Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Bachelorarbeit

Studienrichtung
Informatik

Julian Feller

Cross-Plattform App-Entwicklung: Evaluation und prototypische Implementierung eines Ansatzes mit Cross-Translator Tool-Chain

Prüfer: Prof. Dr. Michael Kipp
Abgabe der Arbeit am: 20.10.2014

Betreuer im Unternehmen MaibornWolff GmbH:
Dipl. Inf. Robert Schmitz
Dipl. Inf. Sebastian Krieg

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences
An der Hochschule 1
D-86161 Augsburg
Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Verfasser der Bachelorarbeit:
Julian Feller
Wilhelm-Busch-Weg 12
86161 Augsburg
Telefon:+49 821 552980
julian.feller@t-online.de

Abstract

In dieser Bachelorarbeit wird ein möglicher Cross-Plattform-Ansatz unter Verwendung eines Cross-Translators zur Entwicklung mobiler Apps thematisiert. Für die Bewertung bereits bestehender Lösungen werden Kriterien erarbeitet, die ein Cross-Plattform-Ansatz erfüllen soll und vier mögliche Ansätze anhand dieser Kriterien untersucht und bewertet. Auf Basis dieser Zwischenergebnisse wird ein Cross-Plattform-Ansatz unter Einsatz einer komponentenbasierten Architektur konzipiert und prototypisch implementiert. Hierfür wird eine Tool-Chain mit dem Cross-Translator j2ObjC umgesetzt und anschließend auf Teile des Quelltextes einer dafür prototypisch entwickelten App angewendet. Das Konzept wird mit der entstandenen Implementierung anhand der Kriterien evaluiert. Alle vorab ausgewählten Teile des Quelltextes konnten übersetzt und somit plattformübergreifend wiederverwendet werden. Der Ansatz hat sich im Hinblick auf zukünftige Erweiterungen als tragfähiges Konzept erwiesen, das für weitere Forschung Potenzial bietet.

Inhaltsverzeichnis

Quelltextverzeichnis

Abbildungsverzeichnis

1 Einleitung

Der Markt für Smartphones und damit auch für mobile Betriebssysteme expandierte in den vergangenen Jahren wie kaum ein anderer. Während im Jahr 2010 noch ca. 300 Millionen Geräte weltweit ausgeliefert wurden, stieg der Absatz bis zum Jahr 2014 auf über eine Milliarde Smartphones. Es wird erwartet, dass diese Zahl in den nächsten zwei Jahren auf ca. 1,6 Milliarden verkaufter Einheiten pro Jahr ansteigen wird. [BI12]

Zwar gibt es eine ganze Reihe von mobilen Betriebssystemen, der Fokus lässt sich jedoch durch den jeweiligen Marktanteil auf wenige wichtige Systeme eingrenzen. Nachdem im Jahr 2010 Nokia mit seinem Feature-Phone-Betriebssystem Symbian erstmals von Android überholt wurde, um anschließend in der Bedeutungslosigkeit zu verschwinden, RIM mit dem Blackberry OS weiterhin stetig Marktanteile verlor und Apple mit iOS zwar sehr erfolgreich war, jedoch keine Führungsposition erreichen konnte, schaffte es Microsoft nicht, mit dem Betriebssystem Windows Mobile (und später mit Windows Phone) in diesem hart umkämpften Markt Fuß zu fassen. [GAR10, GAR11, GAR 11a, GAR 11b]

Die beiden am meisten verbreiteten Smartphone-Betriebssysteme waren im zweiten Quartal 2014 iOS mit einem Marktanteil von 11,7 Prozent, sowie Android mit 84,7 Prozent. Da diese beiden Plattformen zusammen eine Abdeckung von 96,4 Prozent bilden, werden diese als die einzig relevanten Zielplattformen für die in dieser Arbeit behandelten Untersuchungen definiert. [IDC14]

1.1 Hinführung zum Thema

Unternehmen setzen sowohl in der internen Prozesssteuerung und Kommunikation (B2B) als auch bei der Kommunikation mit Kunden (B2C) immer mehr auf mobile Geräte. Dabei spielen natürlich die Entwicklungs- und Wartungskosten von solchen Apps eine wichtige Rolle. Die Entwicklungskosten einer mobilen Applikation reichen von wenigen Tausend Euro für eine sehr einfache App bis zu einigen Hunderttausend Euro für komplexere Anwendungen.

Wenn nicht nur eine Zielplattform bedient werden soll, sondern die Anwendung sowohl auf iOS als auch auf Android lauffähig sein soll, muss hier auch mit Mehrkosten gerechnet werden. Diese belaufen sich durchschnittlich auf ca. 50 Prozent des Grundaufwands für die primäre Plattform. [IBU13]

Dieser finanzielle Mehraufwand für die Portierung auf zusätzliche Plattformen lässt sich durch den Einsatz einer Cross-Plattform-Lösung deutlich reduzieren, da hier Teile des Quelltextes wiederverwendet werden können. Durch die gemeinsame Codebasis verringern sich nicht nur der initiale Entwicklungsaufwand, sondern auch die Investitionen in die Wartung und Weiterentwicklung der Anwendung.

Im Rahmen dieser Arbeit wird ein Cross-Plattformansatz für die Erstellung von Business-Apps gesucht. Business-Apps sind mobile Anwendungen, die in einem Unternehmen eingesetzt werden, um die Bearbeitung bestimmter Unternehmens-Prozesse auf mobilen Endgeräten zu ermöglichen.

1.2 Problemstellung und Vorgehensweise

Unter den Softwareentwicklern, die mobile Plattformen bedienen, besteht Einigkeit darüber, dass sich bisher kein allgemeingültiger Standard für Cross-Plattform-Entwicklung etabliert hat, der sich für jeden Anwendungsfall gleichermaßen eignet. Trotz den zahlreichen Möglichkeiten, die einem bei der Auswahl eines Cross-Plattform Ansatzes zur Verfügung stehen, gibt es keinen adäquaten Ansatz, der alle Voraussetzungen erfüllt, die für den nachhaltigen Einsatz innerhalb eines Unternehmens benötigt werden. Basierend auf dieser grundlegenden Annahme werden in Kapitel 2 Grundlagen erarbeitet, die im weiteren Verlauf der Arbeit sowohl für die Beurteilung bestehender Ansätze als auch für die Konzeption und Implementierung eines Lösungsansatzes von Bedeutung sind. [VB13]

In Kapitel 3.1 werden Gütekriterien zur Bewertung von Cross-Plattform-Ansätzen festgelegt und beschrieben, die sowohl als Basis für die Evaluation von bestehenden Konzepten als auch des im Rahmen dieser Arbeit entwickelten Lösungsansatzes dienen. Anschließend werden in Kapitel 3.2 vier verschiedene Cross-Plattform-Konzepte unter Verwendung existierender Techniken anhand der festgelegten Kriterien untersucht und bewertet. Auf Basis dieser Zwischenergebnisse wird in Kapitel 4 ein Konzept für einen Ansatz entwickelt, welcher die Gütekriterien möglichst vollständig erfüllt. In Kapitel 5 wird eine konkrete prototypische Implementierung des erarbeiteten Cross-Plattform-Modells

vorgestellt. Hier wird zum einen eine Tool-Chain entwickelt, die die benötigten Werkzeuge und Prozesse des Entwicklungsansatzes enthält. Zum anderen wird ein exemplarischer Anwendungsfall implementiert, der als Basis für die Evaluation des erarbeiteten Konzepts dient. Diese wird in Kapitel 6 anhand der vorher festgelegten Gütekriterien durchgeführt.

2 Grundlagen

In dieser Arbeit werden Konzepte zur Cross-Plattform Entwicklung behandelt. Um diese besser einordnen zu können, werden im ersten Teil des Kapitels die drei grundsätzlichen Ansätze der App-Entwicklung erläutert. Im zweiten Teil werden Entwurfsmuster vorgestellt, welche in der späteren Implementierung verwendet wurden. Außerdem wird das *Kommandozeilentool j2ObjC* beschrieben. Dieses von Google entwickelte Werkzeug ist ein elementarer Bestandteil der prototypischen Implementierung des in dieser Arbeit als Lösung vorgeschlagenen Konzepts.

2.1 Arten der mobilen Entwicklung

Will man eine App für eine mobile Plattform umsetzen, hat man bei der Auswahl der Entwicklungsmethode verschiedene Möglichkeiten. Soll die App für mehr als eine Plattform umgesetzt werden, ergeben sich durch die Auswahl der Entwicklungsmethode Vor- und Nachteile, die im folgenden Teil der Arbeit erläutert werden.

2.1.1 Native Entwicklung

Als native Entwicklung bezeichnet man die Entwicklung einer App in der von der Plattform dafür vorgesehenen Programmiersprache mit dem dafür bereit gestellten *Software Development Kit* (SDK).

Wird beispielsweise eine native App für das Betriebssystem iOS entwickelt, so ist der Quelltext in Objective-C geschrieben. Das von Apple zur Verfügung gestellte SDK bietet Programmierschnittstellen in der Sprache Objective-C, mit denen grafische Benutzeroberflächen oder der Zugriff auf Hardwarekomponenten des Smartphones implementiert werden können.

Das SDK von Android bietet vergleichbare Funktionalitäten in der Programmiersprache Java. Das *Native Development Kit* (NDK) ermöglicht es darüber hinaus, durch den Einsatz der *Java Native Interfaces* (JNI) auch C bzw. C++ Quelltext innerhalb einer Android-App auszuführen, die vorwiegend verwendeten Programmierschnittstellen (UI, Hardwarezugriff) sind jedoch ausschließlich in der Programmiersprache Java verfügbar.

Neben der Plattformsprache unterscheiden sich die mobilen Plattformen häufig durch grundlegend verschiedene Konzepte bei der Programmierung von Benutzeroberflächen oder dem Zugriff auf die Gerätehardware. Eine nativ entwickelte App ist somit eng an die jeweiligen Konzepte der Plattform gebunden und schwierig von einer Plattform auf eine andere Plattform portierbar. Besteht der Wunsch, die App für mehrere Plattformen zu veröffentlichen, muss sie folglich für jede der gewünschten Plattformen separat implementiert werden. Dies führt dazu, dass der Entwicklungsaufwand linear mit der Anzahl der Plattformen steigt.

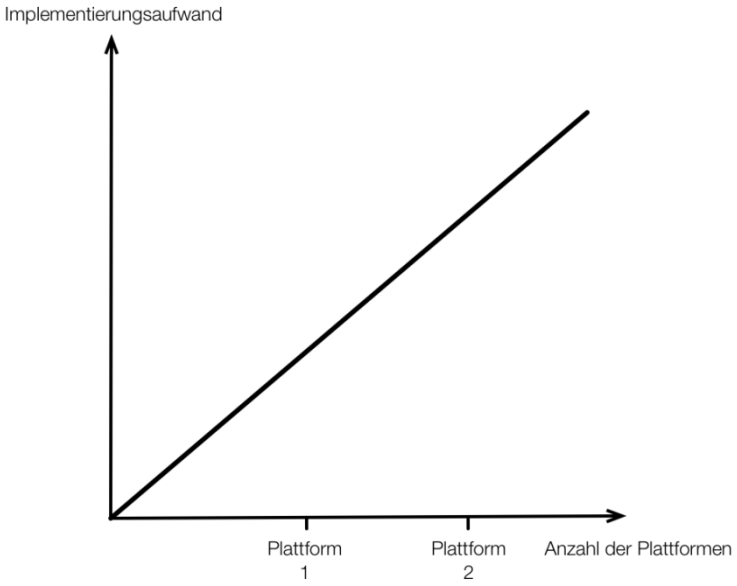

Abbildung 1: Implementierungsaufwand pro Plattform bei nativen Anwendungen

Bei der nativen Entwicklung entsteht also ein erheblicher Mehraufwand. Somit ist die rein native Umsetzung einer App die teuerste der vorgestellten Alternativen.

2.1.2 Webbasierte Entwicklung

Die webbasierte Entwicklung unterscheidet sich von der nativen Entwicklung durch die zur Erstellung der App eingesetzten Technologien. Hier kommen für die Gestaltung der Benutzeroberflächen HTML5 und CSS zum Tragen, während die Logik in der Programmiersprache JavaScript geschrieben werden kann.

Das Ergebnis dieses Ansatzes ist eine mobile Web-App. Diese läuft typischerweise auf einem Webserver, wird im Standard-Browser des mobilen Betriebssystems angezeigt und kann das Aussehen und Verhalten einer nativen App nachahmen. Da die mobile Web-App aber im Grunde nur eine optimierte Website ist, bietet sie auch nur die technischen Möglichkeiten einer solchen. Es ist also nicht möglich, auf die Hardware oder andere Systemfunktionen des Endgerätes zuzugreifen. Durch diese Einschränkung ist die Umsetzung einer mobilen Applikation als Web-App nicht für jeden Anwendungsfall geeignet.

Typischerweise findet man eine Web-App auch nicht in den Appstores der mobilen Plattformen. Durch das Hosting der Web-App auf eigenen Webservern, die somit nicht unter dem Verschluss der jeweiligen Anbieter der mobilen Plattformen stehen, kann man die Web-App beliebig oft und schnell aktualisieren. Ein oftmals mehrwöchiger unbeeinflussbarer Freigabeprozess zur Aufnahme in den jeweiligen Appstore kann somit umgangen werden.

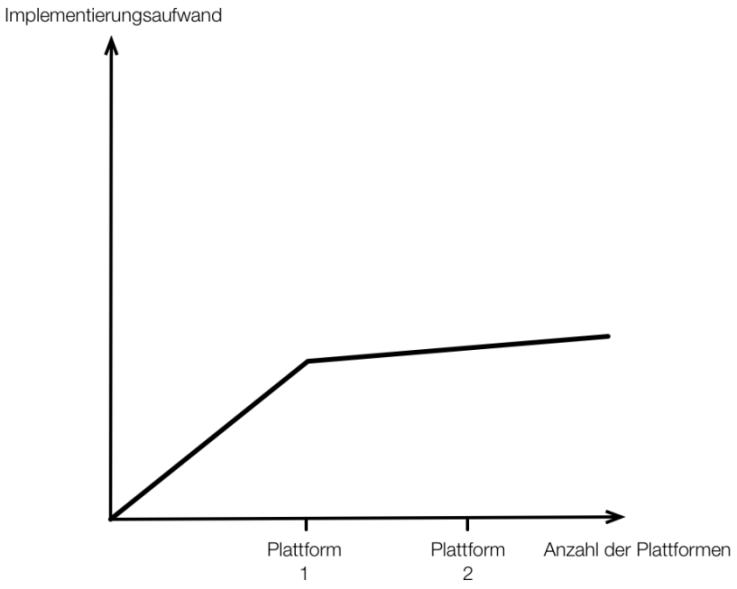

Abbildung 2: Implementierungsaufwand pro Plattform bei webbasierten Anwendungen

Wie für eine Webseite üblich ist es abgesehen von kleineren Anpassungen nicht notwendig, diese für jede Plattform separat zu implementieren, wodurch sich ein nahezu gleichbleibender Aufwand bei steigender Anzahl der Plattformen ergibt.

Dieser Vorteil und auch die Tatsache, dass es viele Entwickler gibt, die die Webtechnologien beherrschen, machen diesen Ansatz für viele Anwendungsfälle attraktiv.

2.1.3 Hybride Entwicklung

Der populärste hybride Entwicklungsansatz ist dem webbasierten Ansatz sehr ähnlich. Er vereint die Vorteile von nativ entwickelten mit denen der webbasierten Apps. Der hybride Ansatz verwendet, ebenso wie der webbasierte Ansatz, Webtechnologien für die Benutzeroberfläche und die Anwendungslogik. Hierbei ist der Quelltext (JavaScript / HTML5) nicht wie bei einer Web-App auf einem Server gespeichert, sondern befindet sich innerhalb des App-Containers lokal auf dem Endgerät. Dies ermöglicht eine performantere Ausführung der Anwendung.

Zusätzlich ermöglicht es der Einsatz einer Abstraktionsebene (Bridge), auf Hardware und Systemfunktionen des Gerätes zuzugreifen. Das Ergebnis ist eine die App in Form einer Binärdatei, die in den jeweiligen Appstores angeboten werden kann.

Durch die Verwendung einer gemeinsamen, plattformunabhängigen Sprache, in welcher Teile des Quelltextes geschrieben sind, werden Entwicklungskosten eingespart.

Je nach eingesetzten Frameworks und Anforderungen an die App, ist es aber vorteilhaft, plattformspezifische Anpassungen individuell zu programmieren. Zum Beispiel können so die unterschiedlichen Interaktionsmuster der jeweiligen Zielplattform adaptiert werden.

Der Implementierungsaufwand bei steigender Anzahl von Plattformen liegt zwischen dem der nativen und der webbasierten Entwicklung.

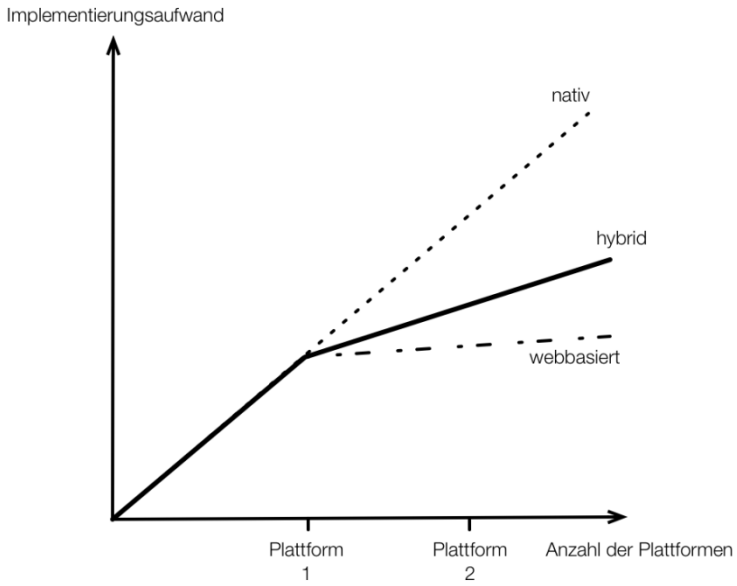

Abbildung 3: Implementierungsaufwand pro Plattform bei hybriden Anwendungen

Wie oben beschrieben ist das populärste Beispiel hierfür die Programmiersprache JavaScript. Es gibt aber, je nach gewünschten Plattformen, auch andere Sprachen die eine hybride Entwicklung ermöglichen.

2.2 Entwurfsmuster und verwendete Technologie

Die nachfolgend allgemein vorgestellten Entwurfsmuster und das Übersetzungswerkzeug j2ObjC sind Bestandteil der Implementierung (Kapitel 5). Im nachfolgenden Teil der Arbeit wird deren allgemeine Funktionsweise erläutert.

2.2.1 Entwurfsmuster

Entwurfsmuster sind Ansätze für die Lösung von allgemeinen wiederkehrenden Problemen, die während der Softwareentwicklung auftreten. Sie bieten konzeptionelle, abstrakte Vorlagen für die Implementierung von bestimmten wiederkehrenden Zusammenhängen, die die Softwarearchitektur betreffen. [FRE07]

Jedes Entwurfsmuster erfüllt einen bestimmten Zweck und hat einen Namen. Dadurch lassen sich nicht nur bewährte Prinzipien bei der Softwareentwicklung wiederverwenden, es vereinfacht auch die Kommunikation über die Struktur einer Softwarearchitektur unter Entwicklern. Die in diesem Kapitel aufgeführten Entwurfsmuster wurden bei der Erstellung der Arbeit verwendet und werden nachfolgend allgemein beschrieben.

2.2.1.1 Chain Of Responsibility

Das Entwurfsmuster *Chain Of Responsibility* ist ein Verhaltensmuster der „Gang of Four“. Die Idee hinter dem Muster ist, dass mehrere Objekte des gleichen Typs eine Kette bilden. Sie implementieren alle dieselbe Schnittstelle `AbstractHandler`, welche die Methode `handleRequest` zum Bearbeiten einer Anfrage definiert. Jedes Objekt der Kette kann einen Nachfolger mit gleichem Interface haben, an den die Weiterverarbeitung der Anfrage delegiert werden kann. Dies ermöglicht einem `Client`, eine Anfrage über diese Schnittstelle an die Zuständigkeitskette zu stellen, ohne dabei den tatsächlichen Empfänger der Anfrage zu kennen.

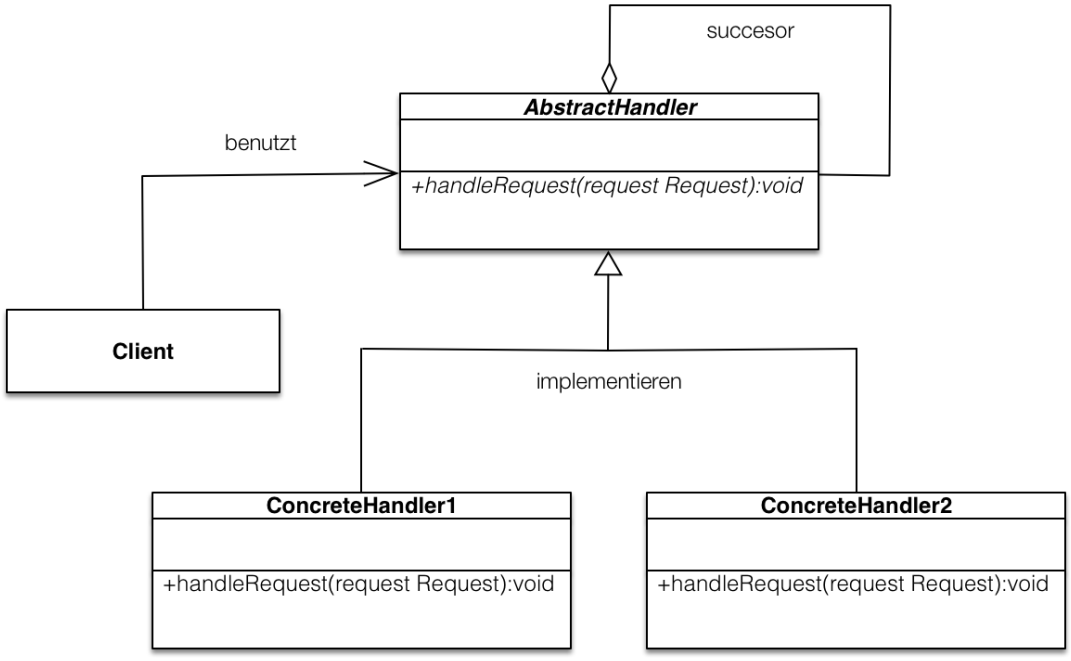

Abbildung 4: Chain Of Responsibility

Das erste Element der Zuständigkeitskette bekommt die Anfrage vom `Client` und entscheidet, ob die Anfrage bearbeitet werden kann. Anschließend wird die Anfrage an das nächste Element der Kette weitergereicht und analog entschieden.

Durch die Aneinanderreihung von Empfängern wird erreicht, dass jeder Empfänger eine Teilaufgabe bearbeitet, das Anfrage-Objekt um Informationen erweitert und dann an seinen Nachfolger weiterreicht. So lassen sich einzelne Aspekte eines komplexen Algorithmus in Einzelschritte kapseln und sind zu einem späteren Zeitpunkt leicht austauschbar.
Durch eine andere Zusammenstellung der Kette lassen sich solche Teile eines Algorithmus darüber hinaus wiederverwenden. [FRE07] 616f

2.2.1.2 Factory

Die Ursprünge des Entwurfsmusters *Factory* lassen sich ebenfalls auf die „Gang of Four“ zurückführen. Es ist ein in der Softwareentwicklung sehr weit verbreitetes Erzeugungsmuster. Es ermöglicht, Objekte zu instanziieren ohne dabei die eigentliche Implementierung oder die für die Instanziierung benötigte Logik zu kennen.

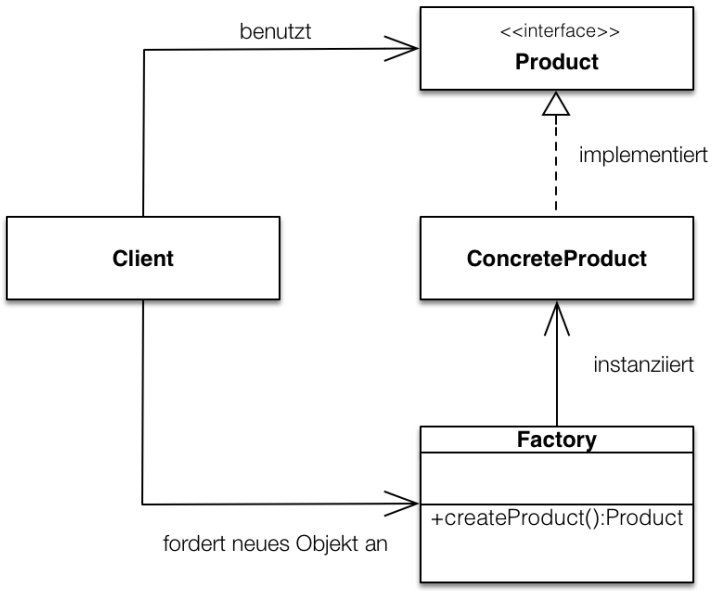

Abbildung 5: Factory

Prinzipiell funktioniert das Muster wie folgt:

Die Factory wird von einem Client aufgerufen und angeleitet, ein neues Objekt zu erstellen, welches eine vorher definierte Schnittstelle implementiert. Die Factory kennt die konkrete Implementierung der Schnittstelle, kapselt die Logik, die notwendig ist, um ein

solches Objekt zu instanziieren, erzeugt letztendlich eine Instanz und gibt diese an den Client zurück.

Durch Anwendung des Musters lässt sich der Grundsatz der losen Kopplung gut umsetzen, da innerhalb der Factory-Methode die konkrete Implementierung der Schnittstelle leicht ausgetauscht werden kann und der Client die konkrete Implementierung der Schnittstelle nicht kennt. [FRE07] 109ff

2.2.1.3 Dependency Injection

Das Entwurfsmuster *Dependency Injection* ist ein Konzept von Martin Fowler. Es ermöglicht dem Anwender, das Prinzip der Single-Responsibilty bei der Entwicklung konsequent umzusetzen. Das Single-Responsibility-Prinzip besagt, dass jede Klasse genau eine klar definierte Aufgabe hat.

Bei der objektorientierten Programmierung gibt es viele Abhängigkeiten zwischen den unterschiedlichen Objekten. Diese Abhängigkeiten lassen sich durch Anwendung von Dependency Injection an einer zentralen Stelle innerhalb der Software verwalten. Das Factory Entwurfsmuster wird oft für diese Aufgabe verwendet. Dadurch ist nicht jedes beteiligte Objekt dafür verantwortlich, seine Abhängigkeiten aufzulösen, sondern bekommt diese von außerhalb injiziert. Dieses Prinzip wird auch *Inversion of Control* genannt. [FOW04]

2.2.2 j2ObjC

j2ObjC ist ein quelloffenes Kommandozeilen-Tool von Google und Grundlage der im Rahmen dieser Arbeit entwickelten Tool-Chain. Es ist derzeit in der Version 0.9.3 unter der Apache 2.0 Lizenz veröffentlicht und befindet sich zwischen Alpha und Beta Status. [GOO14]

j2ObjC ist ein *Source-to-Source Compiler* oder auch *Transcompiler* genannt. Ein Transcompiler übersetzt Quelltext einer Programmiersprache in den Quelltext einer anderen Programmiersprache. Im Gegensatz zu einem Compiler erzeugt ein Transcompiler also keinen lauffähigen Maschinencode, sondern transformiert den Quelltext, welcher dann später von dem Compiler der anderen Programmiersprache übersetzt wird. [NEO14]

Das Kommandozeilen-Tool j2ObjC wird verwendet, um Java Quelltext in Objective-C Quelltext zu übersetzen. Die meisten Java-Sprach-Konstrukte und Laufzeit-Eigenschaften wie Exceptions, innere Klassen, anonyme Klassen, Threads und Reflection werden unterstützt. [GO014]

Durch die Verwendung von j2ObjC ist es also möglich, die in Java geschriebene Logik einer Android App innerhalb einer iOS App zu nutzen. Quelltext, der keine Android-Plattform-Abhängigkeiten besitzt, wie beispielsweise Quelltext, der Geschäftslogik implementiert oder für den Datenzugriff innerhalb einer App zuständig ist, lässt sich transformieren. Dagegen wird Quelltext, welcher grafische Benutzeroberflächen beschreibt und somit Abhängigkeiten zum Android SDK hat, nicht unterstützt und das soll sich laut Google auch in Zukunft nicht ändern. [GO014]

Im Release von j2ObjC sind die Kommandozeilen-Tools *j2objc* und *j2objcc* enthalten.

Abbildung 6: Aufruf j2objc

Das Kommandozeilen-Tool j2objc ist in Java geschrieben und somit ist die Source-to-Source Übersetzung sowohl unter OSX und Linux als auch unter Windows möglich. Unter Windows benötigt man dazu allerdings eine Linux-Kompatibilitätsschicht wie zum Beispiel cygwin [GO014]
Das Programm j2objcc ist ein Wrapper für den Objective-C Compiler *clang*. Für die Kompilierung der übersetzten Java-Klassen wird die Bibliothek *libjre_emul.a* benötigt. Der Wrapper inkludiert diese und andere für die Kompilierung notwendigen Bibliotheken und

Header-Dateien und reicht den Aufruf inklusive aller ihm übergebenen Compiler Flags an den Kompilierer weiter. Da clang Bestandteil des Apple SDKs ist, welches nur unter OSX zur Verfügung steht, ist ein Aufruf des Wrappers auch nur unter OSX Systemen möglich. Die Klasse Test.java (siehe Quelltext 1**Fehler! Ungültiger Eigenverweis auf Textmarke.**) wird mit dem j2ObjC-Aufruf (siehe Abbildung 6) transformiert und mit Hilfe

```
package de.eins.zwei.drei;

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

Quelltext 1: Test.java

des j2objcc-Aufrufs übersetzt.

In Java müssen die Dateinamen dem Namen der in der Datei definierten Klasse entsprechen. Java-Klassen sind immer einem Package zugeordnet und ihr Klassen-Name muss innerhalb des Packages eindeutig sein. Der Speicherort einer Java-Datei ergibt sich aus dem Paketnamen.

```
#ifndef _DeEinsZweiDreiTest_H_
#define _DeEinsZweiDreiTest_H_

@class IOSObjectArray;

#import "JreEmulation.h"

@interface DeEinsZweiDreiTest : NSObject {
}

+ (void)mainWithNSStringArray:(IOSObjectArray *)args;

- (id)init;

@end

__attribute__((always_inline)) inline void DeEinsZweiDreiTest_init() {}

#endif
```

Quelltext 2: Test.h

In Objective-C gibt es keine Packages zur Steuerung von Namensräumen im Sinne von Java. Es ist für den Compiler unerheblich, an welchem Ort die Quelltext-Dateien

gespeichert sind. Die Klasse wird einzig und allein über ihren Klassennamen in einem globalen Namensraum identifiziert.

```
#include "IOSObjectArray.h"
#include "de/eins/zwei/drei/Test.h"
#include "java/io/PrintStream.h"
#include "java/lang/System.h"

@implementation DeEinsZweiDreiTest

+ (void)mainWithNSStringArray:(IOSObjectArray *)args {
  [((JavaIoPrintStream *) nil_chk(JavaLangSystem_get_out_())) println:@"Hallo Welt!"];
}

- (id)init {
  return [super init];
}

+ (J2ObjcClassInfo *)__metadata {
  static J2ObjcMethodInfo methods[] = {
    { "mainWithNSStringArray:", "main", "V", 0x9, NULL },
    { "init", NULL, NULL, 0x1, NULL },
  };
  static J2ObjcClassInfo _DeEinsZweiDreiTest = { "Test", "de.eins.zwei.drei", NULL, 0x1, 2, methods, 0,
NULL, 0, NULL};
  return &_DeEinsZweiDreiTest;
}

@end
```

Quelltext 3: Test.m

Übersetzt man ein Java Objekt mit Hilfe von j2ObjC in die Sprache Objective-C, nimmt der Transcompiler einige Änderungen am Naming vor. Wird beispielsweise die Datei *Test.java* übersetzt, welche die Java-Klasse *Test* beschreibt und zu dem Paket *de.eins.zwei.drei* gehört, erstellt der Transcompiler die Objective-C Implementierungs-Datei *Test.m* und die dazugehörige Header-Datei *Test.h.* Diese wird im Pfad *de/eins/zwei/drei/* abgespeichert. Der Klassenname *Test* des Java Quelltextes wird im Objective-C Quelltext in *DeEinsZweiDreiTest* übersetzt.

3 Untersuchung möglicher Cross-Plattformansätze

Für die Entwicklung von Business-Apps wird ein Cross-Plattform Ansatz gesucht. Um bestehende Cross-Plattform Ansätze in Bezug auf deren Eignung einzuschätzen, wurden dazu in Kapitel 3.1 Kriterien für eine Bewertungsmatrix erstellt, die als Basis für die Beurteilung der untersuchten Ansätze in Kapitel 3.2 dienen.

3.1 Festlegung von Bewertungskriterien für Cross-Plattform-Ansätze

Im Rahmen dieser Forschung haben sich die nachfolgenden vier Kriterien als wesentlich herauskristallisiert. Die inhaltliche Relevanz der jeweiligen Kriterien wird an entsprechender Stelle ausgeführt.

3.1.1 Kriterium 1: keine Logikredundanz

Ein wesentliches Qualitätsmerkmal für jeden Cross-Plattform Ansatz ist die Zeilenanzahl des Quelltextes („Lines of Code“), der plattformübergreifend verwendet werden kann, ohne dabei neu implementiert werden zu müssen. Werden die unterschiedlichen Plattformen bedient, indem komplett eigenständige, native Anwendungen entwickelt werden, kann man mit einem Implemetierungsmehraufwand von nahezu 100% pro Zielplattform rechnen (siehe Kap. 2.1). Kann ein Teil des Quelltextes für verschiedene Plattformen wiederverwendet werden, so sinken neben der eingesparten Entwicklungszeit auch die Entwicklungskosten. Um den Aufwand, der bei der Implementierung entsteht, zu reduzieren, soll beispielsweise Geschäftslogik, welche frei von technischen Abhängigkeiten ist, nicht mehrfach, sondern nur einmal implementiert werden. Die Benutzeroberfläche, sowie auch weite Teile der Adapterschicht sind plattformspezifisch (vgl Kap. 4.2.1), lassen sich also nicht auf andere Systeme übertragen. Es bietet sich jedoch an, die Geschäftslogik der Applikation übertragbar zu gestalten, wodurch sich hier der Aufwand deutlich reduzieren lässt. Die Erfüllung dieses Kriteriums lässt sich in der Praxis gut messen, da einfach die Zeilenanzahl des wiederverwendeten Quelltextes in Relation zum gesamten Programmcode gesetzt werden kann.

3.1.2 Kriterium 2: plattformgetreue, performante Benutzeroberfläche

Der Cross-Plattform-Ansatz soll eine plattformgetreue Benutzeroberfläche bieten, um eine möglichst hohe Akzeptanz der App für den Benutzer zu erreichen.

Jede Plattform hat für die Gestaltung der Benutzeroberfläche eigene Leitfäden, die unter anderem Interaktionsmuster und Navigationskonzepte definieren. Bei Apple sind dies die *„iOS Human Interface Guidelines“.* Bei Einhaltung dieser plattformspezifischen Konventionen kann der Anwender die App intuitiv bedienen, wie er es von seiner

Plattform gewohnt ist. Um dies zu gewährleisten, muss es möglich sein, für jede Plattform eine angepasste Benutzeroberfläche zu erstellen, die dann den Prinzipien der Plattformen folgt.

Die *Norm EN ISO 9241 i*st ein mehrteiliger, international gültiger Standard, der Richtlinien zum Thema Mensch-Computer-Interaktion spezifiziert. Durch Einhaltung dieser Standards soll unter anderem eine hohe Usability von Software erreicht werden. Seit 2006 beschreibt der Teil EN ISO 9241-110 die Grundsätze der Dialog-Gestaltung. Ein dort genannter Grundsatz ist die Erwartungskonformität:

"Ein Dialog ist erwartungskonform, wenn er konsistent ist und den Merkmalen des Benutzers entspricht, z.B. seinen Kenntnissen aus dem Arbeitsgebiet, seiner Ausbildung und seiner Erfahrung sowie den allgemein anerkannten Konventionen." [HTW13]

Der Benutzer einer mobilen App hat die Navigationskonzepte und Interaktionsmuster der Plattform im besten Fall schon intuitiv verinnerlicht. Weichen die Konzepte, die bei der Umsetzung der Benutzeroberfläche verwendet wurden, ab, muss der Benutzer diese erst neu erlernen.

Der japanische Robotik-Wissenschaftler Masashiro Mori hat in den Siebziger Jahren die Hypothese des „Uncanny Valley“ aufgestellt, in welcher der Zusammenhang zwischen dem Realitätsgrad eines Roboters und seiner Akzeptanz gegenüber dem Menschen empirisch untersucht wird. Die gemessene Akzeptanz steigt paradoxerweise nicht wie angenommen linear bei steigendem Realitätsgrad, sondern bricht ab einem gewissen Grad rapide ein und nähert sich erst ab einer sehr hohen Menschenähnlichkeit wieder an den erwarteten Verlauf. [BIL07]

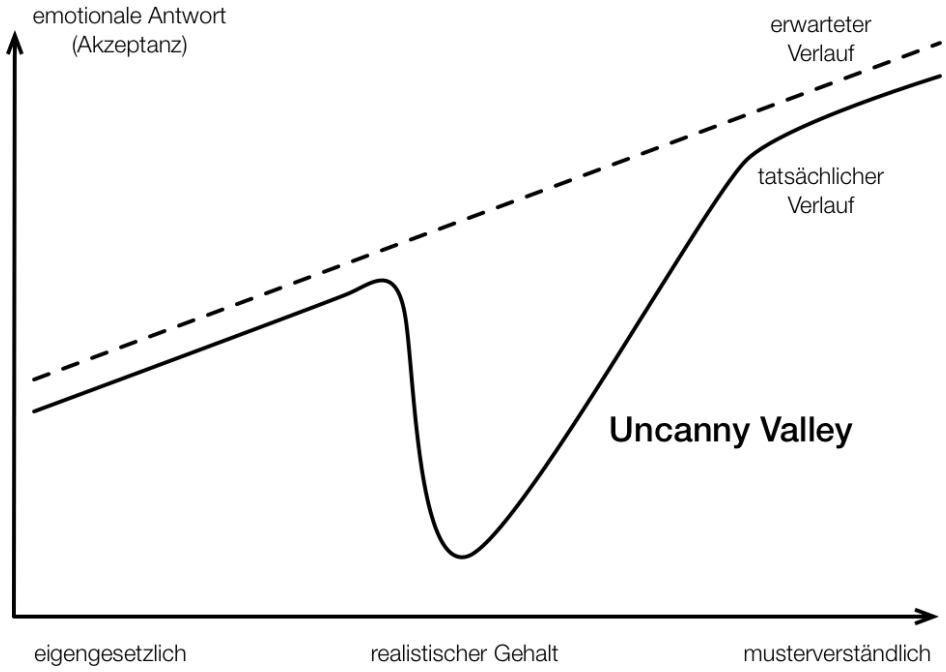

Abbildung 7: Uncanny Valley

Bill Higgins hat diesen Effekt 2007 in einem Beitrag auf seinem Blog auf Web-Benutzeroberflächen übertragen. Eine Benutzeroberfläche hat ein bestimmtes Verhalten und Aussehen. Eine Benutzeroberfläche, die wenig musterverständlich ist und ihren eigenen Gesetzen folgt, wird wahrscheinlich weniger akzeptiert, als eine, die alle Konventionen umsetzt. Dies würde dem erwarteten linearen Verlauf entsprechen. [BIL07]

Ahmt beispielsweise eine Web-Benutzeroberfläche das Aussehen und Verhalten einer nativen App nach, erwartet der Benutzer auch das Verhalten einer nativen App. Kleinere Abweichungen bei Animationen oder längere Reaktionszeiten bei der Interaktion mit UI-Elementen könnten durch eine falsche Erwartungshaltung zur Enttäuschung des Benutzers führen und somit dessen Akzeptanz deutlich senken (Uncanny Valley).

Prinzipiell ergäben sich hier also zwei Optionen, um den angenommenen negativen Effekt des Uncanny Valley zu vermeiden:

- Es wird versucht, alle Richtlinien und Interface-Konzepte zu 100 Prozent erwartungskonform umzusetzen
- Es wird ein Ansatz verfolgt, der sich bewusst deutlich von den plattformspezifischen Konzepten unterscheidet

Die zweite Option steht in Konflikt zum oben genannten Anspruch, die Interface-Konzepte der Zielplattform zu adaptieren. Für die angestrebte Lösung wird also eine native Implementierung der Benutzeroberfläche als Gütekriterium festgelegt.

3.1.3 Kriterium 3: gute Unterstützung durch IDE und Tooling

Um es dem Programmierer zu ermöglichen, eine mobile App effizient zu entwickeln, sollte für den Cross-Plattform Ansatz eine Entwicklungsumgebung vorhanden sein, die ihn beim Entwicklungsprozess unterstützt.

Werden Technologien kombiniert, die ursprünglich nicht für einen gemeinsamen Einsatz gedacht waren, entstehen dadurch Grenzen. Diese Technologiegrenzen können die Ursache für erschwerte Bedingungen im Entwicklungsprozess und bei der Fehleranalyse *(Debugging)* sein. Setzt man beispielsweise das *Framework Phonegap* ein, hat man nicht die gleichen Bedingungen wie beim Debuggen von nativem Code. Durch die Grenze zwischen JavaScript, welches innerhalb des Web-Containers ausgeführt wird, und dem nativen Programmcode wird das schrittweise Debuggen der App durch Breakpoints erschwert. [GIT14]

Besteht die Anwendung aus unterschiedlichen Technologien oder aus statisch verlinkten, vorkompilierten Bibliotheken, ist es nicht oder nur mit großen Einschränkungen möglich, effektive Fehlersuche mittels Debuggingtools durchzuführen. Verwendet man beispielsweise die Standard-Entwicklungsumgebung *XCode,* um eine iOS-App zu programmieren, und will JavaScript-Quelltext für Teile der App verwenden, gibt es keine Möglichkeit, während des Debugging-Vorgangs Breakpoints im JavaScript-Teil zu setzen oder den Ablauf der Programmausführung schrittweise zu steuern. Bei der standardmäßig

eingesetzten, auf Eclipse basierenden IDE *Android Developer Tools* (ADT) von Google, gibt es die Möglichkeit, Plugins von Drittanbietern zu installieren, die dies ermöglichen. [HYB14]

Um Fehler, die bei der Programmierung entstehen, nachzuvollziehen, soll der Cross-Plattform-Ansatz zu jeder Zeit transparent sein. Es soll an jedem Punkt nachvollziehbar sein, was geschieht. Dies bezieht sich sowohl auf die Compile-Zeit, wo der Compiler dem Entwickler konstruktives Feedback zur Vermeidung von fehlerhaftem oder fehleranfälligem Code geben kann, als auch auf die Laufzeit, in der durch sinnvoll angewandtes Logging und Debugging die Fehlersuche erleichtert bzw. überhaupt erst ermöglicht wird.

3.1.4 Kriterium 4: wenig Abhängigkeiten zu Dritten

Die Verwendung externer Frameworks bringt neben einer Reihe von Vorteilen auch potentielle Nachteile mit sich. Je mehr externe Frameworks verwendet werden, desto mehr begibt man sich in Abhängigkeiten zu Drittanbietern. Dabei hat nicht nur die Anzahl der Abhängigkeiten (Kriterium 4a) Einfluss auf Erfüllung des Kriteriums, auch der Grad der Abhängigkeit (Kriterium 4b) ist von Bedeutung.

Eine essentielle Frage ist außerdem, ob die verwendete Fremdkomponente nur eine klar abgegrenzte Teilfunktionalität bedient oder gar die gesamte Applikation von proprietären Systemen abhängt. Hiervon leitet sich die Austauschbarkeit der jeweiligen Komponente ab.

Als problematisch zu bewerten ist es, wenn ein großer Teil des Quelltextes nur unter Verwendung eines bestimmten Frameworks nutzbar ist. Dies kann beispielsweise dann zu Problemen führen, wenn Änderungen oder Erweiterungen in Programmierschnittstellen der mobilen Betriebssysteme eingeführt, diese aber nicht von den Entwicklern der Frameworks aktualisiert bzw. neu implementiert werden.

Mit jeder Veröffentlichung neuer Versionen der mobilen Betriebssysteme ändern sich auch deren Programmierschnittstellen. Diese sollten verwendet werden können, um neu hinzugekommene Funktionalitäten der aktuellen Gerätegenerationen zu nutzen. [AND14]

Bei der Nutzung eines Cross-Plattform-Ansatzes sollte daher nicht nur der einmalige Entwicklungsprozess sondern auch die Wartbarkeit evaluiert werden. Es empfiehlt sich,

die Anzahl der verwendeten Frameworks und den Grad der Abhängigkeiten möglichst gering zu halten. Man ist somit bei Aktualisierungen oder Änderungen der Rahmenbedingungen nicht auf die Leistungen der Drittanbieter angewiesen. Die Verwendung von quelloffenen Frameworks erlaubt notfalls eine unabhängige Weiterentwicklung.

3.2 Beurteilung vorhandener Ansätze anhand der festgelegten Kriterien

Im nachfolgenden Kapitel werden vier verschiedene Konzepte zur Cross-Plattform-Entwicklung vorgestellt und anhand der vorher festgelegten Kriterien bewertet.

3.2.1 Ansatz 1: Interpreteransatz

Beispiele für diesen Ansatz sind Appcelerators Framework Titanium und das *Framework Kony* der Firma Kony Inc. Kony verfolgt einen ganzheitlichen Ansatz und bietet Cloud- und Plattform-Lösungen, die den kompletten Lebenszyklus einer App abdecken. Für die Entwicklung stellt Kony die auf *Eclipse* basierende IDE *Kony Studio* bereit.

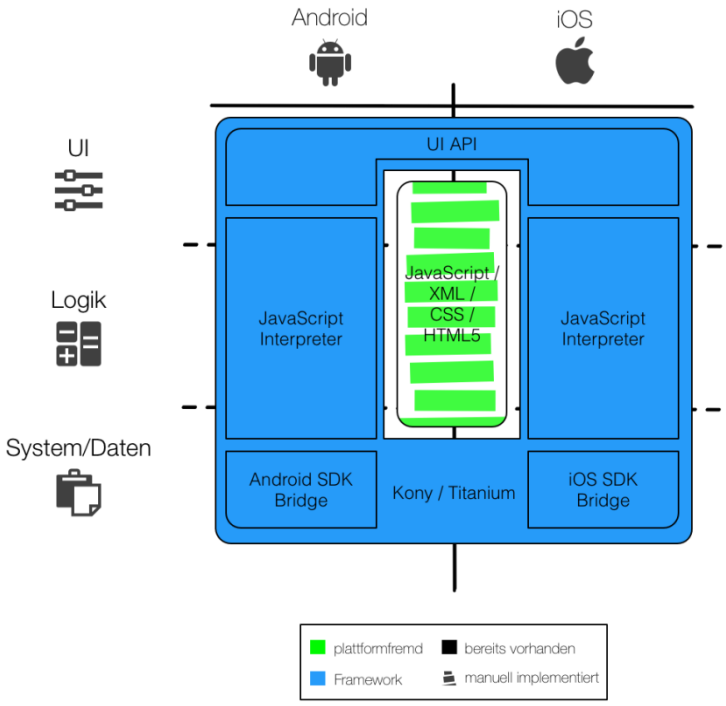

Abbildung 8: Interpreteransatz

Der Ansatz basiert auf der Interpretation von JavaScript-Quelltext zur Laufzeit. Die mobile App wird in JavaScript implementiert. Die Frameworks bieten Programmierschnittstellen an, die es ermöglichen, zur Laufzeit aus interpretiertem JavaScript-Quelltext native UI-Elemente der jeweiligen Plattform zu generieren.

Wird die App gestartet, interpretiert die JavaScript-Umgebung der Plattform den JavaScript-Quelltext. Werden native Komponenten der jeweiligen Plattform benötigt, dienen Teile der Frameworks als Bridge und überführen die Aufrufe in die jeweiligen nativen Pendants. Bei Kony wird der JavaScript-Quelltext in der JavaScript-Engine V8 ausgeführt. [KON14]

Da der Ansatz voll auf proprietären Lösungen aufbaut, begibt man sich in eine hohe Abhängigkeit zu dem Anbieter.

Kriterium	**erfüllt**	**Begründung**
1 UI	Teilweise	Native UIs möglich, bei Erstellung an JavaScript API gebunden
2 Logik	Ja	Anwendungslogik wird wiederverwendet
3 Tooling	Ja	Kony Studio / Titanium Studio
4a Anz Abh.	Ja	Nur an jeweiliges Framework gebunden
4b Grad Abh.	Nein	Abhängigkeitsgrad hoch

3.2.2 Ansatz 2: Hybrider webbasierter Ansatz

Bei diesem Ansatz werden Technologien eingesetzt, die typischerweise in der Webentwicklung Anwendung finden. Die Idee ist, mit HTML5 und CSS die Benutzeroberfläche zu definieren, während die Anwendungslogik in JavaScript geschrieben wird. Die App wird auf dem Endgerät innerhalb eines Web-Containers ausgeführt, verhält sich also ähnlich wie eine lokal gespeicherte, hochoptimierte mobile Website. Der Einsatz des Frameworks PhoneGap steht eine *JavaScript-Bridge* bereit, die es dem Entwickler ermöglicht, auf plattformspezifische Systemfunktionen zuzugreifen wie beispielsweise GPS- und Lagesensoren.

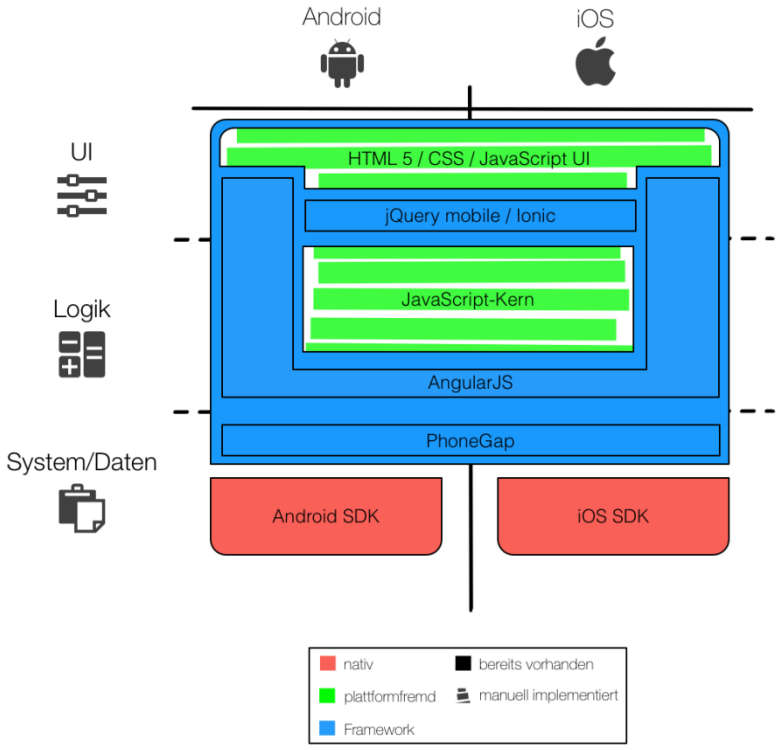

Abbildung 9: hybrider webbasierter Ansatz

Ionic und *jQuery mobile* bieten eigene Bibliotheken von UI-Elementen, die zu einer Benutzeroberfläche zusammengefügt werden können. Hier kann sich der Entwickler zwischen zwei Strategien entscheiden:

Zum einen kann eine Oberfläche gestaltet werden, die auf allen Systemen identisch aussieht, also keine plattformspezifischen Layouts oder Grafiken enthält. Dies hat zur Folge, dass elementare Gestaltungs- und Navigationsmuster der jeweiligen Zielplattform nicht eingehalten werden können.

Die andere Strategie besteht aus dem Versuch, die Benutzeroberfläche von Android bzw. iOS möglichst genau nachzuahmen. Das Framework *AngularJS* stellt hier entsprechende Methoden bereit, die die Differenzierung der Zielplattformen ermöglichen. Da die originalen Systemgrafiken hier aber nicht benutzt werden können, ist es auch nicht möglich, die UI exakt so zu gestalten wie der Nutzer es eigentlich erwarten würde.

Kriterium	**erfüllt**	**Begründung**
1 UI	Nein	keine nativen UIs
2 Logik	Ja	Anwendungslogik wird wiederverwendet
3 Tooling	Nein	schlechtes Debugging
4a Anzahl Abh.	Nein	Hohe Abhängigkeit zu den o.g. Frameworks
4b Grad Abh.	Ja	Zu jedem Zeitpunkt sind alle Quellen lesbar

3.2.3 Ansatz 3: C / C++

Die Programmiersprache C++ eignet sich sowohl für die Plattform Android als auch für iOS. Unter Android ist es möglich, C++ Code per NDK zu übersetzen und über JNI auszuführen, unter iOS kann man C++ Quelltext mit Objective-C Quelltext innerhalb der Implementierung mischen und mit dem Compiler clang übersetzen. Denkbar wäre also auch ein Ansatz, der auf eine komplette Implementierung in C++ setzt.

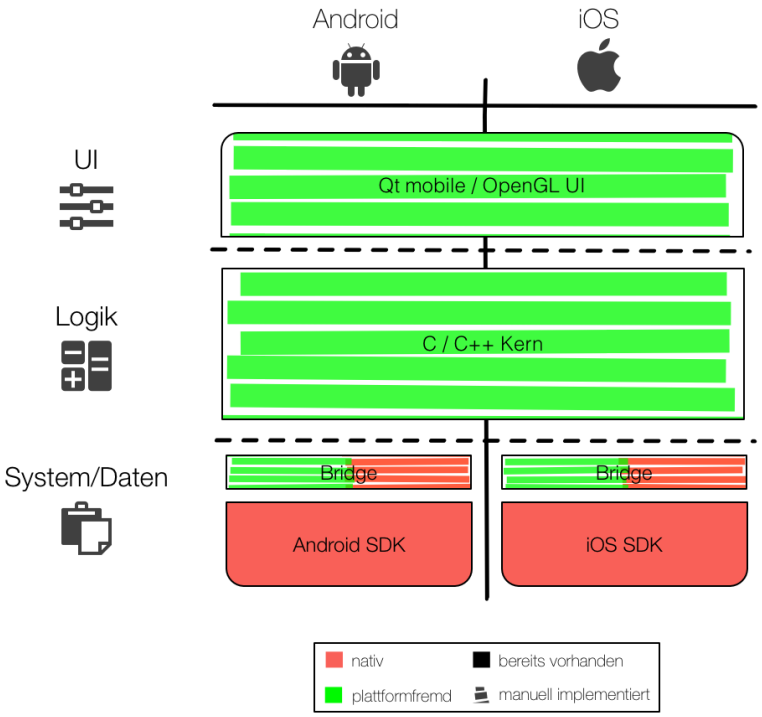

Abbildung 10: C++ Ansatz

Für die Programmierung der UI kann jedoch nicht auf die jeweiligen Programmierschnittstellen der Plattform zugegriffen werden. Es ist aber denkbar, die UI mit Hilfe einer C++ Bibliothek wie beispielsweise *Qt mobile* zu programmieren. Auch denkbar wäre, ein Benutzerinterface in *OpenGL* zu implementieren.

Für die Verwendung von Systemfunktionen müssen bei diesem Ansatz Bridges programmiert werden, die von systemabhängigen Schnittstellen abstrahieren.

Falls nur eine Anbindung an ein Internet-Backend notwendig ist, kann man die *Bibliothek libcurl* in die jeweiligen Projekte mit einbinden. Diese Bibliothek steht sowohl für iOS als

auch Android zur Verfügung und ermöglicht es, unter anderem per HTTP Verbindungen aufzubauen.

Im Hinblick auf Spieleentwicklung ist dieser Ansatz auch durch den Einsatz von OpenGL eine interessante Möglichkeit, Redundanz im Quelltext zu vermeiden. Nahezu alle Teile des Quelltextes lassen sich wiederverwenden.

Kriterium	**erfüllt**	**Begründung**
1 UI	Nein	UI ist nicht plattformgetreu
2 Logik	Ja	Anwendungslogik wird wiederverwendet
3 Tooling	Ja	Keine Technologiegrenzen vorhanden
4a Anzahl Abh.	Teilweise	Je nach Einsatz von zusätzlichen Bibliotheken
4b Grad Abh.	Ja	Quelltext zu jeder Zeit lesbar

3.2.4 Ansatz 4: C / C++ / JS mit nativer UI

Dieser Ansatz ähnelt dem dritten Ansatz insofern, dass für die Geschäftslogik die Programmiersprache C++ verwendet werden kann. Wie bereits oben beschrieben kann somit der Quelltext auf beiden Plattformen genutzt werden. Anstatt C++ kann für die Logik auch JavaScript verwendet werden, da auch diese Sprache auf beiden ausgeführt werden kann. Unter iOS steht hierzu seit iOS 7 das Framework *JavaScriptCore* zur Verfügung. Die Benutzeroberfläche wird bei diesem Ansatz jeweils in nativem Quelltext geschrieben.

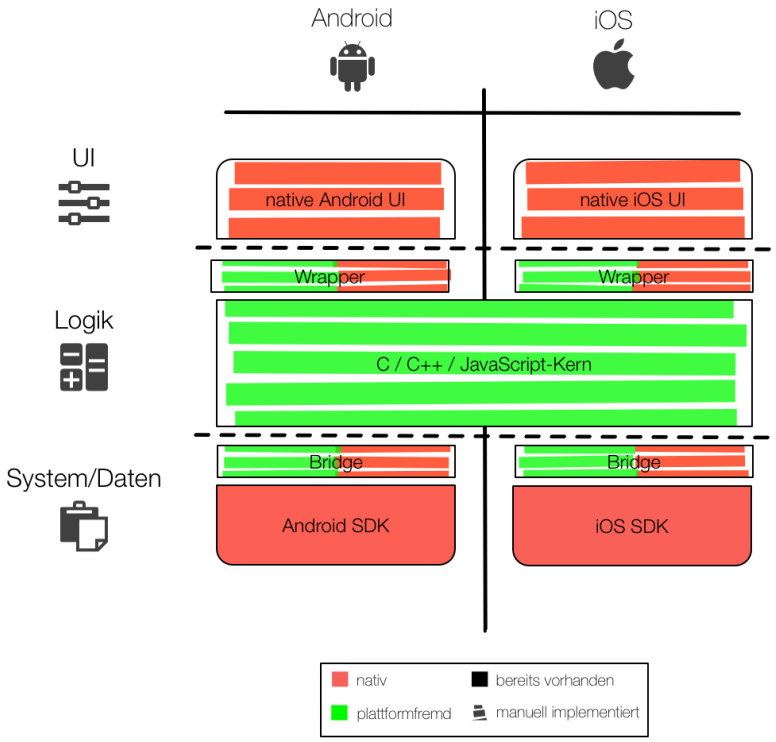

Abbildung 11: nativ gemischter Ansatz

Kriterium	erfüllt	Begründung
1 UI	Ja	Native UI
2 Logik	Ja	Anwendungslogik wird wiederverwendet
3 Tooling	Nein	Keine Tools verfügbar, Grenzen durch JS/C/C++
4a Anzahl Abh.	Ja	Je nach Einsatz von zusätzlichen Bibliotheken
4b Grad Abh.	Ja	Je nach Einsatz von zusätzlichen Bibliotheken

4 Lösungsansatz

In diesem Kapitel wird das Cross-Plattform App-Entwicklungs-Konzept vorgestellt. Es verwendet den im Kapitel 2.2.2 vorgestellten Transcompiler j2ObjC und eine komponentenbasierte Architektur. Im ersten Teil des Kapitels wird erläutert, wie sich der Transcompiler konzeptionell für die mobilen Plattformen nutzen lässt. Im zweiten Teil des Kapitels wird die Architektur und Umgebung beschrieben, in welcher der gewählte Ansatz funktionieren soll. Im dritten Teil des Kapitels wird das Konzept des Cross-Translator beschrieben, mit dem die mehrfache Implementierung von Quelltext vermieden werden soll.

4.1 Grafische Benutzeroberfläche

Wie in Kapitel 2.1.1 erläutert, entsteht bei der nativen Entwicklung einer Benutzeroberfläche ein Mehraufwand bei der Implementierung, wenn man die App für unterschiedliche Plattformen bereitstellen will.

Um dem Benutzer der App aber eine möglichst intuitive und flüssige Bedienung der Benutzeroberfläche auf den jeweiligen Plattformen zu ermöglichen, sollen alle Teile der grafischen Benutzeroberfläche in nativem Quelltext geschrieben werden.

Darüber hinaus hat dieser Ansatz den Vorteil, dass alle Interface-Konzepte von iOS bzw. Android ohne Einschränkung umgesetzt werden können. Ein webbasierter oder hybrider Ansatz kann dieser Forderung nur eingeschränkt gerecht werden, da hier für die Erstellung der Benutzeroberflächen keine nativen UI-Elemente verwendet werden können.

4.2 Anforderungen an die Architektur

Durch eine geeignete Architektur soll es möglich sein, den fachlichen vom technischen Code zu trennen. Um dieses Ziel zu erreichen, soll eine komponentenbasierte Drei-Schichten-Architektur umgesetzt werden.

4.2.1 Drei-Schichten-Architektur

Die einzelnen Schichten haben keinerlei Implementierungswissen von anderen Schichten, sondern kommunizieren mit jenen ausschließlich über öffentliche Schnittstellen. Außerdem darf jede Schicht nur die öffentliche Schnittstelle der jeweils darunter liegenden Schicht kennen. Muss eine Schicht mit der Nächsthöheren kommunizieren, kann beispielsweise das *Listener-Muster* angewandt werden. Diese strikte Schichtentrennung gewährleistet die Austauschbarkeit einzelner Komponenten. Um die lose Kopplung zwischen den einzelnen Komponenten umzusetzen und dabei den Zugriff auf selbige möglichst komfortabel zu ermöglichen, soll hier das Prinzip *Dependency Injection* umgesetzt werden.

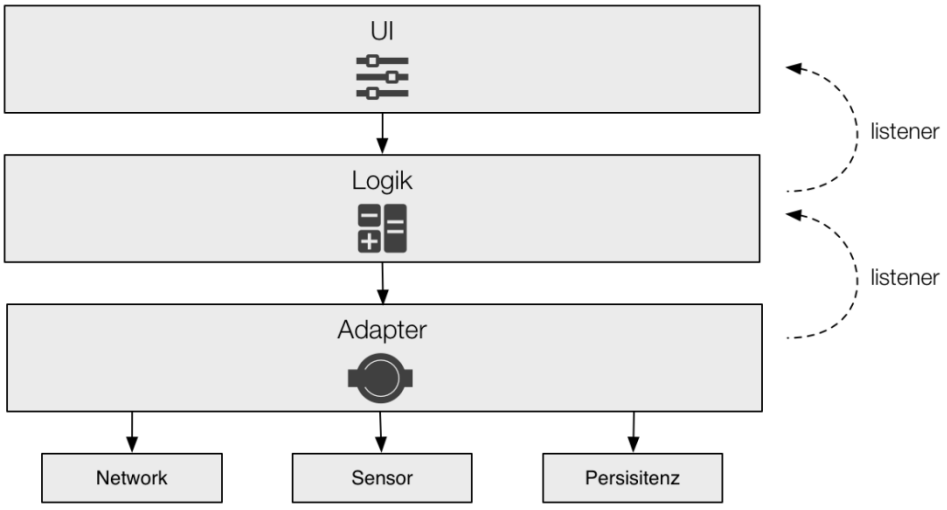

Abbildung 12: Drei-Schichten-Architektur

In der obersten Schicht wird die Benutzeroberfläche gesteuert. Es können Daten in nativen UI-Widgets dargestellt werden und es kann auf Nutzereingaben reagiert werden. Die UI-Schicht hält keinerlei Daten, sie bildet nur die Daten ab, die in der Logikschicht gespeichert sind. Über Zustandsänderungen wird die Benutzeroberfläche von der Logikschicht informiert.

In der darunterliegenden Logikschicht werden Anwendungslogik definiert und fachliche Prozesse implementiert. Hier werden außerdem sämtliche Daten während der Laufzeit gehalten und bei Bedarf für die UI-Schicht zur Verfügung gestellt. Da die hier definierten Komponenten möglichst komplett übersetzt werden sollen, muss deren Quelltext frei von technischen Abhängigkeiten sein.

In der untersten Schicht, der Adapterschicht, sind technische Abhängigkeiten erlaubt. Hier kann beispielsweise auf Gerätehardware zugegriffen, eine Persistenz angebunden oder Netzwerkfunktionalität zur Anbindung eines *Backends* verwendet werden. Da in dieser Schicht keine Daten gespeichert werden dürfen, müssen diese direkt an die Logikschicht weitergereicht werden. Hierfür können *Daten-Transfer-Objekte* (DTO) verwendet werden. Die komponentenbasierte Architektur ermöglicht, dass auch Teile der Adapterschicht übersetzbar sind, sofern diese keine plattformabhängigen Funktionen implementieren.

4.2.2 Zugriff auf Systemfunktionen durch Abstraktionskomponenten

Um auch Funktionen innerhalb der App zu nutzen, die nur durch plattformabhängigen Quelltext implementiert werden können, besteht die Möglichkeit, Schnittstellen zu definieren, welche von der jeweiligen API des SDKs auf eine gemeinsame abstrahieren. Implementiert man die Schnittstelle auf den jeweiligen Plattformen, ist es möglich, die plattformspezifische Hardware zu benutzen. Da die Implementierungen der Komponenten plattformabhängigen Quelltext enthalten, können diese nicht übersetzt werden. Weil aber die Schnittstellen bereits vorher definiert wurden, kann auch übersetzter Code auf die Funktionalität in den Abstraktionskomponenten zugreifen. Prinzipiell lassen sich so für alle benötigten Systemfunktionen Schnittstellen definieren. Einmal implementierte Komponenten lassen sich bei weiteren Projekten wiederverwenden. Sollen Abstraktionskomponenten mit neu hinzukommender Funktionalität erweitert werden, müssen die gemeinsamen Schnittstellen angepasst und die jeweiligen Implementierungen aktualisiert werden.

4.3 Cross-Translator

Um den Mehraufwand, der bei der nativen Entwicklung durch mehrfaches Implementieren entsteht, zu reduzieren, soll Quelltext wiederverwendet werden. Dies soll durch den Einsatz eines Transcompilers geschehen.

Es soll ein Transcompiler eingesetzt werden, der die gesamte Logikschicht sowie plattformunabhängige Komponenten der Adapterschicht automatisch übersetzt. Da hier die Betriebssysteme iOS und Android betrachtet werden, muss der eingesetzte Transcompiler entweder Java-Quelltext in Objective-C Quelltext überführen oder anders herum.

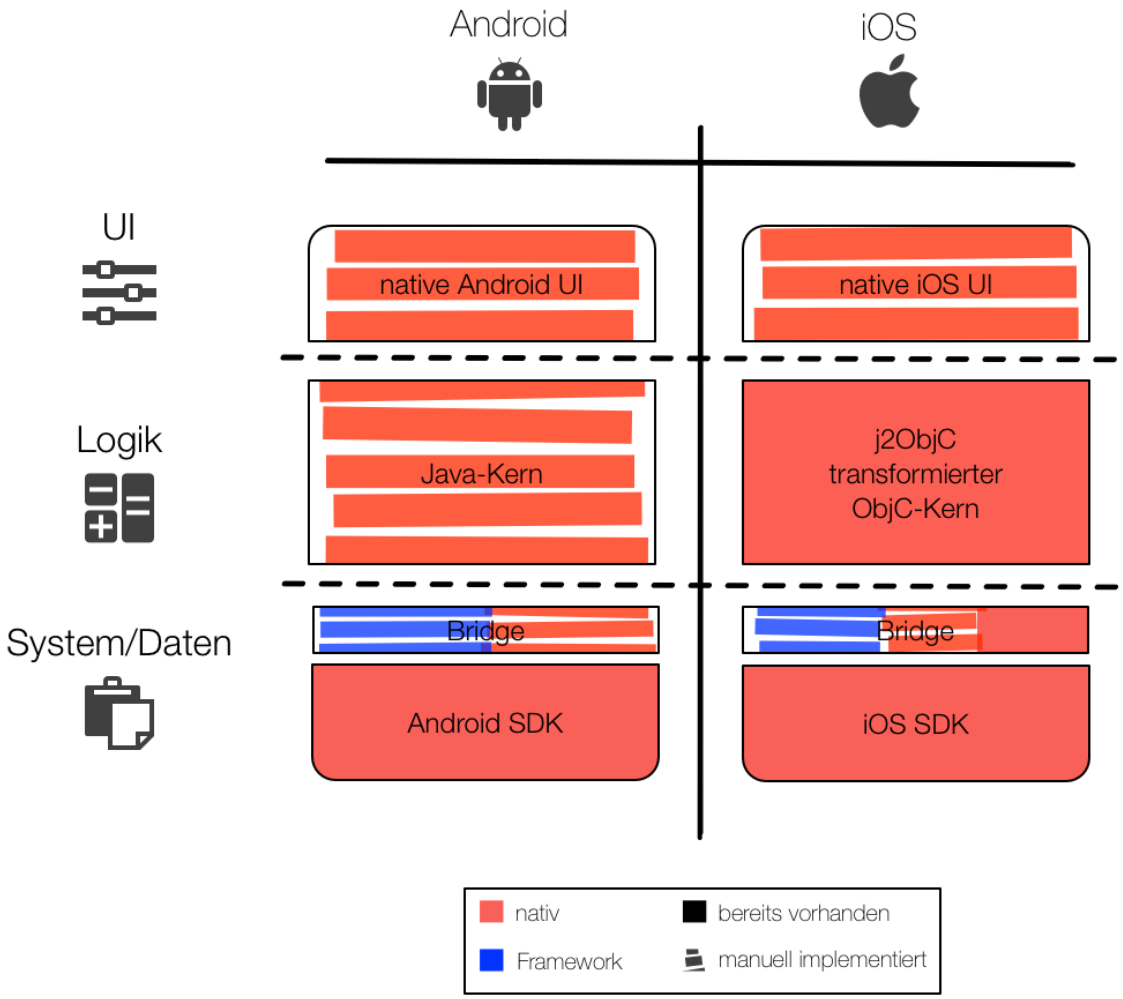

Abbildung 13: Cross-Translator Ansatz

Wie bereits beschrieben, kann Quelltext, der Abhängigkeiten zum Android SDK hat, nicht mit j2ObjC übersetzt werden. Quelltext, der frei von technischen Abhängigkeiten ist, wie zum Beispiel Quelltext, der fachliche Geschäftslogik implementiert, lässt sich durch j2ObjC von Java-Quelltext in Objective-C-Quelltext transformieren.

5 Vorstellung der Implementierung

In diesem Kapitel wird die Implementierung vorgestellt, die nötig war, um das entwickelte Konzept zu evaluieren. In Kapitel 5.1 wird die Implementierung der Tool-Chain vorgestellt, die für den Ansatz entwickelt wurde. In Kapitel 5.2 wird ein Framework behandelt, das entwickelt wurde, um die mit der Tool-Chain übersetzten Komponenten komfortabel in eine iOS-App zu integrieren. Abschließend behandelt das Kapitel 5.3 die prototypische Implementierung eines Anwendungsfalls, welcher die vorher entwickelten Werkzeuge und Frameworks nutzt.

5.1 Implementierung Tool-Chain

Für die Übersetzung des Quelltextes wurde der Transcompiler j2ObjC verwendet, dessen grundlegende Funktionsweise im Kapitel 2.2.2 vorgestellt wurde.

Damit die im Entwicklungsprozess benötigten Werkzeuge komfortabel vom Entwickler eingesetzt werden können, werden diese zu einer Tool-Chain zusammengefasst. Es soll möglich sein, die automatische Übersetzung der plattformunabhängigen Komponenten zu jedem beliebigen Zeitpunkt auszuführen.

5.1.1 Architektur XTranslator

Die Tool-Chain wurde in Hinblick auf Wiederverwendbarkeit und Erweiterbarkeit so entworfen, dass die Konfiguration unabhängig von der Übersetzung erfolgt. Das *XTranslator Command Line Interface* (XTranslator CLI) ist für die Verarbeitung der Konfigurationsdateien verantwortlich. Die *XTranslator Library* (XTranslator LIB) ist für die Bearbeitung und Transformation des Quelltextes zuständig.

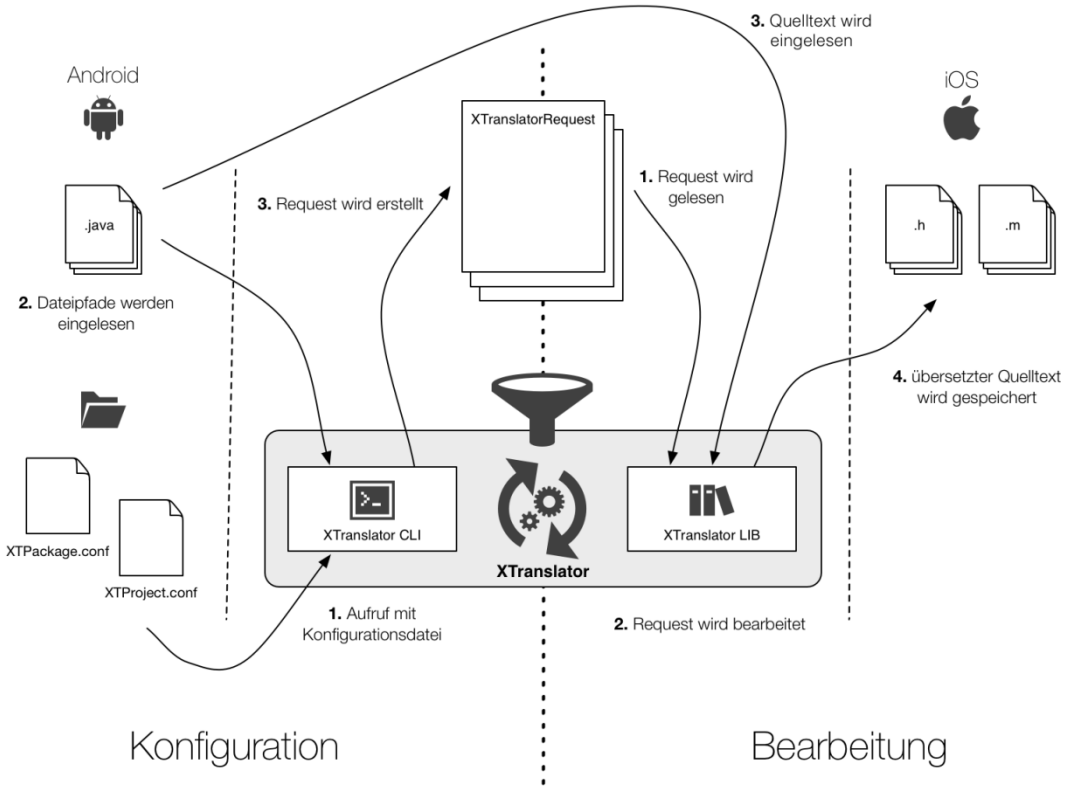

Abbildung 14: Funktionsweise XTranslator

- Konfiguration

  Das XTranslator CLI wird per Kommandozeile mit einer `XTProject.conf`-Datei als Parameter aufgerufen. Die enthaltenen Informationen werden interpretiert und führen zu den Java-Quelltext-Dateien, die übersetzt werden sollen. Für jede zu übersetzende Quelle wird ein `XTranslatorRequest`-Objekt erstellt, das alle Informationen enthält, die für die anschließende Verarbeitung benötigt werden.

- Bearbeitung

  Der Bilbiothek XTranslatorLIB werden die zu verarbeitenden `XTranslatorRequest`-Objekte übergeben. Im Laufe der Interpretation und Verarbeitung der Informationen wird die Java-Quelltext-Datei eingelesen, nachbearbeitet, übersetzt und das entstandene Objective-C Pendants im vorab im Request hinterlegten Pfad gespeichert.

Durch die Trennung lässt sich so jeder `XTranslatorRequest` einzeln übersetzen was eine Parallelisierung möglich macht. Außerdem kann so das Konfigurationsinterface leicht durch eine grafische Benutzeroberfläche ausgetauscht werden.

5.1.2 XTranslator Library

Die XTranslator Bibliothek beinhaltet die Hauptfunktionalität der Tool-Chain. Sie nimmt die zu übersetzenden `XTranslatorRequests` entgegen, überprüft Pfade, parst den Java-Quelltext und führt den `j2objc` Befehl in der Systemumgebung aus. Nachdem der `j2objc` Befehl ausgeführt wurde, werden die entstandenen Objective-C Dateien nachbearbeitet und am gewünschten Pfad abgespeichert.

5.1.2.1 Schnittstellen

Die Bibliothek veröffentlicht zwei Schnittstellen nach außen: zum einen die Schnittstelle `XTranslator` und zum zweiten die Schnittstelle `XTranslatorRequest`.

Die Klasse `XTranslatorRequest` bündelt alle Daten, die für die Übersetzung einer Java-Klasse benötigt werden. Ein `XTranslatorRequest` Objekt wird über den Konstruktor `XTranslatorRequest(…)` instanziert. Alle Parameter sind absolute Pfade, die als Strings übergeben werden. Folgende Parameter werden benötigt:

- Der Pfad der zu übersetzenden Java-Quelltext Datei
- Der Zielpfad, in dem die übersetzten Objective-C Dateien gespeichert werden
- Der Pfad zum Quellen-Hauptverzeichnis des dazugehörigen Projekts
- Der Pfad zum j2objc Kommandozeilen Tool

Besitzt die Java-Quelltext-Datei beispielsweise Abhängigkeiten zu anderen Projekten, so lassen sich solche optionalen Parameter nach der Instanziierung über *setter-Methoden* setzen.

Außerdem ist es möglich, den `j2objc` Aufruf, je nach Anforderungen des jeweiligen zu übersetzenden Projekts durch benutzerdefinierte Parameter zu erweitern, mit denen das Java Objekt übersetzt werden soll. Die zusätzlichen Optionen werden als String übergeben.

Die Schnittstelle `XTranslator` stellt Methoden zur Verarbeitung von `XTranslatorRequest`-Objekten zur Verfügung.
Zum Übersetzen von `XTranslatorRequests` existiert die Methode `translateRequests(…)`. Sie nimmt einen oder mehrere `XTranslatorRequest` Objekte entgegen und übergibt sie an die für die Übersetzung verantwortliche Zuständigkeitskette.

Um die im Rahmen einer Übersetzung erstellten Objective-C Dateien zu löschen, stellt die Schnittstelle die Methode `cleanRequests(…)` zur Verfügung.

5.1.2.2 Chain of Responsibility

Die Algorithmen zur Verarbeitung der `XTranslatorRequest` Objekte wurden mit Hilfe des Entwurfsmusters „Chain of Responsibility“ implementiert. Wie im Kapitel 2.2.1.1 Chain Of Responsibility beschrieben, gibt es innerhalb des Entwurfsmuster drei unterschiedliche Rollen:

- Client

  Die Klasse `XTranslator` nimmt die Rolle des Clients ein. Sie initiiert den Aufruf an die Zuständigkeitskette.

- Request

  Die Klasse `XTranslatorRequest` nimmt die Rolle des `Requests` ein. Eine Instanz wird vom Client an die Zuständigkeitskette übergeben und durch die Kette gereicht.

- RequestHandler

  Die abstrakte Klasse `XTranslatorChainCommand` beschreibt die Schnittstelle, die alle Kettenglieder implementieren. Sie stellt drei Methoden zur Verfügung:

  Die Methode `setNextLink` dient zum Aufbau der Kette und setzt den direkten Nachfolger des Kettenglieds. Ihr wird eine Instanz des Typs `XTranslatorChainCommand` übergeben.

  Die Methode `executeNextLink(XTranslatorRequest request)` gibt den Request an das nächste Kettenglied weiter und ruft dessen `handleRequest`-Methode auf.

Die Methode `handleRequest(XTranslatorRequest request)` ist eine abstrakte Methode, die von den konkreten `ChainCommands` implementiert wird. Sie beinhaltet die Logik, wie der `Request` verarbeitet werden soll. Ist die Bearbeitung des `Requests` innerhalb eines `ChainCommands` abgeschlossen, sollte am Ende der `handleRequest`-Methode die `executeNextLink`-Methode aufgerufen werden.

5.1.2.3 ChainCommands

ChainCommands sind konkrete Implementierungen der Klasse `XTranslatorChainCommand`. Die folgenden ChainCommands wurden implementiert:

5.1.2.3.1 JavaFileReaderChainCommand

Das `JavaFileReaderCommand` überprüft, ob die Java-Quelltext Datei an dem im `XTranslatorRequest` hinterlegten Pfad existiert. Falls die Datei nicht lesbar ist oder es sich nicht um eine Datei handelt, die mit dem Suffix `.java` endet, wirft das Command eine `XTranslatorChainException`, wodurch die Verarbeitung abgebrochen wird. Ansonsten wird der Inhalt der Datei in einen String eingelesen und mit Hilfe des Setters `setJavaString()` im `XTranslatorRequest` Objekt gespeichert. Darüber hinaus wird der Name des Objekts im Request gespeichert.

5.1.2.3.2 JavaFileParserChainCommand

Das `JavaFileParserChainCommand` parst den Inhalt der Java-Quelltext Datei. Es durchsucht den Quelltext nach `import` und `package` Anweisungen. Diese werden dann in *CamelCase* Schreibweise abgespeichert. Die erstellten Strings entsprechen den von `j2objc` während dem Umbenennen der Klassennamen hinzugefügten Präfixen und können zu einem späteren Zeitpunkt aus dem Objective-C Code gelöscht werden, um die Änderungen des Namings zu widerrufen.

5.1.2.3.3 j2ObjCExecuteChainCommand

Das `j2ObjCExecuteChainCommand` extrahiert alle für den Aufruf notwendigen Informationen aus dem `XTranslatorRequest`, führt das `j2objc` Kommando in der Systemumgebung des Betriebssystems aus und speichert die Kommandozeilenausgabe des Aufrufs zu Debugging Zwecken im `XTranslatorRequest` in Form eines Strings ab.

5.1.2.3.4 ObjCFileReaderCommand

Analog zum `JavaFileReaderChainCommand` überprüft das `ObjCFileReaderCommand`, ob sich am im `XTranslatorRequest` gespeicherten output Pfad lesbare Dateien mit dem Präfix des Objektnamens und der Endung .m und .h an den gegebenen Pfaden existieren. Diese Dateien sind die im Laufe der Übersetzung entstandenen Implementierungs- und Header- Dateien. Der Inhalt dieser Dateien wird eingelesen und jeweils als String im Request-Objekt gespeichert.

5.1.2.3.5 ObjCFileParserCommand

Das `ObjCFileParserCommand` parst den Inhalt der Objective-C Quelltextdateien. Es durchsucht den Quelltext nach den vom `JavaFileParserCommand` im Schritt 5.1.2.3.2 gefundenen Präfixen und löscht die Substrings, die den Präfixen entsprechen. Somit sind am Ende des Command Aufrufs die j2ObjC Änderungen am Naming wieder rückgängig gemacht worden.

5.1.2.3.6 ObjCFileWriterCommand

Das `ObjCFileWriterCommand` schreibt die veränderten Strings in die jeweilige Objective-C Datei zurück.

5.1.2.3.7 RequestPrinterCommand

Im `RequestPrinterCommand` wird ein Statusbericht des bearbeiteten `XTranslatorRequest` Objekts auf der Kommandozeile ausgegeben. Das `RequestPrinterCommand` bildet das letzte Glied der Kette.

5.1.3 XTranslator Command Line Interface

Um mit der XTranslator Bibliothek `XTranslatorRequests` zu übersetzen, wurde ein Kommandozeilen-Tool implementiert. Es wird mit einer Java-Projekt-Konfigurationsdatei als Parameter aufgerufen. Das Tool durchsucht das Dateisystem nach Java-Dateien und erstellt für jede Java-Datei einen `XTranslatorRequest`. Diese Requests werden anschließend an die XTranslator Bibliothek übergeben, um die Übersetzung zu starten.

Um das Command Line Interface nutzen zu können, muss mit der `-D` Option des `java` Kommandos die Systemeigenschaft `j2objc.path` auf den Pfad des j2ObjC Release Verzeichnis gesetzt werden.

```
java -Dj2objc.path=/j2objc-0.9.3 -jar xtranslate.jar XTProject.conf
```

5.1.3.1.1 Hierarchie und Aufbau der Konfigurationsdateien

Jedes zu übersetzende Android-Projekt muss über eine Projektkonfigurationsdatei mit dem Namen `XTProject.conf` verfügen. Diese Konfigurationsdatei gilt projektglobal und kann folgende Eigenschaften enthalten:

```
useXTranslator=true
outputPath=../../../Desktop/output
packages=de.maibornwolff.foo.bar,de.maibornwolff.keine,de.maibornwolff.projek
tabhaengigkeit
projectSourcePath=src
dependenciesSourcePaths=../TestProject1/src,../TestProject2/src
j2ObjCOptions=-use-arc
```

Quelltext 4: XTProject.conf

Pfade können immer entweder als absolute oder als relative Pfade angegeben werden. Die relativen Pfade beziehen sich dabei immer auf den Ort der Projektkonfigurationsdatei.

- `useXTranslator`

  Der Wert kann entweder `true` oder `false` sein und entscheidet, ob die Datei bearbeitet werden soll. So lassen sich einzelne Projekte temporär von der Übersetzung ausschließen.

- `outputPath`

  Dieser Pfad beschreibt den Ort, an welchen alle übersetzten Dateien standardmäßig gespeichert werden sollen.

- `projectSourcePath`

  Dieser Parameter beschreibt den Pfad des Wurzelverzeichnisses der Quellen des Projektes. In diesem Verzeichnis befinden sich die Pakete, die übersetzt werden können.

- `packages`

  Die hier aufgeführten Pakete werden übersetzt. Soll mehr als ein Paket übersetzt werden, sind die Paketnamen durch Kommas getrennt. Es können auch Paketnamen mit Wildcard (*) angegeben werden. Alle Unterpakete werden dann automatisch mit übersetzt.

- `dependenciesSourcePaths`

  Benötigt das Projekt andere Projekte, kann man mit dieser Option weitere Quell-Pfade hinzufügen. Gibt es mehr als eine Abhängigkeit, werden die Pfade mit Kommas getrennt.

- `j2ObjCOptions`

  Um dem Transcompiler benutzerdefinierte Parameter zu übergeben, kann man mit Hilfe dieser Option die Parameter setzen. Um beispielsweise Objective-C Dateien zu erzeugen, die Automatic Reference Counting verwenden, gibt man an dieser Stelle den `j2objc` Parameter `-use-arc` mit an.

```
outputPath=../../iOS/DemoApp/DemoApp/AComponent/GeneratedCode/src
j2ObjCOptions=-use-arc
```

Quelltext 5: XTPackage.conf

Die Einstellungen in der XTProject.conf gelten projektglobal. Will man für einzelne Pakete oder Paketbäume abweichende Optionen verwenden als die globalen, kann man mit Hilfe der `XTPackage.conf` Optionen überschreiben. Hierfür wird im Verzeichnis des entsprechenden Pakets eine Paketkonfigurationsdatei mit dem Namen `XTPackage.conf` erstellt. Die Optionen `outputPath` und `j2ObjCOptions` können dadurch überschrieben werden und werden außerdem an alle Unterpakete vererbt. Existieren in den

Unterpaketen weitere `XTPackage.conf` Dateien, gelten die dort definierten Optionen analog.

5.1.3.1.2 ConfigLoader

Die Klassen `ProjectConfigLoader` und `PackageConfigLoader` wurden implementiert, um die Hierarchie der Konfigurationsdateien zu laden. Dazu werden im Laufe der Verarbeitung Objekte der Klassen `XTProjectConfig` und `XTPackageConfig` erstellt. Ist die Verarbeitung abgeschlossen, beinhalten diese Objekte alle Informationen, die notwendig sind, um die `XTranslatorRequests` zu erstellen.

Der `ProjectConfigLoader` hat die Methode `loadXTProjectConfigFromFile()`, um eine `XTProject.conf` in ein `XTProjectConfig` Objekt zu transformieren. Alle relativen Pfade, die in der Konfigurationsdatei enthalten sind, werden in absolute Pfade umgewandelt. Die `packages` und `dependenciesSourcePaths` werden jeweils in einem *Array* gespeichert.

Der `PackageConfigLoader` hat die Methode `loadXTPackageConfigs()`, die mit Hilfe der ihr übergebenen `XTProjectConfig` Instanz eine Liste von `XTPackageConfig` Objekten erstellt.

Dies geschieht mit einem rekursiven Aufruf einer privaten Methode, welche die Pakete nach `XTPackage.conf` Dateien durchsucht und die darin enthaltenen Optionen für alle weiteren Unterpakete übernimmt.

Die entstehenden `XTPackageConfig` Objekte beinhalten am Ende des Aufrufs alle Pfade zu den Java-Dateien, die innerhalb des Paketes übersetzt werden sollen.

5.2 Integration in iOS App

Da der Komponentenschnitt der App auf iOS und Android-Seite identisch ist und die Abstraktionskomponenten plattformübergreifend die gleichen Schnittstellen anbieten, können die nachfolgenden Komponenten per Dependency Injection verbunden werden.

5.2.1 Abstraktionskomponenten

Damit technische Schnittstellen der SDKs plattformunabhängig genutzt werden können, wurden die drei folgenden Abstraktionskomponenten implementiert:

- DICNetworkManager
  Die Komponente ermöglicht es, http-Anfragen an eine URL zu stellen. Die Antwort auf den Request kann entweder direkt oder über den DICNetworkListener beim Empfänger ankommen.

- DICSensorManager

  Durch diese Abstraktionskomponente kann der Neigungssensor des Smartphones angesprochen werden. Neben Schnittstellen für Neigungswerte bietet der DICSensorManager Schnittstellen für die Abfrage von Beschleunigung und Orientierung des Gerätes.

- DICLocationManager
  Der DICLocationManager abstrahiert die GPS-Sensor API. Bereitgestellt wird sowohl eine synchrone Schnittstelle für die einmalige Anfrage der GPS-Position, als auch die Schnittstelle DICLocationListener, die eine fortwährende Übertragung neuer Positionen ermöglicht.

5.2.2 Dependency Injection mit DIComponents

Um die Verknüpfung der Komponenten per Dependency Injection zu ermöglichen, wurde für Android ein bereits im Unternehmen vorhandenes Komponenten-Framework verwendet. Damit die übersetzten Java-Komponenten in die iOS App integriert werden können, wurden Teile des bestehenden Frameworks prototypisch portiert.

Die Singleton-Instanz der Klasse `DIComponents` steht über eine statische Methode applikationsübergreifend zur Verfügung. Komponenten können mit Hilfe von `plist`-Konfigurationsdateien registriert werden. In den Konfigurationsdateien sind die Bindings der Interfaces auf die jeweiligen Implementierungen definiert.

```
@protocol DICF <NSObject>

- (void)setConfigurationForComponentWithName:(NSString *)name
                            andConfiguration:(NSDictionary *)config;

- (id<SubComponent>)get:(id)classOrProtocol;

- (NSSet *)getAll:(id)classOrProtocol;

@end
```

Quelltext 6: DICF.h

Die Schnittstelle `DICF` definiert die Methode `get:(id)classOrProtocol`, mit der Instanzen zu einer ihr übergebenen Schnittstelle oder Klasse angefordert werden können. Sie wird von der Klasse `DIComponents` realisiert.

```
@protocol DICF;

@protocol CFObject <NSObject>

- (void)setDependenciesWithCF:(id<DICF>)componentContainer;

@end
```

Quelltext 7: CFObject.h

Klassen, welche Dependency Injection verwenden wollen, realisieren die Schnittstelle `CFObject`. Im Rumpf der Methode `setDependenciesWithCF` definiert man die Abhängigkeiten, die in die Instanz injiziert werden sollen.

Wird eine benötigte Schnittstelle bei der `DIComponents`-Instanz angefragt, wird überprüft ob schon eine Instanz für die angefragte Schnittstelle zur Verfügung steht. Ist diese bereits verfügbar, wird sie zurückgegeben. Ist dies nicht der Fall, wird die zugehörige neue Instanz mit Hilfe des vorher konfigurierten Mappings erzeugt. Bevor die Instanz zurückgegeben werden kann, wird noch die Methode `setDependenciesWithCF` aufgerufen, um eventuelle Abhängigkeiten des neuen Objekts aufzulösen.

5.3 Prototypische Implementierung iOS-App

Um den im Rahmen dieser Arbeit entwickelten Ansatz zu evaluieren, wurden die erarbeiteten Konzepte in einer mobilen App umgesetzt.

5.3.1 Usecase CityChat

Die CityChat-App bietet dem Anwender die Möglichkeit, sich mit anderen CityChat-App-Benutzern zu unterhalten. Durch eine integrierte Kartenfunktion kann man Chatpartner in der unmittelbaren Umgebung finden und eine Unterhaltung beginnen.

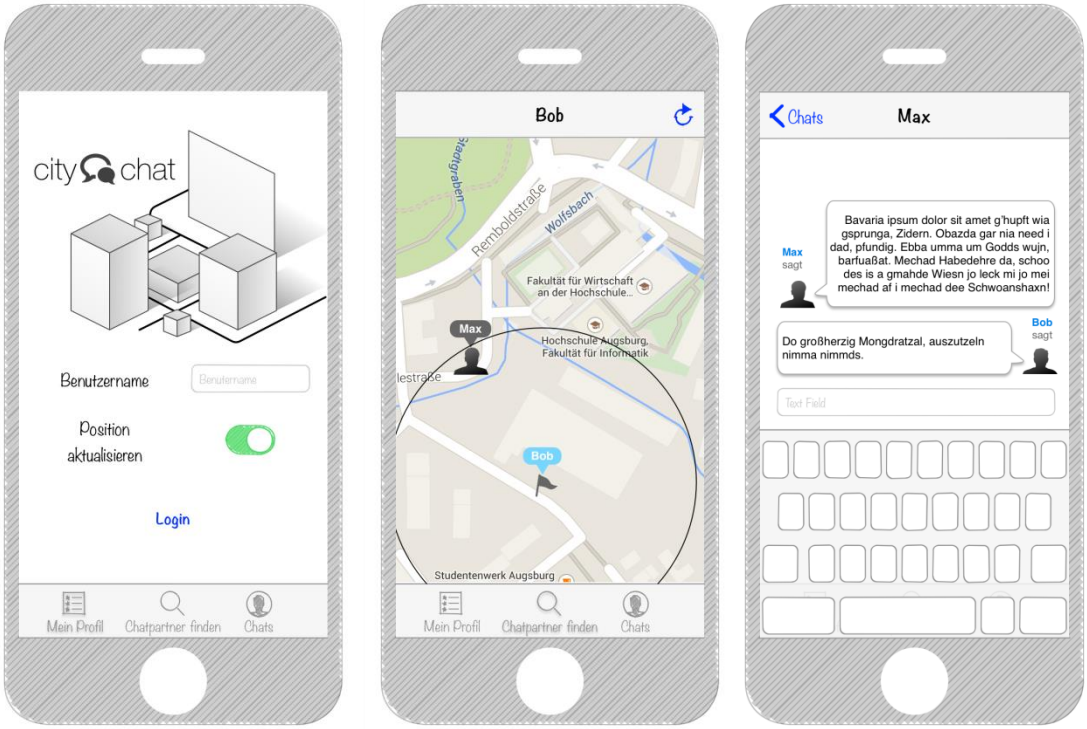

Abbildung 15: Wireframes CityChat

5.3.2 Vorgehensweise

Für die Nutzung des GPS-Sensors und für die Netzwerkkommunikation mit dem Backend wurden wiederverwendbare Komponenten implementiert, die den jeweiligen Teil der Android / iOS API abstrahieren. Für die plattformunabhängigen Schnittstellen wurden Schnittstellen in Java definiert und mit der XTranslator-Tool-Chain übersetzt. Auf iOS-

Seite wurden die Schnittstellen mit Hilfe der iOS APIs `CoreLocation` und `NSURLConnection` implementiert.

Für die Backend-Kommunikation wurde *JSON* als Austausch-Format gewählt. Um JSON Objekte innerhalb der App plattformunabhängig zu verarbeiten, wurde die quelloffene JavaME-Bibliothek *org.json.me* mit der Tool-Chain transformiert.

Die komplette Programmlogik der App wurde in Java programmiert und per XTranslator-Tool-Chain übersetzt.

5.3.3 Architektur

Für die Architektur der App wurde die in Kapitel 4.2.1 vorgestellte 3-Schichten-Architektur umgesetzt. Die gelb markierten Komponenten sollen von Java nach Objective-C übersetzt werden können.

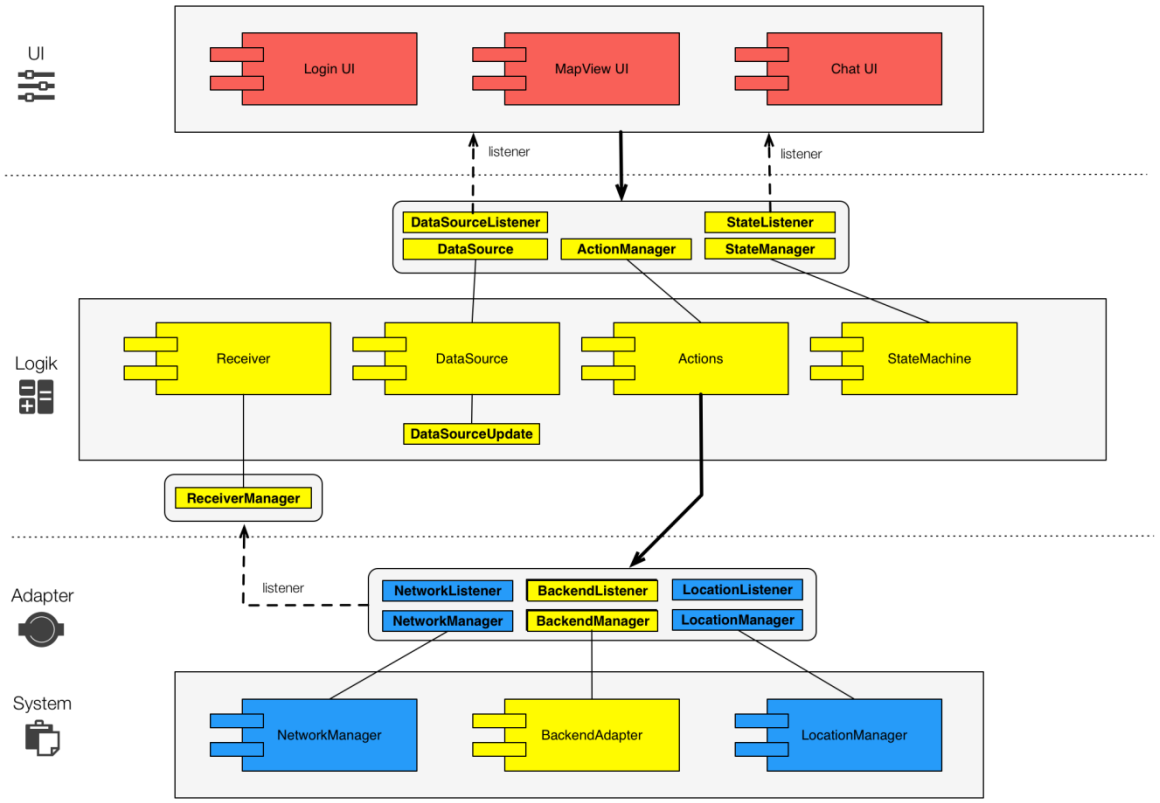

Abbildung 16: Architektur CityChat

- UI-Schicht

Die UI führt Aktionen in der Logik-Schicht über die Action-Komponente aus. Sie registriert sich bei der `DataSource`-Komponente als `DataSourceListener` und holt sich bei Benachrichtigung über Änderungen diese bei der Komponente ab. Über Änderungen von Programm-Zuständen wird die Benutzeroberfläche von der `StateMachine`-Komponente per Listener informiert.

- Logik-Schicht

Die `DataSource`-Komponente hält das Model der App. Das Model besteht aus den Klassen User, Chat und Message. Sie stellt die Schnittstellen `DataSource`, `DataSourceUpdate` und `DataSourceListener` zur Verfügung.

Die Action-Komponente bietet die Schnittstelle `ActionManager` an und ist für die Ausführung von Aktionen der Benutzeroberfläche zuständig. Sie ändert die Zustände der `StateMachine`-Komponente und steuert den Ablauf des Programms.

Die `StateMachine`-Komponente bietet die Schnittstellen `StateMachineManager` und `StateMachineListener` an. Sie hält die Zustände in der sich die App befindet und informiert ihre Listener über deren Änderungen.

Die `Receiver`-Komponente empfängt die Änderungen vom `BackendAdapter` über die Schnittstelle `BackendAdapterListener` und ändert als einzige Komponente das Model über die Schnittstelle `DataSourceUpdate`. Zustandsänderungen erfährt die Komponente von der `StateMachine` per Listener.

- Adapter-Schicht

  Die `BackendAdapter`-Komponente kennt die API des JSON-Backends und kommuniziert mit Hilfe der Abstraktionskomponente `NetworkManager` mit der Serverinstanz.

- Backend

  Als Backend wird eine MarkLogic (NoSQL) Datenbank Instanz verwendet. Alle benötigten Backend-Methoden wurden in XQuery prototypisch implementiert und stehen als Webservice bereit. Die entstehenden Daten werden in Form einer XML Struktur auf dem Server persistiert und beim Zugriff serverseitig in JSON transformiert.

6 Evaluation

Im nachfolgenden Kapitel werden die Ergebnisse der Implementierung vorgestellt und anschließend bewertet.

6.1 Ergebnisse

Im Rahmen der prototypischen Implementierung des Anwendungsfalles unter Benutzung der erarbeiteten Tool-Chain konnte die Geschäftslogik und die Ablaufsteuerung der CityChat-Anwendung von Java nach Objective-C übersetzt werden und die automatisch erstellten Klassen in das iOS Projekt integriert werden. Die nachfolgenden Ergebnisse konnten erzielt werden:

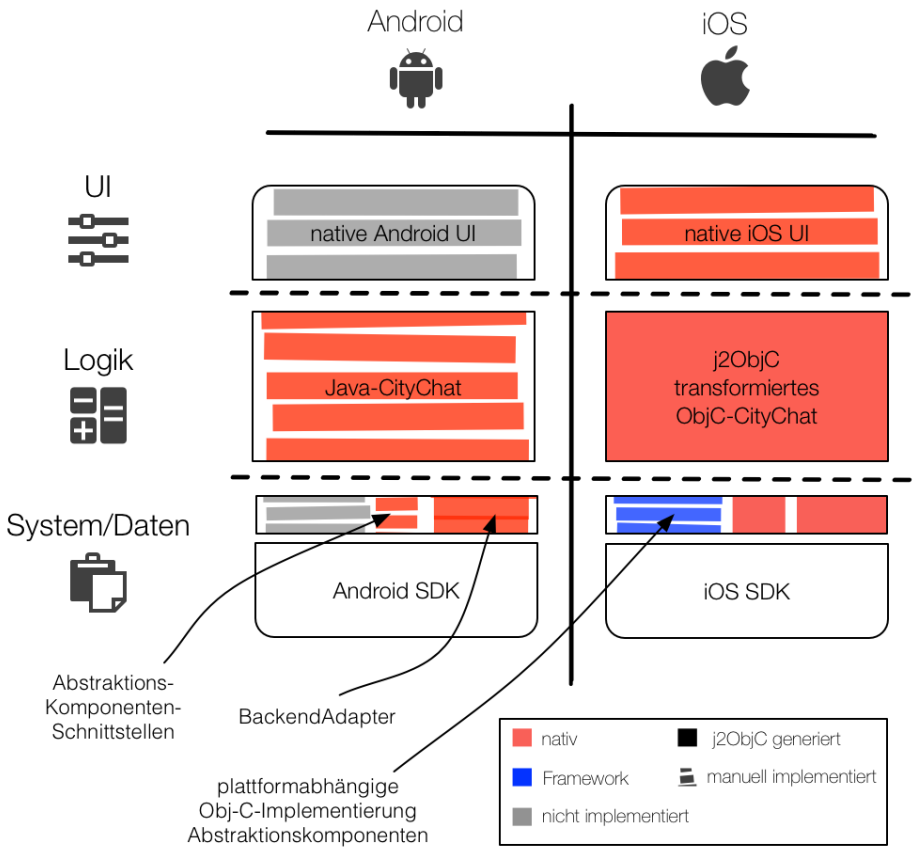

Abbildung 17: Übersetzte und implementierte Teile CityChat-App

- Tool-Chain Übersetzung und Integration in iOS-App

  Nach initialer Konfiguration der Tool-Chain mittels Konfigurationsdateien, konnte der XTranslator über die Kommandozeile angesprochen und in den Entwicklungsprozess integriert werden. Die Übersetzung des Quelltextes bedarf keiner weiteren Anpassung und lief voll automatisch. Durch das eingesetzte Dependency Injection Framework konnten die übersetzten Komponenten ohne weitere Anpassungen ausgeführt werden.

- iOS-App CityChat

  Die Benutzeroberfläche des iOS-Projekts konnte über die zuvor definierten Schnittstellen mit den übersetzten Komponenten kommunizieren. Die Abstraktionskomponenten, die in Objective-C implementiert wurden, konnten ohne Einschränkungen in Verbindung mit den übersetzten Komponenten verwendet werden.

  Durch Übersetzen des Java Quelltextes konnten Teile des Quelltextes wiederverwendet werden. Im Prototyp wurde die gesamte Programmlogik sowie der Backend-Adapter übersetzt. Es wurden die in Abbildung 16 gelb markierten Komponenten inklusive Schnittstellen übersetzt. Zusätzlich wurden die Schnittstellen der Abstraktionskomponenten und die JSON-Bibliothek übersetzt.

  Konkret:

  Gelb markierte Bereiche (Abbildung 16):

  896 LoC Java aus 32 Dateien übersetzt zu 2122 LoC Objective-C Quelltext in 64 Dateien (davon 728 LoC in 32 Header-Dateien, 1394 LoC in 32 Objective-C Implementierungen)

  Schnittstellen Abstraktionskomponente:

  108 LoC Java aus 18 Dateien übersetzt zu 439 LoC Objective-C Quelltext in 36 Dateien (davon 199 LoC in 18 Header-Dateien, 240 LoC in 18 Objective-C Implementierungen)

JSON-Bibliothek

1712 LoC Java aus 10 Dateien übersetzt zu 2503 LoC Objective-C Quelltext in 10 Dateien (davon 338 LoC in 10 Header-Dateien, 2165 LoC in 10 Objective-C Implementierungen)

In Relation zu den 733 LoC Code aus 28 Dateien (davon 97 LoC in 14 Header-Dateien, 636 LoC in 14 Objective-C Implementierungen) der UI und Abstraktionskomponenten, die nicht übersetzt wurden, ergibt sich ein Anteil von insgesamt 78,75 Prozent an automatisch generiertem Quelltext.

- JSON-Bibliothek

  Die verwendete Java-Bibliothek org.json.me konnte vollständig mit der Tool-Chain übersetzt werden und innerhalb der App genutzt werden.

- Sonstige

  Der mit j2ObjC übersetzte Quelltext in den generierten Objective-C Dateien ist aufgrund automatischer Formatierung nicht optimal lesbar, gängige Konventionen wie etwa die Cocoa Coding Guidelines werden nicht vollständig eingehalten. Außerdem werden einige Java-Datentypen nicht in das entsprechende Cocoa-Pendant portiert, sondern deren Java-Implementierung wird in Objective-C-Code nachgebildet. Beispielsweise wird aus dem Java-Typ `java.util.ArrayList` bei der Übersetzung nicht der Objective-C-Typ `NSArray` sondern der Objective-C-Typ `JavaUtilArrayList`, welcher wiederum in der mit j2ObjC mitgelieferten Bibliothek *libjre_emul.a* definiert ist.

  Es wurden in den übersetzten Komponenten Techniken angewandt wie beispielsweise *Threading, Anonyme Klassen, Reflection* und *JSON Parsing,* die auf beiden Zielplattformen erwartungsgemäß funktionieren.

6.2 Bewertung

Obwohl für die Umsetzung des Prototyps aus Zeitgründen keine Android-Oberfläche erstellt wurde, beweist die erfolgreiche automatische Portierung der kompletten Anwendungslogik und die funktionale Verknüpfung mit einer nativen iOS-Benutzeroberfläche die Tragfähigkeit des in dieser Arbeit behandelten Konzepts.

Um Quelltext wiederzuverwenden, ist bei diesem Ansatz der Einsatz eines Transcompilers nötig. Der im Rahmen dieser Arbeit verwendete Transcompiler führt zu einer hohen Abhängigkeit zu einem Drittanbieter. Da j2ObjC aber quelloffen ist, könnte man bei Einstellung des Projekts Änderungen an diesem vornehmen, um eventuell auftretenden Problemen zu entgehen. Ist dies aufgrund des Mangels an dafür notwendig einzusetzenden Ressourcen nicht möglich, bestünde immer noch die Möglichkeit, übersetze Komponenten nachträglich nativ zu implementieren. Diese Möglichkeit besteht beim Einsatz proprietärer Lösungen nicht.

Die schlechte Lesbarkeit des übersetzten Quelltextes ist nur im geringen Maße von Bedeutung, da eine nachträgliche Bearbeitung des transformierten Quelltextes nicht Bestandteil des Cross-Plattform-Konzeptes ist.

Für Debugging Zwecke ist die Lesbarkeit des Quelltextes ausreichend. Durch die Verwendung von jeweils nativem Quelltext hat man bei der Auswahl der IDE und eventuell eingesetzten Tools die gleichen Möglichkeiten wie bei einem rein nativen Entwicklungseinsatz.

Kriterium	**erfüllt**	**Begründung**
1 UI	Ja	UI ist plattformgetreu
2 Logik	Ja	Anwendungslogik wird wiederverwendet
3 Tooling	Ja	Standard IDEs in vollem Umfang verwendbar
4 Anzahl Abh.	Ja	durch eigene Abstraktionskomponenten und j2ObjC gering
5 Grad Abh.	Teilweise	Cross-Plattform Funktionalität an j2ObjC gebunden

7 Fazit und Ausblick

Das Konzept des Cross-Plattform-Ansatzes bei der Entwicklung mobiler Apps bringt große Vorteile mit sich, sowohl was den reduzierten Arbeitsaufwand angeht, als auch in finanzieller Hinsicht. In dieser Arbeit wurden vier Cross-Plattform Lösungen vorgestellt und im Hinblick auf ihre Einsatzmöglichkeiten zur Entwicklung einer mobilen Anwendung untersucht. Jeder der vier Ansätze konnte nicht alle gewünschten Kriterien vollständig erfüllen. Anhand dieser Unzulänglichkeiten wurde ein Ansatz konzipiert der auf eine komponentenbasierte Architektur zurückgreift und einen Cross-Translator verwendet.

Anschließend wurde dieser Ansatz prototypisch implementiert. Mit dem Cross-Translator j2ObjC wurde eine Tool-Chain umgesetzt und auf Teile des Quelltextes einer dafür prototypisch entwickelten App angewandt.

Dabei konnten knapp 80 Prozent des Quelltexts auf der Seite von iOS automatisch generiert werden und mussten somit nicht portiert werden. Im Hinblick auf den Anwendungsfall, der weit weniger aufwändig ist als eine Business-App, ist je nach Logikumfang und Anzahl der Benutzeroberflächen ein anderes Verhältnis in beide Richtungen möglich. Durch die automatische Übersetzung des Quelltexts ist eine Bewertung anhand der LoC wenig aussagekräftig. Doch die Tatsache dass alle Komponenten der Logikschicht übersetzt werden konnten, lässt eine positive Interpretation zu.

Der Ansatz bietet durch sein positives Ergebnis viel Potenzial für weitere Forschungen. Die nachfolgenden Erweiterungsmöglichkeiten sind für weitere Untersuchungen denkbar:

- Automatische Installationsroutine und Projekt-Setup

  Durch eine automatische Installationsroutine ließe sich der Aufwand der initialen Konfiguration der Projekte minimieren.

- Eclipse-Plugin für direkte Unterstützung innerhalb der IDE

  Um auch den Anwendungsfall außerhalb größerer Entwicklungsumgebungen zu unterstützen, könnte ein Eclipse-Plugin an die Tool-Chain angebunden werden.

- Adapterschicht zwischen Logik und UI

  Um der schlechten Lesbarkeit des übersetzten Quelltextes entgegenzuwirken könnte eine Adapterschicht das Anbinden der UI für den Entwickler komfortabler machen.

- Parallelisierung der Tool-Chain

  Für den Einsatz in größeren Entwicklungslandschaften ist eine Parallelisierung sinnvoll.

- Integration der Tool-Chain in einen CI-Prozess

  Durch die Integration der Tool-Chain in einen CI-Prozess könnte die Tool-Chain auch in größere Build-Prozesse integriert werden.

Quellenverzeichnis

[AND14] <uses-sdk>. URL: http://developer.android.com/guide/topics/manifest/uses-sdk-element.html <Stand: 18.10.2014>

[BI12] Gobry, Pascal-Emmanuel: ANALYST: Smartphone Sales Will Dwarf PC Sales This Year And Reach A Staggering 1.5 Billion Per Year By 2016. URL: http://www.businessinsider.com/smartphone-sales-forecast-2012-2 <Stand: 04.10.2014>

[BIL07] Higgins, Bill: the Uncanny Valley of user interface design. URL: http://billhiggins.us/blog/2007/05/17/the-uncanny-valley-of-user-interface-design/ <Stand: 17.09.2014>

[FOW04] Fowler, Martin: Inversion of Control Containers and the Dependency Injection pattern. URL: http://www.martinfowler.com/articles/injection.html <Stand: 18.10.2014>

[FRE07] Eric Freeman, Elisabeth Freeman: Entwurfsmuster von Kopf bis Fuß, O'Reilly 2006, 3. Korrigierte Nachdruck 2007. ISBN: 3-89721-421-0

[GAR10] Gartner Says Worldwide Mobile Phone Sales Grew 35 Percent in Third Quarter 2010; Smartphone Sales Increased 96 Percent. URL: http://www.gartner.com/newsroom/id/1466313 <Stand: 19.10.2014>

[GAR11] Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010. URL: http://www.gartner.com/newsroom/id/1543014 <Stand: 19.10.2014>

[GAR11a] Gartner Says 428 Million Mobile Communication Devices Sold Worldwide in First Quarter 2011, a 19 Percent Increase Year-on-Year. URL: http://www.gartner.com/newsroom/id/1689814 <Stand: 19.10.2014>

[GAR11b] Gartner Says Sales of Mobile Devices in Second Quarter of 2011 Grew 16.5 Percent Year-on-Year; Smartphone Sales Grew 74 Percent. URL: http://www.gartner.com/newsroom/id/1764714 <Stand: 19.10.2014>

[GAR11c] Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011; Smartphone Sales Increased 42 Percent. URL: http://www.gartner.com/newsroom/id/1848514 <Stand: 19.10.2014>

[GIT14] JOK 101: Debugging in PhoneGap. URL: https://github.com/phonegap/phonegap/wiki/Debugging-in-PhoneGap <Stand: 18.10.2014>

[GOO14] google/j2objc. URL: https://github.com/google/j2objc/wiki <Stand: 27.07.2014>

[HTW13] DIN EN ISO 9241-110. URL: http://www.cheval-lab.ch/was-ist-usability/usability-grundlagen/normen-und-richtlinien/iso-9241-110/ <Stand: 17.09.2014>

[HYB14] js Hybugger: Debugging Android web apps. URL: https://www.jshybugger.com/ <Stand: 04.10.2014>

[IBU13] Rauscher, Heinke Shanti: App-Entwicklung: Frontend- und Backend gehen ins Geld. URL: http://www.ibusiness.de/aktuell/db/857182hr.930419hr.html <Stand: 17.09.2014>

[KON14] Kony, Inc. Third-Party Licenses. URL: http://www.kony.com/oslicenses <Stand: 18.10.2014>

[NEO14] Compiler. URL: http://www.neogrid.de/was-ist/Transcompiler <Stand: 27.07.2014>

[ORE07] O'Reilly, Tim: The Uncanny Valley of User Interface Design. URL: http://radar.oreilly.com/2007/05/the-uncanny-valley-of-user-int.html <Stand: 17.09.2014>

[VB13] HTML5 vs. native vs. hybrid mobile apps: 3,500 developers say all three, please. URL: http://venturebeat.com/2013/11/20/html5-vs-native-vs-hybrid-mobile-apps-3500-developers-say-all-three-please/ <Stand: 17.09.2014>

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Augsburg, 20.10.2014

Julian Feller