Android und iOS abzielt, bleiben die folgenden zwei Möglichkeiten übrig:

- Eine Entwicklung unter MacOS mit Xamarin Studio. Auf dem Gerät muss für eine iOS-Entwicklung zudem Xcode mit dem iOS SDK installiert sein.
- Eine Entwicklung unter Windows mit Visual Studio, bei dem zusätzlich ein MacOS-System mit installiertem Xcode und iOS SDK benötigt wird, da darauf per Remote die iOS-Anwendung kompiliert und debugged wird.

Bei der bisherigen Betrachtung wurde Windows Phone als Zielplattform außen vor gelassen. Da allerdings auch dieses mobile Betriebssystem nach den Anforderungen dieser Arbeit abgedeckt werden soll, muss zusätzlich bedacht werden, dass das Windows Phone SDK nur für Windows verfügbar ist. Daher ist eine Entwicklung unter MacOS ausgeschlossen. Windows mit Visual Studio stellt die einzige Möglichkeit dar, eine effiziente plattformübergreifende Entwicklung, die alle drei genannten Systeme abdeckt, durchzuführen.

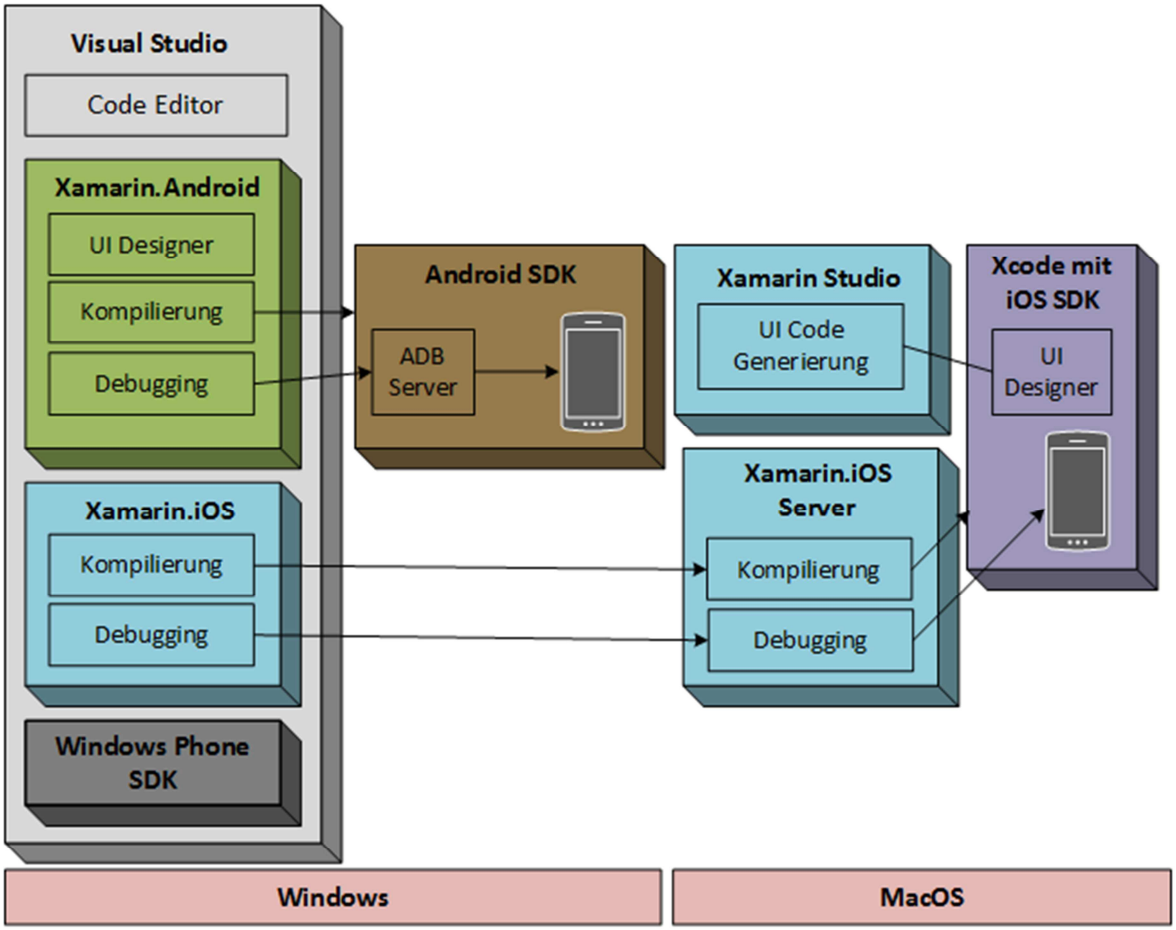


Abbildung 16: Xamarin Cross-Plattform-Entwicklungsumgebung

Abb. 16 zeigt eine vollständige Cross-Plattform-Entwicklungsumgebung, in der man mittels Xamarin für Android, iOS und Windows Phone entwickeln kann. Je nach verwendeter Visual Studio Version ist das Windows Phone SDK bereits enthalten. Innerhalb dieses SDKs befindet sich z.B. der Simulator für Tests, so dass die Vorbereitung für eine Windows Phone Entwicklung im Grunde genommen nur die Installation von Visual Studio bedeutet. Ist man nicht im Besitz einer Visual Studio Version mit integriertem Windows Phone SDK, lässt sich dieses separat von Microsoft downloaden und als Add-In hinzufügen. Anschließend kann man direkt mit der Entwicklung für Windows Phone beginnen, da alle Voraussetzungen erfüllt sind.

Die Entwicklungsumgebung für Android ist die der klassischen mit Eclipse und dem Android SDK sehr ähnlich. Nach dem Herunterladen des Xamarin Installers wird in diesem gefragt, welche Plug-Ins für Visual Studio installiert werden sollen, d.h. *Xamarin.Android* und *Xamarin.iOS* ließen sich auch getrennt voneinander in Visual Studio integrieren, sofern man nicht beide benötigen würde. Vor der Installation von *Xamarin.Android* werden die Abhängigkeiten geprüft, da dass Android SDK und das Java Development Kit (JDK) erforderlich sind. Kann eine Abhängigkeit nicht gefunden werden, wird das entsprechende Produkt automatisch heruntergeladen und installiert. Nach der Installation ist es direkt möglich in die Entwicklung einzusteigen. Die für eine Android-Anwendung benötigten Rechte können in den Projekteigenschaften in einem Menü selektiert werden, wozu der Entwickler jedoch Android-Know-How benötigt, da diese nicht weiter erläutert sind. Die eingangs erwähnte Nähe zur klassischen Android Entwicklungsumgebung ist darin begründet, dass das Android SDK statt in Eclipse hier in Visual Studio integriert ist. So wird bei einem Build-Prozess automatisch auf das Android SDK zugegriffen. Bei einem Debug-Prozess verbindet sich Visual Studio zur Android-Debug-Bridge (ADB) und man erhält – wie bei einer Entwicklung in Eclipse – eine Auswahl zwischen einem eventuell am Rechner angeschlossenem Gerät und dem Android Simulator. Standardmäßig wird für Android eine Just-in-Time Kompilierung durchgeführt. Als Enterprise-Kunde kann man auch eine Ahead-of-Time Kompilierung in den Projekteigenschaften einstellen, wobei die .NET Assemblies zu nativen Code kompiliert und als Library in das Projekt gepackt werden. Zur Gestaltung der Benutzeroberfläche enthält *Xamarin.Android* einen UI-Designer, mit dem sich per Drag & Drop UI-Elemente auf eine View ziehen lassen.

Einzig die Entwicklung für iOS ist unter Windows in Visual Studio ein wenig komplexer, da Apples Compiler und deren Zertifikate bzw. Code-Signierungstools nur auf MacOS-Systeme funktionieren. Um dennoch auf Windows entwickeln zu können, muss sich ein MacOS-Gerät im selben Netzwerk wie der Windows-Rechner befinden, auf dem entwickelt werden soll. Auf diesem MacOS-Gerät muss der Xamarin Server und Xcode mit dem iOS SDK installiert sein, wodurch das System als Build-Server fungiert. Hat man anschließend auf der Windows-Maschine das *Xamarin.iOS* Plug-In für Visual Studio installiert, lässt sich innerhalb der IDE nach Build-Servern im Netzwerk suchen. Nach Auswahl des Servers findet ein Pairing-Mechanismus statt, wodurch der Windows Rechner einen Zugriff auf das MacOS-Gerät erhält. Wenn man die mobile Anwendung in Visual Studio ausführen, kompilieren oder debuggen möchte, delegiert das Visual Studio Plug-In den gesamten Ablauf an den Xamarin Server auf dem MacOS-System. Dieses wiederum managed den Kompilierungs- oder Debug-Prozess mit dem iOS SDK. Bei einer Ausführung der App wird diese im Simulator auf dem MacOS-System oder auf ein daran angeschlossenes mobiles Device ausgeführt.

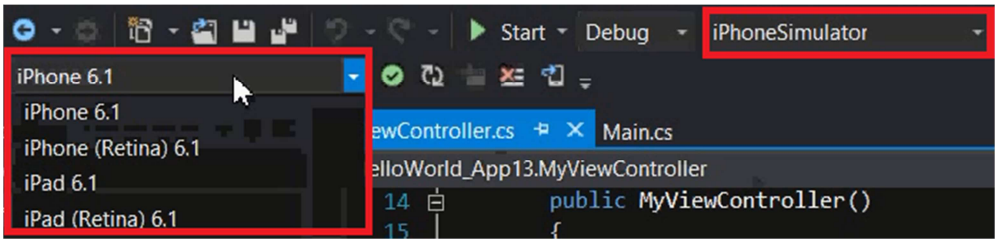


Abbildung 17: iOS Simulator Auswahl in Visual Studio

Vor der Ausführung lässt sich in Visual Studio zudem auswählen, welches Gerät im iOS SDK simuliert werden soll (siehe Abb. 17). In Visual Studio sind die Ausgaben des Build-Prozesses und sonstige Debug-Ausgaben ersichtlich. Für iOS wird immer eine Ahead-of-Time Kompilierung durchgeführt, da eine Just-in-Time Kompilierung seitens Apple untersagt ist [App14].
Um die Benutzeroberfläche für iOS einfacher in Visual Studio programmieren zu können, kann man diese in Xcode erstellen und mit der MacOS-Version von Xamarin Studio zu C# Code transformieren, welcher sich dann in Visual Studio einfügen lässt. Dieser Prozess ist in Abbildung 16 angedeutet.

Sobald eine wie hier beschriebene Entwicklungsumgebung aufgesetzt ist, kann man die anschließende Cross-Plattform-Entwicklung als effizient bezeichnen, da man den Code aller Plattformen innerhalb einer IDE verwalten, steuern und auf Knopfdruck die installierbare Binary einer Zielplattform erstellen kann.

6.2.4 Beispielanwendung

Um die im Kapitel 5 beschriebenen Herausforderungen und Anforderungen lösen zu können, muss die Entwicklung so gestaltet werden, dass möglichst viel Code plattformunabhängig entwickelt wird. Die interessante Frage ist daher, wie viel Code in *Xamarin* tatsächlich zwischen allen Plattformen geteilt wird und wie viel pro Plattform angepasst werden muss. Um diese Frage zu klären, wurden einige bereits existierende *Xamarin*-Projekte, bei welchen der Code frei verfügbar ist, analysiert.

Um die Projekte besser verstehen zu können, muss zunächst noch weiteres Grundwissen zur grundsätzlichen Architektur einer in *Xamarin* geschriebenen mobilen Anwendung vermittelt werden. Im Kapitel 6.2.1 (*„Voraussetzungen und Möglichkeiten“*) wurde bereits erwähnt, dass Anwendungen, die in *Xamarin* entwickelt werden, eine Model-View-Controller (MVC) Architektur umsetzen müssen. Eine klassische MVC-Architektur lässt sich wie folgt darstellen:

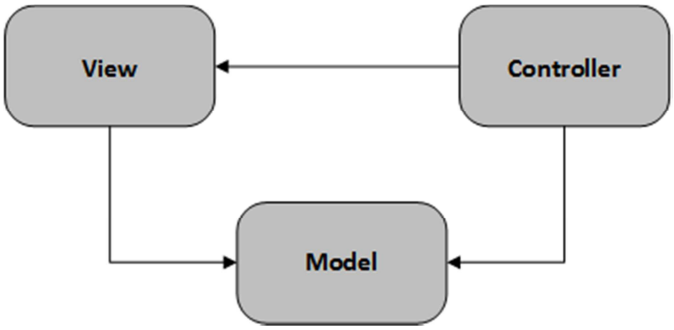


Abbildung 18: Klassische Model-View-Controller Architektur

Dieses Pattern wurde in *Xamarin* durch ein Interface zwischen View und Controller erweitert, um eine Trennung der plattformspezifischen Views zu erleichtern. Das erweiterte Pattern ist in Abb. 19 illustriert.

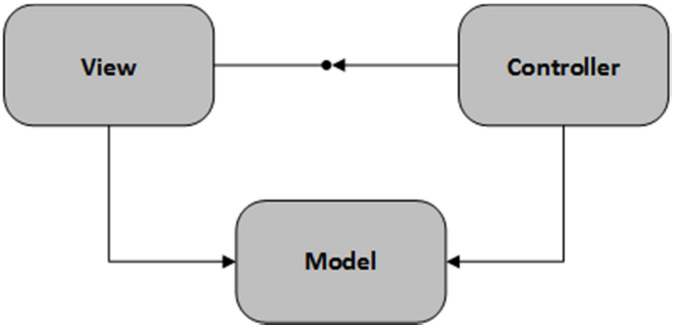


Abbildung 19: Model-View-Controller Architektur mit weiterem Interface

Das modifizierte Entwurfsmuster entkoppelt die plattformspezifischen Views vom gemeinsam genutzten Code und ermöglicht es Entwicklern, angepasste Ansichten pro Plattform zu erstellen, ohne Rücksicht darauf nehmen zu müssen, wie diese Ansichten an das Modell gebunden und in der Anwendung gerendert werden. So lange die Views das separate Interface implementieren, werden sie vom Controller bearbeitet und in geeigneter Weise wiedergegeben.

Als ein Praxisbeispiel in dieser Arbeit soll eine Open-Source App namens Tasky dienen, welche eine einfache ToDo-Liste implementiert und für Android, iOS und Windows Phone 8 verfügbar ist [Xam144] . Die Anwendung deckt die folgenden Features ab:

- Erstellen, Editieren, Speichern und Löschen von ToDo's
- Die Möglichkeit ToDo's als fertig zu markieren
- Eine Liste von ToDo's darstellen

Um die App plattformübergreifend entwickeln zu können, wird in Visual Studio eine Projektmappe mit vier Projekten angelegt, welche sich wie folgt aufteilen:

- Ein Projekt für die pattformübergreifenden Funktionalitäten (Tasky.Core)

- Je ein Projekt für die Zielplattformen (Tasky.Droid, Tasky.iOS, Tasky.Win8)

Das Tasky.Core-Projekt ist, wie Abbildung 20 illustriert, in Business Layer (BL), Data Access Layer (DAL) und Data Layer (DL) aufgeteilt. Dieses Projekt dient als Grundlage aller anderen drei Projekte, da sie später auf diesen Code zugreifen werden.

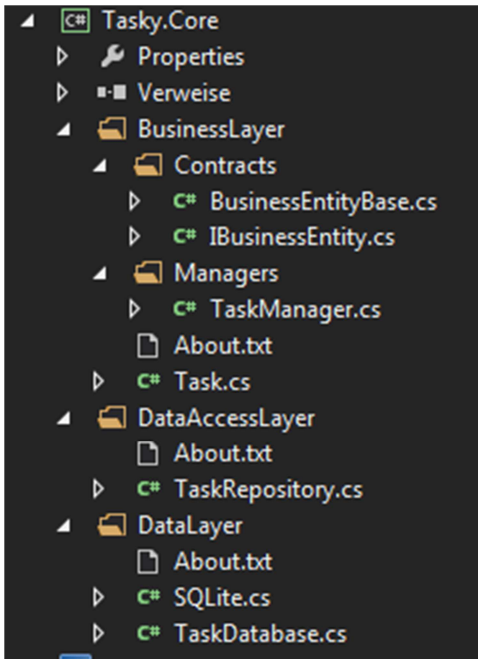


Abbildung 20: Tasky.Core-Projekt

Der Data Layer implementiert die physikalische Speicherung der Daten. In diesem Fall wird eine SQLite-Datenbank, die als .NET Bibliothek vorliegt, verwendet. Ein Zugriff erfolgt durch eine objektrelationale Abbildung und durch die Klasse *TaskDatabase*, welche ein entsprechendes Interface implementiert, um Daten in die SQLite-Datenbank schreiben und von ihr lesen zu können. Um sicherzustellen, dass für einen Datenbankzugriff stets dieselbe Instanz genutzt wird, setzt *TaskDatabase* das Singleton-Pattern um. Der Data Layer hat eine Abhängigkeit zum *IBusinessIdentity* Interface aus dem Business Layer, so dass dort abstrakte Datenzugriffsmethoden für das Speichern und Löschen von Daten implementiert werden können. Jede Business Layer-Klasse, die diese Schnittstelle implementiert, kann so im Data Layer persistent gespeichert werden. Im Data Access Layer instanziiert die Klasse *TaskRepository* den Data Layer, legt eine entsprechende Datenbank an und regelt den Zugriff auf einer solchen. Obwohl man sich hier im plattformübergreifenden Code befindet, muss bereits an dieser Stelle eine Fallunterscheidung für die entsprechenden Zielplattformen vorgenommen werden, da die Speicherung der Datenbank je nach System an unterschiedlichen Orten stattfindet:

```
public static string DatabaseFilePath {
  get {
          var sqliteFilename = "TaskDB.db3";
          #if SILVERLIGHT
            var path = sqliteFilename;
          #else
            #if __ANDROID__
              string libraryPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
            #else
              string documentsPath = Environment.GetFolderPath (Environment.SpecialFolder.Personal);
              string libraryPath = Path.Combine (documentsPath, "../Library/");
            #endif
            var path = Path.Combine (libraryPath, sqliteFilename);
          #endif
  return path;
}
```

Um zwischen den Plattformen unterscheiden zu können, wird der Präprozessor genutzt, der je nach Makro nur den Code kompiliert, der für die jeweilige Zielplattform gedacht ist. Windows Phone, welches mit dem Makro *SILVERLIGHT* zu identifizieren ist, erwartet einen Pfad, der nicht absolut ist. Android hingegen erwartet einen absoluten Pfad. In diesem Fall liefert *Environment.SpecialFolder.Personal* den Pfad */data/data/<PACKAGE_NAME>/files* zurück, wobei <*PACKAGE_NAME* > ein Platzhalter für den Paketnamen darstellt. Für iOS wiederum ist ein anderer absoluter Pfad nötig, weswegen es insgesamt für alle drei Zielplattformen Fallunterscheidungen gibt.

Im Business Layer implementiert die Klasse *Task* das Modell und die Klasse *TaskManager* ein Facade-Pattern. Dieses Entwurfsmuster wird verwendet, um die Get-, Save- und Delete-Methoden der bereits erläuterten *TaskRepository* Klasse zu kapseln, da diese von der Anwendung, d.h. den anderen drei Projekten, in der View referenziert werden. Das Facade-Pattern fördert eine für die Cross-Plattform-Entwicklung nötige lose Kopplung und delegiert die Aufrufe aus der eigentlich Anwendung, wodurch diese auf die Methoden der plattformunabhängigen Bibliothek zugreifen kann, ohne die Klassen, ihre Beziehungen, und Abhängigkeiten zu kennen.

Nachdem der plattformübergreifende Code fertiggestellt ist, müssen – wie im Punkt 6.2.1 („*Voraussetzungen und Möglichkeiten*“) bereits erwähnt - je Plattform der Application Layer und die Benutzeroberfläche erstellt werden. Zwar ähnelt sich dieser Prozess pro Plattform, jedoch ist die Entwicklung nahe am jeweiligen Software Development Kit (SDK), damit eine native Anwendung entsteht. Dadurch wird zum einen plattformspezifisches Wissen benötigt und zum anderen kann dabei kein Code geteilt werden, weshalb dieser Vorgang keinen Vorteil gegenüber einer nativen Entwicklung mit entsprechendem SDK bietet.

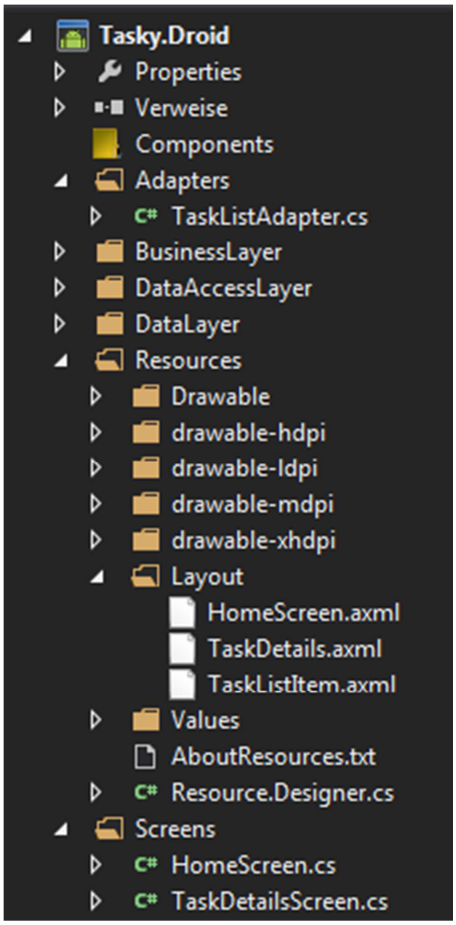


Abbildung 21: Tasky.Droid-Projekt

Die nebenstehende Abbildung illustriert das Tasky.Droid-Projekt. Business Layer, Data Access Layer und Data Layer werden vom bereits beschriebenen Tasky.Core-Projekt geklont. Die Klasse *TaskListAdapter* stellt den Application Layer dar und bindet die sichtbaren Objekte von Tasky.Droid an die Benutzeroberfläche. Konkret wird im

Adapter eine Methode implementiert, welche die Task-Objekte in einer Liste darstellt.

Der *Resource*-Ordner ähnelt dem einer klassischen nativen Android-Entwicklung und beinhaltet die Images und Icons. Dies bedeutet, dass Images und andere Ressourcen nicht plattformübergreifend verwendet werden können und sie daher in jedem plattformspezifischen Projekt importiert werden müssen. Im *Layout*-Ordner liegen die in einer Activity verwendeten Layouts als AXML-Format vor. Diese können entweder programmatisch oder mit dem im *Xamarin.Android* vorhandenem UI Designer erstellt und editiert werden. Da sich die Elemente im UI Designer per Drag & Drop platzieren lassen, ist dieser Vorgang zumindest bei nicht aufwändigen Layouts schnell und unkompliziert abgeschlossen.

Die *Screens* stellen die Android-Activities dar und bestimmen das Verhalten der Benutzeroberfläche. Sie verbinden die UI mit dem Application Layer. Damit die Anwendung weiß, welches die Activity ist, die beim Start angezeigt werden soll – in diesem Fall *HomeScreen* - muss die entsprechende Klasse in *Xamarin* durch *MainLauncher* gekennzeichnet werden:

```
Activity (Label = "TaskyPro", MainLauncher = true, Icon="@drawable/launcher")]
```

Auffällig ist hierbei, dass man auch im *Xamarin*-Umfeld von *Activity* spricht. Diese Nähe zum Android SDK zieht sich durch das komplette Tasky.Droid-Projekt. Beispielsweise wird bei einem Wechsel der View vom *HomeScreen* auf den *TaskDetailsScreen* ein Intent verwendet:

```
var taskDetails = new Intent (this, typeof (TaskDetailsScreen));
```

Die fertige Android-Version von Tasky ist von einer Anwendung, die im Android SDK entwickelt wurde, nicht zu unterscheiden:

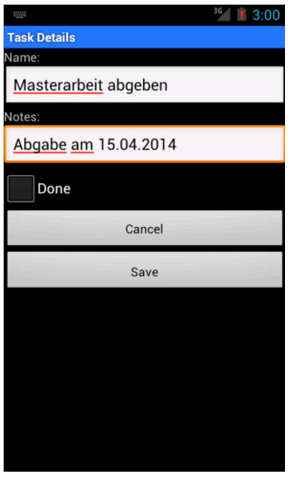


Abbildung 22: Tasky.Android - ToDo hinzufügen

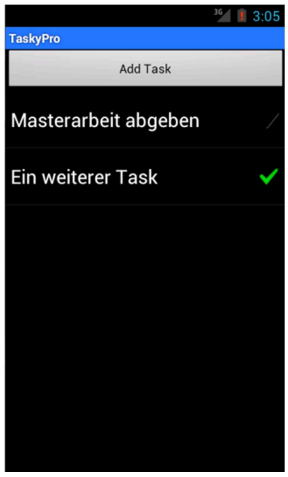


Abbildung 23: Tasky.Android - Listenansicht

Tasky.iOS und Tasky.Win8 implementieren jeweils ebenso einen Application Layer und eine Benutzeroberfläche. An dieser Stelle soll auf die Beschreibung der genauen Umsetzung verzichtet

werden, da dies für weitere Erkenntnisse unnötig ist. Anhand von Tasky.Core wurde gezeigt, welche Entwurfsmuster umgesetzt und welche Aufgaben in *Xamarin* plattformübergreifend entwickelt werden können. Die Beschreibung von Tasky.Android gibt einen Einblick, wie nahe sowohl Projektstruktur als auch Code (Activites, Intends, etc.) sind. Alle drei spezifischen Implementierungen unterscheiden sich deutlich voneinander, wodurch für jede Plattform eine Anwendung entsteht, welche von einer nativen, die im jeweiligen SDK entwickelt wurde, von einem Benutzer nicht zu unterscheiden ist.

Um die eingangs aufgeworfene Frage, wie viel Code in *Xamarin* plattformübergreifend entwickelt werden kann, beantworten zu können, wurden auf die beschriebene Art und Weise einige in *Xamarin* geschriebene Open-Source Anwendungen ([Xam145], [Ebe14]) analysiert. Dabei wurden - abgesehen von Projektstruktur und Entwurfsmustern - auf die plattformspezifischen und plattformunabhängigen Codeanteile und deren Verhältnis zueinander geachtet.

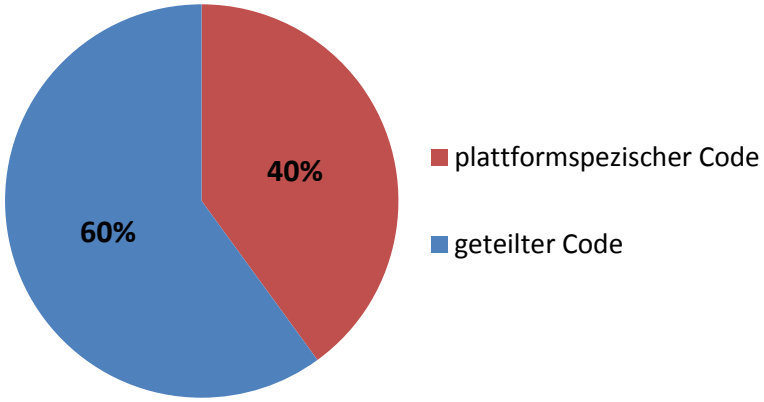


Abbildung 24: Xamarin - Code Separation in der Praxis

Abbildung 24 illustriert das Ergebnis der Analyse. Je nach Komplexität der Anwendung müssen demnach etwa 30-40% des Codes plattformspezifisch entwickelt werden. 40% können als Obergrenze angesehen werden, da beide Anteile je nach Komplexität der Anwendung gleichermaßen zu- oder abnehmen. Eine komplexe Anwendung hat in der Regel nicht nur eine komplexe Business Logik, die im Code plattformübergreifend geteilt wird, sondern auch eine entsprechend plattformspezifische Benutzeroberfläche, um die Workflows abbilden zu können. Zwar gibt es auch Ausnahmen, bei denen möglicherweise nur eine einzige View benötigt wird, während die Business Logik sehr komplexe Prozesse bearbeitet, deren Ergebnisse in der Präsentationsschicht nicht angezeigt werden. In der Regel ist jedoch davon auszugehen, dass eine mobile Business-Anwendung mehrere Views benötigt. Andersherum betrachtet, steigt bei mehreren Views im Normalfall auch der Code-Anteil in der Business Logik, da besagte Views nicht statisch sind und einen Workflow abbilden, der im geteilten Code implementiert ist.

6.2.5 Debugging

Um das Debugging in Visual Studio demonstrieren zu können, wird das bereits beschriebene Tasky.Droid-Projekt verwendet. Grundsätzlich lässt sich eine Anwendung im Simulator oder auf ein angeschlossenes Device debuggen. In diesem Beispiel wird der Android Emulator verwendet, allerdings unterscheidet sich der Vorgang nicht von einem Debugging mit angeschlossenem Gerät. Folgende Punkte soll der Debug-Prozess abdecken:

- Breakpoint beim Speichern eines ToDo's im UI Layer und Ansicht der Objekte und deren Eigenschaften
- Breakpoint beim Speichern eines ToDo's im Data Layer und Editieren eines Tasks im Debugger

Ein Task ließe sich bereits im UI Layer im Debugger editieren. Durch den zweiten Punkt soll jedoch sichergestellt werden, dass ein Debugging im plattformübergreifenden Code möglich ist. Der erste Breakpoint wird im UI Layer in der Klasse *TaskDetailsScreen* in der Save-Methode gesetzt:

```
protected void Save()
{
        task.Name = nameTextEdit.Text;
        task.Notes = notesTextEdit.Text;
        task.Done = doneCheckbox.Checked;
        Tasky.BL.Managers.TaskManager.SaveTask(task); // Breakpoint hier
        Finish();
}
```

Die Methode liest die Daten der Textfelder ein und ruft anschließend den *Taskmanager* auf, welcher den Aufruf an den plattformübergreifenden Code delegiert. Die Methode wird ausgeführt, sobald der Benutzer auf den Save-Button in der View klickt. Anschließend pausiert Visual Studio am Breakpoint. Die in Abb. 25 dargestellten Informationen sind sichtbar.

Name	Wert	Typ
this	{Tasky.Droid.Screens.TaskDetailsScreen}	Tasky.Droid.Screens.TaskDetailsScr
base	{Android.App.Activity}	Android.App.Activity
Non-public members		
cancelDeleteButton	{Android.Widget.Button}	Android.Widget.Button
doneCheckbox	{Android.Widget.CheckBox}	Android.Widget.CheckBox
nameTextEdit	{Android.Widget.EditText}	Android.Widget.EditText
notesTextEdit	{Android.Widget.EditText}	Android.Widget.EditText
saveButton	{Android.Widget.Button}	Android.Widget.Button
task	{Tasky.BL.Task}	Tasky.BL.Task
Done	false	bool
ID	0	int
Name	"Masterarbeit abgeben"	string
Notes	"Abgabe am 15.04.2014"	string
Non-public membe		

Abbildung 25: Tasky.Droid Breakpoint im UI Layer

Es sind sämtliche in der Klasse verfügbaren Objekte ersichtlich. Beispielsweise sind alle Eigenschaften des Task-Objekts sichtbar. Die Debug-Ausgabe ist genau so umfangreich, wie sie auch in einer reinen Android-Entwicklung mit Eclipse wäre, weshalb keine Funktionalität vermisst wird. Die Navigierung ist leicht und selbsterklärend, denn sie lässt sich mit nur drei Pfeilen steuern:

Abbildung 26: Visual Studio 2013 Debug Steuerung

Der erste Pfeil bedeutet *Step In*. Ist der Breakpoint auf einen Funktionsaufruf gesetzt, lässt sich mit F11 diese Funktion aufrufen und Zeile für Zeile durchgehen, während sich mit F10, *Step Over*, die Funktion überspringen lässt. Mit Shift+F11, *Step Return*, kann zu den vorherigen Zuständen zurückgekehrt werden.

Für den zweiten Test wird ein weiterer Breakpoint im Data Layer in der Klasse *TaskDatabase* gesetzt:

```
public int SaveItem<T> (T item) where T : BL.Contracts.IBusinessEntity
{
 lock (locker) {
    if (item.ID != 0) {          // Breakpoint hier
       Update (item);
       return item.ID;
    } else {
       return Insert (item);
    }
 }
}
```

Die Klasse setzt das Singleton-Pattern um und hat Zugriff auf die Datenbank-Methoden. In der SaveItem-Methode wird der Zugriff synchronisiert, damit in der Datenbank keine Inkonsistenzen entstehen. Nach setzten des zweiten Breakpoints wird der pausierte Prozess fortgeführt, wonach anschließend der zweite Break erfolgt, in dem man den erfolgreichen Lock noch einmal an *s_LockTaken = true* beoachten kann:

Name	Wert	Typ
this	{Tasky.DL.TaskDatabase}	Tasky.DL.TaskDatabase
item	{Tasky.BL.Task}	Tasky.BL.Task
Done	true	bool
ID	2	int
Name	"Masterarbeit abgeben"	string
Notes	"Abgabe am 15.04.2014"	string
Non-public members		
<>s__LockTaken8	true	bool

Abbildung 27: Tasky.Droid Breakpoint im Data Layer

Mit Doppelklick auf einen Wert lässt sich dieser bearbeiten. In diesem Fall wird, wie in Abb. 27 illustriert, die boolesche Eigenschaft *Done* des Task-Objektes von *false* auf *true* modifiziert. Wird ein nicht boolescher Wert in das Feld eingetragen, macht Visual Studio auf diesen Fehler aufmerksam und setzt die Variable auf den ursprünglichen Wert zurück. Nach der Änderung auf *true* wird das Task-Objekt mit den geänderten Daten erfolgreich in die Datenbank geschrieben, weshalb der Task in der App daraufhin als *done* markiert ist.

Das Debugging funktioniert auch mit den anderen plattformspezifischen Projekten ohne Probleme. Für iOS muss jedoch bedacht werden, dass der Simulator remote auf einem MacOS-System läuft, so dass man an zwei Geräten parallel arbeiten muss.

Das Debugging in Xamarin Studio soll an dieser Stelle aus zwei Gründen nicht vorgestellt werden. Zum einen funktioniert es analog zur vorgestellten Methode, zum anderen ist eine Cross-Plattform-Entwicklung, welche Android, iOS und Windows Phone abdeckt, aus den in Punkt 6.2.3 („*Entwicklungsumgebung und Build-Prozess*“) genannten Gründen, nur in Visual Studio möglich. Dennoch wurde das Debugging in Xamarin Studio getestet. Während vieler Debugging-Tests kam es vereinzelt zu unerklärlichen Abstürzen der mobilen Anwendung im Simulator. Daher ist Visual Studio auch dann für eine Entwicklung zu empfehlen, wenn keine Cross-Plattform-Entwicklung stattfinden und lediglich eine Android-Applikation entstehen soll.

Insgesamt sind die Eindrücke, vor allem in Visual Studio, positiv. Eine in *Xamarin* entwickelte Anwendung lässt sich mit gleichen Mitteln debuggen wie eine native Anwendung, welche mit entsprechendem SDK entwickelt wird.

6.2.6 Reverse Engineering

Mobile Anwendungen können unternehmenskritische Daten enthalten, weswegen der Quelltext schützenswert ist. Dieses Kapitel untersucht, wie hoch die Hürden einer in *Xamarin* geschriebenen Applikationen sind, die ein potenzieller Angreifer überwinden muss.

Standardmäßig werden Android-Anwendungen in *Xamarin Just-in-Time* (*JIT*) kompiliert. Mit Enterprise-Lizenz erhält man die Möglichkeit auf eine *Ahead-of-Time* (*AOT*) - Kompilierung umzustellen. Für iOS besteht keine Wahl und es wird immer *AOT* angewendet. Die Art der Kompilierung beeinflusst das Reverse Engineering, weshalb beide Methoden anhand von Tasky.Droid kurz vorgestellt werden.

Abbildung 28 zeigt den Inhalt des Android Packages (APK) von Tasky.Droid, welches mit *JIT*-Option kompiliert wurde.

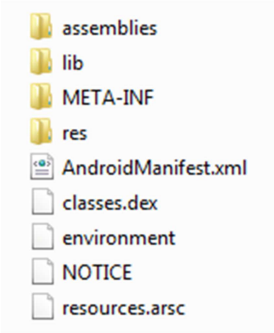


Abbildung 28: Xamarin APK Inhalt

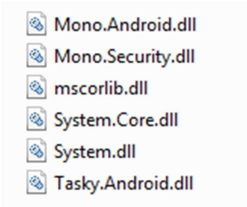


Abbildung 29: Xamarin Assemblies

Zusätzlich zu den typischen APKs, enthält das Paket einen Ordner namens *assemblies*, welcher die .NET Assemblies enthält und dessen Inhalt in Abb. 29 ersichtlich ist. Tasky.Android.dll beinhaltet sowohl den plattformspezifischen als auch den plattformübergreifenden Code, weshalb sie für einen potenziellen Angreifer das Ziel darstellt. Aus diesem Grund wird die Datei in einem .NET Decompiler, in diesem Fall *Dis#* [NET14], dekompiliert:

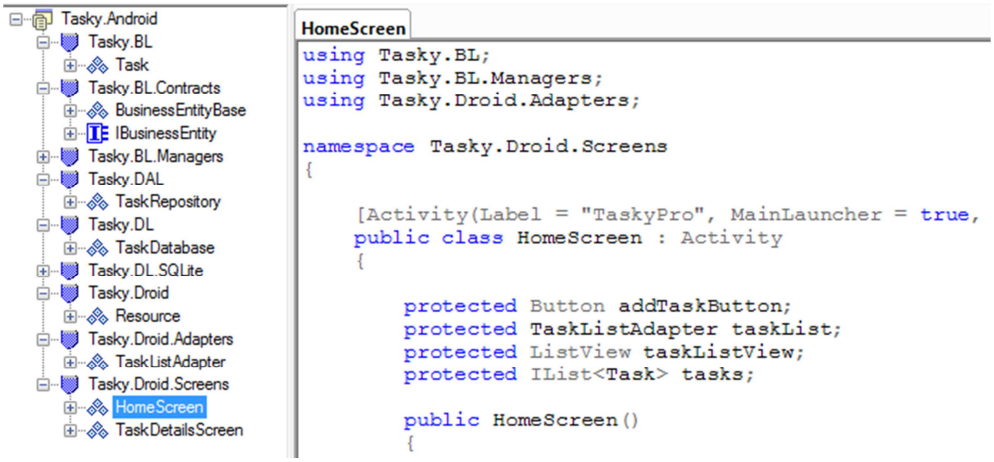


Abbildung 30: Tasky.Droid (JIT) – dekompilierter Code

Der Code wird inklusive aller Klassen-, Methoden- und Variablennamen ausgegeben, wodurch es ein Angreifer in diesem Fall sehr leicht hätte, sensitive Informationen zu stehlen. Dies lässt sich verhindern, indem ein .NET Obfuscator eingesetzt wird:

```
public HomeScreen()
{
}

[CompilerGenerated]
private void c37a51ea9575e7ad89511105671b951d6(object ca9220a41a8f6ffbdb206e638ccfe
{
    StartActivity(typeof(TaskDetailsScreen));
}

[CompilerGenerated]
private void c69606d8faadf7262e0567b36e26e6654(object ca9220a41a8f6ffbdb206e638ccfe
{
    Intent intent = new Intent(this, typeof(TaskDetailsScreen));
    c44c9e6dc3eae6e6310bb8c3f767f544d.cb0418255b7a1bc445dc4a48cc5f5125c(2153);
    tasks.get_Item(cfe9c77f2039c9054fe647569a585b78f.Position);
    intent.PutExtra(c44c9e6dc3eae6e6310bb8c3f767f544d.cb0418255b7a1bc445dc4a48cc5f5
    intent.PutExtra(c44c9e6dc3eae6e6310bb8c3f767f544d.cb0418255b7a1bc445dc4a48cc5f5
    StartActivity(intent);
}
```

Abbildung 31: Tasky.Droid (JIT) – obfuscated Code

Der durch einen Obfuscator modifizierte Code ist für einen Menschen nur noch sehr schwer lesbar, funktioniert auf dem Gerät aber weiterhin ohne Nachteile. In *Xamarin* ist standardmäßig kein Obfuscator enthalten, jedoch können beliebige .NET Obfuscator genutzt werden. Für dieses Beispiel wurde der *Crypto Obfuscator For .Net* verwendet [Log14], welcher sich schnell und unkompliziert in .NET Projekte einbinden lässt, so dass der Verschleierungsprozess des Quelltextes automatisiert erfolgen kann.

Wählt man eine *Ahead-of-Time* – Kompilierung, benötigt man keinen .NET Obfuscator, da es keine Assemblies mehr in der APK gibt und diese zu nativen Code kompiliert und als Library unter */lib* ins Projekt gepackt werden. *Xamarin* nennt diese Datei *libmonodroid_bundle_app.so*, welche alle benötigten Assemblies als Maschinencode enthält:

```
.rodata:000004E0 ; Segment type: Pure data
.rodata:000004E0                 AREA .rodata, DATA, READONLY, ALIGN=5
.rodata:000004E0                 ; ORG 0x4E0
.rodata:000004E0 aTasky_android_ DCB "Tasky.Android.dll",0
.rodata:000004E0                                         ; DATA XREF: .data.rel.ro:assembly_bundle_Tasky_Android_dll↓o
.rodata:000004E0                                         ; .data:image_name↓o
.rodata:000004F2                 ALIGN 4
.rodata:000004F4 aMono_android_d DCB "Mono.Android.dll",0
.rodata:000004F4                                         ; DATA XREF: .data.rel.ro:assembly_bundle_Mono_Android_dll↓o
.rodata:00000505                 ALIGN 4
.rodata:00000508 aSystem_dll     DCB "System.dll",0      ; DATA XREF: .data.rel.ro:assembly_bundle_System_dll↓o
.rodata:00000513                 ALIGN 4
.rodata:00000514 aMono_security_ DCB "Mono.Security.dll",0
.rodata:00000514                                         ; DATA XREF: .data.rel.ro:assembly_bundle_Mono_Security_dll↓o
.rodata:00000526                 ALIGN 4
.rodata:00000528 aSystem_core_dl DCB "System.Core.dll",0 ; DATA XREF: .data.rel.ro:assembly_bundle_System_Core_dll↓o
.rodata:00000538 aMscorlib_dll   DCB "mscorlib.dll",0    ; DATA XREF: .data.rel.ro:assembly_bundle_mscorlib_dll↓o
.rodata:00000545                 ALIGN 0x20
```

Abbildung 32: Tasky.Droid (AOT) - disassemblierter Code

Diese native Library ist wesentlich schwerer zu reengineeren als .NET Assemblies, da ein Angreifer den Code nicht mehr dekompilieren kann und ihn stattdessen disassemblieren muss, wodurch es keine Rückschlüsse mehr auf Klassen-, Methoden- oder Variablennamen gibt, wie sie in einer Hochsprache verwendet werden. Ohne Kenntnisse in ARM Assembler ist es nur noch schwer möglich den Code nachvollziehen zu können.

6.2.7 Bewertung

In diesem Kapitel wird das Framework anhand der im Punkt 5.3 vorgestellten Methodologie bewertet. Die Bewertung soll pro Entscheidungskriterium kurz begründet werden.

Funktionalität

Xamarin unterstützt mit Android, iOS und Windows Phone alle geforderten mobilen Plattformen. Durch die C# Bindings lassen sich die nativen Schnittstellen auf Android und iOS nutzen, so dass es in dieser Hinsicht keine Einschränkungen im Vergleich zu einer nativen Entwicklung mit entsprechendem Software Development Kit gibt. Gleichwohl gibt es auf Android und iOS kleine Limitierungen, weshalb z.B. generische Typen oder generische Subklassen nicht bzw. nur in Einzelfällen

unterstützt werden. Da Visual Studio mit C# die native Entwicklungsumgebung für Windows Phone darstellt, gibt es für dieses Betriebssystem keine solchen Einschränkungen.
Das Framework erfüllt die Anforderungen an die Architektur, da Anwendungen in der Regel eine Model-View-Controller – Architektur umsetzen und Libraries implementiert werden können.

Insgesamt wird Xamarin in dieser Kategorie mit 3,5 bewertet. 4 würde bedeuten, dass die Erwartungen erfüllt wurden. Der Benchmark ist an dieser Stelle das jeweilige SDK einer Plattform. Durch die angesprochenen Limitierungen, die jedoch nicht gravierend sind, wird ein halber Punkt abgezogen.

Score: 3,5

Benutzbarkeit

Aus Anwendersicht wird erwartet, dass sich die App wie eine im Software Development Kit entwickelte native Anwendung anfühlt, d.h. die Benutzeroberfläche soll nicht nur wie eine im SDK entwickelte aussehen, sondern genauso flüssig laufen. Diese Erwartungshaltung kann *Xamarin* zweifelsohne erfüllen, da für einen Benutzer nicht ersichtlich ist, wie eine mobile Anwendung entwickelt wurde. Das Look & Feel entspricht dem einer nativen App.

Aus Entwicklersicht fällt eine Einarbeitung in Xamarin Studio oder Visual Studio leicht. Eine Cross-Plattform-Entwicklung, die Android, iOS und Windows Phone abdecken soll, ist allerdings nur in Visual Studio möglich, welches lediglich für Windows verfügbar ist. Für Android und Windows Phone ist der Build-Prozess innerhalb dieser Entwicklungsumgebung leicht durchführbar. Ein Build-Prozess für iOS ist jedoch insofern kompliziert, als dass stets ein MacOS-System benötigt wird, welches so eingerichtet werden muss, dass darauf per Remote von Windows aus kompiliert und debugged werden kann.

Die Benutzbarkeit des Frameworks wird mit 3,5 bewertet. Die Erwartungen aus Anwendersicht wurden erfüllt. Der Abzug um 0,5 Punkte erfolgt auf Grund des etwas komplizierten Build-Prozesses, den iOS erfordert. Des Weiteren werden mit Xamarin Studio und Visual Studio zwar zwei Entwicklungsumgebungen angeboten, allerdings ist nur Visual Studio für eine Cross-Plattform-Entwicklung, die den Anforderungen dieser Arbeit gerecht wird, geeignet.

Score: 3,5

Zuverlässigkeit und Effizienz

Während einiger Debugging-Sessions kam es innerhalb Xamarin Studio vereinzelt zu Abstürzen. Da eine plattformübergreifende Entwicklung, die Android, iOS und Windows Phone, abdeckt, jedoch ohnehin nur in Visual Studio möglich ist, geht dieser negative Fakt nicht in die Bewertung ein. Die Entwicklung in Visual Studio erfüllt die Erwartungen.

Aus Benutzersicht wurde u.a. geprüft, wie effizient eine Liste mit 10000 Elementen geladen wird. Es wurden weder Ruckler, noch Ladezeiten wahrgenommen, weshalb die Erwartungen aus Benutzersicht erfüllt wurden.

Score: 4

Wartbarkeit und Support

Die Dokumentation auf der Webseite ist teilweise irreführend, so dass z.B. Features erwähnt werden, die offiziell noch nicht existieren. Abgesehen davon gibt es Tutorials für Einsteiger und eine vollständige API-Dokumentation. Für fortgeschrittene Entwickler gibt es die Xamarin University, welche pro Jahr $1995 kostet und an der Zertifikate erworben werden können.

Ein Support, bei dem Hotfixes und garantierte Antwortzeiten in Form von Service Level Agreements verfügbar sind, ist erst ab der Enterprise-Version erhältlich, welche die teuerste aller Lizenzen darstellt. Sie kostet $1899 pro Jahr und pro Entwickler. Die Lizenzen sind immer für ein ganzes Jahr ausgelegt und lassen sich daher nicht für bestimmte Projekte zeitweise aufstocken.

Eine spezifische Anforderung seitens *TK* ist, dass die in der Benutzeroberfläche sichtbaren Texte (z.B. Labels) in einer App exportierbar sein sollen, damit diese vom Produktmanagement gewartet werden können. Eine solche Funktionalität existiert in *Xamarin* nicht, allerdings lassen sich die Strings aus den Dateien in den Projektverzeichnissen extrahieren. Da aber die UI pro Plattform separat entwickelt wird, muss der beschriebene Vorgang ebenso pro Plattform erfolgen.

Insgesamt werden die Wartbarkeit und der Support mit 3 bewertet. Der Abzug erfolgte zum einen auf Grund der unflexiblen Lizensierung und zum anderen ist sie der Tatsache geschuldet, dass in keiner Lizenz festgehalten ist, was mit dem Framework bzw. dessen Quelltext im Falle einer Insolvenz o.Ä. seitens *Xamarin Inc.* geschieht.

Eine Lösung, die z.B. vorsieht, dass der Quelltext an den Kunden übergeht, wäre für eine sichere Zukunft wünschenswert.

Score: 3

Kompromissbereitschaft

Die Kompromissbereitschaft soll den Zusammenhang zwischen dem Portierbarkeitsgrad und den damit verbundenen funktionalen Einbußen bewerten. Eine hohe Bewertung dieses Faktors bedeutet, dass mit wenigen Kompromissen zu rechnen ist. Im Falle von *Xamarin* können etwa 60% des Codes geteilt werden, wobei gleichzeitig lediglich die im Punkt *Funktionalität* genannten Limitierungen für Android und iOS auftreten. Die Entwicklung für Windows Phone unterliegt keinen Einschränkungen. Die Erwartungen wurden geringfügig übertroffen, dafür müssen jedoch bis zu 40% plattformspezifisch implementiert werden.

Score: 4,5

Gewichtung

Die Bewertungen der einzelnen Kategorien im Überblick:

Entscheidungskriterium	Bewertung	Gewichtung
Funktionalität	3,5	4
Benutzbarkeit	3,5	4
Zuverlässigkeit und Effizienz	4	4
Wartbarkeit und Support	3	5
Kompromissbereitschaft	4,5	3

Der gemittelte Score beträgt demnach **3,625**.

6.3 Codename One

Codename One, auch *CN1* genannt, ist ein Open-Source Framework, welches eine *Write once, run anywhere*-Philosophie verfolgt und daher in die Kategorie der vollständigen Lösungen (vgl. Kapitel 4.2.8) eingeordnet wird. Eine erste Betaversion wurde am 13. Januar 2012 von Shai Almog und Chen Fishbein, ehemaliger Sun-Mitarbeiter und Gründer des Lightweight User Interface Toolkits (LWUIT), der Öffentlichkeit vorgestellt [Alm14]. Stand März 2014 befindet sich *CN1* in der Release-Version 2.0.

6.3.1 Voraussetzungen und Möglichkeiten

Anwendungen in *Codename One* werden in Java geschrieben und können für Android, iOS, Windows Phone, Blackberry OS und J2ME-basierte Geräte kompiliert werden. Darüber hinaus werden die Desktop-Systeme Windows und MacOS X unterstützt. Der Java-Quelltext wird während des Build Prozesses (siehe Kapitel 6.3.3) zur jeweiligen plattformspezifischen Sprache übersetzt, wofür hauptsächlich *XMLVM* (siehe Kapitel 4.2.2) zum Einsatz kommt. Die zugrunde liegenden Techniken werden von einem Entwickler jedoch nicht wahrgenommen, weshalb diesbezüglich kein Know-How erforderlich ist.

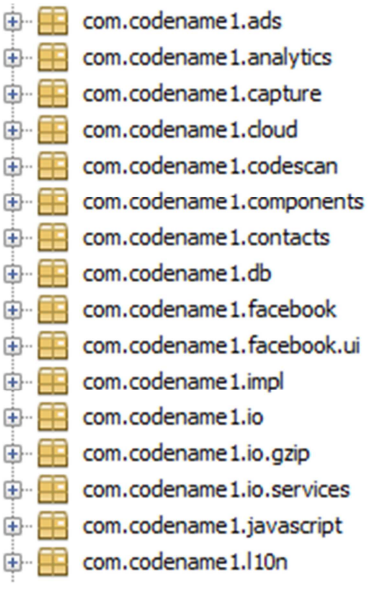


Abbildung 33: CN1 Packages (Ausschnitt)

Die plattformübergreifende Funktionalität wird durch eine weitere Abstraktionsebene erreicht, welche in Form einer Bibliothek bereitgestellt wird. Diese implementiert eigene Schnittstellen, die statt der nativen aufgerufen werden. Um die native Funktionalität zu erlangen, fungieren die Schnittstellen der Bibliothek als eine Art Wrapper hinter denen sich die nativen Aufrufe pro Plattform verbergen. Die Bibliothek implementiert bereits alle gängigen Features wie z.B. GPS, Maps, Network, Storage, L10n, Push, E-Mail, XML, JSON, SMS, Camera, Contacs und viele mehr (vgl. Abb. 33).

Plattformspezifische Kenntnisse sind für eine Entwicklung nicht erforderlich, so dass ein Entwickler ausschließlich Java beherrschen muss. Auch unabhängig von der Programmiersprache benötigt ein Entwickler kein spezifisches Know-How bzgl. einer Plattform. Beispielsweise muss man nicht wissen, welche Rechte eine Android-Anwendung erfordert, da diese von *Codename One* automatisch gesetzt werden. Dabei werden ausschließlich die Rechte verwendet, die tatsächlich genutzt werden.

Plattformspezifische Kenntnisse sind höchstens dann gefragt, wenn man eigene Bibliotheken für das Framework entwickeln möchte, damit sie es um native Funktionen erweitern. Grundsätzlich lassen sich nicht nur native, sondern auch in Java geschriebene Bibliotheken implementieren, weshalb *Codename One* ideal für eine modulare Programmierung geeignet ist.

Für *CN1* geschriebene Libraries liegen nicht im JAR, sondern im „CN1Lib"-Format vor. Das eigene Format hat drei Ursachen:

- *Codename One* ist für eine mobile Cross-Plattform-Entwicklung mit geringem Ressourcen-Verbrauch optimiert und unterstützt daher „nur" ein Subset von Java 5. Konkret sind z.B. keine Reflections verfügbar. Auch die java.io.* Klassen sind für eine plattformübergreifende Entwicklung ungeeignet, weshalb solche Operationen im Packet com.codename1.io enthalten sind.
- Das Format bietet die Möglichkeit zur Inklusion von Android-spezifischen JAR- oder iOS-spezifischen .a Libraries, damit die bereits erwähnte Erweiterung um native Funktionen möglich ist.
- Eine CN1Lib enthält generierte Signaturen und Javadocs und ermöglicht somit Code Completion in den Entwicklungsumgebungen. Eine JAR-Library würde dies ebenfalls unterstützen, allerdings wäre dafür ein gewisser Aufwand erforderlich, wohingegen es im CN1Lib-Format automatisch funktioniert.

Es gibt bereits einige CN1Libs, die zur Erweiterung des Frameworks dienen und kostenlos sind. Die Entwicklung erfolgte teilweise durch Fremdentwickler bzw. durch die Community. Unter den Bibliotheken befinden sich u.a. Erweiterungen für die Erstellung von Charts und welche, die eine Verbindung zu Dropbox und Salesforce erleichtern.

Die Benutzeroberfläche kann entweder programmatisch oder mit dem *Codename One Designer* erstellt werden, welcher aus vier Kernkomponenten besteht:

- Der Designer beinhaltet einen GUI Builder, in dem die UI per Drag & Drop erstellt werden kann. Für eine Cross-Plattform-Benutzeroberfläche wird auf das Lightweight User Interface Toolkit (LWUIT) aufgesetzt, welches für mobile Geräte optimiert ist. *Codename One* erreicht eine hohe Performanz, in dem die Elemente durch native Gaming-APIs, z.B. durch OpenGL ES auf iOS, gezeichnet werden.

- Ein Theme Creator ermöglicht ein einheitliches Design, wobei pro Plattform unterschiedliche Themes angewendet werden können, wodurch man auf die verschiedenen Design-Guidelines der Plattformen reagieren kann.
- Der Designer implementiert eine einfache Lokalisierung zur Unterstützung mehrerer Sprachen. Die in einer Anwendung befindlichen Strings für Labels, Textfelder usw. können im XML- oder CSV-Format exportiert und importiert werden. Dadurch können sie von anderen Personen in anderen Programmen bearbeitet werden.
- Ein im Designer integriertes Image Management sorgt für eine übersichtliche Verwaltung aller Bilddateien. Dabei werden die unterschiedlichen Auflösungen der Geräte berücksichtigt und multi-DPI Images unterstützt, die je nach Auflösung passend dargestellt werden.

Der Designer funktioniert unabhängig vom Rest des Frameworks, wodurch sich eine Benutzeroberfläche von einem Designer erstellen lässt, der keine Programmierkenntnisse besitzt. Es wird eine *.res*-Datei erstellt, die das Design enthält und dem Programmierer übergeben werden kann.

Abbildung 34: CN1 Simulator

In *Codename One* entwickelte Apps lassen sich in einem Simulator, welcher in Abb. 34 dargestellt ist, testen und debuggen. Dieser startet ohne Verzögerungen und ist somit z.B. spürbar schneller als der Android Emulator. Der Simulator verfügt über veränderbare Eigenschaften:

- Es werden unterschiedliche Skins für eine Vielzahl an Devices angeboten. Dabei variieren die Skins nicht nur im Aussehen, sondern beeinflussen auch die Funktionalität. In dieser Art und Weise lassen sich verschiedene Plattformen und unterschiedliche Auflösungen usw. testen.
- Der Simulator lässt sich rotieren.
- Die Geschwindigkeit der Netzwerkverbindung ist variierbar, wodurch sich ein schlechter Empfang simulieren lässt.

Weiterhin lassen sich mit dem Simulator Eigenschaften wie die Netzwerkverbindung überwachen, so dass z.B. Requests und Responses mitgeloggt werden können. Durch einen Performance Monitor werden Schwachstellen aufgedeckt, welche die Benutzbarkeit der Anwendung negativ beeinflussen.

Limitierungen oder Kompromisse bzgl. der Funktionalität sind insofern zu erwarten, als dass die Abstraktionsschicht nicht alle Funktionen jeder Plattform in der Art und Weise abdeckt, wie man es vielleicht gewohnt ist. Beispielsweise ist das Map-Feature pro Plattform unterschiedlich: Android setzt auf Google Maps, iOS auf Apple Maps und Windows Phone auf Nokia Maps. Es gibt sogar Android-Geräte, wie z.B. Amazons Kindle, die kein Google Maps unterstützen. Eine Cross-Plattform-Implementierung eines Map-Features ist demzufolge schwer umsetzbar. *Codename One* löst diese Problematik, indem es eine abstrakte Klasse namens *MapProvider* implementiert und es bisher zwei konkrete Umsetzungen dieser Klasse gibt. Zum einen gibt es den Google Maps- und zum anderen den OpenStreet Map-Provider. Sie erben die Methoden des MapProviders, wodurch sich nur der Konstruktor unterscheidet und die konkreten Methoden identisch sind. Der Vorteil dieser Implementierung ist, dass sich sehr schnell und ohne große Code-Anpassungen der Kartendienst ändern lässt und dass diese Dienste plattformunabhängig sind. Je nach Konstruktor werden die Tiles vom entsprechenden Server heruntergeladen. Der Nachteil ist allerdings, dass man möglicherweise nicht von Optimierungen profitieren kann, die z.B. Google in Form von Vektorgrafiken für seinen Kartendienst implementiert hat.

Sollte man bei einem Feature auf einer solchen Implementierung stoßen und sich daran stören, ließe sich das Framework, wie bereits erwähnt, durch Libraries erweiterten. Die für das Map-Feature verwendete Strategie war eine Design-Entscheidung, die sich ändern lässt. Es ist daher möglich, das Framework in der Art und Weise zu erweitern, dass z.B. die auf Android-Geräten nativen Google Maps genutzt werden. Im späteren Verlauf dieser Arbeit wird dieser Punkt noch einmal bei der Implementierung eines Prototyps aufgegriffen (siehe Kapitel 7.4.2 „*Plattformübergreifende Karten*“).

6.3.2 Support und Dokumentation

Die gleichnamige Firma, die hinter *Codename One* steht, bietet fünf Lizenztypen an. Das Framework ist zwar Open-Source und damit für jeden kostenlos zugänglich, der Build-Prozess der programmierten Anwendungen findet jedoch hauptsächlich innerhalb einer Cloud statt, deren Funktionsweise das Geschäftsgeheimnis von *Codename One* ist. Um die Cloud nutzen zu können, muss eine entsprechende Registrierung stattfinden, bei der keine Angabe persönlicher Daten notwendig sind.

Die fünf Lizenztypen teilen sich in Free, Basic, Pro, Enterprise und Corporate auf. Alle Lizenzen enthalten einen Community-Support, welcher nicht nur von den Usern selbst, sondern ebenso von den *Codename One* Mitarbeitern getragen wird. So besteht zwar keinerlei Anspruch auf eine qualitative Antwort, jedoch ist eine solche während der Erstellungszeit dieser Arbeit (Oktober

2013 – April 2014) die Regel, weshalb der Community-Support als hochwertig einzustufen ist. Auf Grund des Open-Source Charakters können neben den Quelltexten ebenfalls Changesets und Tickets eingesehen werden, wodurch die Entwicklung des Frameworks transparent gestaltet ist. Unabhängig von der Lizenz ist es jedem Entwickler gestattet, eigene Tickets zu öffnen, die nicht nur Bugs, sondern auch Feature-Requests enthalten können. Für die Priorisierung der Tickets fallen zwei Kriterien ins Gewicht. Auf der einen Seite werden die Tickets je nach Lizenztyp priorisiert, auf der anderen Seite wird ebenso das „Wohl" des Projektes mit einbezogen. Daher werden schwerwiegende Bugs oder Sicherheitslücken unabhängig von der Art der Lizenz mit höchster Priorität eingestuft.

Es ist eine vollständige Dokumentation aller Schnittstellen einsehbar. Zusätzlich gibt es einen 212 Seiten langen Developer-Guide, der viele Beispiele enthält. Weiterhin existiert eine „How do I?"-Webseite, auf der sich 28 Video-Tutorials befinden (Stand: April 2014), ebenso Anfänger wie Fortgeschrittene ansprechen und daher eine erste Anlaufstelle für Probleme darstellt.

Die Free-Version erlaubt die kostenlose, aber limitierte Nutzung des Build-Servers für mobile Plattformen, wohingegen die Desktop-Plattformen nicht unterstützt werden. Ein Free-User erhält pro Monat ein Kontingent von einhundert Build-Credits. Für jede iOS-Anwendung, die in der Cloud kompiliert wird, werden zwanzig Credits, für alle anderen mobilen Plattformen jeweils ein Credit verbraucht. Der hohe Credit-Verbrauch für iOS gegenüber den anderen Systemen wird mit den einhergehenden hohen Kosten begründet, die für das Hosting eines Mac-mini Servers anfallen.

Die Basic-Lizenz veranschlagt pro Entwickler $9 im Monat und hebelt das Credit-System aus, wodurch ein Entwickler unendlich viele Anwendungen in der Cloud kompilieren kann. Zusätzlich kann man ab dieser Lizenzstufe nicht nur die in der Cloud kompilierten Binary-Dateien downloaden, sondern auch den dazugehörigen nativen Quelltext. Abgesehen davon gibt es in dieser Version keinen Support, der über den Community-Support hinausgeht.

Die Pro-Lizenz, welche pro Entwickler monatlich $79 kostet, stellt deutlich mehr Features als die Basic-Version zur Verfügung. In dieser Version sind parallele Builds mehrerer Anwendungen möglich. Weiterhin werden Push Notications, welche über den Cloud Server versendet werden, unterstützt und Cloud Storage bereitgestellt, auf dem Dateien oder Java-Objekte gespeichert werden können. Mit „Crash Reporting" wird ein Feature angeboten, bei dem ein Entwickler Strack-Traces von Abstürzen automatisiert per Email erhält. In der Pro-Version lassen sich sowohl Desktop-Apps, als auch Unit-Tests auf dem Build-Server kompilieren. Der Email-Support wertet den Support auf, wenngleich keine Antwortzeiten in Form von Service-Level-Agreements o.Ä. definiert

sind. Des Weiteren erhalten Kunden ab dieser Lizenzstufe einen Zugriff auf 45 weitere Video-Trainings, die User mit Free- oder Basic-Lizenz für einmalig $199 erhalten könnten [Cod14].

Die Enterprise-Version ist die Lizenz, die prinzipiell für Unternehmen zu empfehlen ist, da sie den Support enorm aufwertet. Durch Service-Level-Agreements wird eine Antwortzeit von 48h garantiert. Aus eigener Erfahrung stellte sich in der Praxis bisher eine Antwortzeit von etwa 10 Minuten heraus, weshalb die Entwicklung bei auftretenden Problemen nicht ins Stocken geriet. Die Enterprise-Lizenz enthält einen telefonischen und einen Onsite-Support, wobei bei letzterem individuelle Kosten hinzukommen. In den Service-Level-Agreements ist zusätzlich ein Escrow-Service verankert, bei dem sämtliche interne Quellen, d.h. auch die des Build-Servers, zum Escrow-Provider übermittelt werden, wodurch ein Kunde gegen eine Insolvenz seitens der Firma Codename One abgesichert ist. Weiterhin gibt es einen Offline-Build-Support, wodurch eine Unabhängigkeit zum Cloud-Server erlangt werden kann.

Die Corporate-Lizenz wird erst seit Januar 2014 angeboten und ist die einzige, die mit $14.900 jährlich statt monatlich bezahlt werden muss. Bei ihr erhält ein Kunde weder Support per Telefon, noch per Mail, da diese Lizenz prinzipiell nicht für einen bestimmten Entwickler gedacht ist. Stattdessen wird ein privater Cloud-Server lizensiert, um von der fremden Cloud unabhängig arbeiten zu können. Die Corporate-Version eröffnet Unternehmen die Möglichkeit, eigene Zugriffe zu verwalten, wodurch externe Entwickler ohne spezielle CN1-Lizenz und ohne eine Limitierung durch Credits auf dem eigenen Build-Server kompilieren können. Ein eigener Build-Server sorgt zudem dafür, dass für eine Kompilierung keine eventuell sensitiven Daten in eine fremde Cloud geschickt werden müssen.

Für eine effiziente und von fremden Servern unabhängige Entwicklung ist für Unternehmen eine Corporate-Lizenz mit entsprechenden Enterprise-Lizenzen zu empfehlen, da diese alle Features vereinen. Des Weiteren wird ein Support lizensiert, bei dem schnelle Antwortzeiten garantiert werden. Eventuelle Tickets und Feature-Requests bzgl. des Frameworks genießen die höchste Priorität.

6.3.3 Entwicklungsumgebung und Build-Prozess

Als Entwicklungsumgebung für *Codename One*, kann Netbeans, Eclipse oder IntelliJ IDEA verwendet werden, da für alle drei genannten IDEs Plug-Ins angeboten werden. Das Plug-In integriert die spezifischen Features des Frameworks, wodurch *Codename One*- statt gewöhnliche Java-Projekte erstellt werden können. Zum Plug-In gehören zudem der Simulator zum Testen und Debuggen, sowie der Codename One Designer zum Erstellen der Benutzeroberfläche. Sowohl der Simulator

als auch der Designer wurden bereits im Kapitel 6.3.1 („*Voraussetzungen und Möglichkeiten*“) vorgestellt.

Bevor man eine der drei genannten Entwicklungsumgebungen auswählt, ist es ratsam, entweder in das Changelog von *Codename One* zu sehen oder vorher das jeweilige Plug-In selbst zu testen, da sich während der Evaluierung herausstellte, dass nicht alle drei Plug-Ins den vollen Funktionsumfang implementieren (Stand: März 2014). So ist es in Eclipse zwar möglich *CN1*-Projekte, jedoch keine *CN1*-Library-Projekte zu erstellen, welche für eine modulare Entwicklung wichtig sind. Das IntelliJ IDEA Plug-In ist das neuste der drei genannten, dennoch lassen sich darin im Gegensatz zu Eclipse ebenso *CN1*-Libraries entwickeln. Allerdings traten dafür Stabilitätsprobleme auf, wodurch es während der Entwicklung zu Exceptions kam. Das Netbeans Plug-In war das erste aller Plug-Ins und wird bis zum heutigen Tage am besten supported, so dass es alle *CN1*-Features abdeckt und als stabil zu bezeichnen ist, wenngleich es während einer dreimonatigen Nutzungszeit zu zwei Abstürzen im Designer kam.

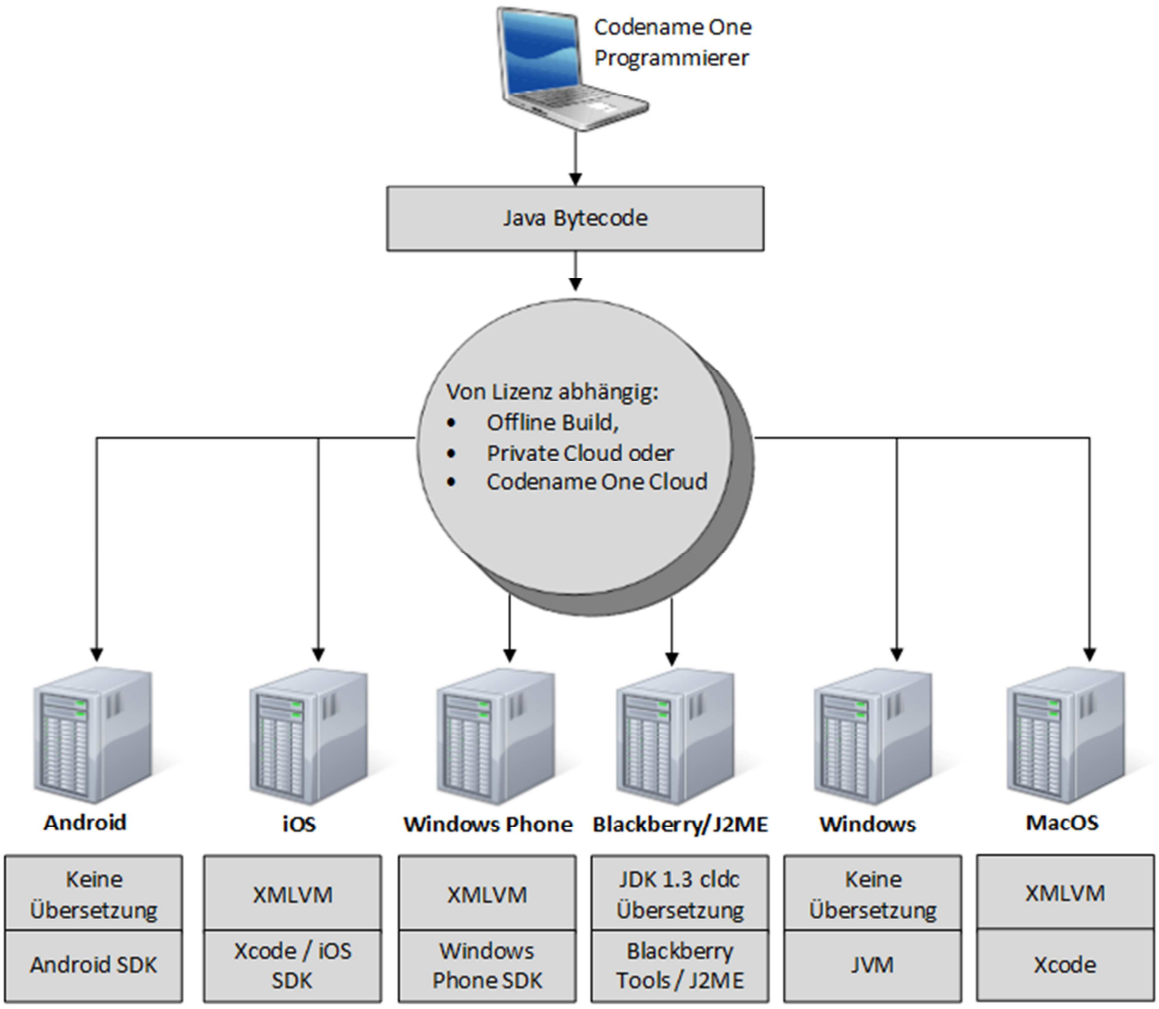


Abbildung 35: CN1 Build-Prozess

Eine Kompilierung einer in *Codename One* entwickelten mobilen Anwendung ist je nach Lizenz auf drei unterschiedliche Arten möglich:

- Codename One Cloud Server
- Privater Cloud Server (bedingt Corporate-Lizenz)
- Offline-Build (bedingt Enterprise-Lizenz)

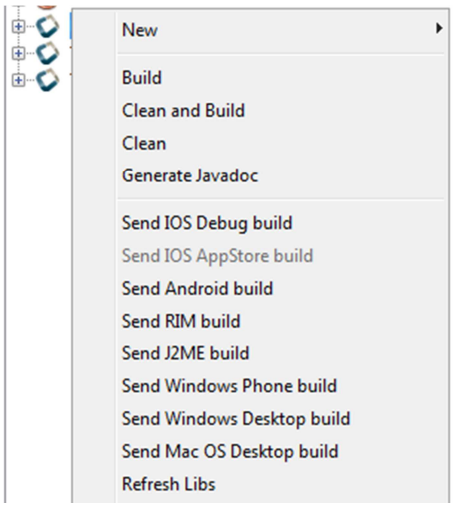


Abbildung 36: CN1 Build-Menü

Die Cloud besteht aus mindestens zwei Linux-Maschinen, einem Windows 8 – Rechner, sowie einem MacOS-Server [Cod141]. Je nach Bedarf lässt sich diese Anzahl erhöhen, damit Skaleneffekte erzielt werden können.

Wird innerhalb der Entwicklungsumgebung das CN1-Projekt mit rechts angeklickt, erscheint das in Abbildung 36 illustrierte Menü, in welchem die gewünschte Zielplattform ausgewählt wird. Nach der Auswahl wird der Java-Quelltext lokal zu Java-Bytecode kompiliert, welcher anschließend in die Cloud geschickt wird. Je nach Zielplattform findet in der Cloud ein unterschiedlicher Prozess statt, welcher in Abb. 35 dargestellt ist.

Für Android findet keine Übersetzung des Codes statt, da Java eine native Sprache unter Android ist. Mit Hilfe des Android SDKs und des dx Cross-Compilers entsteht ein natives Android Package, welches im Anschluss vom Cloud Server heruntergeladen werden kann. Lokal lässt sich in den Eigenschaften des Projekts ein Keystore anlegen, wodurch eine signierte Anwendung kompiliert wird. Ohne Hinterlegung eines Keystores wird ein Debug-Package generiert.

Für Blackberry und J2ME wird die Java 5 Syntax auf Bytecode-Level zu JDK 1.3 cldc transformiert, wodurch die plattformübergreifende Kompatibilität zu den nativen Schnittstellen entsteht. Die für Blackberry benötigten Signaturen können ebenfalls in den Projekteigenschaften hinterlegt werden.

Für Windows Phone, iOS und MacOS wird der Java-Bytecode mittels *XMLVM* (siehe Kapitel 4.2.2) zu C# für Windows Phone respektive zu C bzw. Objective-C für iOS und MacOS transformiert. Aus dem nativen Code wird anschließend im jeweiligem Software Development Kit die installierbare Anwendung kompiliert. Für iOS müssen dafür zuvor die Developer-Zertifikate in den Projekteigenschaften eingetragen worden sein.

Für Windows findet keine Transformation statt. Es wird ein Setup erstellt, welches die Anwendung und die Java-Laufzeitumgebung (JRE) enthält, so dass der Java-Code auf der Java Virtuellen Maschine (JVM) ausgeführt wird.

All diese Prozesse finden automatisiert nach einem Klick auf den Build-Button in der Cloud statt. Ein Entwickler muss aus diesem Grund kein Know-How bzgl. der zugrunde liegenden Techniken besitzen. Ohne spezielle Nachforschungen wird dieser nicht wissen, dass z.B. *XMLVM* für die Generierung des nativen Codes eingesetzt wird. Die Code-Generierung in der Cloud bringt zudem den Vorteil mit, dass für eine iOS- oder MacOS-Entwicklung kein lokales MacOS-Gerät nötig ist. Dieser Umstand macht *Codename One* zum heutigen Zeitpunkt einzigartig. Nutzt man den in der Enterprise-Version angebotenen Offline-Build Support, muss eine iOS- oder MacOS-Entwicklung weiterhin auf einem MacOS-Gerät stattfinden.

6.3.4 Beispielanwendung

Um den Entwicklungsablauf einer mobilen Anwendung in *Codename One* nachvollziehen zu können, soll in diesem Abschnitt eine einfache ToDo-Liste mit folgenden Features implementiert werden:

- Erstellen, Editieren, Speichern und Löschen von ToDo's
- Möglichkeit ToDo's als fertig zu markieren
- Darstellung von ToDo's in einer Liste

Eine Benutzeroberfläche lässt sich sowohl programmatisch als auch mit dem *Codename One Designer* erstellen. Der Designer ist auf Grund der Einfachheit und der bereits im Punkt 6.3.1 („*Voraussetzungen und Möglichkeiten*“) beschriebenen Feature-Vielfalt zu bevorzugen.

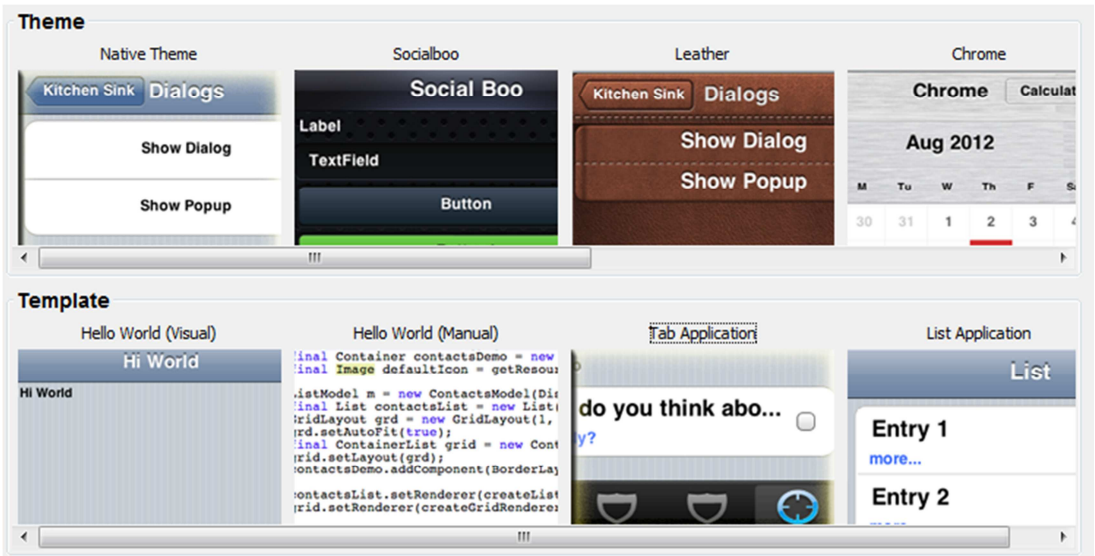


Abbildung 37: CN1 Theme und Template Wahl

Bei der Erstellung eines Projektes kann bereits ein Theme und ein Template ausgewählt werden. Dieser Auswahlprozess ist in Abb. 37 illustriert. Zu beachten ist hierbei, dass für die Entwicklung Netbeans eingesetzt wird und die Abbildung in Eclipse und IntelliJ IDEA abweichen kann, jedoch funktional kein Unterschied besteht. Für die ToDo-App werden das native Theme und die Tab-Application ausgewählt, wodurch eine Dummy-Benutzeroberfläche generiert wird.

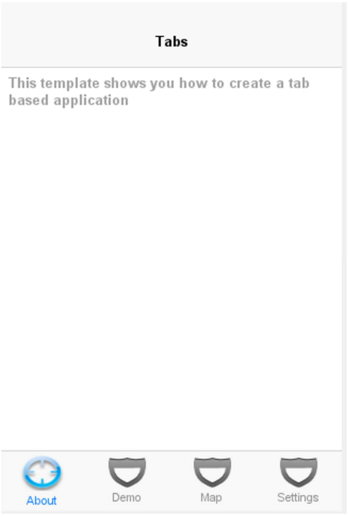


Abbildung 38: CN1 iOS 7 Native Theme

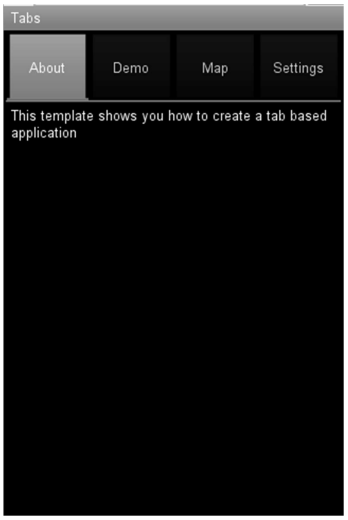


Abbildung 39: CN1 Android Native Theme

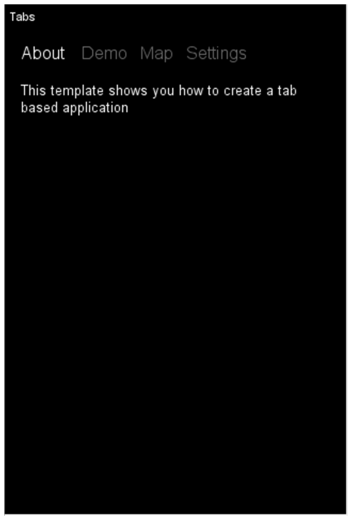


Abbildung 40: CN1 WP Native Theme

Erwähnenswert ist an dieser Stelle, dass die generierte Benutzeroberfläche je nach Zielplattform nativ aussehen und unterschiedlich ausfallen kann. Die nativen Oberflächen von iOS, Android und Windows Phone sind in den Abb. 38 – 40 dargestellt. Für iOS ist auf der Abbildung das native Theme für iOS 7 zu sehen. Auf iOS 6 sähe die Oberfläche ein wenig anders aus, da nicht nur zwischen den unterschiedlichen Plattformen eine native UI generiert wird, sondern auch unter den unterschiedlichen Versionen einer einzigen Plattform, weshalb eine in *Codename One* entwickelte App sich stets in das bestehende Ökosystem einfügt.

Die Dummy-Benutzeroberfläche muss anschließend an die eigenen Bedürfnisse angepasst werden, wozu unnötige UI Elemente im GUI Builder gelöscht und benötigte per Drag & Drop hinzugefügt werden. Jedes UI Element hat bestimmte Eigenschaften, u.a. *Enabled*, *Focusable*, *Visibile*, *Scrollable* und viele mehr, wodurch sie ganz individuell angepasst werden können. Auch die Eigenschaft *Name* kann modifiziert werden, damit sich das entsprechende UI Element innerhalb der Zustandsmaschine eindeutig identifizieren lässt. Die Main-Form wurde so angepasst, dass es statt

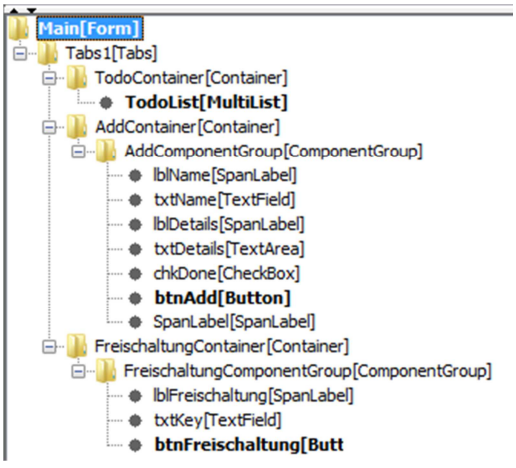


Abbildung 41: CN1 ToDo App Main-Form UI Elemente

der vier generierten Tabs nur noch drei gibt (vgl. Abb. 41). Im ersten Tab werden die ToDo's in einer MultiList ausgegeben. Eine MultiList ist eine Liste, in der ein Eintrag aus mehreren Spalten und Zeilen bestehen kann. Für jede Liste muss ein Modell erstellt werden, damit das Framework interpretieren kann, wie die Liste gerendert werden muss. In der ToDo-App wird in der ersten Zeile der Name des ToDo's und in der zweiten Zeile die dazugehörigen Details ausgegeben. Zusätzlich wird durch ein Image illustriert, ob das ToDo fertiggestellt wurde oder nicht. Im zweiten Tab können ToDo's hinzugefügt werden. Das dritte Tab ist eine Vorbereitung für Kapitel 6.3.6 („*Reverse Engineering*") und implementiert eine einfache Freischaltung der App. Um ToDo's bearbeiten zu können, wird eine weitere Form erstellt, auf der ein User durch einen Klick auf ein ToDo gelangen soll.

Der bisherige Prozess, d.h. die Erstellung der Benutzeroberfläche, fand ausschließlich im *Codename One Designer* statt. Da für diesen keine Programmierkenntnisse nötig sind, könnte diese Arbeit ebenso von einem Designer übernommen werden. *Codename One* erstellt eine *theme.res*, die anschließend in ein vorhandenes Projekt importiert werden kann, wodurch von einem Programmierer Funktionalitäten hinzugefügt werden können. Die *theme.res* kann allerdings nicht nur eine starre Benutzerfläche abbilden, sondern auch den Workflow innerhalb einer App. Im Falle der hier beschriebenen ToDo-App wurde der Workflow bereits bei Erstellung des Templates mitgeneriert, weshalb mit einem Klick auf einem Tab oder durch Rechts/Links-Wischen die Darstellung wie gewünscht geändert wird. Für die Form, auf welcher ein ToDo bearbeitet werden kann, wurde ein „Back-Command" hinzugefügt, der den Workflow bei Betätigung des Back-Buttons definiert. Dabei passt das Framework automatisch das plattformspezifische Design an, wodurch z.B. für iOS links oben innerhalb der Form ein Back-Button generiert wird, während für Android kein Button in der Form zu sehen ist. Stattdessen ist die Funktionalität bei Android-Geräten auf den Sensortasten bzw. bei älteren Devices auf den Hardware-Buttons.

Das Framework generiert aus besagter *theme.res* eine Zustandsmaschine in der Klasse *StateMachineBase*, in welcher die UI-Elemente und deren Events definiert sind. In Abb. 41 sind drei fettgedruckte UI-Elemente sichtbar. Die fettgedruckte Schriftart ist ein Indiz dafür, dass ein Event über den Designer hinzugefügt wurde, beispielsweise ein ActionEvent. Um ein solches Event zu implementieren, muss die entsprechende Methode im User-Space in der Klasse *StateMachine.java* überschrieben werden.

Zunächst muss jedoch die Klasse ToDo mit entsprechenden Gettern und Settern implementiert werden. Die Klasse ist als UML-Klassendiagramm in Abbildung 42 illustriert. Auf eine genaue Erklärung der Implementierung soll an dieser Stelle verzichtet werden, da die Klasse keine spezifi-

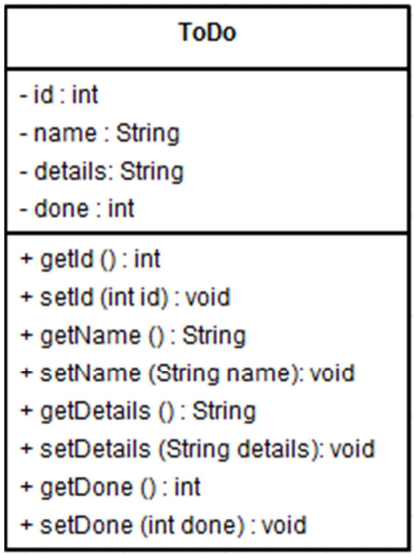


Abbildung 42: CN1 ToDo Klasse (UML)

schen Anpassungen für *Codename One* benötigt und daher mit gewöhnlichem Java-Wissen umgesetzt werden kann.

Um die ToDo's persistent speichern zu können, werden sie in einer SQLite-Datenbank abgelegt. Korrekterweise müsste das Attribut *done* als boolean modelliert werden, da es den Zustand eines ToDo's, d.h. ob dieser offen oder bereits bearbeitet ist, repräsentiert. SQLite kennt jedoch keine booleschen Werte, weshalb der Zustand als ein Integer mit 0 (= ToDo ist offen) oder 1 (=ToDo ist bereits bearbeitet) abgebildet wird.

Die Klasse *DatabaseControl* setzt das Singleton-Entwurfsmuster um, wodurch sichergestellt wird, dass in der Datenbank keine Inkonsistenzen entstehen. Die Klasse implementiert alle Methoden, die eine Operation auf der Datenbank ausführen (vgl. Abb. 43). In *Codename One* müssen dafür die folgenden Klassen importiert werden:

- com.codename1.db.Database
- com.codename1.db.Row
- com.codename1.db.Cursor

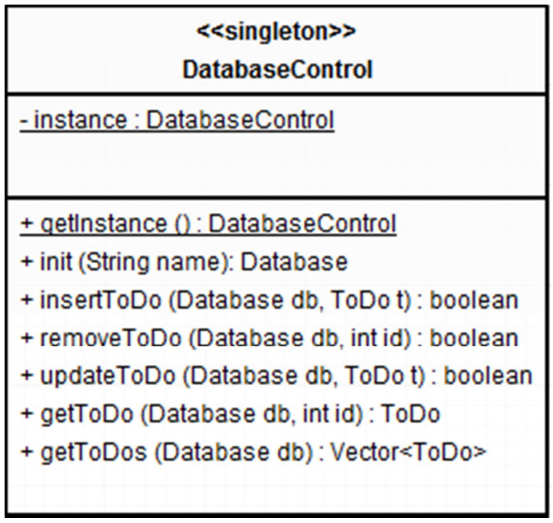


Abbildung 43: CN1 DatabaseControl Klasse (UML)

Die zwei letztgenannten Klassen werden für die Get-Methoden benötigt, damit das Ergebnis einer SQL-Abfrage gelesen werden kann.

Die *Init*-Methode prüft, ob eine Datenbank bereits im Dateisystem liegt und öffnet diese im Erfolgsfall. Falls keine Datenbank vorhanden ist, wird diese neu erstellt:

```
public Database init(String name) throws IOException {
   Database db;
   if (Database.exists(name)) {
      db = Database.openOrCreate(name);
      return db;
   } else {
      db = Database.openOrCreate(name);
      try {
         db.beginTransaction();
         db.execute("CREATE TABLE TODO"
               + "(ID INTEGER PRIMARY KEY AUTOINCREMENT,"
```

```
                + "NAME TEXT,"
                + "DETAILS TEXT,"
                + "DONE INTEGER)");
            db.commitTransaction();
            return db;
        } catch (IOException ex) {
            Log.p(ex.getMessage(), Log.ERROR);
            try {
                db.rollbackTransaction();
                return null;
            } catch (IOException ex1) {
                Log.p(ex1.getMessage(), Log.ERROR);
                return null;
            }
        }
    }
}
```

Die Init-Methode wird nur einmalig beim Start der Applikation aufgerufen. Sollte während der Erstellung der Datenbank ein Fehler auftreten, wird eine Exception geworfen und die Operation zurückgerollt. Die im Code-Ausschnitt zu sehende Log-Methode wird von *Codename One* bereitgestellt und bietet unterschiedliche Log-Levels an, wodurch einfache Fehler relativ schnell zu debuggen sind.

Stellvertretend für alle Get- und Set-Methoden soll im Folgenden die Methode *getToDos* erläutert werden, welche alle in der Datenbank befindlichen ToDo's ausliest:

```
public Vector<ToDo> getToDos(Database db) {
    try {
        Vector<ToDo> hsVec = new Vector<ToDo>();
        db.beginTransaction();
        Cursor cs = db.executeQuery("SELECT * FROM TODO");
        while (cs.next()) {
            ToDo t = new ToDo();
            Row r = cs.getRow();
            t.setId(r.getInteger(cs.getColumnIndex("ID")));
            t.setName(r.getString(cs.getColumnIndex("NAME")));
            t.setDetails(r.getString(cs.getColumnIndex("DETAILS")));
            t.setDone(r.getInteger(cs.getColumnIndex("DONE")));
            hsVec.addElement(t);
        }
        db.commitTransaction();
        cs.close();
        return hsVec;
    } catch (IOException ex) {
        Log.p(ex.getMessage(), Log.ERROR);
    }
    return null;
}
```

Ein Cursor iteriert über alle Ergebnisse der SQL-Abfrage. Dabei wird jeweils ein ToDo instanziiert, welches die aus der Datenbank gelesenen Werte zugewiesen bekommt. Jedes ToDo wird zu einer

Vector-Liste hinzugefügt, welche nach dem Auslesen der Datenbank der aufrufenden Methode zurückgegeben wird. Die anderen Get- und Set-Methoden funktionieren analog, jedoch nutzen sie andere SQL-Queries.

Wenn alle benötigten Klassen implementiert sind, müssen die bereits erwähnten Methoden in der Zustandsmaschine *StateMachineBase* überschrieben werden. *Codename One* verfolgt bei den Methoden eine intuitive Namenskonvention. Als Beispiel sollen drei generierte Methodennamen dienen:

- onMain_BtnAddAction: *Main* heißt die Form und *BtnAdd* ein darauf existierender Button. Wenn sich ein User auf der Main-Form („onMain“) befindet und auf den erwähnten Button klickt, wird eine Action ausgelöst, welche den Code der überschriebenen *onMain_BtnAddAction*-Methode ausführt.
- beforeMain: Wenn diese Methode überschrieben ist, wird der Code ausgeführt, sobald auf die Main-Form gewechselt wird. Analog zum Schlüsselwort *before* existiert auch *post*. *postMain* wird erst nach dem Anzeigen der Form aufgerufen.
- initListModelTodoList: In diesem Fall heißt das UI-Element *TodoList* und repräsentiert eine Liste. Die Methode wird aufgerufen, noch bevor die Liste gerendert wird, damit der Liste ein Modell übergeben werden kann (*initListModel*).

Exemplarisch soll die Methode *beforeMain* vorgestellt werden:

```
@Override
protected void beforeMain(Form f) {
   List list = (List) findByName("TodoList", f);
   Image notdone = fetchResourceFile().getImage("work-in-progress.png");
   Image done = fetchResourceFile().getImage("Done.png");
   Vector<ToDo> todos = DatabaseControl.getInstance().getToDos(db);
   Vector vec = new Vector();
   for (int i = 0; i < todos.size(); i++) {
      Hashtable h = new Hashtable();
      ToDo t = (ToDo) todos.get(i);
      h.put("id", t.getId());
      h.put("name", t.getName());
      h.put("details", t.getDetails());
      if (t.getDone() == 1)
         h.put("icon", done);
      else
         h.put("icon", notdone);
      vec.add(h);
   }
   list.setModel(new DefaultListModel(vec));

   list.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
         List list = (List) evt.getSource();
```

```
            Hashtable h = (Hashtable) list.getSelectedItem();
            selected_id = (Integer) h.get("id");
            showForm("ToDoBearbeiten", null);
        }
    });
}
```

Da am Anfang der Entwicklung die Entscheidung getroffen wurde, die UI mit dem *Codename One Designer* zu erstellen, können die UI-Elemente mittels *findByName* gefunden, zugewiesen und genutzt werden. Hätte man eine andere Entscheidung getroffen, müsste man an dieser Stelle die UI-Elemente programmatisch erzeugen. In diesem Fall wird für diese View nur eine List benötigt, in der die ToDo's dargestellt werden sollen. Mit Hilfe von *fetchResourceFile* können ebenso die im Designer hinzugefügten Ressourcen, in diesem Fall zwei Bilder, die den Status der ToDo's illustrieren sollen, gelesen werden. Nach dem Abrufen der ToDo's aus der Datenbank, befinden sich diese – wie bereits erläutert – in einer Vector-Liste, über die iteriert wird. Das Listenmodell der ToDo-Liste wird durch Key/Value-Paare definiert, weshalb während der Iteration jeweils eine Hashtable instanziiert wird. Das Modell ist prinzipiell von folgenden Schlüsseln abhängig:

- name
- details
- icon

Die Groß- und Kleinschreibung muss dabei beachtet werden, so dass die Schlüsselnamen identisch zu denen im Modell sind, welches im *Codename One Designer* erstellt wurde. Die Zuweisung des Icons erfolgt je nach Status des ToDo's, welcher im Attribut *done* repräsentiert wird. Jede befüllte Hashtable wird einem zuvor deklarierten Vector hinzugefügt, welcher nach der Iteration der Liste übergeben wird, die den Inhalt letztendlich wie im Modell vorgegeben darstellt. Die *id* wird für das Listenmodell und für die bloße Darstellung nicht benötigt, jedoch wird der Liste ein ActionListener angehangen, in welchem ein angeklicktes ToDo identifiziert werden muss. Daher wird die Hashtable während der Iteration ebenso mit dem Schlüssel *id* befüllt. Im ActionListener wird besagte *id* ausgelesen und der globalen Variable *selected_id* zugewiesen, mit der stets die zuletzt ausgewählte *id* getracked wird. Im Anschluss wird die Form gewechselt, um das ToDo bearbeiten zu können.

Die anderen überschriebenen Methoden sollen an dieser Stelle nicht weiter erläutert werden, da sie zum einen ähnlich umgesetzt sind und zum anderen keine neuen Erkenntnisse zum Funktionsverständnis des Frameworks liefern.

Die fertige ToDo-Anwendung kann für jede Plattform, die *Codename One* unterstützt, nativ kompiliert werden. Die folgenden Abbildungen zeigen sie für Android, Windows Phone, iOS und Blackberry OS:

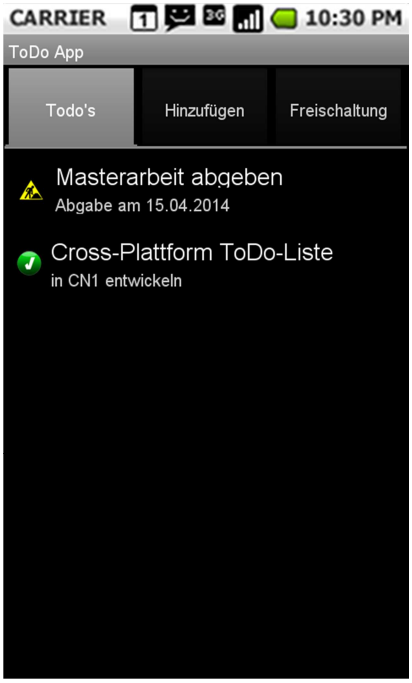


Abbildung 44: CN1 ToDo App Android

Abbildung 45: CN1 ToDo App Windows Phone

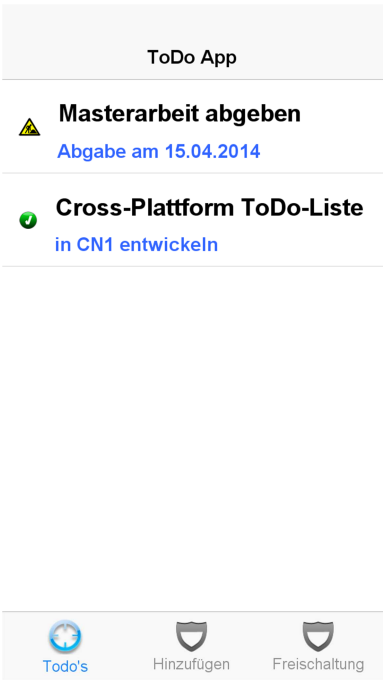


Abbildung 46: CN1 ToDo App iOS

Abbildung 47: CN1 ToDo App Blackberry OS

6.3.5 Debugging

Eine in *Codename One* entwickelte Anwendung kann in allen drei unterstützten Entwicklungsumgebungen (siehe Kapitel 6.3.3) debugged werden. Die einfachste Form des Debuggings ist die Verwendung der Klasse *com.codename1.io.Log*, welche die folgenden Log-Levels unterstützt:

- DEBUG
- INFO
- WARNING
- ERROR

Das Level *DEBUG* ist dabei das niedrigste, *ERROR* das höchste Level. Diese Art des Debugging ist für viele Fehler hilfreich, weswegen besagte Klasse in jedem Entwicklungsprojekt implementiert werden sollte. Die Debug-Ausgaben werden nicht nur auf der Konsole, sondern z.B. bei Android-Anwendungen ebenso im Dalvik Debug Monitor Server (DDMS) ausgegeben.

Ab der Pro-Lizenz lassen sich zusätzlich die folgenden Level nutzen:

- REPORTING_DEBUG
- REPORTING_PRODUCTION

Damit können die Debug-Ausgaben von einem Gerät in einem bestimmten Intervall in die Cloud geschickt werden. Das Feature lässt sie wie folgt nutzen:

```
Log.setReportingLevel(Log.REPORTING_DEBUG);
DefaultCrashReporter.init(true, 2);
```

In diesem Beispiel werden die Debug-Ausgaben alle zwei Minuten in die Cloud geschickt. Während *REPORTING_DEBUG* alle Meldungen in die Cloud sendet, meldet *REPORTING_PRODUCTION* nur aufgetretene Fehler.

Ebenfalls ist ab der Pro-Version der Build-Parameter *crash_protection: true* nutzbar. Dieser bewirkt, dass bei einem Absturz der App eine Email mit dem Stack-Trace an den Entwickler gesendet wird. Der Parameter sollte aber nur während der Entwicklung genutzt werden, da er die Performanz signifikant verschlechtert.

Für ein lokales Debugging sind die wichtigsten Tools die verwendete Entwicklungsumgebung und der *Codename One Simulator*. Abb. 48 zeigt beispielhaft einen Breakpoint innerhalb der vorgestellten ToDo-App. Es sind alle Objekte und deren Eigenschaften sichtbar, weshalb diese Art des Debuggings besonders für komplexe Anwendungen und Fehler geeignet ist, die nicht auf den ersten Blick ersichtlich sind.

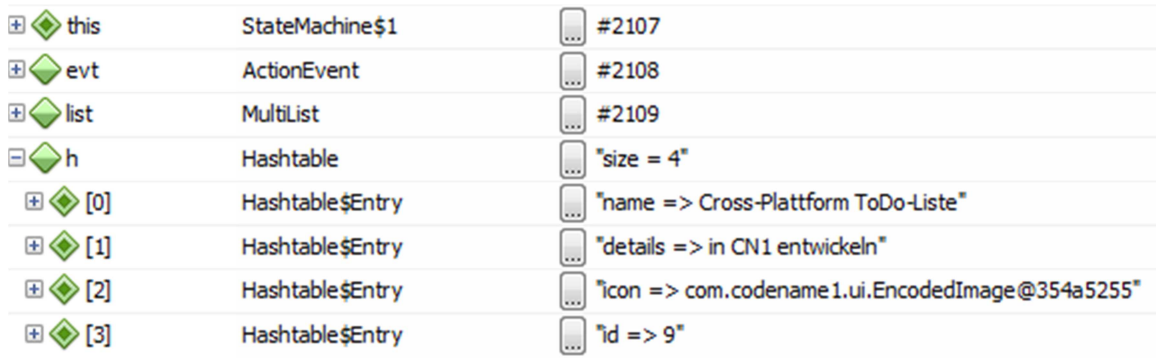


Abbildung 48: CN1 Netbeans Breakpoint

Darüber hinaus lässt sich mit dem im Simulator integrierten *Performance Monitor* die Performanz monitoren und debuggen:

Component Details (times are in nano-seconds: one billionth of a second)

Name	Type	UIID	Parent N...	Icon	Invocations	Fastest	Slowest	Average
	com.cod...	Tab			11	112584	1967010	561473
	com.cod...	Tab			11	167340	1602159	534171
	com.cod...	Tab			8	167060	462629	318162
	com.cod...	TabsCon...	Tabs1		8	567949	4441905	2223362
	com.cod...	Title			4	46654	131022	99035
	com.cod...	TitleArea	Main		4	134095	342502	274546
TodoContain...	com.cod...	Container			4	5588	2245817	829924
	com.cod...	icon	lblName		2	4749	6146	5447
	com.cod...	Label	lblName		2	90514	105321	97917

Abbildung 49: CN1 Performance Monitor

Mit Hilfe des *Performance Monitors* lassen sich daher schnell UI-Elemente identifizieren, die einen negativen Einfluss auf die Performanz haben.

6.3.6 Reverse Engineering

Je nach Zielplattform wird ein unterschiedlicher Build-Prozess angewandt (siehe Kapitel 6.3.3: „*Entwicklungsumgebung und Build-Prozess*“). Bei den Plattformen, bei denen *XMLVM* für die Code-Generierung zum Einsatz kommt, entsteht zwar ein effizienter Code, jedoch ist dieser für einen Menschen schwer lesbar (vgl. Kapitel 4.2.2: „*XMLVM*“). Für einen potenziellen Angreifer sind daher eher die Plattformen interessant, auf denen der Java-Code ausgeführt wird, da dieser relativ leicht zu dekompilieren ist. Für Blackberry OS kann in den Eigenschaften eines *Codename One* Projekts ein Obfuscating aktiviert werden. Für Android existiert solch eine Option nicht, allerdings kann zwischen Debug- und Releaseversion unterschieden werden. Bei Letzterem findet automatisch ein Obfuscating statt, sofern in den Projekteigenschaften ein Keystore hinterlegt ist. Aus diesem Grund sollte bei einer Veröffentlichung der App stets die Releaseversion verwendet werden. Um den Unterschied zwischen Debug- und Releaseversion zu verdeutlichen, wird das Reengineering der Android-Version von der im Punkt 6.3.4 entwickelten ToDo-App beschrieben.

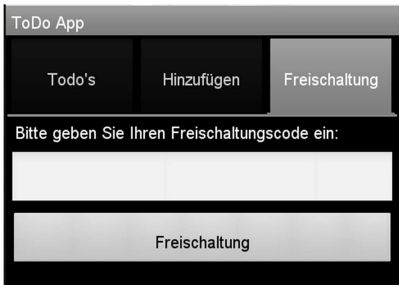


Abbildung 50: ToDo App Freischaltung

Dafür wurde in der ToDo-App ein Tab namens „Freischaltung“ implementiert (siehe Abb. 50). Sobald der Nutzer eine korrekte Seriennummer einträgt und auf den Freischalt-Button klickt, wird eine Erfolgsmeldung ausgegeben – anderenfalls eine Fehlermeldung, die darauf aufmerksam macht, dass der eingegebene Freischaltcode nicht korrekt ist.

Für die Prüfung des Freischaltcodes wurde eine Klasse namens *RegCheck* implementiert:

```
public class RegCheck {

   private char firstChar = 'M';
   private int secretA = 1337;
   private int secretB = 5;

   public boolean checkSerial(String serial) {
      if (serial.length() < 5 || serial.charAt(0) != firstChar)
         return false;
      int key = tryParse(serial.substring(1, 5));
      return ((key%secretA) == secretB);
   }

   private static Integer tryParse(String text) {
      try {
         return Integer.parseInt(text);
      } catch (NumberFormatException e) {
         return 0;
      }
   }
}
```

Bei Betätigung des Freischalt-Buttons wird die Methode *checkSerial* aufgerufen, wobei die Variable *serial* den im Textfeld eingegeben String enthält. Der Freischaltcode muss aus mindestens 5 Zeichen bestehen, die sich wie folgt ergeben:

- Das erste Zeichen ist immer ein „M“
- Die Zeichen an der Stelle 2,3,4 und 5 werden als ein Integer geparsed. Anschließend wird der Modulo zwischen dem Wert und dem *secretA* (=1337) errechnet. Entspricht das Ergebnis der Modulo-Operation dem *secretB* (=5), ist die Eingabe korrekt.

Eine gültige Eingabe wäre demnach z.B. M1342.

Das kompilierte Android-Package (APK) der ToDo-Anwendung hat den gleichen Aufbau wie eine beliebige native Android-Anwendung, weshalb sich für Android vorhandene Reengineering-Tools verwenden lassen. Das Angriffsziel ist die in der APK-Datei befindliche *classes.dex*, welche vom dx-Cross-Compiler – ein Bestandteil des Android SDKs – erstellt wurde und die alle Java-Klassen in

Form von Dalvik-Bytecode enthält. Für Android existieren prinzipiell zwei unterschiedliche Möglichkeiten des Reverse Engineerings:

- Mit dem frei erhältlichen *APK-Tool* [Wiś11] ist es möglich, eine APK-Datei und insbesondere die darin gepackte DEX-Datei, die den in Java geschriebenen Code einer Anwendung enthält, zu dekodieren. Das Tool rekonstruiert alle Klassen-, Methoden- und Variablennamen. Der Quelltext liegt im Dalvik-Bytecode vor, welcher mit Hilfe der Dalvik-Opcodes lesbar ist.
- Die zweite Möglichkeit nimmt den dx-Cross-Compiler zum Vorbild, welcher Java-Bytecode zu Dalvik-Bytecode kompiliert, und setzt einen Cross-Compiler in die andere Richtung um, wodurch man wieder Java-Bytecode erhält. Der Vorteil dieser Methode ist, dass auf den Java-Bytecode handelsübliche Java-Decompiler angewendet werden können.

Im Folgenden soll die zweite Methode für die Debugversion der ToDo-App verwendet werden, wozu der Cross-Compiler *dex2jar* [Pan14] zum Einsatz kommt. Dafür wird die *classex.dex* der ToDo-App aus dem Archiv extrahiert und *dex2jar* aufgerufen:

```
$ d2j-dex2jar classes.dex
dex2jar classes.dex -> classes-dex2jar.jar
```

Das Tool generiert eine Datei namens *classes-dex2jar.jar*, welche anschließend mit einem Java-Decompiler geöffnet werden kann. In diesem Fall wird *JD-GUI* [Dup14] verwendet. Die dekompilierte Methode checkSerial:

```
public boolean checkSerial(String paramString) {
  int i = 1;
  if ((paramString.length() < 5) || (paramString.charAt(0) != this.firstChar)) {
    i = 0;
  }
  while (tryParse(paramString.substring(i, 5)).intValue() % this.secretA == this.secretB) {
    return i;
  }
  return false;
}
```

Die Methode entspricht zwar nicht vollständig dem Original, dennoch kann man die Funktionsweise nachvollziehen. Die Klasse, die Methoden und die globalen Variablen sind mit Klarnamen bekannt, wodurch sich Rückschlüsse ziehen lassen und der Freischaltcode leicht zu errechnen ist.

Für einen besseren Schutz vor Angriffen sollte daher nur die Releaseversion veröffentlich werden. Ein Reengineering der Releaseversion ergibt:

```
public class c
{
  private char a = 'M';
  private int b = 1337;
  private int c = 5;

  public static Integer b(String paramString)
  {
    try
    {
      Integer localInteger = Integer.valueOf(Integer.parseInt(paramString));
      return localInteger;
    }
    catch (NumberFormatException localNumberFormatException) {}
    return Integer.valueOf(0);
  }

  public boolean a(String paramString)
  {
    int i = 1;
    if ((paramString.length() < 5) || (paramString.charAt(0) != this.a)) {
      i = 0;
    }
    while (b(paramString.substring(i, 5)).intValue() % this.b == this.c) {
      return i;
    }
    return false;
  }
}
```

Die Klassen-, Methoden- und Variablennamen besitzen in der Releaseversion bedeutungslose Namen, wodurch ein Verständnis unberechtigter Dritte vor allem in komplexen Anwendungen deutlich erschwert wird. Da es jedoch ebenso in Releaseversionen zu Abstürzen kommen kann, muss der Entwickler die Strack-Traces verstehen. Aus diesem Grund generiert der *CN1*-Build-Server zu jeder Releaseversion eine *mapping.txt*, in welcher ein Entwickler die ursprünglichen Namen einsehen kann. Das Mapping der Klasse *RegCheck* sieht beispielsweise wie folgt aus:

```
de.mahlert.cn1_todo_liste.RegCheck -> de.mahlert.cn1_todo_liste.c:
    char firstChar -> a
    int secretA -> b
    int secretB -> c
    20:23:boolean checkSerial(java.lang.String) -> a
    28:30:java.lang.Integer tryParse(java.lang.String) -> b
```

6.3.7 Bewertung

In diesem Kapitel wird das Framework anhand der im Punkt 5.3 vorgestellten Methodologie bewertet. Die Bewertung soll pro Entscheidungskriterium kurz begründet werden.

Funktionalität

Codename One unterstützt mit Android, iOS und Windows Phone alle geforderten mobilen Plattformen, wobei sich der Windows Phone Support noch im Beta-Stadium befindet. Darüber hinaus werden Blackberry OS und J2ME-basierte Geräte unterstützt. Durch eine weitere Abstraktionsebene kann man auf die nativen Funktionen der jeweiligen Plattformen zugreifen. Dadurch entstehen insofern Einschränkungen, als dass nicht jede native Funktion jeder Plattform in der Abstraktionsschicht abgebildet ist. Solche Unzulänglichkeiten lassen sich durch eigene Libraries umgehen, da jede Funktionalität selbst implementiert werden kann. Da das Framework über eine lange Zeit genutzt werden soll, ist jedoch nicht absehbar, welche Funktionen zukünftig benötigt werden, so dass es schwer abzuschätzen ist, wie viele solcher Anpassungen nötig sind.
Das Framework erfüllt die Anforderungen an die Architektur, da Anwendungen in der Regel eine Model-View-Controller – Architektur umsetzen und Libraries implementiert werden können.

Insgesamt wird *Codename One* in dieser Kategorie mit 3,5 bewertet. Es werden zwar mehr Plattformen als gefordert unterstützt, allerdings erfolgt ein Abzug auf Grund der möglicherweise nötigen Anpassungen, die sich im Vorhinein schwer abschätzen lassen.

Score: 3,5

Benutzbarkeit

Für einen Benutzer ist es nicht ersichtlich, wie eine mobile Anwendung entwickelt wurde. Das Look & Feel entspricht dem einer nativen App. Je nach Theme wird nicht nur zwischen den verschiedenen Plattformen eine native UI generiert, sondern auch unter den unterschiedlichen Versionen einer einzigen Plattform.

Aus Entwicklersicht fällt eine Einarbeitung in *Codename One* sehr leicht, da eine Wahl zwischen den drei sehr verbreiteten Entwicklungsumgebungen Netbeans, Eclipse und IntelliJ IDEA besteht. Es ist jedoch darauf zu achten, dass die jeweiligen Plug-Ins nicht komplett die in *CN1* verfügbaren Features abdecken. Netbeans wird am besten unterstützt und ist daher zu empfehlen.
Der Build-Prozess findet hauptsächlich innerhalb einer Cloud statt, wodurch die Entwicklung auf einem beliebigen Gerät möglich ist. Daher kann z.B. eine iOS-App auf einem Windows- oder Linux-Rechner implementiert werden. Die zugrunde liegenden Techniken wie *XMLVM* (siehe Kapitel 4.2.2) sind für einen Entwickler nicht ersichtlich, so dass diesbezüglich kein Know-How erforderlich ist.

Die Benutzbarkeit des Frameworks wird mit 4,5 bewertet, da sie die Erwartungen übertrifft. Code Name ist Stand März 2014 das einzige Framework, welches für eine native iOS-Entwicklung kein MacOS-System und für eine native Windows Phone-Entwicklung kein Windows-System benötigt.

Score: 4,5

Zuverlässigkeit und Effizienz

Das Netbeans Plug-In war das erste aller Plug-Ins und wird bis zum heutigen Tage am besten supported, so dass es alle *CN1*-Features abdeckt und als stabil zu bezeichnen ist, wenngleich es während einer dreimonatigen Nutzungszeit zu zwei Abstürzen im Designer kam, die jedoch keinen wesentlichen Einfluss auf die Entwicklung nahmen.

Aus Benutzersicht wurde u.a. geprüft, wie effizient eine Liste mit 10000 Elementen geladen wird. Es wurden weder Ruckler, noch Ladezeiten wahrgenommen, weshalb die Erwartungen aus Benutzersicht erfüllt wurden.
Bei der Evaluierung von *XMLVM* wurde festgestellt, dass eine generierte iOS-Anwendung schneller als eine solche sein kann, die eigenhändig in Objective-C entwickelt wurde (vgl. Kapitel 4.2.2). Da *Codename One* auf *XMLVM* aufsetzt, profitiert es von diesem Effekt.

Score: 4

Wartbarkeit und Support

Die Dokumentation auf der Webseite ist klar und verständlich. Es gibt eine vollständige API-Dokumentation und Video-Tutorials für Einsteiger und Fortgeschrittene. Da das Framework Open-Source ist, kann die Entwicklung transparent eingesehen werden. Die Tickets und Feature-Requests werden je nach Lizenz priorisiert. Weiterhin existiert durch den Open-Source-Gedanken eine aktive Community, welche z.B. Erweiterungen in Form von Libraries anbietet.

Die Lizenzen pro Entwickler lassen sich monatlich aufstocken oder kündigen, weshalb eine flexible Lizenzierung je nach Auslastung möglich ist. Weiterhin kann ein privater Cloud-Server lizensiert werden, auf den externe Entwickler ohne spezielle Lizenz zugreifen können.

Eine spezifische Anforderung seitens *TK* ist, dass die in der Benutzeroberfläche sichtbaren Texte (z.B. Labels) in einer App exportierbar sein sollen, damit diese vom Produktmanagement gewartet werden können. Eine solche Funktionalität implementiert der *Codename One Designer*, in welchem die Strings wahlweise im XML- oder CSV-Format exportierbar sind.

Insgesamt werden die Wartbarkeit und der Support mit 4,5 bewertet. Der Support deckt alle gewünschten Features ab. Mit Enterprise-Lizenz lagen die Antwortzeiten nach eigener Erfahrung bei etwa zehn Minuten, weshalb eine Entwicklung nicht ins Stocken geriet.
Darüber hinaus wird ein Escrow-Service angeboten, wodurch ein Lizenznehmer auch im Falle einer Insolvenz von *Codename One* abgesichert ist und weiter mit dem Framework planen und arbeiten kann.

Score: 4,5

Kompromissbereitschaft

Die Kompromissbereitschaft soll den Zusammenhang zwischen dem Portierbarkeitsgrad und den damit verbundenen funktionalen Einbußen bewerten. Eine niedrige Bewertung dieses Faktors bedeutet, dass mit sehr vielen Kompromissen zu rechnen ist. Im Falle von *Codename One* können 100% des Codes geteilt werden, wobei jedoch, wie bereits im Punkt *Funktionalität* erwähnt, Limitierungen insofern auftreten können, als dass die Abstraktionsschicht des Frameworks nicht jede plattformspezifische Funktion bis ins Detail implementiert. Zwar lassen sich diese Funktionen selbst implementieren oder je nach Lizenz requesten, dennoch erfolgt ein Abzug, da das Framework im Urzustand nicht alle Funktionen abbildet.

Score: 3

Gewichtung

Die Bewertungen der einzelnen Kategorien im Überblick:

Entscheidungskriterium	Bewertung	Gewichtung
Funktionalität	3,5	4
Benutzbarkeit	4,5	4
Zuverlässigkeit und Effizienz	4	4
Wartbarkeit und Support	4,5	5
Kompromissbereitschaft	3	3

Der gemittelte Score beträgt demnach **3,975**.

6.4 Vergleich und Wahl des Frameworks

Sowohl *Codename One* als auch *Xamarin* konnten überzeugen. Beide vorgestellten Frameworks verfolgen das Ziel einer nativen Cross-Plattform-Entwicklung, gehen dabei aber unterschiedliche Wege. Im Folgenden werden noch einmal die wichtigsten Für und Wider jeder Plattform abgewogen.

Die wichtigsten Gründe, die für eine Wahl von *Xamarin* sprechen:

- Es sind kaum Kompromisse auf Kosten der Portierbarkeit einzugehen.
- Es lassen sich via C#-Bindings direkt die Schnittstellen der jeweiligen Plattformen aufrufen.
- Im Schnitt lassen sich etwa 60% des Codes teilen.
- Die Anwendungen lassen sich für jede Plattform auf ein am Rechner angeschlossenes Gerät debuggen.
- Für C#-Entwickler ist der Einstieg sehr einfach und mit entsprechender Lizenz in gewohnter Entwicklungsumgebung (Visual Studio) möglich.

Die gravierendsten Gründe, die gegen *Xamarin* sprechen:

- Es bestehen Unsicherheiten im Falle einer Insolvenz der Firma Xamarin Inc. In keiner Lizenz ist festgehalten, was in einem solchen Fall mit dem Framework bzw. dessen Quelltext geschieht.
- Es müssen bis zu 40% des Codes plattformspezifisch entwickelt werden.
- Die Dokumentation auf der Webseite ist teilweise irreführend, so dass z.B. Features erwähnt werden, die offiziell noch nicht existieren.

Die wichtigsten Gründe, die für eine Wahl von *Codename One* sprechen:

- Es lassen sich 100% des Codes teilen.
- Ein sehr guter Support, bei dem selbst Benutzer ohne Lizenz in der Regel eine Antwort von *CN1*-Mitarbeitern im Forum erhalten. Zusätzlich existiert eine Open-Source-Community, die z.B. Libraries bereitstellt.
- In den Service-Level-Agreements ist zusätzlich ein Escrow-Service verankert, bei dem sämtliche interne Quellen, d.h. auch die des Build-Servers, zum Escrow-Provider übermittelt werden, wodurch ein Kunde gegen eine Insolvenz seitens der Firma Codename One abgesichert ist.
- iOS-Anwendungen lassen sich ohne MacOS und Windows Phone-Anwendungen ohne Windows entwickeln.

- Für Java-Entwickler ist der Einstieg sehr einfach und in gewohnten Entwicklungsumgebungen (Netbeans, Eclipse, IntelliJ IDEA) möglich.

Die gravierendsten Gründe, die gegen *Codename One* sprechen:

- Für den hohen Portierungsgrad müssen ein paar funktionale Kompromisse eingegangen werden.
- Ein Debugging findet überwiegend im Simulator statt. Ab der Pro-Lizenz sind jedoch nicht nur die Binary-Dateien, sondern ebenfalls die generierten Quellen downloadbar. Diese könnten zwar direkt auf dem Device debugged werden, allerdings ist es umständlich, da man z.B. für iOS trotzdem ein MacOS-System bräuchte.

Wahl des Frameworks

Die Scores der evaluierten Frameworks im Überblick:

Entscheidungskriterium	Bewertung Codename One	Bewertung Xamarin	Gewichtung
Funktionalität	3,5	3,5	4
Benutzbarkeit	4,5	3,5	4
Zuverlässigkeit und Effizienz	4	4	4
Wartbarkeit und Support	4,5	3	5
Kompromissbereitschaft	3	4,5	3
gemittelter Score	**3,975**	**3,625**	

Auf Grund der vorgenommen Gewichtung wurde sich für *Codename One* entschieden. Diese Gewichtung spiegelt die Herausforderungen und Anforderungen des Unternehmens *TK* wider. Leser, die andere Anforderungen an eine mobile Cross-Plattform-Entwicklung stellen, sollten daher eine eigene Gewichtung vornehmen.

7 Implementierung eines Prototyps

Nachdem durch die Evaluierung der Frameworks die Wahl auf *Codename One* fiel, wurde mit diesem ein Prototyp entwickelt, welcher einen Anwendungsfall der Firma *TK* implementiert.

7.1 Ziel und Auswahl des Prototyps

Codename One erhielt Punktabzüge in den Kategorien Kompromissbereitschaft und Funktionalität. Bei letzterer wurde bemängelt, dass möglicherweise benötige plattformspezifische Anpassungen im Vorhinein schwer abzuschätzen sind. Zwar wurden durch die Evaluierung des Frameworks und durch die Entwicklung kleinerer Beispielanwendungen bereits erste Erfahrungen in *Codename One* gesammelt (siehe Kapitel 6.3.4 „*Beispielanwendung*“), jedoch kann nur mit einem realen Anwendungsfall und durch die tägliche Arbeit in diesem Framework ein Know-How erworben werden, welches für die Einschätzung bzgl. der Machbarkeit späterer Entwicklungsprojekte unabdingbar ist.

Für die Implementierung eines Prototyps standen seitens *TK* drei unterschiedliche Anwendungsfälle zur Verfügung, von denen einer ausgesucht werden musste:

1) eine Anwendung im Bereich Service Assurance
2) ein Programm-Manager
3) ein Hotspotfinder

Im ersten Fall soll eine neue unternehmensinterne Anwendung entstehen, die von Technikern im Außendienst einzusetzen wäre. Die jetzigen Abläufe und Prozesse bei einer Provisionierung oder Entstörung beim Kunden Vorort wurden von *TK* als ineffizient eingestuft. Heute werden beispielsweise Techniker im Außendienst mit einem Notebook ausgestattet, um die nötigen Prozesse im Backend auslösen zu können. So muss bei bestimmten Prozessen z.B. die Media-Access-Control-Adresse (MAC-Adresse) des Kundengerätes abgetippt und eingetragen werden, was zum einen fehleranfällig und zum anderen zeitaufwendig ist. Mit einer mobilen Anwendung könnte man solche Prozesse automatisieren und beschleunigen, in dem die MAC-Adresse von einem Barcode-Scanner gelesen wird und die Informationen automatisch zum Backend übertragen werden.

Der Programm-Manager ist eine bereits existierende webbasierte mobile Anwendung, mit welcher ein Nutzer den Electronic Program Guide (EPG) auf seinem mobilen Endgerät anzeigen und mit jenem remote TV-Aufzeichnungen programmieren kann. Der Programm-Manager liegt in vier unterschiedlichen Versionen vor. Konkret existiert je eine iOS- und Android-Anwendung für Smartphone und Tablet, welche durchweg von den Nutzern in den jeweiligen App Stores negativ

bewertet werden. Diese negativen Rezensionen sind hauptsächlich auf die webbasierte Entwicklung zurückzuführen (vgl. Kapitel 5.1 „*Herausforderungen*“), weshalb die Anwendung nativ neu entwickelt werden soll.

Auch der Hotspotfinder ist eine existierende mobile Anwendung, welche nativ und separat von zwei unterschiedlichen externen Firmen für iOS und Android entwickelt wurde. Die Firma *TK* besitzt über ganz Deutschland verteilt ca. 350.000 WLAN-Hotspots, über die ein Kunde von unterwegs Zugriff auf ein schnelles Internet erhalten kann. Mit Hilfe der App können diese Hotspots gefunden, verwaltet und genutzt werden. Ein in *Codename One* entwickelter Prototyp ist auf Grund mehrerer Punkte interessant:

- Eine native Cross-Plattform-Entwicklung lässt sich direkt mit den schon existierenden nativen Anwendungen in Bezug auf Performanz und nativer Nutzererfahrung vergleichen.
- Innerhalb der App werden die Hotspots auf einer Karte dargestellt. Das Map-Feature ist auf den ersten Blick keine typische Cross-Plattform-Funktion, da es auf jeder Plattform unterschiedlich umgesetzt ist (Google Maps auf Android, Apple Maps auf iOS, Nokia Maps auf Windows Phone, usw.). Aus der Evaluierung weiß man bereits, dass typische UI-Elemente wie eine Liste, Buttons oder Textfelder plattformübergreifend einfach zu erstellen sind und je nach System nativ aussehen. Doch wie genau sieht eine Entwicklung aus, welche UI-Elemente benötigt, die – wie die Karte – sich je nach Plattform komplett voneinander unterscheiden? Falls eine Erweiterung des Frameworks nötig ist, ist der Aufwand der Implementierung ein wichtiger Erfahrungswert für zukünftige Projekte.
- Die App wurde zwar für Android und iOS nativ entwickelt, jedoch hat sie auf beiden Systemen viele Bugs, weshalb Anwender im App Store z.B. über Abstürze klagen. Daher steht sie unabhängig dieser Arbeit für eine neue native Entwicklung zur Ausschreibung. Externe Firmen sollen nach der Einführung der Cross-Plattform möglichst schnell auf dieser entwickeln, damit *TK* die im Kapitel 5 beschriebenen Anforderungen und Herausforderungen meistern kann. Der Autor dieser Arbeit hat die Möglichkeit an den Präsentationen der externen Firmen teilzunehmen und diese auf eine Cross-Plattform-Strategie anzusprechen. Insbesondere wäre deren Reaktionen und Feedback interessant, wenn Ihnen mitgeteilt wird, dass auf einer bestehenden Cross-Plattform mit einem festgelegten Framework entwickelt werden soll.

Die Wahl der Umsetzung fiel aus zwei Gründen auf den Hotspotfinder. Zum einen stellte sich für die ersten beiden Anwendungsfälle heraus, dass für eine native Entwicklung weitere Schnittstellen im Backend vonnöten sind. Im Rahmen dieser Arbeit wäre eine Backend-Entwicklung zu zeitin-

tensiv gewesen und es wäre ungewiss, inwiefern die dann verbliebene Zeit für die Implementierung des eigentlichen Prototyps gereicht hätte. Für den Hotspotfinder hingegen existieren alle benötigten Schnittstellen im Backend. Zum anderen erschien der Hotspotfinder eine steile Lernkurve zu versprechen, da die Kartenansicht aus den schon erwähnten Gründen ein Feature ist, welches prinzipiell nicht für eine native Cross-Plattform-Entwicklung prädestiniert ist. Weiterhin konnte auf Grund der parallelen Ausschreibung der App Feedback externer Entwicklungsfirmen eingeholt werden.

7.2 Projektplan

Nach der Evaluierung der Frameworks wurde das Ergebnis Ende Januar 2014 den Entscheidungsträgern der Technologieabteilung von *TK* vorgestellt. Im Rahmen der Präsentation wurden diese von *Codename One (CN1)* als zukünftige Cross-Plattform überzeugt. Als einen wichtigen Diskussionspunkt ergab sich dabei der Build-Prozess der in *CN1* geschriebenen mobilen Anwendungen (vgl. Kapitel 6.3.3 „*Entwicklungsumgebung und Build-Prozess*“). Die Einwände bezogen sich auf den Build-Prozess in der Cloud und daraus resultierende Bedenken in Bezug auf die Konzernsicherheit. Diese Bedenken wurden jedoch schnell verworfen, als den Entscheidungsträgern die Corporate-Lizenz, d.h. ein privater Build-Server im unternehmensinternen Netz, vorgestellt wurde. Diese spezielle Lizenz existiert erst seit der zweiten Januarwoche 2014 und war neben der eigentlichen Bewertung des Frameworks ein wichtiges Kriterium für *TK*, um sich auf *Codename One* festlegen zu können. Da am Entstehungsprozess einer App bei *TK* allerdings nicht nur die Technologieabteilung, sondern auch andere Business-Einheiten beteiligt sind (vgl. Kapitel 5.1 „*Herausforderungen*“), vergingen weitere zwei Wochen, in denen sämtliche Stakeholder von *Codename One* überzeugt wurden.

Nachdem alle Stakeholder informiert waren, wurde ein Projektplan für den Prototyp ausgearbeitet, der aus vier größeren Arbeitspaketen bestand:

1) Antrag und Bestellung der Plattform
2) Aufbau der Plattform
3) Anforderungsanalyse Hotspotfinder
4) Implementierungsphase Hotspotfinder Prototyp

Die einzelnen Arbeitsschritte und Abhängigkeiten der Arbeitspakete sind in Abbildung 51 dokumentiert.

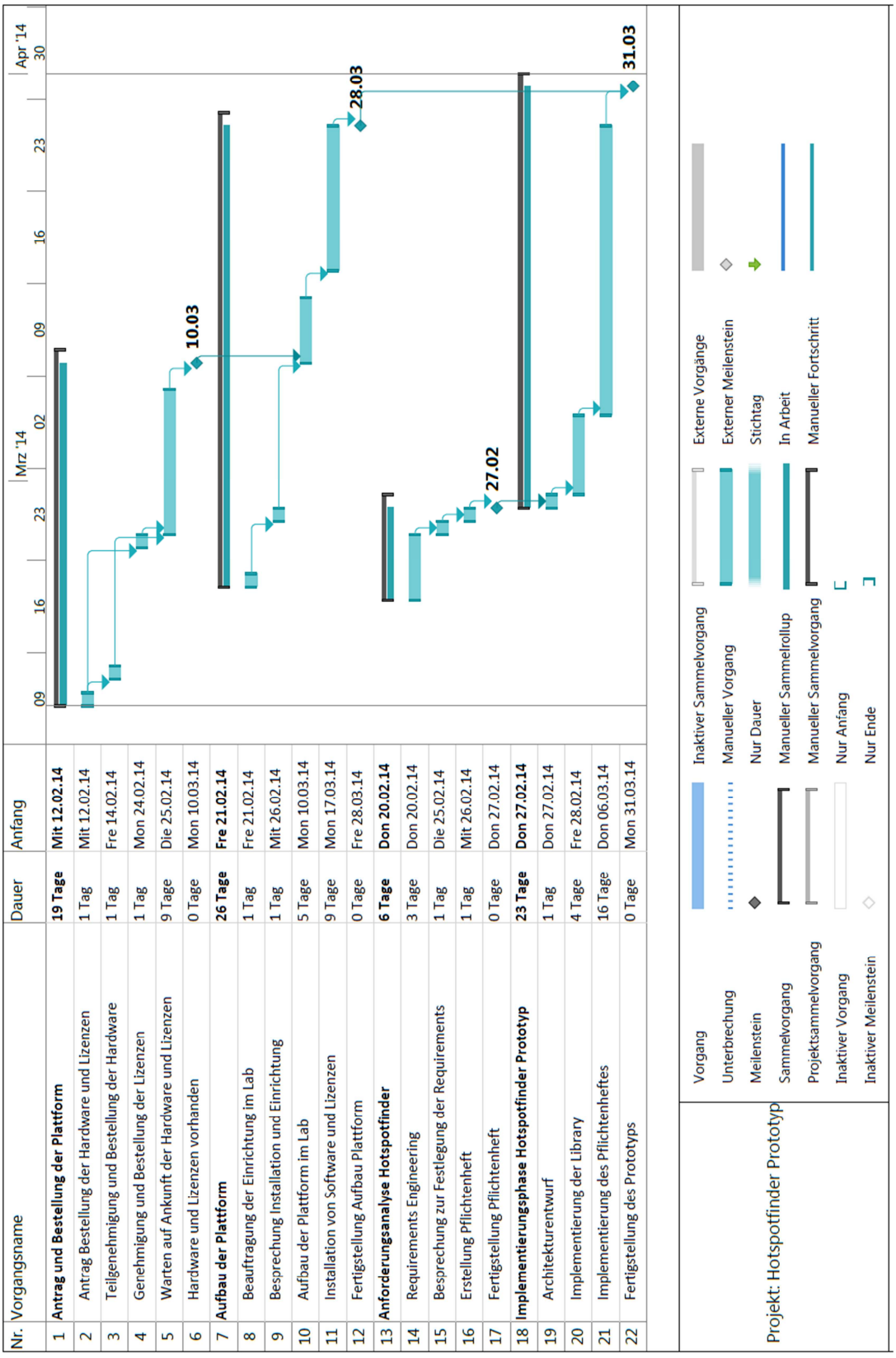


Abbildung 51: Projektplan Hotspotfinder Prototyp

Während ein *TK*-Mitarbeiter aus dem Technologiebereich große Teile aus den zwei zuerst genannten Arbeitspaketen übernahm, wurden die Anforderungsanalyse und die Implementierungsphase vom Autor dieser Arbeit durchgeführt.

Parallel zum dargestellten Projektplan fanden im März die Präsentationen der externen Entwicklungsfirmen statt, die sich auf die Ausschreibung der Hotspotfinder App für Android und iOS beworben hatten. Eine Ausschreibung für andere mobile Systeme wie z.B. Windows Phone lag nicht vor.

Auf die Reaktionen der externen Firmen wird im letzten Kapitel dieser Arbeit eingegangen. In den folgenden Abschnitten werden zunächst die Anforderungen an den Prototyp und die besonderen Herausforderungen während der Implementierungsphase erwähnt. Anschließend werden die im Projekt gesammelten Erkenntnisse dokumentiert.

7.3 Anforderungen an den Hotspotfinder

Es soll eine native mobile Cross-Plattform-Applikation für Android, iOS und Windows Phone 8 entwickelt werden, welche zum Finden von WLAN-Hotspots dient. *TK* kategorisiert die Hotspots in drei unterschiedliche Arten:

- Homespots
- Outdoor-Hotspots
- Indoor-Hotspots

Während bei den zwei letztgenannten Hotspot-Arten die dazugehörigen Adressen abgerufen und ausgegeben werden können, ist dies bei den Homespots aus datenschutzrechtlichen Gründen nicht möglich, da diese Art der Hotspots bei den Privatkunden Vorort in deren Routern integriert sind. Daher sind im Rahmen der mobilen Anwendung nur die GPS-Koordinaten von Homespots bekannt.

Outdoor- und Indoor-Hotspots sollen sowohl in einer Liste, als auch auf einer Karte angezeigt werden. Homespots hingegen sollen auf Grund der fehlenden Adressen nur auf der Karte illustriert werden. Der Benutzer soll Details von Hotspots abrufen, sich an diesen einloggen und eine Verwaltung vornehmen können.

Die genauen Anforderungen, die sich während des Requirements Engineering ergaben, wurden in Musskriterien, Wunschkriterien und Abgrenzungskriterien eingeordnet:

Musskriterien

- Hotspots vom Backend abrufen
- Hotspots auf einer Karte darstellen
- Details von Hotspots aufrufen (Kategorie des Hotspots, ggf. Adresse eines Hotspots)

- Outdoor- und Indoor-Hotspots in einer Liste darstellen
- Outdoor- und Indoor-Hotspots in einer Liste suchen
- Outdoor- und Indoor-Hotspots offline speichern
- Login am Hotspot
- Verwaltung von Hotspots
- Status einer Verbindung zum Hotspot abrufen

Wunschkriterien

- Darstellung von Hotspots durch Scannen eines QR-Codes
- Homespots offline speichern
- Verbesserter Workflow im Vergleich zur bestehenden App durch ein modernes Sidemenü

Abgrenzungskriterien

- Der Prototyp muss keine aufwendige Benutzeroberfläche besitzen

Die Produktfunktionen (Musskriterien) des Prototyps als UML-Anwendungsfalldiagramm:

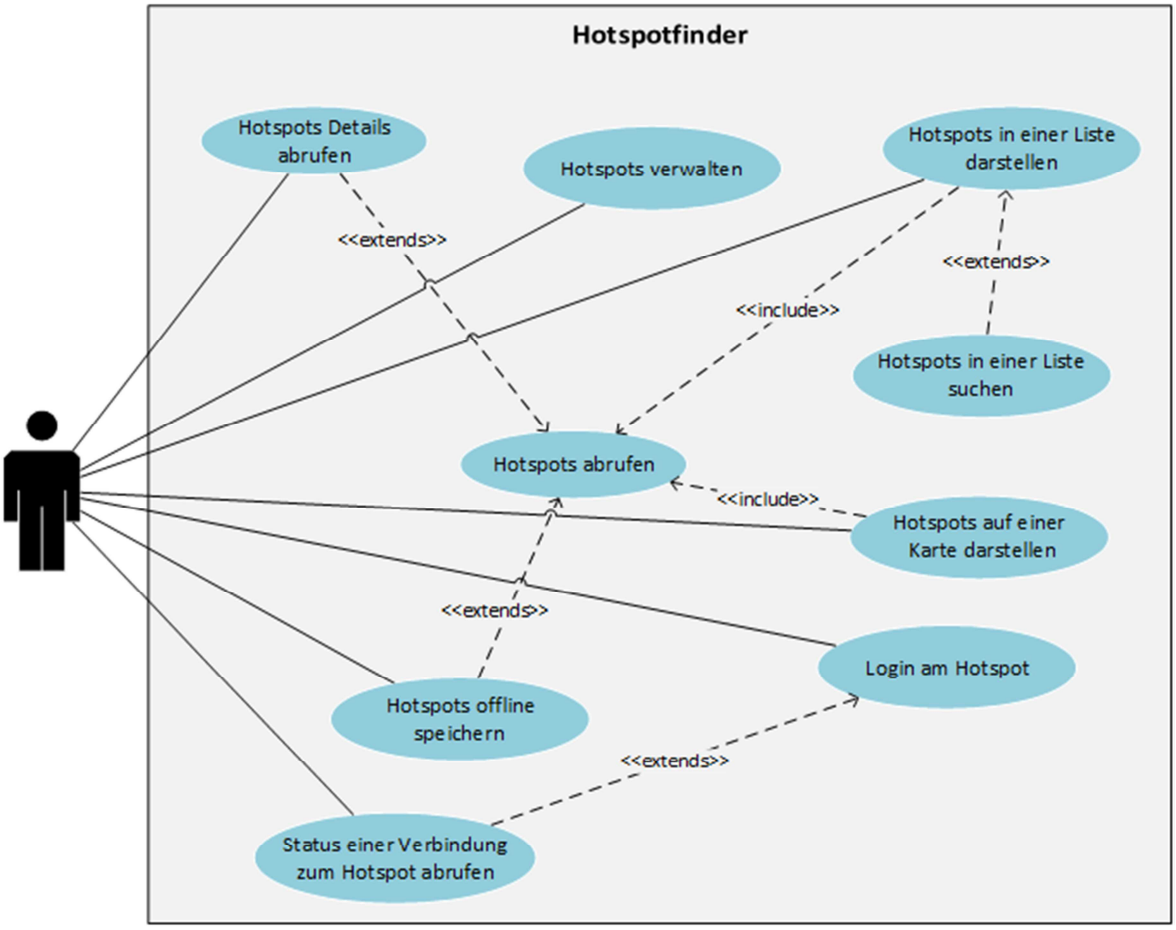


Abbildung 52: Prototyp Produktfunktionen als UML-Anwendungsfalldiagramm

Die Applikation soll auf einem mobilen Endgerät (Android, iOS, Windows Phone 8) mit GPS-Sensor zum Einsatz kommen. Bei der Implementierung des QR-Code-Features wird optional auch eine Kamera im Gerät benötigt.

Die Zielgruppe der Applikation bilden alle Kunden von *TK* mit einem mobilen Endgerät mit Android, iOS, Windows Phone 8, welche unterwegs mit Hilfe der GPS-Positionsbestimmung WLAN-Hotspots finden wollen, um über diese das Internet nutzen zu können.

Die Produktumgebung des Prototyps ist in Abb. 53 illustriert. Ein Benutzer ist eine Person mit einem mobilen Endgerät und installierter Hotspotfinder App. Der Benutzer möchte einen WLAN-Hotspot finden, um sich über diesen mit dem Internet zu verbinden. Ein WLAN-Hotspot ist ein von *TK* aufgestellter WLAN Router, mit dem man sich in das Internet verbinden kann. Das Backend liefert alle wesentlichen Prozesse und Nutzerdaten, die vom Hotspotfinder genutzt werden.

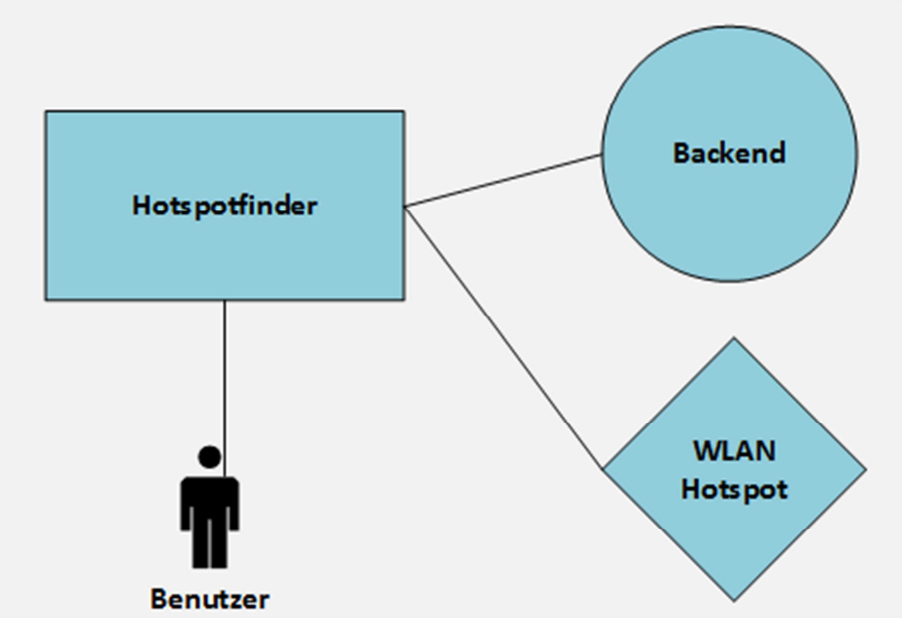


Abbildung 53: Produktumgebung des Prototyps

Die Produktdaten sind vom Backend vorgegeben und werden als JSON-Objekte übertragen. Für das Offline-Speichern von Hotspots wird eine SQLite-Datenbank auf dem lokalen mobilen Endgerät angelegt.

Als Tests sollen die folgenden Szenarien durchgeführt werden:

- /T10/ Der Benutzer meldet sich mit fehlerhaften Daten bei einem Hotspot an
- /T20/ Die App wird ohne aktiviertes GPS gestartet
- /T30/ Die App wird ohne aktivierte Internetverbindung gestartet

Als Entwicklungsumgebung dienen Netbeans mit installiertem *Codename One*-Plugin (Enterprise-Lizenz), sowie eine private *CN1*-Cloud (Corporate-Lizenz).

7.4 Besondere Herausforderungen während der Implementierung

In diesem Unterkapitel findet keine detaillierte Beschreibung der Implementierung des Prototyps statt, da sie etwa 3000 Zeilen Code umfasst. Wie ein Entwicklungsablauf in *Codename One* im Detail aussehen kann, wird im Kapitel 6.3.4 („*Beispielanwendung*“) erläutert. In diesem Kapitel sollen stattdessen neue Erkenntnisse durch die besonderen Herausforderungen gewonnen werden, welche während der Implementierungsphase aufgetreten sind.

7.4.1 Implementierung einer Library

Ein Bestandteil der allgemeinen Architekturanforderungen ist die Entwicklung einer Library, in der die Kommunikation mit dem Backend implementiert ist und die von zukünftigen Projekten genutzt werden soll (vgl. Kapitel 5.2 „*Anforderungen*“). Durch die Kapselung der Backend-Kommunikation haben externe Entwicklungsfirmen keinen direkten Zugriff mehr auf das Backend und nutzen stattdessen die Schnittstellen der Library.

Stand April 2014 liegt eine API-Dokumentation des Backends in der Version 4.2 vor. Aus Gründen der Konzernsicherheit kann die Dokumentation in dieser Arbeit nicht angehangen werden. Um die Umsetzung der Library verstehen zu können, sollen zumindest ein paar grundlegende Fakten erwähnt und die Funktionsweise anhand eines Beispiels erläutert werden.

Die API nutzt das HTTP-Protokoll, wobei ein Zugriff auf die Daten des Backends über ein REST-Interface stattfindet. Für alle Transaktionen wird SSL genutzt. Die allgemeinen Spezifikationen sind:

- JSON ist in UTF-8 kodiert.
- Alle POST-Aufrufe, die kein JSON-Objekt enthalten, müssen URL-kodiert (nicht multipart-kodiert) sein.
- Schlägt ein Request fehl, wird ein 5xx Return-Code zurückgegeben.

Folgende Methoden sind in besagter API-Dokumentation enthalten und in der Library implementiert:

- Session Creation & Portal-Information
- Login im Kundencenter
- Logout aus dem Kundencenter
- Abrufen von festgelegten URLs, auf denen die FAQ, AGB oder das Impressum von *TK* zu finden ist

- Hotspotverwaltungsfunktionen
  - Liste aller Benutzer
  - Anlegen neuer Benutzer
  - Update eines Benutzers
  - Löschen eines Benutzers
  - Prüfung, ob ein Benutzername gültig und verfügbar ist
- Hotspotfunktionen
  - Abrufen aller Hotspots für die Listendarstellung (aus Datenschutzgründen werden Stand April 2014 keine Homespots zurückgeliefert)
  - Abrufen von Meta-Informationen wie z.B. Revision der Hotspot-Datenbank
  - Inkrementelle Updates für die lokale Datenbank
  - Abrufen von Hotspots für einen definierten Kartenausschnitt

Die Library besteht aus den zwei Klassen *HSManage*r und *ServerUrl*. Letztere enthält lediglich alle HTTP-URLs, die vom *HSManager* für die jeweiligen Requests verwendet werden. *HSManager* ist als ein Singleton implementiert und beinhaltet die bereits erwähnten Methoden. Um die Funktionsweise der Klasse zu verstehen, wird im Folgenden das Abrufen von Hotspots für einen definierten Kartenausschnitt erläutert. Um Hotspots für einen Kartenausschnitt zu bekommen, muss die im *HSManager* implementierte Methode *getHotSpotForMap* aufgerufen werden:

```
/**
 * Returns a JSON-Objekt as Hashtable, which contains the hotspot-list for this piece of map.
 *
 * @param latUp upper right latitude
 * @param lngUp upper right longitude
 * @param latDown bottom left latitude
 * @param lngDown bottom left longitude
 * @param width viewport-width
 * @param height viewport-height
 * @param dpi (optional) the dpi of the viewport
 * @param filter (optional) comma-separated list of hotspot-types that should be displayed
 * @return Hashtable with the hotspot-list for this piece of map
 * @throws Exception
 */
public Hashtable getHotSpotForMap(double latUp, double lngUp, double latDown, double lngDown, double width, double height, int dpi, String filter) throws Exception {
   Hashtable<String, String> attributes = new Hashtable<String, String>();
   attributes.put("lat1", String.valueOf(latUp));
   attributes.put("lat2", String.valueOf(latDown));
   attributes.put("lng1", String.valueOf(lngUp));
   attributes.put("lng2", String.valueOf(lngDown));
   attributes.put("width", String.valueOf(width));
   attributes.put("height", String.valueOf(height));

   if (dpi != 0) {
      attributes.put("dpi", String.valueOf(dpi));
```

```
    }
    if (filter != null) {
        attributes.put("filter", filter);
    }
    return executeQuery(URL.getMapUrl(), attributes, null, null);
}
```

Das Backend erwartet die GPS-Koordinaten vom rechten oberen und vom linken unteren Kartenausschnitt, woraus es ein Rechteck bildet, aus welchem die Hotspots aus der Datenbank abgefragt werden. Da *TK* über 350.000 Hotspots verfügt, gibt es Ballungszentren, in denen es selbst bei höchster Zoom-Stufe zu viele Hotspots für die vernünftige Darstellung eines Kartenausschnittes geben würde und sich eine Überlappung der Marker/Grafiken nicht verhindern ließe. Höhe, Breite und DPI-Werte des mobilen Endgerätes nutzt das Backend daher für Optimierungen, um ggf. mehrere einzelne Hotspots zu einer GPS-Koordinate zusammenzufassen, um so die Kartendarstellung zu verbessern. In der JSON-Antwort des Backends sind zusammengefasste Hotspots als Cluster gekennzeichnet, weshalb dem Benutzer durch entsprechende Marker illustriert werden kann, dass sich an diesem Standort mehr als nur ein Hotspot befindet. Durch das Filter-Attribut ließen sich die Hotspots nach den Kategorien Homespots, Indoor-Hotspots und Outdoor-Hotspots filtern. Alle Attribute werden in eine Hashtable geschrieben, die der Methode *executeQuery* übergeben wird. Nach diesem einfachen Prinzip funktionieren alle anderen in der Library implementierten Methoden, die nach außen freigegeben werden.

Die Methode *executeQuery* ist die zentrale Funktion des *HSManagers*, da sie den Request zum Server ausführt und dessen Response parsed:

```
private static Hashtable executeQuery(String url, Hashtable<String, String> attributes, String method, String
token) throws Exception {
    Log.p("execute http request", Log.DEBUG);
    ConnectionRequest conn = new ConnectionRequest();
    if (conn != null) {
        conn.setPost(false);
        if (method != null) {
            if (method.equals("POST"))
                conn.setPost(true);
            else
                conn.setHttpMethod(method);
        }
        conn.setFailSilently(true);
        Log.p("Connection: " + conn, Log.DEBUG);
        conn.setUrl(url);
        if (token != null)
            conn.addRequestHeader("X-CSession-Token", token);

        if (attributes != null) {
            Log.p("Try to read attributes", Log.DEBUG);
            Enumeration e = attributes.keys();
            while (e.hasMoreElements()) {
```

```
            String key = e.nextElement().toString();
            String value = attributes.get(key);
            Log.p("Key of attribute: " + key, Log.DEBUG);
            Log.p("Value for key Below: " + value, Log.DEBUG);
            conn.addArgument(key, value);
          }
        }

        try {
          NetworkManager nm = NetworkManager.getInstance();
          Log.p("Add request to queue", Log.DEBUG);
          nm.addToQueueAndWait(conn);
          Log.p("ResponseCode: " + conn.getResponseCode());
        } catch (Exception e) {
          e.printStackTrace();
        }
        if (conn.getResponseData() == null) {
          throw new Exception("No connection available");
        } else {
          if (url.equals(URL.getDatenschutz())) {
            return null;
          }
          JSONParser parser = new JSONParser();
          ByteArrayInputStream bais = new ByteArrayInputStream(conn.getResponseData());
          InputStreamReader isr = new InputStreamReader(bais, "UTF-8");
          Hashtable jsonObject = parser.parse(isr);
          Log.p("Parsed json Objekt: " + jsonObject.toString(), Log.DEBUG);
          return jsonObject;
        }
      } else {
        Log.p("Connection couldn't be established", Log.DEBUG);
        return null;
      }
}
```

Je nach Aufruf der Methode wird ein GET- oder ein POST-Request durchgeführt. Der Aufruf *setFailSilently(true)* erfolgt für einen *ConnectionRequest*, da in einem solchen Fehler auftreten können, die man nicht behandeln kann. Besteht beispielsweise ein Problem mit dem DNS-Server und kann der Hostname nicht aufgelöst werden, wird ohne diesen Aufruf vom Framework eine Fehlermeldung innerhalb der App in Form eines Dialogs ausgegeben, auf die man als Entwickler keinen direkten Einfluss nehmen kann. Mit Hilfe der Klasse *NetworkManager*, welche das Singleton Entwurfsmuster umsetzt, lassen sich in *Codename One* die Requests synchron oder asynchron versenden. Im Falle der Library erfolgt durch die Methode *addToQueueAndWait* ein synchroner Aufruf. *Codename One* implementiert bereits einen JSON-Parser, der die Antwort als Hashtable repräsentiert, welche dann in der eigentlichen Anwendung iteriert werden muss.

Eine Problematik, die während der Implementierung der Library offengelegt wurde, ist die Tatsache, dass die von *TK* bereitgestellte Dokumentation Fehler enthält, obschon sie in Version 4.2 vorlag. So wurde z.B. für das Abrufen von Hotspots für einen Kartenausschnitt dokumentiert, dass die

GPS-Koordinaten vom linken oberen (statt rechten oberen) und vom rechten unteren (statt linken unteren) Kartenausschnitt benötigt werden. In diesem Fall würde sich am hier vorgestellten Code zwar nichts ändern, jedoch wäre die Schnittstelle falsch dokumentiert und der Aufruf eben solcher fehlerhaft. Der Fehler fiel auf, da das Backend für den gewählten Kartenausschnitt entweder keine oder falsch geclusterte Hotspots zurücklieferte, die wenig Sinn ergaben. Weitere Dokumentations- und API-Fehler fielen in den Funktionen auf, die für die Hotspotverwaltung und für ein inkrementelles Update der lokalen Hotspot-Datenbank zuständig sind. Da der Hotspotfinder bereits separat von zwei unterschiedlichen externen Firmen für Android und iOS entwickelt wurde, stellt sich die Frage, ob dieser Fehler zuvor noch nie aufgefallen ist und warum er noch falsch dokumentiert ist. Da die App parallel zu dieser Arbeit neu ausgeschrieben würde, könnte ohne den Prototyp bzw. ohne Bemerken des Fehlers erneut eine falsche Dokumentation herausgegeben werden. Auf Grund der API-Fehler im Backend im Bereich der Hotspotverwaltungsfunktionen wurden in der App die Menüpunkte „Hotspot Accounts“ und „Manage Accounts“ hinzugefügt. Letzterer implementiert die Hotspotverwaltung nativ über die Library-Schnittstelen, welche auf Grund der Fehler im Backend jedoch Bugs aufweisen. Daher wurde zusätzlich als Workaround der Punkt „Hotspot Accounts“ hinzugefügt, welcher ein WebView implementiert, in dem eine mobile Webseite aufgerufen wird, die einen Login zum Kundencenter verlangt. Dort ist es ebenfalls möglich Accounts zu verwalten.

An dieser Stelle wurde noch einmal die Stärke einer zentralen und von *TK* gepflegten Library deutlich. Auf Grund der derzeit vorherrschenden unterschiedlichen Codebasen der einzelnen nativen Projekte und auf Grund der stetigen Neu-Implementierung gemeinsam genutzter Funktionen entstünde nach der Feststellung eines Fehlers ein enormer Wartungsaufwand, da eine Anpassung pro App und pro Plattform erfolgen müsste. Im Falle einer Library muss jedoch nur diese gewartet und in den jeweiligen Projekten ausgetauscht werden.

7.4.2 Plattformübergreifende Karten

Die Darstellung der Hotspots auf einer Karte ist ein Hauptfeature der App. Die Implementierung dieses Features stellte die größte Herausforderung während der Entwicklung der App dar. Ursächlich dafür ist, dass das Map-Feature pro Plattform unterschiedlich implementiert ist: Android setzt auf Google Maps, iOS auf Apple Maps und Windows Phone auf Nokia Maps. Es gibt sogar Android-Geräte, wie z.B. Amazons Kindle, die kein Google Maps unterstützen. Eine Cross-Plattform-Implementierung eines Map-Features ist demzufolge schwer umsetzbar. In *Codename One* ist diese Problematik so gelöst, dass das Framework eine abstrakte Klasse namens *MapProvider* implementiert und es bisher zwei konkrete Umsetzungen dieser Klasse gibt. Zum einen gibt es den Google Maps- und zum anderen den OpenStreet Map-Provider. Sie erben die Methoden des

MapProviders, wodurch sich nur der Konstruktor unterscheidet und die konkreten Methoden identisch sind. Der Vorteil dieser Implementierung ist, dass sich sehr schnell und ohne große Code-Anpassungen der Kartendienst ändern lässt. Je nach Konstruktor werden die Tiles entweder vom einen, oder vom anderen Server heruntergeladen. Tiles sind dabei typischerweise 256 x 256 Pixel Images, die je nach Bedarf automatisch heruntergeladen und angezeigt werden. Durch diese Art und Weise können Google- oder OpenStreet Maps völlig plattformunabhängig angezeigt werden.

Der große Nachteil gegenüber den nativen Karten, die je nach System unterschiedlich implementiert sind, ist jedoch, dass man nicht von Optimierungen profitieren kann, die z.B. Google in Form von Vektorgrafiken für seinen Kartendienst implementiert hat. Die nativen Karten sind aus diesem Grunde wesentlich performanter als die statischen Bilddateien, die nach jeder Zoomstufe neu heruntergeladen werden müssen.

Die Map-Implementierung in *Codename One* hatte darüber hinaus weitere Einschränkungen. Die im vorherigen Teilkapitel vorgestellte Methode *getHotSpotForMap* benötigt stets die GPS-Koordinaten des aktuellen Kartenausschnitts, welcher dem Benutzer angezeigt wird. Das Framework bot aber keinen Event-Listener an, den man für eine Überwachung des Kartenausschnitts nutzen konnte. Benutzerinteraktionen mit der Karte wie z.B. Scrollen oder Zoomen konnten dementsprechend nicht registriert werden, weshalb auch *getHotSpotForMap* nicht aufgerufen werden konnte. Aus diesem Grund sah der Benutzer keine Hotspots auf der Karte.

Auf Grund dieser Einschränkung und der schlechteren Performanz gegenüber den nativen Karten wurde der *CN1*-Support am 07.03.2014 kontaktiert. Folgende Vereinbarungen wurden getroffen:

1) Das *CN1*-Team implementiert einen Event-Listener, mit dessen Hilfe sich Benutzerinteraktionen mit der Karte registrieren lassen.
2) Das *CN1*-Team entwickelt eine Library, welche auf Android und iOS die Google Maps API implementiert, wodurch sich die Vektorgrafiken von Google nutzen lassen. Konkret setzt die Library für iOS auf das *Google Maps SDK for iOS* auf [Goo143], während für Android die *Google Maps Android API v2* genutzt wird [Goo144]. Für alle anderen Plattformen soll ein Fallback auf die bereits implementierte *MapComponent* stattfinden. Ein solcher Fallback findet außerdem für Android-basierte Geräte statt, welche keinen Play Store implementiert haben (z.B. Amazon Kindle).

Beide Features sollten innerhalb von zwei Wochen zur Verfügung stehen. Für Windows Phone 8 wurde aus Zeitgründen keine spezielle Kartenimplementierung angefordert, da die Fertigstellung

den Projektrahmen des Prototyps gesprengt hätte. Auf der anderen Seite hatten Android und iOS eine höhere Priorität als Windows Phone 8. Zukünftig wäre eine Implementierung von Nokia Maps für Windows Phone 8 jedoch möglich. Für den Prototyp sollte jedoch ein Fallback auf die *MapComponent* stattfinden.

Der Event-Listener für die Karte wurde nur zwei Tage nach dem Feature-Request mit Revision 1782 implementiert [Alm141]. Allerdings wies dieser noch Bugs auf – z.B. wurde kein Event bei Zoom-Änderungen getriggert, welche in Revision 1785 behoben wurden [Alm142].

Die Library mit der nativen Google Maps Implementierung wurde am 17.03.2014 fertiggestellt und ist ebenso Open-Source wie das Framework [Alm143]. Die Implementierung der Library in das bestehende Projekt sorgte jedoch für neue Probleme, welche in enger Zusammenarbeit mit dem Support gelöst werden konnten:

1) Auf Android wurde der Inhalt von Dialogen, die über die Karte gelegt wurden, nur weiß und ohne Inhalt angezeigt. Das Problem lag im nativen Android-Code der Library und wurde durch die Implementierung eines *ViewRenderer* in Revision 4 der Library behoben [Alm144].
2) Die Library bot keine Schnittstelle an, um GPS-Koordinaten einer bestimmten Stelle in der Karte abrufen zu können. Dies ist für einen Aufruf von *getHotSpotForMap* zwingend erforderlich, da die Methode die GPS-Koordinaten vom rechten oberen und vom linken unteren Kartenausschnitt verlangt (vgl. vorheriges Teilkapitel). Diese Funktionalität wurde neu angefordert und in Revision 5 der Library nachgeliefert [Alm145].
3) Das Projekt ließ sich nach der Implementierung der Library nicht mehr für Windows Phone 8 kompilieren. Dabei handelte es sich um einen Fehler im nativen Windows Phone 8 Code, bei dem die Signatur einer Methode fehlerhaft definiert war [Alm145].
4) Die Android- und iOS- Implementierungen der Google Karten wiesen Inkonsistenzen auf. So wurde z.B. auf Android bei Veränderung des Blickwinkels ein Kompass angezeigt, während dieser bei iOS fehlte. Aus diesem Grund wurden der Map-Library Schnittstellen hinzugefügt, mit denen sich solche UI-Elemente explizit steuern lassen [Alm146].
5) Bei einer Veränderung des Blickwinkels wurden keine Hotspots auf der Karte mehr angezeigt. Nach Analyse des Problems stellte sich heraus, dass das Backend von *TK* die GPS-Koordinaten der rotierten Karte nicht verstand und daher keine korrekten Antworten lieferte. Da die Ursache im Backend lag, welches kurzfristig nicht angepasst werden konnte, wurde der Map-Library eine Schnittstelle hinzugefügt, die das Rotieren der Karte unterbindet [Alm146].

6) Sobald sich der Benutzer auf der Karte befand und mit dieser interagierte, kam es auf Android nach einer kurzen Zeit zu Abstürzen.

Der letzte Punkt erwies sich als die größte Herausforderung des gesamten Projektes, da der *CN1*-Support die Abstürze nicht nachstellen konnte. Daher wurden viele Stunden in Code-Analyse und Debugging investiert, wobei zwei Ursachen gefunden werden konnten. Zum einen lag eine Race Condition im Event-Listener der Karte vor, welcher unvorhersehbare Bugs bis hin zu Abstürzen produzierte. Zum anderen bestand in der Marker-Implementierung im nativen Android-Code der Map-Library ein Memory Leak, wodurch es zu einer Out-of-Memory-Exception kam und Android die App selbständig beendete. Nach jedem Map-Event müssen die Hotspots für den aktuellen Kartenausschnitt neu vom Backend abgerufen und auf der Karte als Marker angezeigt werden, wobei ältere Marker zuvor gelöscht werden müssen. Es stellte sich heraus, dass in der Android-Implementierung die Marker zwar von der Karte entfernt, aber weiterhin im Speicher gehalten wurden. Diese Erkenntnisse wurden dem *CN1*-Support mitgeteilt, welcher umgehend die Fixes ins SVN commitete [Alm147] [Alm148].

7.4.3 Wunschkriterien

Die Anforderungen an die App beinhalteten drei Wunschkriterien (vgl. Kapitel 7.3 „*Anforderungen an den Hotspotfinder*“):

- Darstellung von Hotspots durch Scannen eines QR-Codes
- Homespots offline speichern
- Verbesserter Workflow im Vergleich zur bestehenden App durch ein modernes Sidemenü

Während der erste und der letzte Punkt umgesetzt werden konnten, war dies für den zweiten Punkt nicht möglich, da hierfür eine Änderung im Backend nötig wäre, wofür im Rahmen dieses Projekts nicht ausreichend Zeit bestand. Das Backend liefert Stand April 2014 nur GPS-Koordinaten für Homespots, wenn dies für einen definierten Kartenausschnitt abgefragt wird. Für ganz Deutschland lassen sich zwar Indoor- und Outdoor Hotspots in Gänze und ohne Clustering abrufen und speichern, jedoch aus Datenschutzgründen keine einzelnen Homespots.

Die Implementierung eines QR-Code-Scanners war hingegen ohne Probleme möglich, da das Framework mit dem Paket *com.codename1.codescan* einen vollständigen Scanner implementiert [Cod142]. Das Scannen eines QR-Codes funktioniert auf Android ohne eine spezielle Berechtigung für den Zugriff auf die Kamera.
Das Sidemenü konnte ohne Code und nur mit Hilfe des *Codename One Designers* implementiert werden. Als Hilfestellung diente ein offizielles Tutorial [Alm149].

7.4.4 Debugging der generierten Windows Phone 8 Anwendung

Das Debugging der generierten Windows Phone 8 Anwendung auf einem Device stellte insofern eine Herausforderung dar, als das dies nur umständlich funktionierte. Für Android lassen sich die Debug-Meldungen mit der Android Debug Bridge (ADB) und dem Dalvik Debug Monitor Server (DDMS) verfolgen. Für iOS ist dies mit Xcode und der im Organizer zur Verfügung stehenden Konsole möglich. Für Windows Phone 8 ist es selbst als registrierter Developer nicht möglich, Konsole-Ausgaben zu verfolgen. Ein Windows Phone 8 Developer hat jedoch Zugriff auf den internen Speicher seiner Applikation. Diese Funktionalität macht sich *Codename One* zu Nutze. Mit folgendem Aufruf wird im internen Speicher eine Datei geschrieben, welche die Log-Ausgaben enthält:

```
Log.getInstance().setFileWriteEnabled(true);
```

Die Problematik bestand jedoch darin, dass das Schreiben der Log-Datei auf Windows Phone 8 nicht zuverlässig funktionierte, weshalb ein Debugging zu einem Glücksspiel geriet. Um besser Debuggen zu können, wurden zwei Lösungen gefunden. In *Codename One* ist es ab der Pro-Lizenz möglich, sich Logs per Mail schicken zu lassen (vgl. Kapitel 6.3.5 „*Debugging*“), was in den Tests funktionierte. Für die zweite Möglichkeit ist mindestens eine Basic-Lizenz nötig, ab der man sich neben der Binary auch den Quelltext generieren lassen kann. Der Windows Phone 8 Quelltext ließ sich in den Tests problemlos in Visual Studio importieren und ausführen. Bei dieser Lösung sollte jedoch bedacht werden, dass der Code mit *XMLVM* generiert wurde und von einem Menschen nur schwer lesbar ist (siehe Kapitel 4.2.2 „*XMLVM*“), weshalb deutlich mehr Know-How benötigt wird.

7.5 Resultat und gesammelte Erkenntnisse

Der Projektplan konnte eingehalten werden, so dass zum 31.03.2014 alle Anforderungen erfüllt waren. Die generierten Android und iOS Versionen des Hotspotfinders können mit einer nativen Nutzererfahrung überzeugen, da sie von einer „echten“ nativen Anwendung nicht zu unterscheiden sind. Die Performanz der generierten Windows Phone 8 Version ähnelt hingegen der einer Web-App. Ursächlich dafür ist nicht nur die Karte, welche im Prototyp in der Windows Phone 8 Version auf statische Bilder setzt (siehe Kapitel 7.4.2 „*Plattformübergreifende Karten*“), sondern auch das Sidemenü und dessen Transitions, welche etwas behäbig wirken. Gleichwohl muss an dieser Stelle erwähnt werden, dass der Windows Phone 8 Support in *Codename One* einen Beta-Status besitzt und in zukünftigen Versionen eine verbesserte Performanz erwartet werden kann. Eine Implementierung der nativen Nokia Maps könnte ein erster Schritt in diese Richtung sein.

Die hohe Gewichtung bei der Evaluierung der Frameworks in der Kategorie „Wartbarkeit und Support“ hat sich als richtig erwiesen (vgl. Kapitel 5.3.3 „*Gewichtete Evaluierung*“). Der sehr gute

CN1-Support ermöglichte eine Erweiterung des Frameworks um die Google Maps mit Vektorgrafiken für Android und iOS innerhalb kurzer Zeit, weshalb das Projekt nicht ins Stocken geriet. Ohne den Support wäre die Implementierung der Kartenansicht zudem nicht möglich gewesen, da Features wie der Map-Listener gefehlt haben. Dadurch wurde die Erkenntnis erlangt, dass zukünftig Projekte durchgeführt werden können, welche Features benötigen, die zum Evaluierungszeitpunkt noch nicht vorhanden sind.

Als negativer Punkt muss erwähnt werden, dass sich der Simulator zwar je nach ausgewähltem Gerät in den Funktionen bzw. in der Interpretation des Codes ändert, jedoch für ein Debugging nicht vollständig ausreicht. Es ist ratsam, die App schon frühzeitig auf den gewünschten Plattformen bzw. auf den echten Geräten zu testen, da sich diese teilweise unterschiedlich zum Simulator verhalten.

Abgesehen von der Windows Phone 8 Performanz mussten während der Entwicklung keine Kompromisse eingegangen werden, so dass man *Codename One* auch in der Praxis als eine *Write once, run anywhere*-Lösung bezeichnen kann. Neben Android, iOS und Windows Phone 8 wurden zusätzlich die Versionen für Windows (Desktop) und MacOS X getestet. Diese liefen auf Anhieb ohne jegliche Anpassungen. Eine Blackberry-Version konnte mangels Testgerät nicht überprüft werden.

Die Anforderungen verlangten keine aufwendige Benutzeroberfläche, weshalb die nativen Themes genutzt wurden. Diese sind auf allen Geräten überzeugend, welches in folgenden Screenshots begutachtet werden kann:

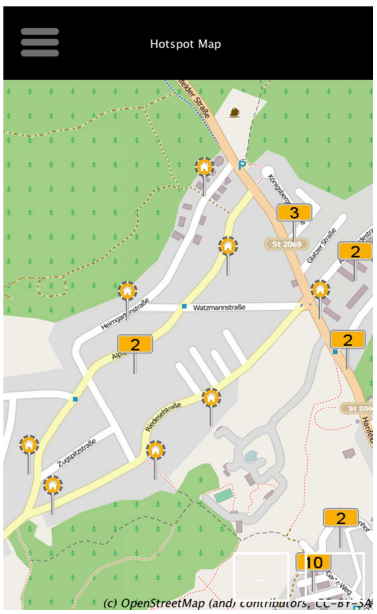


Abbildung 54: Hotspotfinder WP8 Map Fallback

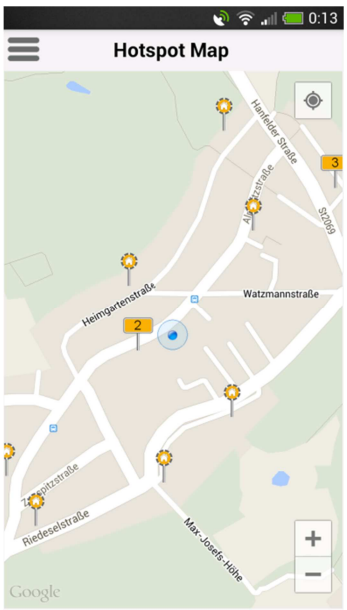


Abbildung 55: Hotspotfinder Android Google Maps

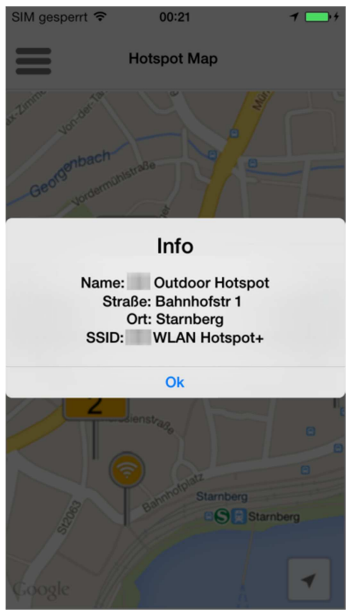


Abbildung 56: Hotspotfinder iOS Hotspot Info

Abbildung 57: Hotspotfinder iOS Homescreen

Abbildung 58: Hotspotfinder Android Sidemenü

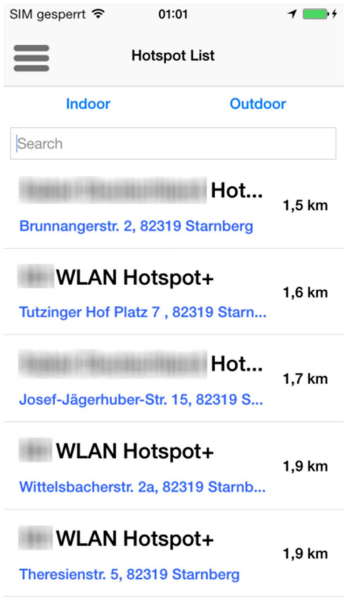


Abbildung 59: Hotspotfinder iOS Hotspot Liste

Abbildung 60: Hotspotfinder WP8 Einstellungen

Als Testgeräte standen die Smartphones HTC One X, iPhone 5C und Nokia Lumia 720 zur Verfügung. Für Tests auf einem Tablet wurde das ASUS Eee Pad Transformer TF300T genutzt, auf welchem die Anwendung ohne jegliche Anpassungen lief.

8 Schlussfolgerung und Empfehlungen

In diesem Kapitel werden die wichtigsten Erkenntnisse dieser Arbeit kurz zusammengefasst. Die technische Sicht soll bei der Wahl eines Frameworks helfen, während die finanzielle Sicht eine Kalkulation des Telekommunikationsunternehmens *TK* enthält, welches auf Grund der gewonnen Erkenntnisse dieser Arbeit zukünftig auf eine native Cross-Plattform-Strategie setzt. Weiterhin werden erste Reaktionen externer Entwicklungsfirmen dokumentiert, welche zukünftig auf der Cross-Plattform, die bereits für den Prototyp genutzt wurde, entwickeln sollen.

8.1 Technische Sicht

Wichtig für die Wahl eines Frameworks ist eine Vergleichbarkeit, die in dieser Arbeit durch eine geeignete Methodologie erreicht wurde. Die Bewertung der Frameworks wurde anhand der Herausforderungen und Anforderungen eines großen Telekommunikationsunternehmens gewichtet, um ein individuelles Ergebnis erhalten zu können. Gleichwohl verliert dieses Ergebnis bei entsprechender Anpassung der Gewichtung für andere Unternehmen bzw. für andere Anwendungsfälle nicht seine Gültigkeit. Aus diesem Grund sollte im Vorhinein genau evaluiert werden, welche Gewichtung den unterschiedlichen Kategorien zukommen könnte und ob in dieser Arbeit nicht berücksichtigte Kriterien mit einbezogen werden sollten.

Die Kompromissbereitschaft erwies sich als ein entscheidender Faktor bei der Wahl eines Frameworks. Für den Prototyp wurde auf eine vollständige Lösung gesetzt, bei der eine Codebasis für sämtliche Plattformen dient. Der Kompromiss, der in diesem Fall eingegangen werden musste, ist eine Windows Phone 8 App, die in Bezug auf Performanz ähnliche Probleme wie eine Web-App besitzt. Die generierten Android- und iOS-Versionen sind hingegen von einer nativen Anwendung, die im entsprechenden Software Development Kit entwickelt wurde, nicht zu unterscheiden. Da das in dieser Arbeit beschriebene Telekommunikationsunternehmen jedoch bereit war, Kompromisse zugunsten einer hohen Portierbarkeit einzugehen, ist dieses Ergebnis im Rahmen der Erwartungen. Weiterhin ist davon auszugehen, dass die Performanz generierter Windows Phone 8 Anwendungen zukünftig steigen wird, da sich der Entwicklungsstand des Frameworks für den Support dieser Plattform während der Erstellung dieser Arbeit in einem Betastatus befand.

Sofern keine Kompromissbereitschaft besteht, sollte bei einer vollständigen Lösung wie *Codename One* die Umsetzbarkeit jedes Features einer mobilen Anwendung vorab im Detail evaluiert werden, da ansonsten während der Entwicklung Herausforderungen auftreten könnten, die man aus den verschiedensten Gründen unter Umständen nicht lösen kann. Anderenfalls sind halbautomatisierte Frameworks wie *Xamarin* zu bevorzugen, bei denen nahezu keine Kompromisse nötig

sind und die den vollen Funktionsumfang einer nativen Entwicklung ausschöpfen. In einem solchen Framework müssen jedoch bis zu 40% plattformspezifisch implementiert werden, weshalb der hohe Portierbarkeitsgrad von vollständigen Lösungen unerreichbar bleibt.

Aus technischer Sicht konnte eine native Cross-Plattform-Entwicklung die hohen Erwartungen erfüllen. Auf Grund der in dieser Arbeit gewonnenen Erkenntnisse, wurden die Entscheidungsträger innerhalb des Telekommunikationsunternehmens *TK* überzeugt, zukünftig auf eine native Cross-Plattform-Strategie zu setzen. Diese Strategie baut auf *Codename One* und einem privaten Build-Server auf, um einerseits von einer fremden Cloud unabhängig zu sein und um andererseits die Bestimmungen der Konzernsicherheit einhalten zu können. Für eine langfristige Strategie erwies sich die hohe Bewertung der Kategorie „Wartbarkeit und Support“ während der Evaluierung als richtig. Ohne den Support bzw. ohne eine Möglichkeit der Erweiterung des Frameworks wäre bereits der Prototyp an Grenzen gestoßen, welche die Implementierung eines Features verhindert hätten.

Das ausgewählte Framework erfüllte alle Herausforderungen und Anforderungen, wodurch sich die folgenden Vorteile gegenüber einer traditionellen nativen Entwicklung ergeben:

- Die für den Prototyp entwickelte Library wird um weitere Backend-Funktionen ergänzt und für verschiedene zukünftige Entwicklungsprojekte genutzt. Einer ersten Kalkulation zufolge können über alle mobilen Anwendungen hinweg etwa 50% der Schnittstellen wiederverwendet werden. Dadurch entsteht eine hohe Standardisierung der Prozesse und Zugriffe. Während der Entwicklung des Prototyps wurden beispielsweise Fehler sowohl in der API-Dokumentation, als auch in deren Funktionalität festgestellt. Bisher erfolgte keine standardisierte Entwicklung, so dass sich ein solcher Fehler in allen Codebasen unterschiedlich äußerte und ggf. verschiedene Workarounds nach sich zog. Durch die nun standardisierte Entwicklung einer Library steigt insgesamt die Qualität der mobilen Anwendungen, da die Schnittstellen zentral gewartet werden und nicht erneut implementiert werden müssen.
- Auf Grund der erhöhten Standardisierung können Guidelines vorgegeben werden, an denen sich externe Entwickler orientieren können. Dadurch wird eine einheitliche Entwicklung gefördert, welche wiederum die Qualität der mobilen Anwendungen positiv beeinflusst.
- Pro App gibt es nur noch eine Codebasis für alle Plattformen, wodurch eine enorme Komplexitäts- und Kostenreduktion stattfindet.

- Der Launch/Release-Termin einer Anwendung ist besser planbar, da es für eine Anwendung jeweils ein Entwicklungsprojekt gibt. Vor der Einführung der Cross-Plattform war für eine App mindestens ein Entwicklungsprojekt pro Zielplattform nötig. Teilweise gab es für eine Plattform zwei Projekte, bei denen nach der Art des Devices unterschieden wurde (Smartphone, Tablet). Die Cross-Plattform ermöglicht hingegen die Konzentration auf ein Entwicklungsprojekt.
- Insgesamt verkürzen sich die Durchlaufzeiten und damit auch die Time-to-Market. Dafür sorgen neben der Komplexitätsreduktion und der Reduktion der Entwicklungsprojekte die besser planbaren Tests. Es wird lediglich gegen eine Codebasis getestet, weshalb kein Abstimmungsbedarf zwischen mehreren Entwicklungsprojekten besteht.

8.2 Finanzielle Sicht

Die Umsetzung einer Cross-Plattform-Strategie und die sich daraus ergebenen technischen Veränderungen haben einen direkten Einfluss auf die Finanzkennzahlen. Im Telekommunikationsunternehmen *TK* existieren momentan sieben verschiedene nativ entwickelte Apps. Damit diese von den im vorherigen Kapitel erläuterten Vorteilen profitieren können, sollen sie im Anschluss dieser Arbeit auf die Cross-Plattform überführt werden. In diesem Kapitel wird mit Hilfe der Net-Present-Value-Methode dokumentiert, welche Einsparungen sich dadurch ergeben.

Jede der sieben vorhandenen Apps hat einen fest definierten Release-Zyklus. Im Falle des Hotspotfinders gibt es beispielsweise drei Releases pro Jahr. Mit einem Release-Zyklus sind keine kleineren Updates, sondern Versionen mit vielen neuen Features gemeint, die im Endeffekt ähnlich hohe Kosten erzeugen, wie die Neuentwicklung der App.

Fünf der sieben Apps werden für Android und iOS entwickelt. Die zwei weiteren mobilen Anwendungen stellen unternehmensinterne Apps im Bereich Service-Delivery und Service-Assurance dar, welche ausschließlich für Android entwickelt werden. Das Entwicklungsbudget der Apps lässt sich in drei Kategorien einteilen:

- Der Umfang der App1, App3 und App4 wird als klein eingestuft. Sie haben jeweils ein Budget von 35.000€ pro Release und pro Plattform.
- App6 und App7 haben je ein Budget von 100.000€ pro Release. Sie werden lediglich unternehmensintern genutzt und für Android entwickelt.
- App2 und App5 sind die größten Projekte. Ihnen steht jeweils ein Entwicklungsbudget von 200.000€ pro Release und pro Plattform zur Verfügung.

Die Entwicklung der Apps fällt in drei unterschiedliche Verantwortungsbereiche und wird unabhängig voneinander und ohne jegliche Standardisierung durchgeführt, weshalb bestimmte Schnittstellen bzw. Funktionalitäten wie z.B. das Login am Kundencenter für verschiedene Apps mehrfach entwickelt werden. Darüber hinaus müssen die drei unterschiedlichen Verantwortungsbereiche für die Funktionalität der App zusammenarbeiten. Pro Bereich fallen daher innerhalb eines Geschäftsjahres Integrationskosten in Höhe von 180.000€ an.

Für die Kalkulation wird das Geschäftsjahr, welches im März 2014 endete, als Grundlage genommen. Für eine Berechnung über die kommenden zwei Geschäftsjahre werden diese Zahlen fortgeschrieben. Des Weiteren wird mit einem gewichteten durchschnittlichen Kapitalkostensatz (WACC) von 13% gerechnet:

Produkt Roadmap ohne Cross-Plattform	**Plattform-Anzahl**	**Releases pro Jahr**	**März FY 13/14**	**März FY 14/15**	**März FY 15/16**
APP1 (Hotspotfinder)	2	3	*210.000 €*	*210.000 €*	*210.000 €*
APP2 (TV Online)	2	1,15	*460.000 €*	*460.000 €*	*460.000 €*
APP3 (TV Home)	2	2	*140.000 €*	*140.000 €*	*140.000 €*
APP4 (Internet)	2	2	*140.000 €*	*140.000 €*	*140.000 €*
APP5 (Internet OTT)	2	1	*400.000 €*	*400.000 €*	*400.000 €*
APP6 (Service-Delivery)	1	1	*100.000 €*	*100.000 €*	*100.000 €*
APP7 (Service-Assurance)	1	1	*100.000 €*	*100.000 €*	*100.000 €*
Plattform Integration			*540.000 €*	*540.000 €*	*540.000 €*
Cash Flow [aktuell]			**2.090.000** €	**2.090.000** €	**2.090.000** €
Cash Flow [kumuliert]			**2.090.000** €	**4.180.000** €	**6.270.000** €
Net-Present-Value (WACC = 13%)			**1.849.558** €	**3.486.334** €	**4.934.809** €

Abbildung 61: Entwicklungskosten ohne Cross-Plattform

Ohne eine Cross-Plattform wird ein jährlicher Cash-Flow von 2.090.000€ benötigt. Die Summe der diskontierten Zahlungen nach drei Geschäftsjahren beträgt demnach 4.934.809*€.*

Um einen Vergleich zu einer Cross-Plattform durchführen zu können, wird angenommen, dass eine solche im letzten Geschäftsjahr zur Verfügung stand. Weiterhin wird angenommen, dass die Cross-Plattform für jede App eine Android-, iOS- und Windows Phone 8 – Version generiert hätte, wobei gleichzeitig nur das Budget einer Plattform benötigt wird. Für eventuelle plattformspezifische Anpassungen oder ggf. benötigte Erweiterungen am Framework wird auf das Budget ein Puffer von 20% gewährt, wodurch sich folgende Entwicklungsbudgets ergeben:

- App1, App3 und App4 können jeweils auf ein Budget von 42.000€ (35.000€ + 20%) pro Release zurückgreifen.
- App6 und App7 haben je ein Budget von 120.000€ pro Release(100.000€ + 20%).

- App2 und App5 steht jeweils ein Entwicklungsbudget von 240.000€ (200.000€ + 20%) pro Release zur Verfügung.

Zur Einführung der Cross-Plattform müssen im ersten Geschäftsjahr alle Schnittstellen der drei Verantwortungsbereiche in einer gemeinsamen Library integriert werden. Eine weitere Annahme besteht darin, dass über alle mobilen Anwendungen hinweg etwa 50% der Schnittstellen (z.B. Login am Kundencenter) wiederverwendet werden können, so dass sich die Integrationskosten von 180.000€ auf 90.000€ halbieren. Durch die Standardisierung der Library arbeiten zukünftig alle Bereiche gemeinsam an dieser, weshalb in den folgenden Geschäftsjahren keine Integrationskosten mehr anfallen. Stattdessen fallen jährlich 25.000€ Lizenzkosten für die Cross-Plattform an.

Mit diesen Annahmen entstehen folgende Entwicklungskosten:

Produkt Roadmap mit Cross-Plattform	Plattform-Anzahl	Releases pro Jahr	März FY 13/14	März FY 14/15	März FY 15/16
APP1 (Hotspotfinder)	3	3	*126.000 €*	*126.000 €*	*126.000 €*
APP2 (TV Online)	3	1,15	*267.000 €*	*267.000 €*	*267.000 €*
APP3 (TV Home)	3	2	*84.000 €*	*84.000 €*	*84.000 €*
APP4 (Internet)	3	2	*84.000 €*	*84.000 €*	*84.000 €*
APP5 (Internet OTT)	3	1	*240.000 €*	*240.000 €*	*240.000 €*
APP6 (Service-Delivery)	3	1	*120.000 €*	*120.000 €*	*120.000 €*
APP7 (Service-Assurance)	3	1	*120.000 €*	*120.000 €*	*120.000 €*
Plattform Integration			*295.000 €*	*25.000 €*	*25.000 €*
Cash Flow [aktuell]			**1.345.000 €**	**1.075.000 €**	**1.075.000 €**
Cash Flow [kumuliert]			**1.345.000 €**	**2.420.000 €**	**3.495.000 €**
Net-Present-Value (WACC = 13%)			**1.190.265 €**	**2.032.148 €**	**2.777.177 €**

Abbildung 62: Entwicklungskosten mit Cross-Plattform

Mit einer Cross-Plattform reduziert sich der jährlich benötigte Cash-Flow im ersten Jahr auf 1.345.000€, ab dem zweiten Jahr auf 1.075.000€. Die Summe der diskontierten Zahlungen nach drei Geschäftsjahren beträgt demnach 2.777.177€.

Ein Vergleich zwischen den aktuellen Entwicklungskosten und denen, die bei einer Cross-Plattform-Entwicklung entstehen:

Geschäftsjahr	Cash Flow Ziel vs. Aktuell	Cash Flow Ziel vs. Aktuell [kumuliert]	Net-Present-Value Ziel vs. Aktuell (WACC = 13%)
März 13/14	*-745.000 €*	*-745.000 €*	*-659.292 €*
März 14/15	*-1.015.000 €*	*-1.760.000 €*	*-1.454.186 €*
März 15/16	*-1.015.000 €*	*-2.775.000 €*	*-2.157.632 €*

Abbildung 63: Entwicklungskosten Ziel vs. Aktuell

Die Höhe der Einsparungen bei Verwendung der Cross-Plattform beträgt innerhalb von drei Jahren 2.775.000€ bzw. diskontiert 2.157.632€. Aus diesem Grund bietet eine native Cross-Plattform-Entwicklung aus finanzieller Sicht Wettbewerbsvorteile.

8.3 Erfahrungen mit externen Entwicklungsfirmen

Während der Entwicklung des Prototyps fanden Ende März 2014 parallel die Präsentationen der externen Entwicklungsfirmen statt, die sich auf die Ausschreibung der nativen Hotspotfinder App für Android und iOS beworben hatten. Eine Ausschreibung für andere mobile Systeme wie z.B. Windows Phone lag nicht vor.

Die externen Firmen stellten eine traditionelle native Entwicklung vor, d.h. eine separate Entwicklung für Android und iOS. Auf die Nachfrage, warum keine Cross-Plattform-Entwicklung durchgeführt werden solle, war die gängige Argumentation, dass sie damit schlechte Erfahrungen gesammelt hätten. Aus dem Gespräch ergab sich jedoch, dass die Firmen in der Vergangenheit weder mit *Codename One* noch mit *Xamarin* gearbeitet hatten. Ihre Erfahrungen beruhten lediglich auf Web-Frameworks.

Die Reaktionen auf die in dieser Arbeit evaluierten Frameworks fielen je nach Firma sehr verschieden aus. Eine Firma lehnte eine solche Entwicklung schon im Vorhinein ab und behauptete, dass eine Entwicklung von Libraries in nativen Cross-Plattform-Frameworks unmöglich sei. Da zu diesem Zeitpunkt bereits die Library für den Prototyp implementiert war, konnte das Gegenteil bewiesen werden. Dennoch wurde die ablehnende Haltung gegenüber einer nativen Cross-Plattform-Entwicklung ohne weitere Argumente beibehalten.

Eine weitere externe Firma gab zwar offen zu, keine Erfahrungen mit *Codename One* zu haben, neuen Technologien gegenüber jedoch aufgeschlossen zu sein. Nach der offiziellen Präsentation suchte ein Vertreter der Firma im Nachhinein das Gespräch, um weitere Informationen zum genannten Framework erhalten zu können.

Letztgenannte Firma erhielt letztendlich den Auftrag für den Hotspotfinder und wird zukünftig die Entwicklung des Prototyps zur Release-Reife weiterführen.

8.4 Fazit und Ausblick

Diese Arbeit hat verschiedene Möglichkeiten zur mobilen Entwicklung aufgezeigt. Auf Grund der erörterten Probleme, die bei webbasierten Entwicklung die Nutzererfahrung mindern können, setzte sich eine native Entwicklung durch. Da mehrere mobile Betriebssysteme unterstützt werden sollten, wurde eine native Cross-Plattform-Entwicklung, sowie deren Grundlagen und Umsetzbarkeit in Hinblick auf Wiederverwendbarkeit, Effizienz, Erweiterbarkeit und anderen Faktoren evaluiert. Dabei wurden unterschiedliche Ansätze für eine plattformübergreifende Entwicklung gefunden, welche durch eine geeignete Methodologie bewertet wurden. Diese Bewertung wurde anhand der Herausforderungen und Anforderungen eines großen Telekommunikationsunternehmens gewichtet, um ein individuelles Ergebnis erhalten zu können. Gleichwohl verliert dieses Ergebnis bei entsprechender Anpassung der Gewichtung für andere Unternehmen bzw. für andere Anwendungsfälle nicht seine Gültigkeit. Die Entwicklung eines Prototyps für eine reale Business-Anwendung sorgte für weitere Erkenntnisse in der Praxis und bestätigte die Ergebnisse der Evaluierung.

Das Resultat kann sowohl aus technischer, als auch aus finanzieller Sicht den hohen Erwartungen nachkommen, weshalb das in dieser Arbeit vorgestellte Telekommunikationsunternehmen zukünftig auf eine native Cross-Plattform-Strategie setzen wird. Für die Überführung bestehender mobiler Anwendungen auf die Cross-Plattform wurde bereits eine externe Firma beauftragt.

Aus den Gesprächen mit den externen Firmen ging hervor, dass in der Unternehmenswelt zum heutigen Zeitpunkt wenig Know-How im Bereich der nativen Cross-Plattform-Entwicklung besteht. Auch wenn ein Framework wie *Codename One* vom Namen her bekannt ist, sind die technischen Grundlagen wie z.B. *XMLVM* meist unklar. Diese Arbeit stellt jedoch das potential solcher Lösungen vor und beweist, dass eine Nutzung im produktiven Umfeld zu technischen Vorteilen und erheblichen finanziellen Einsparungen führen kann.

9 Literaturverzeichnis

[Ado13] Adobe Systems, Inc: PhoneGap http://phonegap.com, Version vom: 02. 12 2013.

[Ado14] Adobe, Inc.: Adobe AIR Developer Center http://www.adobe.com/devnet/air.html, Version vom: 02. 02 2014.

[Ado141] Adobe, Inc.: Native extensions architecture http://help.adobe.com/en_US/air/extensions/WSb464b1207c184b14-70ccb6bd12937b4f2d6-7ffe.html, Version vom: 02. 02 2014.

[Alm14] Shai Almog: What Is Codename One http://codenameone.blogspot.de/2012/01/what-is-codename-one.html, Version vom: 20. 11 2014.

[Alm141] Shai Almog: r1782 - codenameone http://code.google.com/p/codenameone/source/detail?r=1782, Version vom: 28. 03 2014.

[Alm142] Shai Almog: r1785 - codenameone http://code.google.com/p/codenameone/source/detail?r=1785, Version vom: 28. 03 2014.

[Alm143] Shai Almog: Mapping Natively - Codename One http://www.codenameone.com/3/post/2014/03/mapping-natively.html, Version vom: 28. 03 2014.

[Alm144] Shai Almog: r4 - codenameone-google-maps https://code.google.com/p/codenameone-google-maps/source/detail?r=4, Version vom: 29. 03 2014.

[Alm145] Shai Almog: r5 - codenameone-google-maps https://code.google.com/p/codenameone-google-maps/source/detail?r=5, Version vom: 29. 03 2014.

[Alm146] Shai Almog: r14 - codenameone-google-maps https://code.google.com/p/codenameone-google-maps/source/detail?r=14, Version vom: 29. 03 2014.

[Alm147] Shai Almog: r9 - codenameone-google-maps https://code.google.com/p/codenameone-google-maps/source/detail?r=9, Version vom: 30. 03 2014.

[Alm148] Shai Almog: r12 - codenameone-google-maps https://code.google.com/p/codenameone-google-maps/source/detail?r=12, Version vom: 30. 03 2014.

[Alm149] Shai Almog: Codename One Hamburger Sidemenu Tutorial http://www.youtube.com/watch?v=Ipr1eXsu9L4, Version vom: 30. 03 2014.

[App14] Apple, Inc.: iOS Developer Program License Agreement https://developer.apple.com/programs/terms/ios/standard/ios_program_standard_agreement_20130610.pdf, Version vom: 05. 02 2014.

[App141] Appcelerator, Inc: Hyperloop https://github.com/appcelerator/hyperloop, Version vom: 18. 01 2014.

[Bal12] Marcelo Ballvé: Smartphones Are Outselling PCs Two-To-One http://www.businessinsider.com.au/smartphones-outpacing-pcs-two-to-one-2012-11, Version vom: 16. 10 2013.

[Bal14] Tom Ball: Does j2objc support java.net.Socket and java.net.URL? https://groups.google.com/forum/#!topic/j2objc-discuss/waTJR-AwLUs, Version vom: 10. 02 2014.

[Bal141] Tom Ball: Releases - google/j2objc - GitHub https://github.com/google/j2objc/releases, Version vom: 20. 02 2014.

[Bru04] Bernd Bruegge, Allen H. Dutoit: *Object-Oriented Software Engineering: Using UML, Patterns, and Java*. Prentice Hall, Upper Saddle River, 2004.

[Buc14] Erik M. Buck: https://groups.google.com/forum/#!topic/gnu.gnustep.discuss/JYIa-87IbZA https://groups.google.com/d/msg/gnu.gnustep.discuss/JYIa-87IbZA/jWwUaiFkhM0J, Version vom: 25. 02 2014.

[Cat14] Grégory Catellani: App-Entwicklung für Apple-iOS am Beispiel eines Head-mounted Displays http://kola.opus.hbz-nrw.de/volltexte/2012/798/pdf/thesis.pdf, Version vom: 25. 02 2014.

[Cod14] Codename One 101 - Write Native Mobile Apps In Java https://www.udemy.com/codenameone101#/, Version vom: 14. 03 2014.

[Cod141] Codename One: Installing The Codename One Build Server (beta) http://www.codenameone.com/uploads/9/7/6/3/9763921/corporate_server_install_guide.pdf, Version vom: 28. 02 2014.

[Cod142] Codename One: CodeScanner (Codename One API) http://codenameone.googlecode.com/svn/trunk/CodenameOne/javadoc/com/codename1/codescan/CodeScanner.html, Version vom: 30. 03 2014.

[Cor14] Corona Labs, Inc.: Corona SDK http://coronalabs.com/products/corona-sdk, Version vom: 16. 01 2014.

[del14] Miguel de Icaza: Xamarin Welcomes iOS 7.1, with Day One Support! http://blog.xamarin.com/ios-7.1, Version vom: 10. 03 2014.

[Dup14] Emmanue Dupuy: Java Decompiler project http://jd.benow.ca/, Version vom: 15. 03 2014.

[Ebe14] Colin Eberhardt, Sam Hogarth: Xamarin PropertyCross https://github.com/tastejs/PropertyCross/tree/master/xamarin, Version vom: 30. 01 2014.

[För11] Klaus Förster, Bernd Öggl: *HTML5 Guidelines for Web Developers*. Addison-Wesley Professional, Amsterdam, 2011.

[Ful11] Steve Fulton, Jeff Fulton: *HTML5 Canvas*. O'Reilly Media, Inc., Sebastopol, 2011.

[Gar12] Gartner, Inc.: Gartner Says Two-Thirds of Enterprises Will Adopt a Mobile Device Management Solution for Corporate Liable Users Through 2017 http://www.gartner.com/newsroom/id/2213115, Version vom: 17. 10 2013.

[Gar13] Gartner, Inc.: Gartner Says Smartphone Sales Accounted for 55 Percent of Overall Mobile Phone Sales in Third Quarter of 2013 http://www.gartner.com/newsroom/id/2623415, Version vom: 20. 11 2013.

[Goo13] Google, Inc.: Android NDK http://developer.android.com/tools/sdk/ndk/index.html, Version vom: 22. 11 2013.

[Goo131] Google, Inc.: Memory Management/GC https://code.google.com/p/j2objc/wiki/MemoryManagement, Version vom: 28. 12 2013.

[Goo14] Google, Inc.: Adobe AIR - Android Apps auf Google Play
https://play.google.com/store/apps/details?id=com.adobe.air, Version vom: 05. 02 2014.

[Goo141] Google, Inc.: j2objc
http://code.google.com/p/j2objc, Version vom: 10. 02 2014.

[Goo142] Google Project Hosting: in-the-box - Porting of Gingerbread Android runtime on top of iOS
http://code.google.com/p/in-the-box, Version vom: 12. 02 2014.

[Goo143] Google, Inc.: Google Maps SDK for iOS
https://developers.google.com/maps/documentation/ios, Version vom: 28. 03 2014.

[Goo144] Google, Inc.: Google Maps Android API v2
https://developers.google.com/maps/documentation/android, Version vom: 28. 03 2014.

[Hay14] Jeff Haynie: Introducing Hyperloop
http://www.appcelerator.com/blog/2013/10/introducing-hyperloop/, Version vom: 15. 01 2014.

[Hei14] Heise Zeitschriften Verlag: TZI identifiziert Sicherheitslücken in mit Cordova erstellten Apps
http://www.heise.de/newsticker/meldung/TZI-identifiziert-Sicherheitsluecken-in-mit-Cordova-erstellten-Apps-2117556.html, Version vom: 24. 02 2014.

[Hei141] Heise Zeitschriften Verlag GmbH & Co. KG: Mono-Entwickler gründen eigenes Unternehmen
http://www.heise.de/developer/meldung/Mono-Entwickler-gruenden-eigenes-Unternehmen-1244160.html, Version vom: 28. 01 2014.

[Hen12] Heitkötter Henning, Sebastian Hanschke, Tim A. Majchrzak: Comparing cross-platform development approaches for mobile applications
http://www.wi1.uni-muenster.de/pi/veroeff/heitkoetter/Comparing-Cross-Platform-Development-Approaches-for-Mobile-Applications.pdf, Version vom: 16. 11 2013.

[Hog101] Brian P. Hogan: *HTML5 and CSS3: Develop with Tomorrow's Standards Today*. Pragmatic Bookshelf, Raleigh, 2010.

[Hol11] Anthony T. Holdener III: *HTML5 Geolocation*. O'Reilly Media, Inc., Sebastopol, 2011.

[Köl14] Jürgen Köller: Turm von Hanoi
http://www.mathematische-basteleien.de/hanoi.htm, Version vom: 25. 02 2014.

[Laf14] Eric Lafortune: ProGuard
http://proguard.sourceforge.net, Version vom: 20. 02 2014.

[Lap13] Joan Lappin: Nokia Shareholders Approve Sale Of Mobile Devices To Microsoft
http://www.forbes.com/sites/joanlappin/2013/11/19/nokia-shareholders-approve-sale-of-mobile-devices-to-microsoft/, Version vom: 20. 11 2013.

[Lar14] Frederic Lardinois: Xamarin Launches Test Cloud Automated Mobile UI Testing Platform
http://techcrunch.com/2013/04/16/xamarin-launches-test-cloud-automated-mobile-ui-testing-platform-acquires-mobile-test-company-lesspainful, Version vom: 08. 03 2014.

[Lat14] Chris Lattner: The LLVM Compiler Infrastructure
http://llvm.org, Version vom: 01. 02 2014.

[Law11] Bruce Lawson, Remy Sharp: *Introducing HTML5*. New Riders, Berkeley, 2011.

[Log14] LogicNP Software: Crypto Obfuscator For .Net http://www.ssware.com/cryptoobfuscator/obfuscator-net.htm, Version vom: 28. 01 2014.

[Lum14] Thomas Lumesberger: BlackBerry kann nun offiziell Android-Apps ausführen http://www.androidmag.de/news/technik-news/blackberry-kann-nun-offiziell-android-apps-ausfuhren, Version vom: 12. 02 2014.

[Mah11] Christoph Mahlert: Evaluierung der Umsetzung nativer mobiler Anwendungen im Vergleich zu webbasierten Technologien https://www.os.in.tum.de/fileadmin/w00bdp/www/Lehre/Abschlussarbeiten/Bachelorarbeit_Mahlert.pdf, Version vom: 16. 11 2013.

[Mar14] Marmalade Technologies Ltd.: Marmalade - The fastest cross-platform C++ game development SDK https://www.madewithmarmalade.com, Version vom: 05. 02 2014.

[Nai08] Kshirasagar Naik, Priyadarshi Tripathy: *Software Testing and Quality Assurance Theory and Practice*. John Wiley & Sons, Hoboken, 2008.

[NET14] NETdecompiler.com: Dis# - .NET Decompiler http://www.netdecompiler.com, Version vom: 26. 01 2014.

[Ols12] Scott Olson, John Hunter, Ben Horgen, Kenny Goers: *Professional Cross-Platform Mobile Development in C#*. John Wiley & Sons, Indianapolis, 2012.

[Ora14] Oracle Corporation: ADF Mobile Overview and Frequently Asked Questions http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adfmobilefaq-1866697.pdf, Version vom: 08. 01 2014.

[Paa13] Timo Paananen: Smartphone cross-platform frameworks https://publications.theseus.fi/bitstream/handle/10024/30221/110510_Thesis_Timo_Paananen.pdf, Version vom: 16. 11 2013.

[Pan14] Panxiaobo: dex2jar http://code.google.com/p/dex2jar/, Version vom: 15. 03 2014.

[Pet12] Farago Peter: iOS and Android Adoption Explodes Internationally http://blog.flurry.com/bid/88867/iOS-and-Android-Adoption-Explodes-Internationally, Version vom: 16. 10 2013.

[Pho13] PhoneGap Blog: PhoneGap, Cordova, and what's in a name? http://phonegap.com/2012/03/19/phonegap-cordova-and-what's-in-a-name, Version vom: 26. 11 2013.

[Rea14] ReadyTalk: Avian http://oss.readytalk.com/avian/index.html, Version vom: 04. 02 2014.

[Sen14] Sencha, Inc.: Sencha Touch - Build Mobile Web Apps with HTML5 http://www.sencha.com/products/touch, Version vom: 20. 01 2014.

[Sha12] Greg Shackles: *Mobile Development with C#*. O'Reilly Media, Sebastopol, 2012.

[Sim13] Gary Sims: iOS users spend 20% more of their time using apps compared to Android users http://www.androidauthority.com/time-using-apps-223112/, Version vom: 20. 11 2013.

[Sin11] Inderjeet Singh, Manuel Palmieri: Comparison of cross-platform mobile development tools http://www.idt.mdh.se/kurser/ct3340/ht11/MINICONFERENCE/FinalPapers/ircse11_submission_16.pdf, Version vom: 16. 11 2013.

[Sta13] Statista, Inc.: Global market share held by Nokia smartphones from 1st quarter 2007 to 2nd quarter 2013
http://www.statista.com/statistics/263438/market-share-held-by-nokia-smartphones-since-2007/, Version vom: 20. 11 2013.

[Sta14] Stack Exchange, Inc.: Newest xamarin or monotouch or monodroid or xamarin.mac Questions
http://stackoverflow.com/questions/tagged/xamarin%20or%20monotouch%20or%20monodroid%20or%20xamarin.mac, Version vom: 11. 03 2014.

[Ste14] Hannah Steve: Avian vs XMLVM: AOT Java on iPhone Benchmarks
http://sjhannah.com/blog/?p=226, Version vom: 25. 02 2014.

[The13] The Apache Software Foundation: Apache Cordova
http://cordova.apache.org, Version vom: 02. 12 2013.

[The14] The jQuery Foundation: jQuery Mobile
http://jquerymobile.com, Version vom: 20. 01 2014.

[Tri14] Trillian Mobile AB: RoboVM - Create native iOS apps in Java
http://www.robovm.org, Version vom: 12. 01 2014.

[W3C13] W3C: W3C Confirms May 2011 for HTML5 Last Call, Targets 2014 for HTML5 Standard
http://www.w3.org/2011/02/htmlwg-pr.html.en, Version vom: 21. 11 2013.

[W3C131] W3C: HTML Media Capture
http://www.w3.org/TR/html-media-capture/, Version vom: 25. 11 2013.

[W3C132] W3C: Web Notifications
http://www.w3.org/TR/notifications/, Version vom: 25. 11 2013.

[Wiś11] Ryszard Wiśniewski: android-apktool
http://code.google.com/p/android-apktool/, Version vom: 15. 03 2014.

[Xam13] Xamarin, Inc.: Xamarin - iOS Limitations
http://docs.xamarin.com/guides/ios/advanced_topics/limitations, Version vom: 05. 12 2013.

[Xam14] Xamarin, Inc.: What is Mono
http://www.mono-project.com/What_is_Mono, Version vom: 26. 02 2014.

[Xam141] Xamarin, Inc.: Xamarin - Android Limitations
http://docs.xamarin.com/guides/android/advanced_topics/limitations, Version vom: 05. 12 2013.

[Xam142] Xamarin, Inc.: Xamarin Forums
http://forums.xamarin.com, Version vom: 08. 03 2014.

[Xam143] Xamarin, Inc.: Introduction to Test Cloud
http://docs.xamarin.com/guides/testcloud/introduction_to_test_cloud, Version vom: 08. 03 2014.

[Xam144] Xamarin, Inc.: Tasky
http://docs.xamarin.com/content/Tasky, Version vom: 05. 03 2014.

[Xam145] Xamarin, Inc.: xamarin / mobile-samples
https://github.com/xamarin/mobile-samples, Version vom: 10. 02 2014.

[XML14] XMLVM: XMLVM - Overview
http://xmlvm.org/overview, Version vom: 22. 02 2014.

[XML141] XMLVM: XMLVM - Overview: iPhone/Objective-C
http://xmlvm.org/iphone, Version vom: 26. 02 2014.

10 Abbildungsverzeichnis