Ein komponentenbasiertes System zur Erstellung von Smartphone-Anwendungen unter Verwendung einer einfachen Beschreibungssprache

Ralph Harrer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2012

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 21. Juni 2012

Ralph Harrer

Inhaltsverzeichnis

Kurzfassung

Durch die Einführung von *iOS* und *Android* wurde ein komplett neuer Markt für mobile Anwendungen geschaffen. Für Entwickler stellt sich bei der Planung dieser „Apps“ die Frage, für welche Plattform entwickelt werden soll oder welche Multiplattform-Lösung für die Umsetzung in Frage käme. Letztere werden immer beliebter, da sie Kosten und Zeit sparen, doch leider erreichen diese selten die Qualität von nativen Anwendungen, da diese aus technischen Gründen diverse Einschränkungen mit sich bringen.

In Zuge dieser Arbeit wurde ein neuer Ansatz für die native Erstellung von Smartphoneanwendungen entwickelt, der keine Programmierkenntnisse voraussetzt. Über eine sehr einfach gehaltene Beschreibungssprache können Anwendungen mit Hilfe eines Baukastensystems erstellt werden. Dieser experimentelle Ansatz ermöglicht Anfängern einen schnellen Einstieg und nutzt die Vorteile der nativen Anwendungsentwicklung.

Abstract

Since the release of *iOS* and *Android* a completely new market has opened for mobile applications. This is why App developers have to decide which platform to support or which multi platform tool to use to meet specific requirements. Multi platforms have become increasingly popular as they save time and money, but due to technical limitations they stand significantly behind the quality of native written Apps.

This project has offered an approach to native smartphone applications independent of developers' programming proficiency. Using a simple markup language and a modular software system, it enables the creation of entirely new Apps from scratch. The suggested experimental approach thus scaffolds the advantages of native applications and allows beginners to get started easily.

Kapitel 1

Einleitung

Die mobile Revolution ist seit dem Erscheinen von *Apples iOS* und *Googles Android* nicht mehr zu stoppen. Der Touchscreen hat herkömmliche Tastenfelder abgelöst und die darauf abgestimmten Betriebssysteme ermöglichen die einfache und intuitive Bedienung mittels Berührungen und Gesten. Diese neuen Konzepte brachten auch einen komplett neuen Markt mit sich: Den Markt der mobilen Applikationen, kurz „Apps“. Diese können über plattformeigene digitale Marktplätze bezogen und mit nur wenigen Berührungen bezahlt und installiert werden. Noch nie war es so leicht, Anwendungen zu erwerben und zu nutzen. Die Entwickler stehen jedoch vor einem großen Problem: In welche Plattform soll man sich einarbeiten, welches Tool eignet sich dafür am Besten und wie schwer gestaltet sich der Einstieg? Der Markt hat auf diese Fragen reagiert und stellt viele Lösungen zur Verfügung, um die Entwicklung von Apps so einfach wie möglich zu gestalten.

1.1 Problemstellung

Es gibt viele unterschiedliche Arten, wie moderne Apps entwickelt werden können. Jedoch bringt jeder Ansatz seine Vor- und Nachteile mit sich. Der Wunsch nach einer einheitlichen Lösung wird seitens der Wirtschaft immer lauter, da die native Entwicklung von Smartphone Anwendungen für alle Plattformen kostspieliger ist, wie die Umsetzung mit einer „Ein Code für alle Plattformen“-Lösung. Letzteres führt immer wieder zu unerwarteten Problemen: Letztere laufen nicht flüssig, halten sich nicht an das „Look-and-Feel“ der jeweiligen Plattform oder reizen nur sehr eingeschränkt den Vorteil einer mobilen Plattform (GPS, Touchscreen, Sensoren) aus.

Es gibt bereits einige Ansätze, die native Anwendungen für unterschiedlichste Plattformen aus einer Codebasis erstellen. In dieser Arbeit wurden zwei davon evaluiert, jedoch setzen diese gute Programmierkenntnisse und spezielle Entwicklungsumgebungen voraus. Ein neuer Ansatz soll versuchen, die Einstiegshürde der Erstellung von nativen Anwendungen zu reduzieren,

indem auf ein Baukastensystem zurückgegriffen wird. Durch die Verwendung einer einfachen Beschreibungssprache werden keine Programmierkenntnisse vorausgesetzt. Dieser experimentelle Ansatz wird mit bestehenden Systemen verglichen, um dessen Vor- und Nachteile hervorzuheben und dessen Praxistauglichkeit zu prüfen.

1.2 Motivation

Als App-Entwickler steht man oft vor der Frage, für welche Plattform entwickelt werden und welches Werkzeug dafür verwendet werden soll. Es gibt bereits einige Ansätze, die versuchen, dem Entwickler diese Entscheidung zu nehmen, indem sie mit der gleichen Codebasis versuchen, alle gängigen Plattformen zu bedienen. Dies klingt zwar verlockend, führt jedoch nicht immer zum gewünschten Ziel.

Der native Weg führt zwar zum besten Ergebnis, gestaltet sich für Einsteiger als besonders schwierig, da nicht nur eine Programmiersprache beherrscht werden muss, man muss sich auch in das Framework und in die Eigenheiten der jeweiligen Plattform einarbeiten.

Die perfekte Lösung, außer den nativen Weg zu gehen, wird es in diesem Bereich wahrscheinlich nie geben. Aus diesem Grund wäre es doch sinnvoll, auf diesem aufzubauen und dafür die Einstiegshürde so gering wie möglich zu halten. Der in dieser Arbeit vorgestellte Ansatz versucht diese beiden Eigenschaften zu vereinen.

1.3 Zielsetzung

In dieser Arbeit wird ein experimenteller Ansatz vorgestellt, mit dessen Hilfe native Apps zur Verwaltung von benutzerdefinierten Datensätzen ohne Programmierkenntnisse mit einem Baukastensystem erstellt und verändert werden können, ohne diese neu kompilieren zu müssen. Dazu soll eine einfache und leicht zu erlernende Beschreibungssprache verwendet werden, welche optional auch mit Hilfe eines visuellen Editors generiert werden kann.

Um dies zu erreichen, werden bestehende Anwendungen analysiert, um wiederkehrende Muster und User Interface Elemente zu isolieren, um diese als Bausteine wiederverwenden zu können. Dabei wird der Fokus vor allem auf die einfache Bedienung und Erweiterung des Systems gelegt.

1.4 Aufbau der Arbeit

Diese Arbeit setzt sich aus fünf Bereichen zusammen. Im ersten Abschnitt wird ein aktueller Überblick über den mobilen Markt gegeben. Dabei werden die drei Plattformen *iOS*, *Android* und *Windows Phone 7* genauer vorgestellt und deren aktueller Stellenwert auf dem Markt betrachtet. Danach werden

bestehende Ansätze zur Entwicklung von mobilen Anwendungen im Detail vorgestellt und auf deren Vor- und Nachteile untersucht. Dieses Kapitel dient als Ausgangspunkt für den dritten Teil dieser Arbeit, worin der eigene Ansatz im Detail vorgestellt wird. Es wird darauf eingegangen, wie bei der Entwicklung vorgegangen wurde und wie die Architektur des vorgestellten Systems funktioniert.

Im viertel Teil wird im Detail gezeigt, wie und mit welchen Mitteln das im Zuge dieser Arbeit entwickelte komponentenbasierte System zur Erstellung von *iOS*-Anwendungen umgesetzt wurde.

Der letzte Teil dieser Arbeit beschäftigt sich mit der Evaluierung des Systems und klärt dabei die Fragen, wo das System Anwendung finden kann, welche ähnlichen Ansätze es bereits gibt, wie diese Arbeiten und an welchen Stellen Verbesserungen eingearbeitet werden können.

Kapitel 2

Überblick über mobile Betriebssysteme

2.1 Allgemein

Innerhalb der letzten Jahre entwickelt sich kaum ein anderer Markt so schnell, wie der der Mobiltelefone. Neben der (beinahe schon vernachlässigbaren) Funktion des Telefonierens, bieten moderne Smartphones einen Internetzugang samt modernster Übertragungstechnologien an. Nicht selten sind Wi-Fi, Bluetooth, (A-)GPS, ein Zweikernprozessor samt potenter GPU und ein Multitouch-Display in den Geräten verbaut. Die Akkulaufzeit hat sich im Laufe der Smartphonegenerationen auf etwa einen Tag bei normalem Gebraucht eingependelt.

Mit steigender Leistung stiegen auch die Ansprüche an die Software, die auf den jeweiligen Geräten läuft. Deshalb bieten moderne Betriebssysteme neben vorinstallierten Programmen, wie einem Webbrowser, einem E-Mail Client und diversen nützlichen Werkzeugen, auch die Möglichkeit an, neue Software über das Internet herunterzuladen. Diese meist recht überschaubaren Programme werden als „Apps“ bezeichnet, was eine Kurzform des Wortes „Applikation“ darstellt und können über spezielle Online-Märkte bezogen werden. Dieses Konzept ist aber alles andere als jung: Im Desktop-Bereich ist es gang und gäbe, dass Software von Drittanbietern installiert werden kann. Auf mobilen Geräten ist der Ansatz zwar ebenfalls nicht neu, aber erst durch das Internet und der einfachen Bedienung wurde es möglich, Smartphone-Anwendungen für die breite Masse zugänglich zu machen. Mayer beschreibt *Das Erfolgsmodell „App“* in [8] wie folgt:

> Mobile Apps stehen für das derzeit dynamischste Distributionsmodell der Mediengeschichte: Man kann sie passgenau programmieren, individuell anpassen, sie sind überall und jederzeit abrufbar. Hinzu kommen ein klares Kostensystem und die extrem einfache Bedienbarkeit: [...]

Operating System	4Q11 Market Share (%)	4Q10 Market Share (%)
Android	50,9	30,5
iOS	23,8	15,8
Symbian	11,7	32,3
RIM	8,8	14,6
Bada	2,1	2,0
Microsoft	1,9	3,4
Others	0,8	1,5
Total	100,0	100,0

Tabelle 2.1: Marktverteilung mobile Betriebssysteme viertes Quartal 2010 und 2011, Quelle: [19].

Mayer bezieht sich damit auf die „Einfachheit“ der neuen Vertriebswege und der Apps, was einen großen Teil des Erfolgs des neuen Konzepts ausmacht.

Die drei wichtigesten Betriebssysteme für Smartphones sind zur Zeit Apples *iOS*, Googles *Android* und Microsofts *Windows Phone 7*. Alle drei Plattformen bauen auf das oben beschriebene Konzept, der beliebigen Erweiterung des Systems durch Apps auf, unterscheiden sich jedoch in einigen Punkten grundlegend voneinander.

2.2 Aktuelle mobile Betriebbsysteme

2.2.1 Marktverteilung

Der Aktuelle Smartphonemarkt wird von den beiden Plattformen *iOS* und *Android* dominiert. Jedoch befinden sich auch weitere Mitstreiter im Geschäft. Die aktuelle Marktverteilung (Stand: viertes Quartal 2011) ist in 2.1 abgebildet.

Android und *iOS* führen den Markt mit 50,9% bzw. 23,8% an und weisen eine steigende Marktverteilung auf. Microsoft hat im Jahr 2010 bis 2011 Marktanteile mit seinem Betriebssystem *Windows Phone 7* verloren, jedoch plant der Redmonder Softwarehersteller mit dem Release von deren Mobile-Desktop-Hybridbetriebssystem Windows 8 einen größeren Marktanteil zurückzugewinnen. *Symbian* und *RIM*s Betriebssystem sind ebenfalls noch stark vertreten, weisen jedoch ebenfalls große Verluste am Smartphonemarkt auf. So sank die Verteilung von *Symbian* von 32,3% auf 11,7% und *RIM*s Betriebssystem von 14,6% auf 8,8% ab. Der Rest des Marktes wird von Samsungs Plattform namens *Bada* mit einer Verteilung von 2,1%, Microsofts mobilen Betriebssystemen mit 1,9% und restlichen Herstellern mit 0,8% eingenommen.

2.2.2 iOS

Geschichte

Apples mobiles Betriebssystem *iOS* wurde 2007 bei der Vorstellung der ersten iPhone-Generation der Öffentlichkeit präsentiert. Damals wurde es noch als *iPhone OS* bezeichnet und war eher als eine Verschmelzung von Telefon, iPod und mobilem Browser anzusehen. Üblich für Apple, wurde das Betriebssystem nur mit Apple-eigener Hardware und zwar dem iPhone bzw. mit dem iPod Touch ausgeliefert. Neben einem Webbrowser, einem E-Mail Client und diversen Organizer-Funktionen bot das Betriebssystem noch nicht viele Funktionen an. Doch durch die intuitive Bedienung durch Berührungen und Gesten erfreute sich das Betriebsystem großer Beliebtheit. Erst ein Jahr später stellte Apple den AppStore für das *iPhone OS* vor. Durch diesen war es nun möglich, Apps aus dem Internet herunterzuladen. Im Gegensatz zur damaligen Konkurrenz von Google, mussten sich die Entwickler an strenge Design- und Softwarerichtlinien (Näheres zu den Guidelines unter [16]) halten, um ihre Anwendungen im AppStore vertreiben zu dürfen. Im Jahr 2010 stellte Apple das iPad vor. Es handelte sich dabei um einen Tablet-Computer, welcher mit einer eigens für das iPad optimierten Version von *iPhone OS* Version 3.2 ausgeliefert wurde. Der Name des Betriebssystems wurde im Zuge der Vorstellung auf *iOS* abgeändert. Erst mit der Version 4.1 wurden beide parallel laufenden Versionen (speziell für iPad und iPhone optimiert) zu einem Betriebssystem verschmolzen. Von nun an hatten Entwickler die Möglichkeit für beide Plattformen parallel zu entwickeln und so genannte „Universal Apps“ zu veröffentlichen, welche auf beiden Plattformen ausführbar waren, sich jedoch im User Interface unterschieden. Mittlerweile liegt *iOS* in der Version 5.1 vor (Stand: März 2012).

Überblick

Das Betriebssystem *iOS*, welches ein Mac OS X Derivat ist, ist ein rein auf Berührungsbedienung ausgelegtes System. Damit der Endnutzer sich in jeder App möglichst schnell zurechtfindet, hat Apple genaue Guidelines veröffentlicht, an die sich jeder Entwickler zu halten hat. Somit wird garantiert, dass sich jede Anwendung ähnlich verhält. Dieser Punkt wird oft von vielen Entwicklern kritisiert, da diese Einschränkungen die Funktionalität von Apps limitieren, jedoch die Sicherheit des Systems erhöhen. Somit ist zum Beispiel das dynamische Erzeugen oder Nachladen und Ausführen von Code untersagt, was später in dieser Arbeit noch genauer behandelt wird. Ein weiteres neues Konzept stellt das fehlende Dateisystem dar. Dem Benutzer wird der Zugriff auf letzteres verwehrt. Jede Anwendung ist für die Verwaltung seiner eigenen Dateien verantwortlich. Diese werden in der Sandbox des jeweiligen Programms gespeichert und dürfen nur von der dazugehörigen Anwendung gelesen werden, was wiederum die Sicherheit des Systems verstärkt. Appli-

Abbildung 2.1: Links der iPhone Home Screen, rechts das Notification Center mit Widgets.

kationen können jedoch dem Betriebssystem mitteilen, mit welchen Dateien sie umgehen können. Danach ist es möglich, Dateien von einer Applikation an eine andere weiterzureichen.

Jeder Entwickler, der seine Anwendungen im AppStore vertreiben möchte, muss sich bei Apple als Entwickler registrieren und eine jährliche Gebühr von 99$ bezahlen. Ohne diese Registrierung ist es nicht möglich, die Anwendungen auf die jeweiligen Geräte zu übertragen, denn auch diese müssen als Entwicklergeräte bei Apple gemeldet werden. Durch diese strengen Kontrollen wird versucht, die Softwarepiraterie einzuschränken und die Qualität der Anwendungen zu steigern.

Die Hardware stellt nur sehr wenige physikalische Tasten bereit. Außer den Home-, Hold-Buttons, einem Mute-Switch und zwei Lautstärkereglern haben die Geräte keine weiteren haptischen Eingabemöglichkeiten. Der Homescreen (in Abb. 2.1 abgebildet) des iPhones besteht aus einer Auflistung von Applikationssymbolen, die man durch links-rechts-Wischgesten weiter-

blättern kann. Seit der Version 5.0 von *iOS* können Entwickler auch Widgets (ein Beispiel dafür ist in Abb. 2.1 zu sehen) erstellen und diese ins Notification Center integrieren. Android und Windows Phone 7 können im Gegensatz zu Apples Lösung Widgets direkt auf dem Home Screen anzeigen.

2.2.3 Android

Geschichte

Das *Android* Betriebssystem wurde das erste mal mit der Version 1.0 im Oktober 2008 mit dem T-Mobile G1 in den Handel gebracht. Im Gegensatz zu Apples *iOS* Plattform, wurde die erste Version von Googles *Android* bereits mit dem Android Market ausgeliefert. Dem System fehlten aber noch viele Features, wie ein On-Screen Keyboard, Multitouch-Funktionalität und es konnten nur Gratis-Versionen von Apps im Android Market angeboten werden. Im Februar 2009 folgte das erste Update auf Version 1.1 und kurz darauf folgte die Version 1.5 mit dem Namen *Cupcake*. In letzterer wurde das fehlende On-Screen Keyboard nachgeliefert, neben vielen weiteren Verbesserungen an der Plattform. Mit der Version 1.6 namens *Donut* wurde *Android* auflösungsunabhängig, was für die Plattform, welche an keine besondere Hardware gebunden sein sollte, essentiell war. Einen großen Schritt wagte Google mit der Version 3. *Honeycomb* war die erste Version von *Android*, welche speziell auf Tablets optimiert wurde. Das User Interface wurde komplett überarbeitet und die Geräte benötigten keine physikalischen Tasten mehr. Mit Version 4 *Ice Cream Sandwich* wurden die Tablet- und die Smartphone-Version zu einem System zusammengeführt. Seit März 2012 wurde der Android Market in „Google Play Market“ umbenannt und mit Googles Cloud Services verschmolzen. „Google Play“ ist nun ein zentraler Bezugspunkt für Bücher, Musik, Videos und Apps.

Überblick

Im Gegensatz zu Apples *iOS* ist der Quellcode von *Android* Open Source [1] und das System ist nicht an spezielle Geräte eines Herstellers gebunden. In [7, S. 2] wird die Philosophie von Android wie folgt beschrieben:

> Android had a very different philosophy when compared to Apple and the iPhone. Anyone could use Android in their devices for free, anyone could modify Android, and anyone could develop apps for it without seeking permission to put their apps in the Android Market.

Durch diese Offenheit wurde es schnell zum Marktführer, da es von großen Unternehmen wie Samsung und HTC eingesetzt wird. Oft passen die

[1] http://source.android.com/source/licenses.html

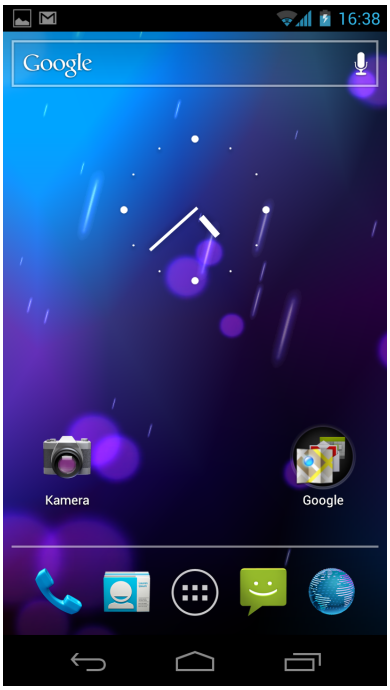

Abbildung 2.2: Android Version 4.x Home Screen mit Apps und Widgets.

Hersteller die Oberfläche des Betriebssystems an die eigenen Vorstellungen an, wie HTC mit *Sense*[2] oder Samsung mit *Touchwiz*[3]. Auch in der Open Source Szene erfreut sich Googles Betriebssystem großer Beliebtheit. Mittlerweile gibt es spezielle Modifikationen des auf Linux basierenden Betriebssystems, welche kostenfrei verwendet werden dürfen. Einer der bekanntesten Modifikationen stellt der „CyanogenMod“[4] dar.

Die tiefe Integration von Googles Services ins System ermöglicht die schnelle Inbetriebnahme der Telefone. Daten, wie zum Beispiel das Adressbuch und die installierten Applikationen werden mit dem angegebenen Google Konto automatisch syncronisiert und abgeglichen. Zwar ist ein Google Konto nicht verpflichtend, aber ohne dieses ist der Funktionsumfang von *Android* stark eingeschränkt. Neben dem offiziellen „Google Play Market“, welcher als Quelle für mobile Applikationen unter *Android* dient, gibt es noch einige andere Anbieter (wie Amazon) die Software für das Betriebssystem offerieren. Im Gegensatz zu Apples AppStore gibt es für *Android* Anwendungen keine Qualitätssicherung. Google stellt den Entwicklern ebenfalls umfangreiche Guidlines zur Verfügung (Näheres dazu unter [14]), die Entwickler werden aber nicht gezwungen diese einzuhalten. Dadurch gibt es

[2] https://www.htcsense.com/
[3] http://en.wikipedia.org/wiki/TouchWiz
[4] http://www.cyanogenmod.com/

bei den Apps große Qualitätsunterschiede und Sicherheitsrisiken mit denen die Nutzer konfrontiert werden.

2.2.4 Windows Phone 7

Geschichte

Microsofts *Windows Phone 7* ist das jüngste der drei vorgestellten Betriebssysteme. Es wurde im deutschsprachigen Raum im Oktober 2011 mit der Versionsnummer 7.0 das erst Mal ausgeliefert. Kurz darauf folgten einige kleinere Updates, die unter anderem die Funktion zum Kopieren und Einfügen von Daten und die Suche im Marketplace verbesserten. Im September 2011 wurde die Version 7.5 mit dem Codenamen „Mango“ vorgestellt (eine detaillierte Auflistung der Verbesserungen kann unter [21] nachgelesen werden). Dieses Update verbesserte nicht nur die Geschwindigkeit des Systems, sondern erweiterte auch die Multitasking Funktionalität für Apps, die nicht aus dem Hause Microsoft stammen. Seitdem dürfen auch diese mit starken Einschränkungen (die zugunsten der Akkulaufzeit ausfallen) im Hintergrund laufen. Der nächste größere Versionsschritt wird dieses Jahr mit *Windows 8* erwartet. Im Gegensatz zu *Windows 7*, welches nicht für mobile Plattformen konzeptioniert wurde, läuft der Nachfolger sowohl auf dem Desktop als auch auf Tablets und Smartphones. Microsoft setzt dafür auf zwei von einander unabhängigen User Interfaces. Auf dem Desktop wird die bekannte Windowsoberfläche dem Benutzer zur Verfügung stehen. Auf Tablets und Smartphones kommt die „Metro UI'", welche bereits auf *Windows Phone 7* zu sehen ist, zum Einsatz.

Überblick

Microsoft setzt unter *Windows Phone 7* die „Metro UI“ ein. Diese Oberfläche hebt sich von den beiden bereits vorgestellten Konkurrenten stark ab. Statt auf Applikations-Icons zu setzen, verwendet Microsoft „Live Tiles“. Diese spiegeln die jeweilige App auf dem Home Screen wider, bieten aber zusätzliche Informationen, nicht nur den Namen der Anwendung an. Je nach Implementierung können diese beliebige Daten der jeweiligen App darstellen und somit dem Benutzer schon auf dem Home Screen (in Abb. 2.3 dargestellt) Informationen anbieten, die dieser sonst nur nach dem Start der App, welches durch Antippen des jeweiligen „Live Tiles“ geschieht, bekommen würde. Dieser Ansatz kombiniert somit Widgets mit dem Applikationsicon. Um ähnliche Applikationen miteinander zu verknüpfen, verwendet Windows Phone 7 so genannte „Hubs“. Der „Kontakte-Hub“ (in Abb. 2.3 abgebildet) verbindet zum Beispiel Facebook, Twitter, SMS und jegliche andere Social-Media App oder Applikation zur Kontaktaufnahme in einem Bereich, von dem aus man direkt eine Person mittels der zur Verfügung gestellten Anwendung kontaktieren kann. Entwickler, die für Microsofts mobile Plattform entwickeln

Abbildung 2.3: Windows Phone 7 Home Screen (links) und Kontakte-Hub (rechts), Quelle: [23].

wollen, müssen wie *iOS*-Entwickler eine jährliche Gebühr von 99$ bezahlen und sich an Microsofts Development Guidelines (mehr dazu unter [22]) halten, um Anwendungen in den „Windows Phone Marketplace“ stellen zu dürfen. Auch den Geräteherstellern werden gewisse Spezifikationen vorgelegt, an die sie sich bei der Entwicklung der Smartphones halten müssen, um den reibungslosen Betrieb von Windows Phone 7 gewährleisten zu können. Einen spannenden Schritt wagt Microsoft mit Windows 8, was vor allem die Entwicklung von Apps angeht: Zukünftig sollen Anwendungen nicht nur mit C# und VB.NET entwickelt werden können, sondern auch mit HTML5 und JavaScript. Durch diesen Schritt ermöglicht Microsoft auch Web-Entwicklern für die zukünftige Plattform Anwendungen zu erstellen.

2.2.5 Sonstige

Neben den drei vorgestellten Plattformen, gibt es noch einige andere Vertreter, die auf modernen Smartphones zu finden sind. Samsung bietet für deren Einsteigergeräte ein eigenes Betriebssystem namens *Bada* an. Dieses basiert auf einem Linux-Kernel und verfügt ebenfalls über einen Anwendungsmarkt namens „Samsung Apps“[5]. Mit einer Marktverteilung von 2,1% liegt es knapp vor *Windows Phone 7*, welches mit 1,9% aufwarten kann (Stand: Viertes Quartal 2011, Abgebildet in Tabelle 2.1).

Symbians Marktverteilung ist in einem Jahr von 32,3% auf 11,7% gesunken. Trotzdem war die von Nokia verwendete Plattform im vierten Quartal 2011 noch der drittgrößte Vertreter von mobilen Betriebssystemen auf dem Markt. Voraussichtlich wird die Marktverteilung von Symbian weiter sinken, da Nokia in Zukunft auf *Windows Phone 7* setzen wird.

Research In Motion oder kurz *RIM* wurde durch die „BlackBerry“ Mobiltelefone (welche mit dem *BlackBerry OS* ausgeliefert werden) bekannt und erfreut sich mit einer Marktverteilung von 8,8% nach wie vor großer Beliebtheit. Auch *RIM* vertreibt Apps über die „BlackBerry App World“[6]. Der Einstieg in den Tablet Markt mit dem „BlackBerry PlayBook“ brachte jedoch nicht den gewünschten Erfolg.

2.3 Unterschiede zu Desktop Software

Wie bereits erwähnt, stellen moderne Smartphones und Tablets einen Touchscreen für die Interaktion mit dem Gerät zur Verfügung. Im Gegensatz zu früheren Touchscreen Produkten, welche noch mit Stifteingaben arbeiteten, sollen aktuelle Geräte ausschließlich mit dem Finger bedienbar sein, weswegen komplett neue Ansprüche an das User Interface gestellt werden.

2.3.1 Benutzeroberfläche

Die Benutzeroberfläche stellt die Schnittstelle zwischen dem Programm und dem Benutzer dar. Diese entscheidet darüber, wie gut oder schlecht ein Programm zu bedienen ist. Auf dem PC haben sich gewisse Metaphern über die vergangenen Jahrzehnte verankert: Der Desktop spiegelt eine Schreibtischoberfläche wieder, Daten werden in Ordnern abgelegt und die Eingabe erfolgt über die Maus und die Tastatur. Hartmann beschreibt es in seinem Buch [5, S. 80] die Desktop Metapher wie folgt:

> Die Hegemonie der bereits existierenden Bürotechnik und die Organisationsform des amerikanischen Büros führten also dazu, dass Computer kein multimediales Interface bekamen, sondern lediglich die obligatorische Schreibtischtastatur (und später dann

[5] http://www.samsungapps.com/
[6] http://appworld.blackberry.com/webstore/?lang=de

Abbildung 2.4: Vergleich zwischen Desktop- und Mobile-Software.

auch eine Maus). In diesem Zusammenhang wurde schließlich von Alan Kay ca. 1976 die Desktop-Metapher generiert: Jede Aufgabe wird in einem Dokument erledigt, das wie ein Blatt Papier erscheint, und diese Dokumente lassen sich im Fenster übereinander stapeln. Die perfekte Anwenderillusion war geschaffen, um den Computer als Arbeitsgerät für das Büro zu vermarkten.

Diese Metapher mag auf dem PC funktionieren, da genug Platz zur Verfügung steht. Auf kleinen Smartphone-Bildschirmen, muss jedoch enorm an Platz gespart werden. Deshalb ließen sich die Hersteller von modernen Betriebssystemen neue Bedienkonzepte einfallen, welche auf die Eingabe mit den Fingern optimiert wurde.

Apple schreibt den Entwicklern z. B. vor, wie groß Schaltflächen in Apps mindestens sein müssen (Details dazu sind in [16, Bereich: Custom Icon and Image Creation Guidelines] zu finden), um diese mit dem Finger ohne größere Probleme berühren zu können. In Abb. 2.4 ist ein Vergleich zwischen der Desktop- und Mobilvariante von Apples iPhoto abgebildet. Man kann gut erkennen, wie wenig Platz den Entwicklern für die Benutzeroberfläche auf dem Smartphone zur Verfügung steht.

Mehr zu diesem Thema kann in [1] nachgelesen werden. In diesem Paper wurden vier Kernprobleme der mobilen Webentwicklung und den damit verbundenen Einschränkungen isoliert und genauer untersucht.

2.3.2 Programmablauf

Der übliche Programmablauf auf einem Smartphone kann am besten mit einem typischen „Wizard“ auf einem PC verglichen werden: Dieser führt den

Abbildung 2.5: Typischer Ablauf eines Programms.

Benutzer Schritt für Schritt durch einen Prozess, um eine gewisse Aufgabe zu erledigen. Jeder dieser Schritte dient dazu, einen Teil der Gesamtaufgabe gezielt abzuarbeiten. Der Benutzer kann jederzeit einen Schritt zurückgehen, um diverse Eingaben zu ändern oder zu korrigieren. Ein Beispiel dafür ist in Abb. 2.5 grafisch dargestellt: Nach der Auswahl eines Fotoalbums, werden alle Bilder aufgelistet. Durch die Selektion des gewünschten Bildes wird eine Detailansicht eingeblendet, wo das Bild nun betrachtet, weitergeleitet und bearbeitet werden kann.

Ein mitunter sehr komplexer Programmablauf wird somit in kleine Teilbereiche heruntergebrochen und einzeln erledigt. Dieser Sachverhalt wird in Kapitel 4 ausführlich behandelt.

2.3.3 Andere Einschränkungen

Ein Smartphone und somit dessen Software unterliegen aufgrund fehlender Rechenleistung und beschränkter Stromreserven gewissen Einschränkungen.

Parallelität

Auf modernen Smartphone-Betriebssystemen können Apps durchaus parallel ausgeführt werden, doch um Energie zu sparen, greifen *iOS* und *Windows Phone 7* auf einen Trick zurück: Apps dürfen nur gewisse Services in Anspruch nehmen, während sie sich im Hintergrund befinden. Eine Radio App, die Musik aus dem Internet streamt, kann dadurch im Hintergrund die Wiedergabe der Musik fortsetzen, aber aufwändige Rechenprozesse können erst dann weiter ausgeführt werden, sobald sie wieder in den Vordergrund gerufen wird.

Auslagerung von Rechenleistung

Moderne Applikationen stellen durchaus hohe Ansprüche an die Rechenleistung der Smartphones. Um diese möglichst gering zu halten und dadurch die Akkulaufzeit zu verlängern, besteht die Möglichkeit, die Berechnung in die „Cloud“ auszulagern. Je nach Bedarf, können z. B. Bilder ins Internet hochgeladen werden, dort einer aufwändigen Operation (z. B. Bildkorrekturen, Filter usw.) unterzogen und wieder aufs Gerät heruntergeladen werden. Mehr zum Thema *Cloud-* und *Grid-Computing* kann in [13] nachgelesen werden.

Speicher

Der Festspeicher von mobilen Geräten ist im Vergleich zu Desktop-Rechnern oder Notebooks stark begrenzt. Hier greifen die Hersteller ebenfalls auf die „Cloud“ zurück und stellen den Nutzern Onlinespeicher, Synchronisationsso wie Backupmöglichkeiten zur Verfügung. Die Daten werden bei Bedarf je nach Anwendungsfall auf das Smartphone übertragen oder gestreamt.

Kapitel 3

Bestehende Arbeiten

Im Bereich der Smartphone-Programmierung gibt es bereits mehrere unterschiedliche Ansätze, wie Applikationen mit oder ohne Kenntnisse der nativen Programmiersprachen erstellt und für eine oder mehrere Plattformen veröffentlicht werden können.

3.1 Bestehende Ansätze

Es gibt bereits eine Reihe von bestehenden Ansätzen, wie man neben dem nativen Ansatz zur Softwareentwicklung, mit ein und dem selben Code Applikationen für mehrere Plattformen entwickeln kann. Im folgenden Abschnitt werden vier Ansätze mit ihren Vor- und Nachteilen beschrieben.

3.1.1 Web-Apps

Bei den Web-Apps handelt es sich um Webseiten, welche speziell für die Verwendung auf dem Smartphone optimiert wurden. In [4] werden mobile Web Applicationen wie folgt beschrieben:

> Mobile web applications are mobile applications that do not need to be installed or compiled on the target device. Using XHTML, CSS, and JavaScript, they are able to provide an application-like experience to the end user while running in any mobile web browser. By “application-like“ experience, I mean that they do not use the drill-down or page metaphors in which a click equals a refresh of the content in view. Web appli- cations allow users to interact with content in real time, where a click or touch performs an action within the current view.

Die „Anwendung“ wird im Browser des mobilen Geräts ausgeführt und hat deswegen kaum Zugriff auf die Hardware. Mit Hilfe von diversen Java-

Abbildung 3.1: Die Twitter Web-App (links) im Vergleich zur nativen iOS Anwendung (rechts).

Script Bibliotheken, wie jQuery Mobile[1] oder Sencha Touch[2] wird die Oberfläche dem Verhalten einer nativen App angepasst. Web-Apps arbeiten oft langsamer und sind in ihrerer Funkionalität stärker eingeschränkt als native Anwendungen. In Abb. 3.1 sind eine Web-App und ihr natives Pendant abgebildet. Man kann erkennen, dass die Web-App dem „Look and Feel“ der App unter iOS angepasst wurde und sich dadurch auf anderen Plattformen nicht in die jeweilige UI nahtlos integriert, wie man es von nativen Anwendungen erwarten würde.

3.1.2 Native Web-Apps

Native Web-Apps sind eine spezielle Art von Web-Apps, welche nicht direkt im Browser des Smartphones ausgeführt werden, sondern als native Apps ausgeliefert werden. Der Hauptteil dieser Anwendungen besteht aus einem Webbrowser, der die Web-App rendert. Der Vorteil dieses Ansatzes ist, dass die Anwendung in den jeweiligen Vertriebswegen der Plattformen als eigenständige Anwendung angeboten werden und auf die Hardwareschnittstellen des jeweiligen Gerätes zugreifen kann. Das Problem mit dem mitunter platt-

[1] http://jquerymobile.com/
[2] http://www.sencha.com/products/touch

formfremden User Interface bleibt auch bei diesem Ansatz bestehen. *PhoneGap*[3] und *Tersus*[4] sind zwei Vertreter, die diesen Ansatz verfolgen.

PhoneGap

PhoneGap wurde von *Nitobi Software* als Open Source Plattform entwickelt und ermöglicht die Erstellung von Applikationen mittels JavaScript und HTML5 (es werden auch Bibliotheken wie *jQuery Mobile* oder *Sencha Touch* unterstützt). Im Oktober 2011 wurde das Unternehmen von *Adobe*[5] aufgekauft.

PhoneGap bietet eine JavaScript API an, welche den Zugriff auf die Hardwareschnittstellen der Smartphones mit der gleichen Codebasis ermöglicht. [12, Kapitel 7. Going Native] beschreibt die Arbeitsweise von *PhoneGap* wie folgt:

> PhoneGap abstracts the APIs for the most widely available mobile phone features so mobile application developers can use the same code everywhere. You still need to deploy your app manually using the SDK provided by the vendor, but you don't need to change your application code.

Durch den Zugriff auf die Hardware kann das Verhalten einer nativen App gut angenähert werden. Mitterweile werden sieben unterschiedliche Plattformen von der API von *PhoneGap* unterstützt. In Tabelle 3.1 ist aufgeführt, welche Funktionen auf der jeweiligen Plattform zur Verfügung stehen. Nähere Informationen zu *PhoneGap* können auf der offiziellen Website[6], in [12, Kapitel 7. Going Native] oder in der Arbeit von Eckl unter [3, Abschn. 4.3.2] nachgelesen werden.

Tersus

Tersus verwendet ein Konzept, welches dem Projekt dieser Arbeit sehr ähnelt, denn deren Ansatz basiert ebenfalls auf Komponenten und visueller Programmierung. Die Entwicklungsumgebung von *Tersus* (in Abb. 3.2 dargestellt) setzt auf *Eclipse* auf und ist allein auf visuelle Programmierung ausgelegt.

Die Komponenten (z. B. Buttons, Labels, Eingabefelder) werden per *Drag and Drop* auf der Arbeitsfläche positioniert und über Ein- und Ausgabeschnittstellen miteinander verbunden. Jede dieser Komponenten kann beliebige Kind-Komponenten oder Aktionen enthalten. Um die Übersicht zu

[3] http://phonegap.com/
[4] http://www.tersus.com/
[5] http://www.adobe.com/aboutadobe/pressroom/pressreleases/201110/AdobeAcquiresNitobi.html
[6] http://www.phonegap.com

	iOS 3GS+	Android	RIM 6.0+	HP	WP7	Symbian	Bada
Accelerometer	✓	✓	✓	✓	✓	✓	✓
Camera	✓	✓	✓	✓	✓	✓	✓
Compass	✓	✓	–	–	✓	–	✓
Contacts	✓	✓	✓	–	✓	✓	✓
File	✓	✓	✓	–	✓	–	–
Geolocation	✓	✓	✓	✓	✓	✓	✓
Media	✓	✓	–	–	✓	–	–
Network	✓	✓	✓	✓	✓	✓	✓
Notifications*	✓	✓	✓	✓	✓	✓	✓
Storage	✓	✓	✓	✓	✓	✓	–

Tabelle 3.1: Unterstützte Technologien von *PhoneGap*, Quelle: [25], Stand: März 2012 (*Es werden Alert-Dialoge, Ton- und Vibrations-Notifications unterstützt).

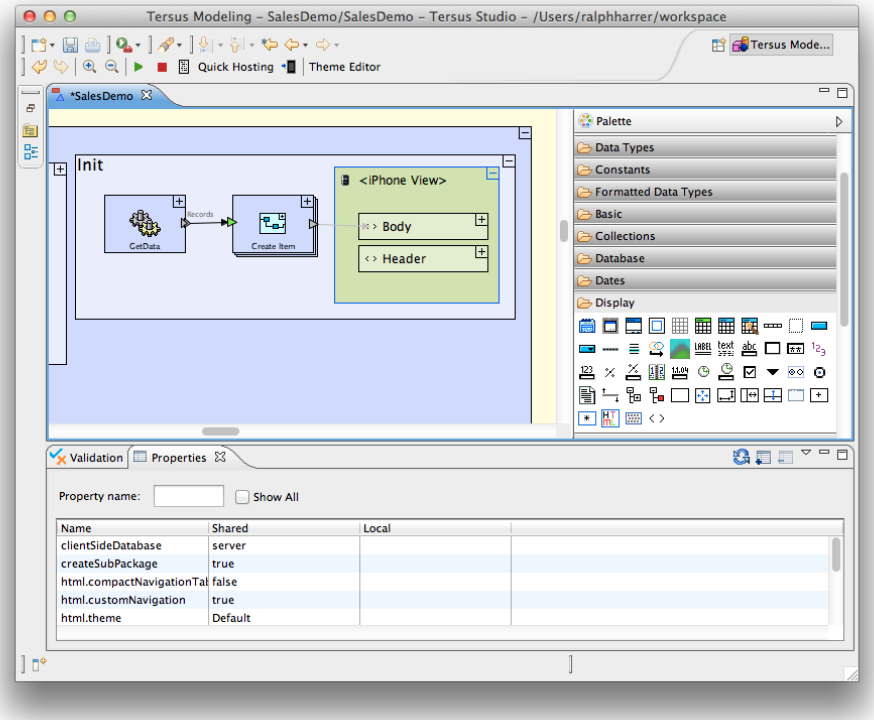

Abbildung 3.2: Die Tersus IDE.

bewahren, lässt sich die Arbeitsoberfläche unendlich in die Tiefe zoomen, um so beliebige Kind-Elemente Übersichtlich darzustellen.

Was die Kommunikation mit der Hardware angeht, gibt Tersus nicht viel preis, außer dass der Zugriff auf die Kamera möglich ist. Die Gestaltung der Anwendungen wird über die Verwendung von Cascading Style Sheets ermöglicht.

3.1.3 Pseudo-Native Ansätze

Bei dem verwendeten Ausdruck „Pseudo-Nativ“ handelt es sich um Anwendungen, welche zwar native Elemente, wie z. B. Teile des User Interfaces, verwenden, aber der Code der Anwendung nicht kompiliert wird und somit nicht nativ vorliegt.

Appcelerator

Appcelerator verfolgt einen durchaus interessanten Ansatz. Die Programmierung der Anwendungen erfolgt mit der zur Verfügung gestellten JavaScript API, welche „native“ Anwendungen erzeugt. Auf der offiziellen Webseite [15] wird das wie folgt beschrieben:

> Everything else is basically a web page. There are lots of ways to build apps, especially on the iPhone and Android. You can build a JavaScript widget in the Safari browser. You can wrap Webkit, throw in a few API's to bridge the 'gap' between web and app, and lob it into the app store. But aside from learning Objective-C or Java, building native apps means doing everything native, not just pieces. When we say “native,“ this is what we mean: [...]

Was in diesem Fall verschwiegen wird, ist, dass selbst diese Apps nicht 100%ig nativ laufen. Die Anwendung führt zur Laufzeit den JavaScript Code aus und generiert aufgrund dessen das User Interface, welches aus nativen UI-Elementen besteht. Das hat zur Folge, dass Anwendungen eventuell langsamer ausgeführt werden, da kein kompilierter Code vorliegt. Der Vorteil dieses Ansatzes liegt aber auf der Hand: Durch diese Methode kann der selbe JavaScript Code verwendet werden, um Anwendungen für *Android* und für *iOS* zu erstellen, welche sich in ihrem plattformspezifischen Erscheinungsbild präsentieren. Eine genaue Beschreibung zur Arbeitsweise von Appcelerator, verfasst von Haynie, Gründungsmitglied und CEO von Appcelerator, kann auf Stack Overflow unter [20] nachgelesen werden.

Da beide Betriebssysteme unterschiedliche Funktionen aufweisen, wie UI-Animation unter *iOS*, bietet Appcelerator spezielle „Weichen“ an, um gewissen Code nur auf der gewünschten Plattform ausführen zu können. In der Praxis zeigt sich, dass diese Weichen sehr oft benötigt werden, um die jeweiligen Eigenheiten der doch sehr unterschiedlichen Plattformen auszugleichen.

Dadurch wird jedoch der Vorteil, einen Code für mehrere Plattformen zu verwenden, ad absurdum geführt, da sehr viel Zeit in die plattformspezifische Programmierung investiert werden muss.

3.1.4 Native Ansätze

Im Gegensatz zu den zuvor beschriebenen Anwendungen, wird der Quellcode von nativen Apps zuerst von einem Compiler in Byte-Code umgewandelt. Dieser kann direkt auf den dafür vorgesehenen Prozessoren der Smartphones ausgeführt werden. Der Vorteil dieses Ansatzes ist, dass die Anwendungen schneller laufen, da der Programmcode direkt auf der Hardware des Gerätes ausgeführt wird. Außerdem können die Apps mit den vom Betriebssystemhersteller vorgesehenen Entwicklungsumgebungen entwickelt werden. Dadurch haben diese auch Zugriff auf alle Schnittstellen, die vom Hersteller freigegeben werden und können somit das gesamte Potential und Vorteile der jeweiligen Plattform ausnutzen. Ein ausführlicher Artikel zu den Unterschieden von nativen Anwendungen zu mobilen Web-Applikationen ist in [2] zu finden.

Der Nachteil an diesem Ansatz ist, dass jede Anwendung speziell für die jeweilige Plattform entwickelt werden muss. Der Code-Austausch zwischen den Plattformen ist nur sehr beschränkt möglich, da sich die Architekturen und Programmiersprachen zu stark voneinander unterscheiden.

Applause

Applause versucht über eine domänenspezifische Programmiersprache ein und den selben Code für mehrere Plattformen zu nutzen. Auf der offiziellen Website [18] wird Applause so beschrieben:

> *Applause* is a domain-specific language and a set of code-generators to produce mobile applications for *iPhone*, *Android*, *Windows Phone* and a mobile website on top of Google App Engine. [...] *Applause* is not a web framework, a cross-compiler nor is it an interpreter. Instead, it uses a simple input language explicitly designed for data-driven mobile applications to produce human-readable source code in Objective-C, Java, C# or Python.

Im Gegensatz zu den zuvor erwähnten Ansätzen, wird bei *Applause* nativer Quellcode über Source-Code Vorlagen (die jeweils für die untersützten Plattformen bereitliegen) aufgrund der domänenspezifischen Programmiersprache mit Hilfe von *xText* erzeugt. Daraus kann mit der jeweiligen Entwicklungsumgebung eine lauffähige native App für die gewünschte Plattform kompiliert werden. Näheres zur Softwareentwicklung mit *xText* ist in [17] zu finden.

Kapitel 4

Eigener Ansatz für komponentenbasierte Systeme

Im Zuge dieser Arbeit wurde ein Prototyp entwickelt, mit dem es möglich ist, Applikationen für *iOS* mit Hilfe einer einfachen Beschreibungssprache erstellen zu können. Im Gegensatz zu den bereits vorgestellten Ansätzen, soll es bei diesem möglich sein, die Arbeitsweise der Anwendung ohne eine Neukompilierung ändern zu können. Dadurch wäre es z. B. möglich, die Anwendung durch Ändern der Beschreibung jederzeit neu anzupassen. Auch dies wird von Apple leider nicht geduldet, da sich das Verhalten einer App nach der Installation nicht verändern darf. Zum Erstellen einer App kann dieses System jedoch ohne Weiteres verwendet werden.

4.1 Architektur und Aufbau

Im folgenden Abschnitt wird beschrieben, wie das komponentenbasierte System aufgebaut wurde. Der Fokus lag vor allem darauf, mit einer einfach zu erlernenden Beschreibungssprache native Anwendungen ohne Programmierkenntnisse erstellen zu können.

4.1.1 Allgemein

Wie der Ausdruck „komponentenbasiertes System“ vermuten lässt, werden Anwendungen durch Verknüpfung und Kombination von Komponenten erstellt. Um dieses Ziel zu erreichen, mussten zuerst die wichtigsten Bausteine einer App gefunden werden. Dazu wurden bestehende Anwendungen genau analysiert. Die gefundenen Bausteine werden in dieser Arbeit als *Components* bezeichnet. Sie dienen zur visuellen Repräsentation und für die Benutzereingabe. Um die Kommunikation zwischen den *Components* zu ermöglichen, wurden so genannte *Events* eingeführt. Diese werden von den *Components* dann aufgerufen, wenn eine gewisse Aktion ausgelöst werden soll. Die Da-

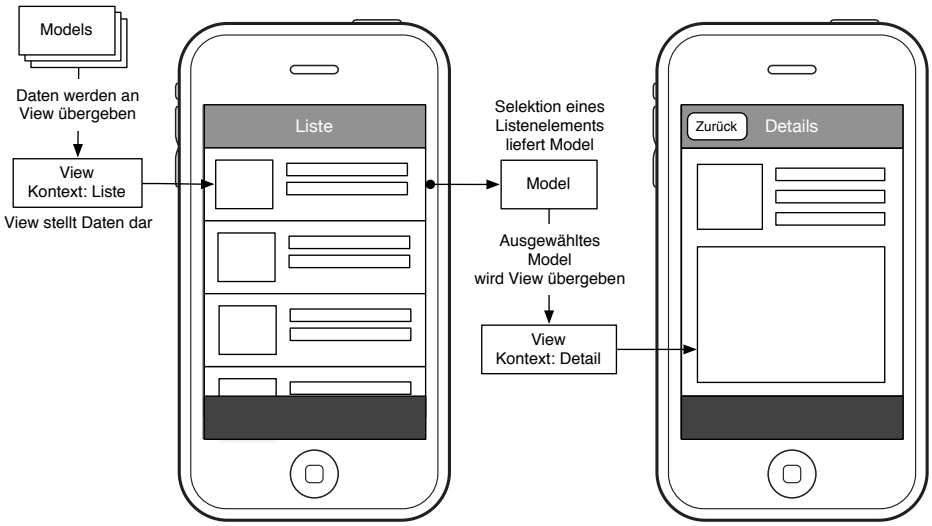

Abbildung 4.1: Vereinfachte Darstellung der Kommunikation.

tenhaltung und der Datenaustausch zwischen den *Events* wird über *Models* realisiert. *Models* bilden eine einfache Abstraktionsschicht zur darunterliegenden Datenbank ab. Die Darstellung der Daten geschieht wiederum über *Components* welche die *Models* über *Views* abbilden. Letztere können mit einer an CSS angelehnten Beschreibungssprache jederzeit angepasst werden. In Abb. 4.1 ist der Zusammenhang der Bausteine dargestellt.

Das gesamte System basiert auf den fünf erwähnten modularen Bausteinen, welche mit einer auf JSON basierenden Beschreibungssprache zu einer App zusammengebaut werden können.

4.1.2 Einschränkungen

Der gewählte Ansatz unterliegt diversen Einschränkungen, um einerseits die Komplexität des Systems in Grenzen zu halten und andererseits gewissen technischen Spezifikationen gerecht zu werden. Da Apple aus Sicherheitsgründen die dynamische Generierung und das Nachladen von Quellcode aus dem Internet und die darauffolgende Ausführung zur Laufzeit der Anwendung untersagt, musste bei der Entwicklung darauf verzichtet werden. Dies brachte Vor- und Nachteile mit sich. Auf der einen Seite wurde dadurch die Komplexität des Systems reduziert und somit die Entwicklung vereinfacht, auf der anderen Seite mussten dadurch die *Components* und deren Funktionalität stärker eingeschränkt werden.

Das System beschränkt sich deshalb auf eine Anwendungsstruktur, die aus zwei Navigationsebenen besteht. Jede dieser Ebenen kann jedoch beliebig viele *Components* bzw. Ansichten enthalten. Ein Beispiel dafür ist in

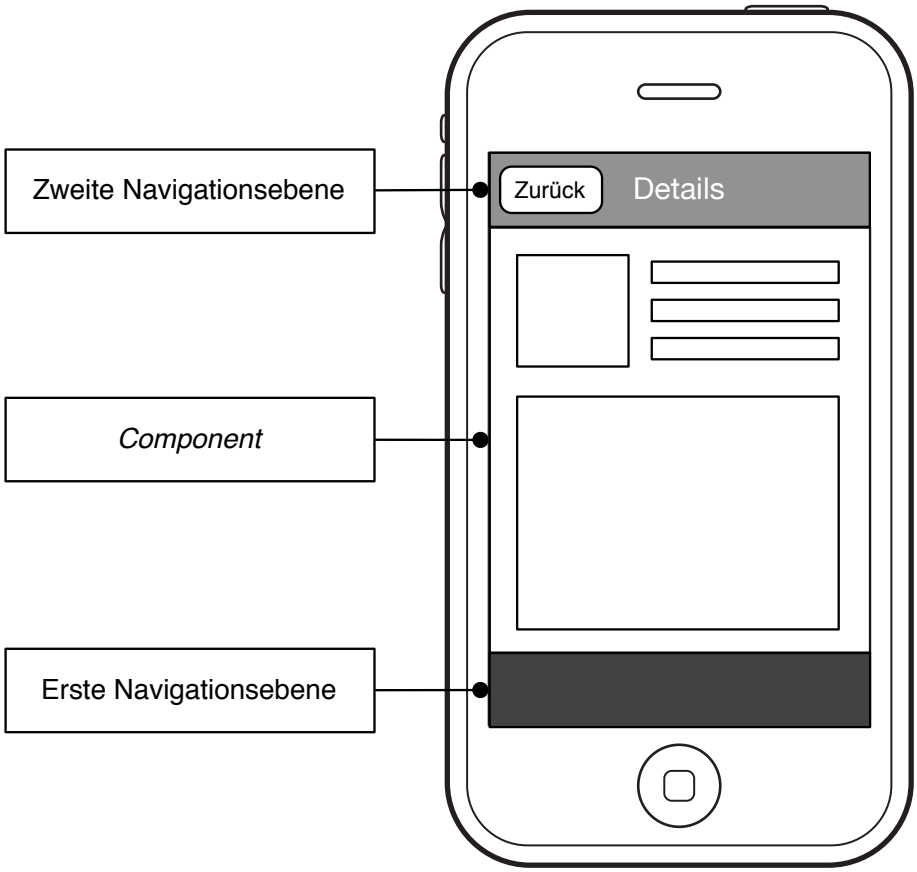

Abbildung 4.2: Navigationsebenen und *Components*.

Abb. 4.2 zu sehen. Was die Funktionalität betrifft, wurde das System auf die Verwaltung von benutzerdefinierten Datensätzen beschränkt. Das bedeutet, dass beliebige Datensätze mit unterschiedlichsten Feldern modelliert werden können. Die Anwendung generiert auf Grund dessen Formulare für die Erstellung dieser Modelle automatisch. Die Speicherung der Daten erfolgt dabei ebenfalls ohne weiteren Aufwand in einer Datenbank.

4.1.3 Findung und Evaluierung der Komponenten

Einer der wichtigsten Schritte war die Findung der richtigen Komponenten. Dabei lag der Fokus vor allem auf der Wiederverwendbarkeit. Dazu wurden bestehende Apps und deren Arbeitsweise genau analysiert. Wie im Abschnitt 2.3 bereits erläutert, werden am Smartphone komplexe Arbeitsabläufe meist in kleinere modulare Schritte zerlegt. Das hat zur Folge, dass der begrenzte Platz am Bildschirm des Smartphones besser ausgenützt werden kann.

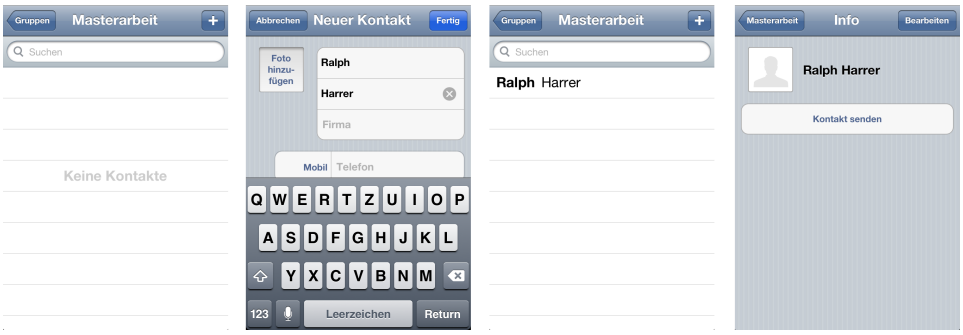

Abbildung 4.3: Schritte zur Erstellung eines neuen Kontakts.

Smartphoneanwendungen bestehen aus einem Grundstock von User-Interface-Komponenten, welche sich in den Apps immer wieder finden lassen. Diese wurden in der Recherchephase ermittelt und als eigenständige Bauteile isoliert. Als gutes Beispiel dafür dient die Listenansicht: Diese wird oft verwendet um Daten aufzulisten. Zusätzlich erlaubt diese aber die Selektion und das Löschen von Daten. Weiters können die Zellen frei formatiert werden, wodurch das Erscheinungsbild nach Belieben angepasst und die Ausgabe der Daten bestimmt werden kann. Das macht die Listenansicht zu einer vielseitigen Komponente, die somit für viele unterschiedliche Zwecke zum Einsatz kommen kann.

In Abb. 4.3 ist ein üblicher Programmablauf unter *iOS* dargestellt. Die aufgeführten Schritte zeigen das Erstellen eines neuen Kontaktes in der „Adressbuch"-App. Dabei wird zuerst die gewünschte Gruppe ausgewählt, was zu einer Auflistung der Kontakte der Gruppe führt. In dieser Ansicht hat man nun die Möglichkeit, sich die Details eines Kontakts anzeigen zu lassen oder einen neuen Eintrag zu erstellen. Letzteres blendet ein Formular ein, welches wieder den gesamten Bildschirm ausfüllt. Nach dem Anlegen erscheint der Kontakt in der Listenansicht und kann durch Auswählen in einer Detailansicht angezeigt werden.

Dieser Ablauf spiegelt die Arbeitsweise des behandelten Ansatzes des komponentenbasierten Systems wieder. Alle Ansichten würden *Components* entsprechen. Das Betätigen der „+"-Schaltfläche oder das Antippen eines Kontakts würde ein *Event* auslösen, welches ein *Model* entgegennimmt und dieses erneut einer neuen *Component* weiterreicht.

Das gesamte System ist so ausgelegt, dass es jederzeit um neue Bausteine erweitert werden kann. Mehr Informationen zur Erweiterung des Systems sind in Kapitel 5 zu finden.

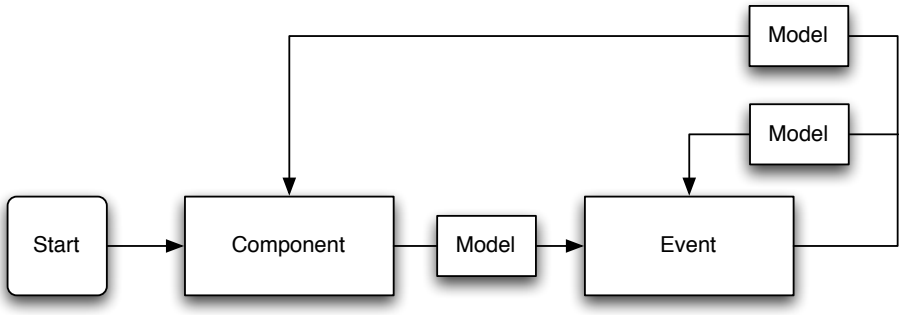

Abbildung 4.4: Ablaufdiagramm des Kombonentensystems.

4.1.4 Aufbau

Wie bereits erwähnt, arbeitet das System mit vier unterschiedlichen Bausteintypen: *Components*, *Events*, *Models* und *Views*. Jede Instanz eines Bausteins trägt einen eindeutigen Namen, über die sie identifiziert werden kann. Die Referenzen untereinander basieren ebenfalls auf diesen eindeutigen Namen. So verweist eine *Component* auf ein *Event* indem der Name des jeweiligen *Events* angegeben wird.

Der Aufbau und die möglichen Kombinationen sind in Abb. 4.4 dargestellt. Jede Anwendung beginnt mit der Startkomponente, welche lediglich eine Referenz auf die erste *Component* hält. Von nun an arbeitet die Anwendung über *Trigger* (näheres zu diesem Thema in Abschnitt 4.1.5). Diese lösen immer ein *Event* aus, das nach der Abarbeitung entweder wieder ein *Event* auslösen kann, oder wieder zu einer *Component* führt. Bei dem Übergang, zwischen zwei Bausteinen, wird immer ein einzelnes *Model* oder eine Liste von *Models* weitergereicht. Der Empfänger entscheidet dann, ob er mit der Liste oder mit dem einzelnen *Model* arbeitet und welche Aktion darauf angewandt wird.

Die Einzelheiten und Arbeitsweisen zu den jeweiligen Bausteinen werden im folgenden Abschnitt genauer beschrieben.

4.1.5 Components

Components stellen die Schnittstelle zwischen dem System und dem Benutzer dar. Sie beinhalten das User Interface der jeweils darzustellenden Ansicht und reagieren auf Benutzereingaben. Um die Verwendung von *Components* möglichst einfach zu gestalten, stellt jede *Component* gewisse Schnittstellen nach außen hin zur Verfügung, welche mit *Events* verknüpft werden können. Diese Schnittstellen werden in dieser Arbeit als *Trigger* bezeichnet. Ein *Trigger* wird dann aufgerufen, sobald ein gewisses Systemereignis, wie das Betätigen

Trigger-Name	Beschreibung
Init	Wird beim ersten Laden der *Component* ausgeführt.
BecomesActive	Führt den Trigger aus, wenn die *Component* aktiv wird. Das ist dann der Fall, wenn diese nach der Initialisierung das erste Mal angezeigt wird und wenn diese beim Navigieren verdeckt und wieder aufgedeckt wird
BecomesInactive	Führt den Trigger aus, sobald die *Component* von einer anderen verdeckt wird.

Tabelle 4.1: Allgemeine *Trigger* einer *Component*.

einer Schaltfläche, eingetreten ist. Welcher *Trigger* bei welchem Systemereignis an das komponentenbasierte System weitergeleitet werden soll, kann vom Entwickler frei entschieden werden. Empfängt das komponentenbasierte System nun einen *Trigger*, wird ein in der Beschreibungssprache verknüpftes *Event* ausgeführt. Durch dieses System kann der Benutzer selbst entscheiden, welches Ereignis, bei gewissen Benutzerinteraktionen ausgelöst werden soll. Wird eine Schnittstelle vom Benutzer nicht genutzt, wird diese deaktiviert und die Quelle dafür ausgeblendet. Verwendet man z. B. den *Trigger* eines Buttons nicht, wird dieser von der *Component* nicht gerendert und bleibt somit vor dem Benutzer verborgen. In 4.1 sind alle *Trigger* aufgeführt, die eine normale *Component* zur Verfügung stellt.

Jede *Component* nimmt je nach Typ entweder eine Liste von *Models* oder ein einzelnes *Model* bei der Initialisierung entgegen. Das Model wird daraufhin darüber informiert, in welchem Kontext es dargestellt wird. Dieses entscheidet dann, welche *View* für die Darstellung der Daten herangezogen werden soll. Die *Component* verwendet dann die Ansicht des *Models* und fügt diese in das User Interface ein.

In Abb. 4.5 ist die Struktur einer *Component* dargestellt. Jeder Baustein des Systems muss über einen einmaligen Namen verfügen, über den die Referenzen hergestellt werden können („ComponentId“-Feld). Der „Type“ legt fest, um welche Art von *Component* es sich handelt. Diese werden in den folgenden Absätzen genauer beschrieben. Das „Events“-Feld besteht aus einer Schlüssel-Wert-Paarung, was in der Programmierung als Dictionary bezeichnet wird. Dadurch wird der Name des *Triggers* (z. B. „ItemClicked“) mit der Referenz des gewünschten *Events* verbunden. Die beiden Felder „Model“ bzw. „Models“ halten die Daten, je nach dem ob es sich um ein einzelnes *Model* oder um eine Liste von *Models* handelt.

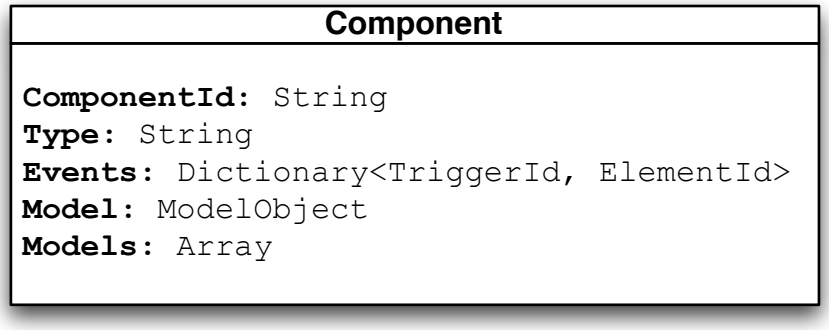

Abbildung 4.5: Vereinfachte Darstellung der internen Struktur einer *Component*.

Trigger-Name	Beschreibung
Add	Wird beim Betätigen der „+“-Schaltfläche ausgelöst.
Delete	Durch das Durchstreichen eines Listenelements wird der „Löschen“-Button eingeblendet. Dieser löst beim betätigen den „Delete“-Trigger aus.
ItemClicked	Sobald ein Listenelement berührt wird, wird dieser *Trigger* ausgeführt.
ItemLongClicked	Wenn eine lange Berührung eines Listenelements stattfindet, wird dieser *Trigger* gestartet.

Tabelle 4.2: Neue *Trigger* einer Listen-*Component*.

Listen

Die Listen-*Component* dient zur Repräsentation von Tabellen bzw. Listen. Die Daten stammen dabei aus einer Auflistung von *Models*, welche im Kontext einer Liste gerendert werden. Das Aussehen der einzelnen Listenelemente wird durch Zellen-*Views* definiert. Jedes *Model* muss dabei wissen, welche *View* es zu verwenden hat und welche Daten auf welche Felder innerhalb der *View* zugewiesen werden müssen. Im Abschnitt 4.1.8 werden alle Einzelheiten zu *Views* und dem Daten-Mapping beschrieben.

In Abb. 4.6 ist der Aufbau einer Listen-*Component* abgebildet. Neben den vererbten *Triggert* von der Basis-*Component*, stellt diese vier weitere Schnittstellen zur Verfügung, welche in 4.2 aufgeführt sind.

Karten

Karten-*Components* sind den Listen sehr ähnlich. Sie übernehmen ebenfalls eine Liste von *Models* welche aber ein Kartenfeld mit Koordinaten aufweisen

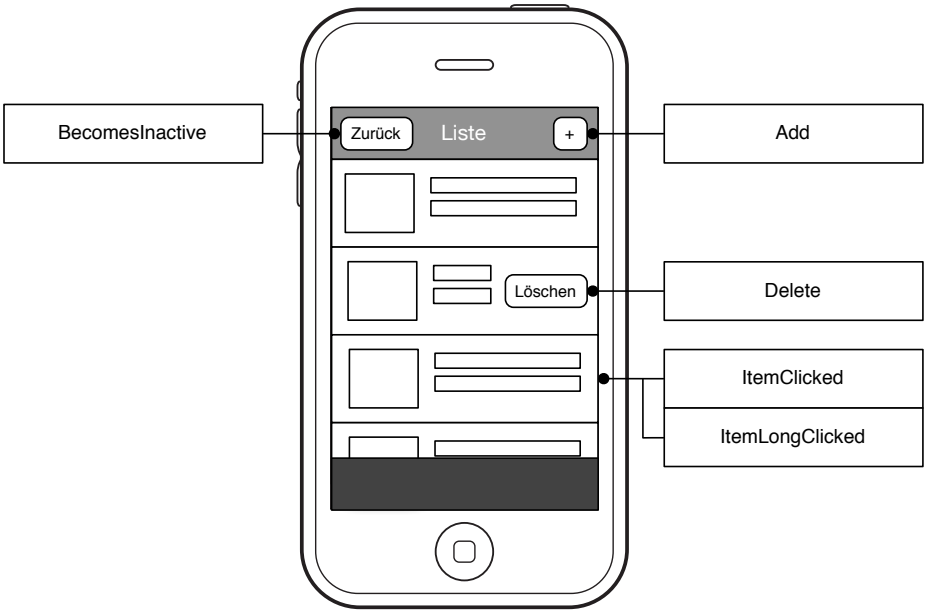

Abbildung 4.6: List-*Component* mit allen möglichen *Triggers*.

müssen. Ist das der Fall, werden die *Models* im Kontext der Karte gerendert. Das bedeutet, dass das jeweilige *Model* darüber informiert wird, dass es in einer Karte dargestellt wird. Daraufhin greift das *Model* auf die zugewiesene Karten-*View* zu, übergibt dieser die Daten und reicht die befüllte *View* der Map-*Component* weiter, welche die Ansicht an den jeweiligen Koordinaten rendert. Das *Model* auf der Karte stellt wie eine Zelle in der List-*Component* zwei *Trigger* (dargestellt in Abb. 4.7) zur Verfügung. Diese reichen das selektierte *Model* an das Folgeevent weiter.

Formulare

Formulare stellen eine spezielle Art der *Components* dar. Diese nehmen ein einzelnes *Model* entgegen und generieren aus dessen Feldern ein Formular. Dieser Vorgang ist in Abb. 4.8 abgebildet. Im Gegensatz zu den bereits vorgestellten List- und Map-*Components*, dient das Formular nicht zur Darstellung sondern zur Bearbeitung der Daten. Es ermöglicht die Befüllung und die Modifikation des übergebenen *Models*. Die Felder des Formulars werden dynamisch auf Grund der Modellbeschreibung generiert. Durch die Selektion eines dieser Felder, wird eine Eingabemaske angezeigt, mit der der Wert des Feldes bearbeitet werden kann. Nähere Informationen zu den möglichen Feldern sind im Abschnitt 4.1.7 zu finden.

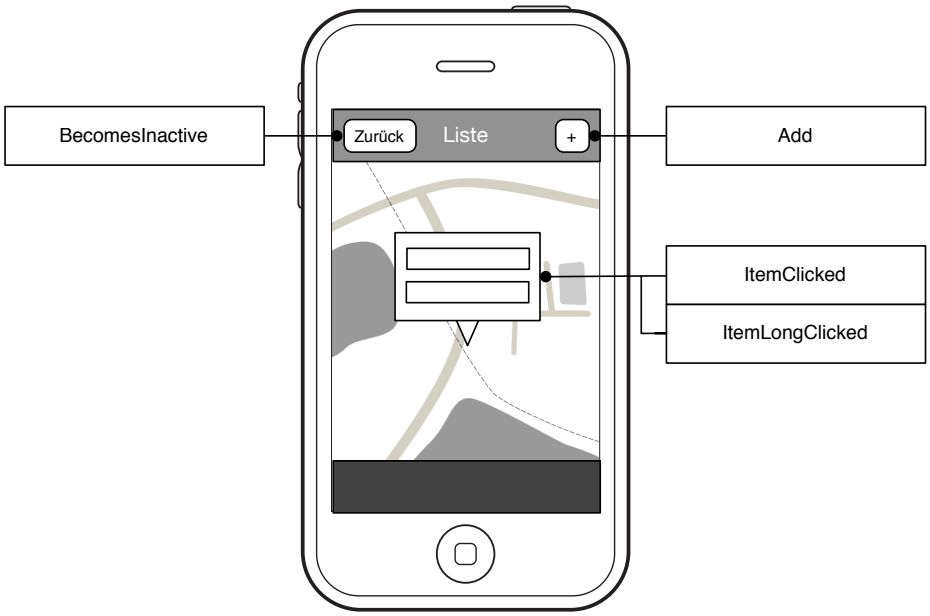

Abbildung 4.7: Übersicht der Karten-*Component*.

Allgemeine Ausgaben

Die *Component* für allgemeine Ausgaben dient dazu, *Models* auf einfache Art und Weise auszugeben. Dabei werden wie bei den Formularen die Felder des übergebenen Models geladen und die Werte in die entsprechende *View* eingesetzt. Um dies zu bewerkstelligen, muss das *Model* über eine Zuweisung zu einer *View* verfügen und die Felder an die entsprechenden User-Interface-Elemente in der visuellen Repräsentation binden. Danach werden die Felder des *Models* in einer Liste ausgegeben, welche wiederum selektierbar sind und die beiden Trigger *ItemClicked* und *ItemLongClicked* zur Verfügung stellen (weitere Informationen dazu sind in Tabelle 4.1 angegeben), um *Events* auslösen zu können.

4.1.6 Events

Events ermöglichen die Modifikation und den Austausch der *Models* zwischen den *Components*. Wird ein *Event* ausgelöst, muss diesem mitgeteilt werden, was es zu tun hat und an welchen Baustein das Ergebnis weiterzureichen ist. Der Baustein kann wieder ein *Event* sein, das erneut eine Änderung an den übergebenen Daten vornimmt, oder ein *Component*, welche die Daten darstellt, oder wie im Falle eines Formulars, die Modifikation der Daten ermöglicht.

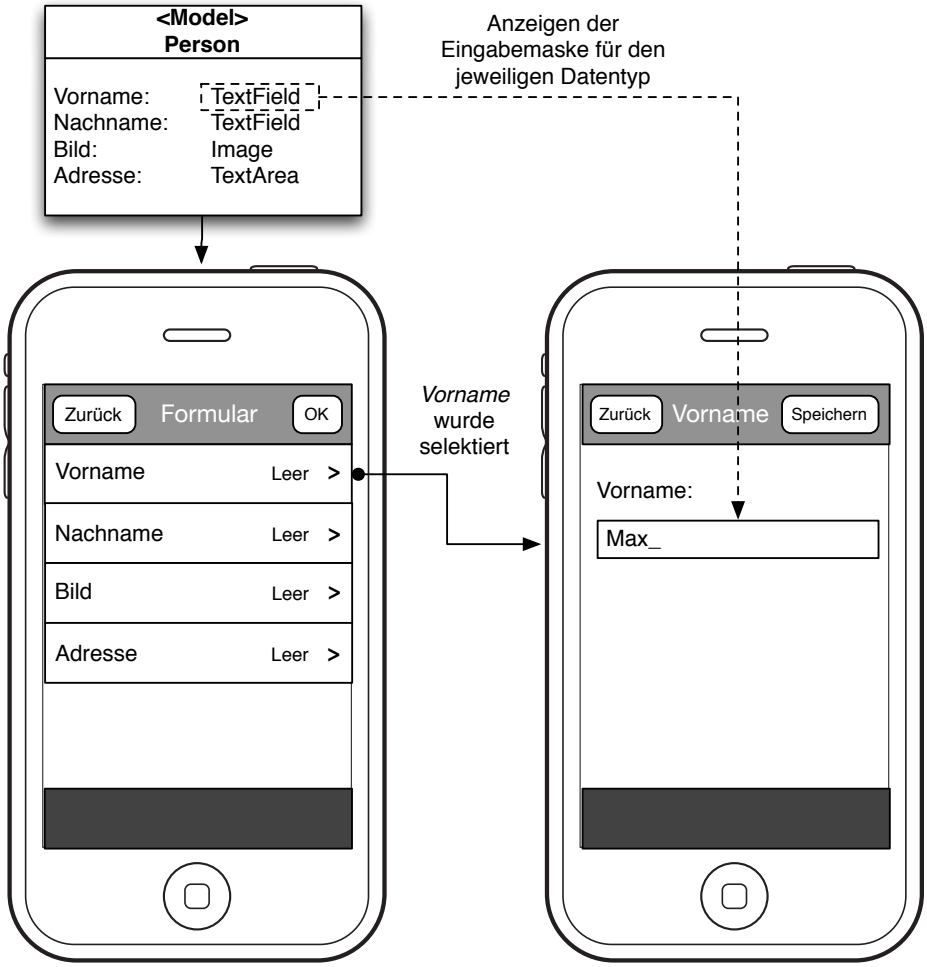

Abbildung 4.8: Darstellung der dynamischen Erzeugung der Formularfelder aufgrund der *Model*-Beschreibung.

Aufbau

Ein *Event* muss über zwei Dinge informiert werden: Einerseits muss ihm mitgeteilt werden, welche Aufgabe es zu erledigen hat („Type“-Feld, Abb. 4.9), was dem *Event*-Typ entspricht, und andererseits benötigt es die Referenz eines Empfängers, an den die Daten weitergereicht werden sollen („ElementId“-Feld, Abb. 4.9). Diese beiden Informationen werden beim Initialisieren des *Events* festgelegt. Optional kann jedem *Event* noch mitgeteilt werden, nach wie vielen Sekunden die jeweilige Aktion auszulösen ist. Je nach *Event*-Typ können aber noch mehr Parameter übergeben werden.

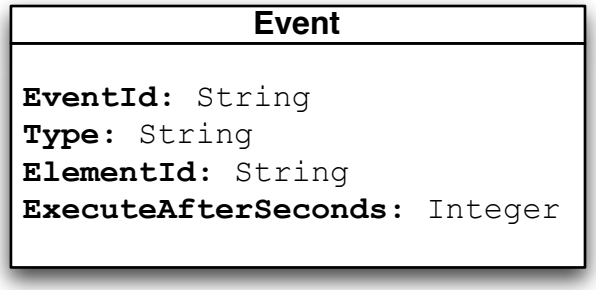

Abbildung 4.9: Vereinfachte Darstellung der internen Struktur eines *Events*.

Typen

Zur Zeit gibt es im System eine sehr überschaubare Menge an vorgegebenen *Event*-Typen, welche für die Erstellung eigener Anwendungen herangezogen werden können. Diese werden je nach Funktion in drei Gruppen unterteilt:

1. Modellmodifikation
2. User-Interface-Modifikation
3. Navigationsmodifikation

Modellmodifikation

Diese *Events* sind dafür zuständig, *Models* zu manipulieren. Dabei werden folgende CRUD-Operationen abgedeckt: `Create`, `Retrieve`, `Update` und `Delete`.

Zum Erstellen von neuen, leeren, temporären *Model*-Instanzen wird ein `CreateModel`-*Event* verwendet. Dieses erwartet zwingend die Referenz auf eine *Model*-Beschreibung und erneut eine Referenz auf den Baustein, an welchen das erzeugte *Model* weitergereicht werden soll. Ein Beispiel dafür ist in Abb. 4.10 dargestellt.

Analog wird bei dem `LoadModels`-*Event* verfahren. Dieses lädt gespeicherte *Models* aus der Datenbank, welche der referenzierten *Model*-Beschreibung entsprechen. Dabei kann optional die Sortierung in an- oder absteigender Ordnung angegeben werden.

Um *Model*-Instanzen permanent in der mitgelieferten Datenbank abzuspeichern, wird das `AddModel`-*Event* verwendet. Im Gegensatz du den zwei bereits erwähnten Typen, benötigt das *Event* keine Referenz auf eine Beschreibung eines *Models*, sondern es erwartet die Instanz des erstellten, noch nicht gespeicherten *Models*. Dieses wird daraufhin in die Datenbank geschrieben und automatisch mit einer einmaligen Identifikationskennung versehen.

Ähnlich arbeitet das `UpateModel`-*Event*. Dieses erwartet ebenfalls eine

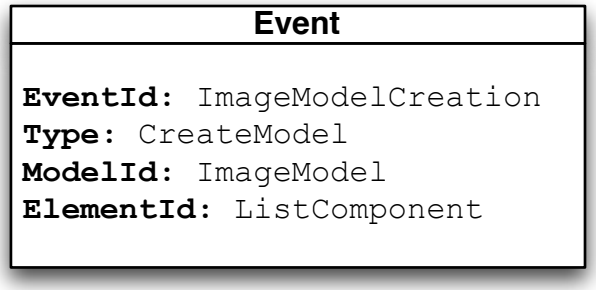

Abbildung 4.10: Ein Beispiel für ein *Event* welches ein neues leeres *Model* aufgrund der Beschreibung mit dem Name „ImageModel“ instanziert.

Instanz eines *Models*. Im Gegensatz zum `AddModel`-*Event* muss das übergebene *Model* bereits in der Datenbank existieren und mit einer einmaligen Identifiktionskennung versehen sein, über die es in der Datenbank gefunden werden kann. Ist das der Fall, werden alle Änderungen, die an dem *Model* vorgenommen wurden permanent in die Datenbank übertragen.

User-Interface-Modifikation

Diese Art von *Events* kümmern sich um die visuelle Modifikation des User Interfaces. Damit können zum Beispiel kurze Informationsnachrichten dem User präsentiert oder selektierte Tabellenzellen wieder deselektiert werden. Durch die Möglichkeit *Events* zweitverzögert ausführen zu können, lassen sich zum Beispiel Animationen zu einem gewählten Zeitpunkt starten.

Wie bereits erwähnt, gibt es die Möglichkeit dem Benutzer kurzzeitig Informationen einblenden zu lassen. Dieses *Event* wird mit dem „Type“ `DisplayMessage` erzeugt. Damit ein Text angezeigt werden kann, muss das „Message“-Feld des *Events* mit der gewünschten Nachricht belegt werden. Optional kann über den Parameter „ForSeconds“ angegeben werden, wie lang die Nachricht auf dem Bildschirm des Smartphones angezeigt werden soll.

Das Gegenstück dazu trägt den Namen `HideMessage` und blendet alle angezeigten Nachrichten, welche nicht von alleine vom Bildschirm entfernt werden, wieder aus.

Zum Deselektieren von angewählten Listenelementen oder Tabellenzellen, gibt es das *Event* mit dem „Type“-Namen `DeselectRow`.

Navigationsmodifikation

Diese Art von *Events* beeinflussen die Navigation zwischen den einzelnen *Components*. Sie benötigen die Art der Navigationsmanipulation und gegebenenfalls die Referenz der zu zeigenden *Component*. Das `Push`-*Event* benötigt letztere, um die dazugehörige *Component* zu instanzieren und auf den inter-

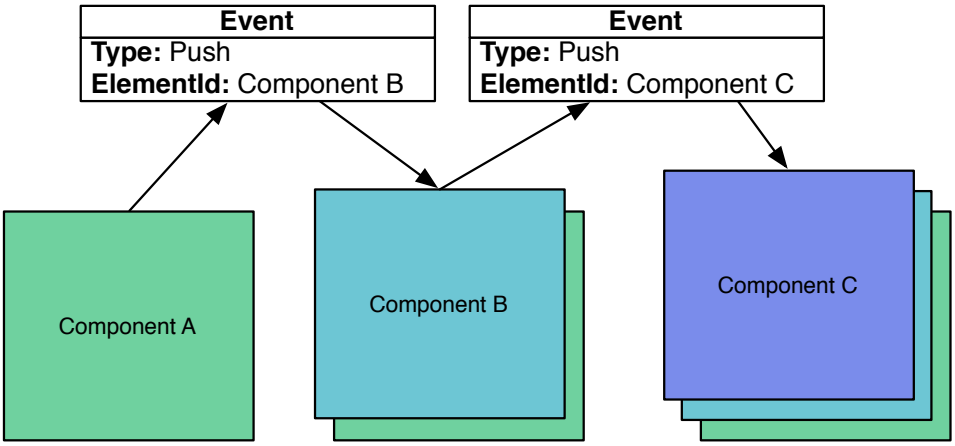

Abbildung 4.11: Arbeitsweise der Navigationsmodifikation mittels `Push`-*Events*.

nen „Component"-Stapel zu legen. Das hat zur Folge, dass sich der gesamte Bildschirm ändert. Dieser Vorgang ist in Abb. 4.11 dargestellt.

Möchte man das umgekehrte Ergebnis erziehlen, so muss ein `Pop`-Event instanziert werden. Diese benötigt jedoch keine Referenz auf eine *Component*, da es lediglich die oberste Ansicht vom internen Stapel entfernt und die darunterliegende *Component* wieder zum Vorschein bringt. So ähnlich arbeitet der letzte Vertreter der navigationsmodifizierenden Events. Es handelt sich dabei um das `GoToRoot`-*Event*. Dabei werden alle, sich auf dem internen Stapel befindlichen *Components* entfernt, bis man beim Ausgangspunkt angelangt ist. In Abb. 4.12 ist die Arbeitsweise der beiden letzten *Events* illustriert. Bei all diesen Operationen werden immer die intern verwendeten *Models* weitergereicht.

4.1.7 Models

Die *Models* dienen zum Austausch, zur temporären Haltung und zur dauerhaften Speicherung von Daten.

Aufbau

Wie die bereits vorgestellten Bausteine, benötigen auch *Models* einen einmaligen Namen, um auf diese referenzieren zu können. Die zwei wichtigsten Bestandteile der *Models* sind aber jene Felder, welche die zu speichernden Daten repräsentieren und die *Views*, welche für die Darstellung der Daten verantwortlich sind.

Jedes *Model* kann beliebig viele Felder beinhalten, welche wiederum über einen einmaligen Namen verfügen müssen. In Tabelle 4.3 sind alle möglichen

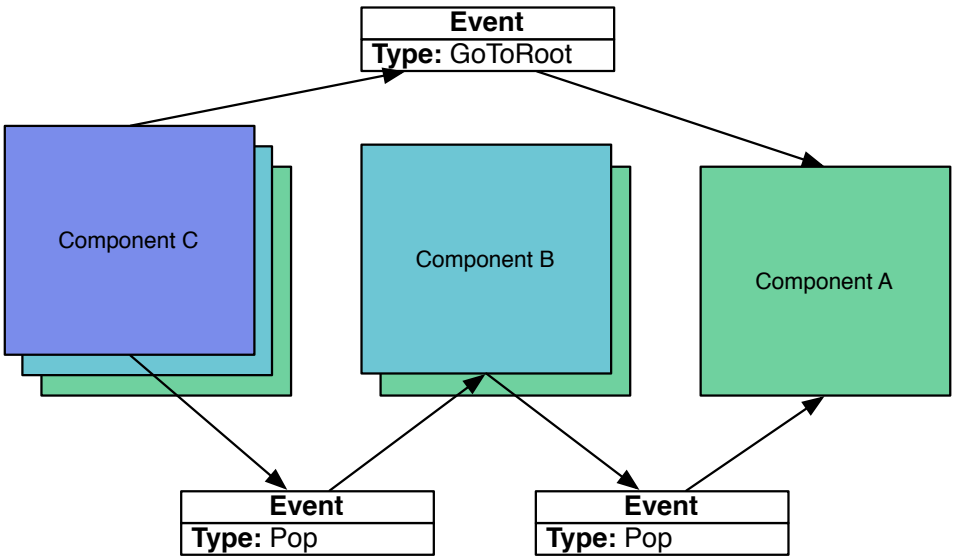

Abbildung 4.12: Arbeitsweise der Navigationsmodifikation mittels `Pop`-*Events*.

Feldtypen mit einer Beschreibung angegeben. Zusätzlich bietet jedes Feld Optionen an, welche zur Validierung und zur Darstellung der Daten dienen. Einem „TextFeld“ kann zum Beispiel eine minimale und/oder eine maximale Länge und ein erwartetes Format (z. B. E-Mail Adresse) zugewiesen werden. Dadurch lassen sich nur gewisse Werte in dieses Feld eintragen. Falls der Benutzer Daten in das Feld einträgt die nicht den erforderten Validierungskriterien entsprechen, wird vom Formular eine Warnung angezeigt, die den Benutzer darüber informiert, welche Daten für das jeweilige Feld gültig sind. Die Darstellung des Feldes, wenn es in einem Formular dargestellt wird, kann ebenfalls über zusätzliche Optionen gesteuert werden.

Um die Darstellung der Daten zu gewährleisten, muss jedes *Model* über Referenzen auf *Views* verfügen. Da ein *Model* in unterschiedlichsten *Components* gerendert werden kann, wurde ein *Kontext* eingeführt. Dieser gibt an, welche Art der Darstellung in der jeweiligen *Component* gewählt werden soll. Die folgenden *Kontexte* gibt es bereits im System: *Table* für die Listen-/Tabellen-, *Map* für Karten- und *Detail* für Detailansichten. Durch dieses System ist es möglich, ein *Model* auf unterschiedliche Arten dem Benutzer zu präsentieren. Wie die Zuweisung im Detail funktioniert, kann in Abschnitt 4.1.8 und 4.2 nachgelesen werden.

Datenbank

Damit die Daten dauerhaft gespeichert werden können, musste ein passendes Datenhaltungsmodell gefunden werden. Da die zu speichernden *Models*

Name	Beschreibung
`NumberField`	Dient zum Speichern von Zahlen. Optional kann der Wertebereich festgelegt werden.
`TextField`	Speichert Texte beliebiger Länge. Optional können eine Mindest- und Maximallänge und das gewünschte Format angegeben werden. Das Erscheinungsbild des Eingabefelds kann nach Belieben auf ein-, mehrzeilig oder Passwortfeld gesetzt werden.
`RangeField`	Dieses Feld dient zum Speichern von Zahlenbereichen. Diese können mit einem beliebigen Start- und Endwert versehen werden.
`MapField`	Dient zum Speichern von Koordinatenpaaren. Zusätzlich können folgende Daten für die Darstellung im Formular angegeben werden: Die Startkoordinaten, die Zoomstufe, die Darstellungsart und ob mit der Karte interagiert werden darf (Scrolling, Zooming).
`ImageField`	Dieses Feld dient zum Abspeichern von einem Bild. Es kann dabei angegeben werden, von welchen Quellen das Bild bezogen werden darf. Zur Auswahl stehen die Kamera, die Smartphone-Bibliothek und die Direkteingabe einer URL.
`DateField`	Erlaubt die Speicherung von einem Datum. Zusätzlich lässt sich das Datumsformat frei wählen.
`TimeField`	Erlaubt die Speicherung von einer bestimmten Zeit.
`URLField`	Ermöglicht das Speichern von einer Webadresse.
`ReferenceField`	Dieses Feld dient zum Speichern von Referenzen auf bestehende *Models*. Zusätzlich kann festgelegt werden, wie viele Referenzen mindestens und maximal gewählt werden dürfen.

Tabelle 4.3: Auflistung und Beschreibung aller möglicher Feldtypen eines *Models*.

in ihrer Gestaltung sehr flexibel sind, konnte ein relationales Datenmodell nur schwer angewendet werden. Aus diesem Grund wurde auf das *Entity, Attribute und Value*-Modell, kurz *EAV*-Modell zurückgegriffen. Dabei werden die einzelnen Attribute (welchen den Felder entsprechen) als eigene Datensätze in einer relationalen Datenbank abgespeichert. Durch diesen flexiblen Ansatz ist es möglich, bestehende *Models* jederzeit zu erweitern oder zu ändern und diese Änderungen auf die Datenbank zu übertragen. In Abb. 4.13 ist der Unterschied der beiden Datenbankmodelle abgebildet.

Die Kommunikation mit der Datenbank wird, wie in Abschnitt 4.1.6 be-

Relationales-Modell

ModelID	Vorname	Nachname	Bild	Adresse
101	Max	Mustermann	Max.jpg	Hauptstr...
102	Susanne	Meister	Susanne.jpg	Feldweg...
...	...	...	...	...

EAV-Modell

ModelID	Type
101	Person
102	Person
...	...

ModelRefID	Attribute	Value
101	Vorname	Max
101	Nachname	Mustermann
101	Bild	Max.jpg
101	Adresse	Hauptstr...
102	Vorname	Susanne
102	Nachname	Meister
...	...	...

Abbildung 4.13: Unterschiede zwischen einem relationalem und dem EAV Datenbankmodell.

reits erwähnt, von den modellmodifizierenden *Events* übernommen. Diese übernehmen die Erstellung, Veränderung, Löschung und das Abrufen von Datensätzen und erstellen daraus wieder *Models*, welche an die nachfolgenden *Events* oder *Components* weitergereicht werden.

4.1.8 Views

Die *Views* dienen zur Darstellung und visuellen Aufbereitung von *Models*. Der Fokus lag bei der Entwicklung des Systems darauf, die *Views* für beliebige *Models* verwendbar zu machen. Aus diesem Grund verfügt jede *View* über einen einmaligen Namen, über den sie referenziert werden kann. Innerhalb einer *View* befinden sich diverse User-Interface-Elemente (wie z. B. Textfelder, Labels, Bilder usw.), welche zur Darstellung diverser Daten herangezogen werden können. Auch diese verfügen über einen einmaligen Namen.

Damit die Daten eines *Models* in einer *View* dargestellt werden können, muss eine Zuweisung der Felder eines *Models* zu den jeweiligen User-Interface-Elementen innerhalb der *View* angegeben werden. In Abb. 4.14 ist ein Beispiel abgebildet, wie die Zuweisung funktioniert. Diese so genannten *View-Field-Mappings* müssen in der Definition des *Models* festgelegt werden. Näheres zu diesem Thema kann im Abschnitt *Beschreibungssprache*, 4.2 nachgelesen werden.

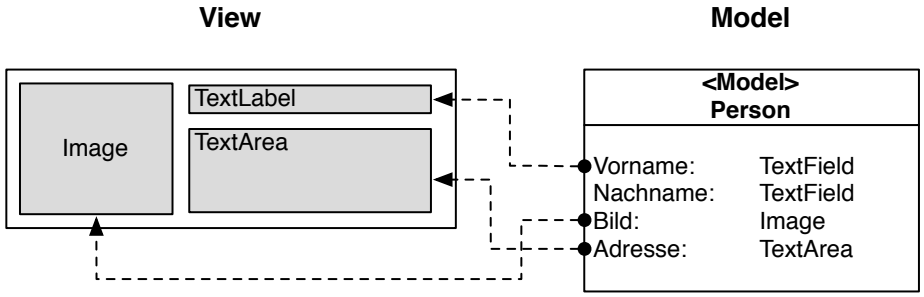

Abbildung 4.14: Zuweisung der Felder eines *Models* an die User-Interface-Elemente einer *View*.

Gestaltung der Views

Die *Views* können nach Belieben vom Benutzer über eine Cascading Style Sheet ähnliche Syntax erstellt werden. Durch das beliebige Hinzufügen und das freie Gestalten von User-Interface-Elementen hat der Benutzer die größtmögliche Freiheit, die Ansichten nach eigenen Wünschen anzupassen.

Jedes hinzugefügte Feld verfügt über einen Typ, einen Namen und Gestaltungsoptionen. Der Typ gibt an, um welches User-Interface-Element es sich handeln soll. Der Name ermöglicht das *View-Field-Mapping*, um die Daten eines Feldes eines *Models* in die Ansicht zu laden. Die restlichen Optionen dienen zur Gestaltung der Ansicht. Wie die im Detail funktioniert kann im Abschnitt *Beschreibungssprache*, 4.2 nachgelesen werden.

4.2 Beschreibungssprache

Die Beschreibungssprache dieses komponentenbasierten Systems dient dazu, Anwendungen ohne Programmierkenntnisse zu erstellen. Dabei werden *Components*, *Events*, *Models* und *Views* angelegt, gestaltet und miteinander verknüpft.

4.2.1 Allgemein

Für die Beschreibungssprache kam die JavaScript Object Notation, kurz JSON zum Einsatz. Diese wurde bewusst gewählt, da sie sehr kompakt, schlicht sowie übersichtlich ist und darüberhinaus intern Datentypen wie Arrays und Objekte zur Datenrepresäntation verwendet. Dadruch lassen sich komplexe Strukturen einfach und mit wenig Aufwand erstellen und abbilden.

4.2.2 Struktur

Bei der Entwicklung der Beschreibungssprache lag der Fokus vor allem auf der Wiederverwendbarkeit der definierten Bausteine. Aus diesem Grund wurde auf eine verschachtelte Anordung verzichtet und im Gegensatz dazu, eine flache Hirarchie gewählt. Dadurch ist jeder Baustein frei zugänglich und über dessen Namen referenzierbar. Bausteine die oft benötigt werden können dadurch wiederverwendet werden, was die Komplexität der Beschreibungssprache verringert und die Übersicht erhöht. Die Umgestaltung von bestehenden Anwendungen lässt sich dadurch ebenfalls leicht realisieren, da lediglich die „Verbindungen“ der Bausteine untereinander getauscht werden müssen, anstatt mitunter große Beschreibungsfragmente an andere Stellen verschieben zu müssen.

Die Bausteine werden in der Beschreibungssprache als JSON-Objekte definiert. Die jeweiligen Eigenschaften werden als Felder im JSON-Objekt angegeben. Als Wert können entweder Strings, Integer, Floats, Boolsche-Werte, Arrays oder erneut Objekte übergeben werden.

Insgesamt werden folgende Bausteintypen mit Hilfe der Beschreibungssprache definiert: *Components*, *Models*, *Events* und *Views*. Um die Übersicht bei großen Anwendungen gewährleisten zu können, werden die Bausteinbeschreibungen in vordefinierten Bereichen abgelegt. Davon gibt es insgesamt fünf verschiedene, wovon vier zur Speicherung der vier Grundbausteine dienen. Der fünfte Bereich stellt den Einstiegspunkt `Root` der Anwendung dar, welcher im Abschnitt 4.2.2 beschrieben wird. Folgender Code bildet die Grundstruktur der Beschreibungssprache ab:

```
1 {
2   "Root": [...],
3   "Components": {...},
4   "Events": {...},
5   "Models": {...},
6   "Views": {...}
7 }
```

Die Reihenfolge der Bereiche (`Root`, `Components`, ...) kann dabei frei gewählt werden, müssen aber für den fehlerfreien Ablauf der Anwendung vorhanden sein. Im Gegensatz zu den restlichen Bereichen nimmt `Root` kein Objekt, sondern ein Array von Objekten entgegen. Diese beinhalten die Referenzen auf die Start-*Components*, welche in der Hauptnavigation der Anwendung abgebildet werden.

Einstiegspunkt

Jede Beschreibung muss mindestens einen Einsprungspunkt für die Anwendung zur Verfügung stellen. Dieser legt unter anderem fest, mit welcher *Component* die Anwendung gestartet werden soll. Zusätzlich kann angegeben werden, welcher Titel und welches Icon in der Hauptnavigation verwendet

und welches *Model* für die gewünschte *Component* initialisiert werden soll. Die Reihenfolge der Startbausteine wird in der Hauptnavigation abgebildet. Folgendes Beispiel zeigt, wie zwei Einstiegspunkte definiert werden können:

```
{
  "Root": [{
    "Title": "Bilder",
    "Icon": "images_icon.png",
    "ElementId": "ImageListComponent",
    "ModelName": "Image"
  }, {
    "Title": "Karte",
    "Icon": "map_icon.png",
    "ElementId": "ImageMapComponent",
    "ModelName": "Image"
  }],
  ...
}
```

Die Felder `Title` und `Icon` legen den Titel und das Symbol für die Hauptnavigation fest. Die *Component* wird über `ElementId` und das *Model* über `ModelName` referenziert. In Abb. 4.15 ist das Resultat des oben angegebenen Quellcodes des Einstiegspunktes dargestellt. Für die Abb. wurden zusätzlich zwei *Components* und ein *Model* definiert, um die Anwendung starten zu können.

Components

Components werden im gleichnamigen Bereich der JSON-Struktur abgelegt. Die beiden wichtigsten Felder, die eine *Component* definieren sind `Type` und `Events`. Ersteres legt fest, um welche Art von *Component* es sich dabei handeln soll. Näheres dazu kann in Abschnitt *Components*, 4.1.5 nachgelesen werden.

Das Feld `Events` erwartet ein JSON-Objekt, welches die vorhandenen *Trigger* mit *Event*-Referenzen verbindet. Es folgt nun ein Beispiel für eine Listen-*Component* namens „MyList“, welche beim ersten Laden das „Load“-*Event* ausführt und Aktionen für das Betätigen des „Add“-Buttons (`PushNewForm`-*Event*) und das Auswählen einer Zelle (`PushDetailView`-*Event*) festlegt:

```
{
  ...
  "Components": {
    "MyList": {
      "Type": "TableComponent",
      "Title": "Bilder",
      "Events": {
        "Init": "Load",
        "Add": "PushNewForm",
        "ItemClicked": "PushDetailView"
      }
```

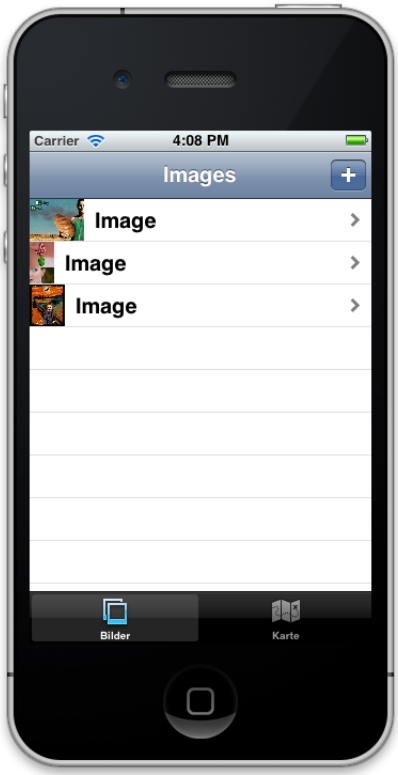

Abbildung 4.15: Anwendung mit zwei Start-*Components*.

```
12      },
13      "AnotherComponent": {
14        ...
15      }
16    }
17    ...
18  }
```

Events

Events benötigen in der einfachsten Konfiguration nur ein Feld namens `Type`. Dieses legt fest, welche Operation das *Event* ausführen soll. Je nach `Type` sind aber weitere Felder notwendig. Bei den meisten *Events* muss zusätzlich angegeben werden, welcher Baustein als nächstes aufgerufen werden soll. Dies geschieht mit dem Feld `ElementId`. Im folgenden Code wird ein *Event* definiert, welches alle *Models* lädt, die im `Root` des jeweiligen Abschnitts der Anwendung angegeben wurde und diese an einen Baustein namens „MyComponent“ weiterreicht:

```
1 {
```

```
  ...
  "Events": {
    "Load": {
      "Type": "LoadModels",
      "ElementId": "MyComponent"
    }
  },
  ...
}
```

Models

Ein *Model* setzt sich aus `Fields` und `Views` zusammen. Das `Fields`-Feld erwartet ein Array von JSON-Objekten, die ein Datenfeld des *Models* beschreiben. Dieses benötigt folgende drei Felder: `Name`, `Type` und `Options`. Ersteres dient zur Identifizierung des Feldes. `Type` gibt an, um welchen Typ es sich bei dem Feld handelt (siehe Tabelle 4.3 im Abschnitt *Models*, 4.1.7). Zur Konfiguration und für Validierungskriterien dient das Feld `Options`.

Das `Views`-Feld ist etwas komplexer aufgebaut: Es nimmt JSON-Objekt entgegen, welches festlegt, in welchem Kontext welche *View* mit welchen *View-Field-Mappings* zu rendern ist. Zur Erinnerung: Je nach *Component* wird dem jeweiligen *Model* mitgeteilt, in welchem Kontext es sich zu rendern hat. So teilt die *Map Component* dem *Model* mit, dass es sich im Kontext der Karte befindet. Das *Model* greift daraufhin auf das eigene `Views`-Feld zurück und lädt die *View* die dem „Map“-Kontext zugewiesen wurde und befüllt deren Felder mit den Daten des *Models*, die über die *View-Field-Mappings* zugewiesen wurden. Folgendes Beispiel zeigt eine Definition eines *Models* mit drei Feldern und dem entsprechenden *View-Field-Mapping*:

```
{
  "Models": {
    "Person" : {
      "Fields": [{
        "Name": "Firstname",
        "Type": "Text",
        "Options": {
          "InputType": "TextField",
          "InputFormat": "Default",
          "Required": true
        }
      }, {
        "Name": "Lastname",
        "Type": "Text",
        "Options": {
          "InputType": "TextField",
          "InputFormat": "Default",
          "Required": true
        }
      }, {
        "Name": "Image",
```

```
22          "Type": "Image",
23          "Options": {
24            "UseCamera": true,
25            "UseGallery": true,
26            "Required": false
27          }
28        }],
29        "Views": {
30          "Detail": {
31            "ViewName": "PersonView",
32            "Mappings": {
33              "FirstnameLabel": "Firstname",
34              "LastnameLabel": "Lastname",
35              "Image": "Image"
36            }
37          }
38        }
39      }
40    }
41 }
```

Views

Views können komplett frei mit Hilfe der Beschreibungssprache gestaltet werden. Dazu stehen einige vordefinierte UI-Elemente zur Verfügung, mit denen die Ansichten gestaltet werden können. Das Aussehen kann mit einer CSS ähnlichen Sprache angepasst werden.

Eine *View* wird über eine Liste von Feldern definiert. Diese werden absolut auf einer Containerfläche ausgerichtet. Die Größe der Containerfläche wird dann aufgrund der darin befindlichen UI-Elemente berechnet. Diese Felder werden als Array dem Property `Fields` des *Views* Objekts übergeben. Jedes Feld muss über einen Typ `Type` verfügen, der angibt, um welches UI-Element es sich handelt. Danach folgen die CSS-ähnlichen Definitionen, welche das Aussehen anpassen.

Neben dem `Fields` Feld gibt es noch das `Order` Property. Dieses bestimmt die Anordnung der Felder, falls diese in einer Liste nacheinander gerendert werden sollen. Falls zwischen den Listenelementen Überschriften eingefügt werden sollen, kann dieses mit spitzen Klammern erreicht werden.

Das Feld `Type` legt fest, um welchen Ansichtstypen es sich handelt. Zur Zeit gibt es nur frei definierbare Ansichten (`Type` wird nicht gesetzt) und Tabellenansichten (`Type` wird auf „Table“ gesetzt). Letztere stellen die Felder untereinander in einer Liste dar und ignorieren somit das `Position`-Property der Style-Informationen.

Der nachfolgende Code zeigt, wie eine *View* definiert werden kann:

```
1 {
2   "Views": {
3     "PersonView": {
4       "Type": "Table",
```

```
5       "Fields": {
6         "Image": {
7           "Type": "Image",
8           "Size": {"Width": 100, "Height": 70},
9           "Position": {"X":0, "Y": 0}
10        },
11        "FirstnameLabel": {
12          "Type": "Label",
13          "Size": {"Width":190, "Height":20},
14          "Position": {"X":110, "Y":0},
15          "BackgroundColor": "#cccccc",
16          "Lines": 3
17        },
18        "LastnameLabel": {
19          "Type": "Label",
20          "Size": {"Width":190, "Height":20},
21          "Position": {"X":110, "Y":20},
22          "Lines": 3
23        }
24      },
25      "Order": ["<Image>", "Image", "<Details>", "FirstnameLabel", "
        LastnameLabel"]
26     }
27   }
28 }
```

In Abb. 4.16 ist die Ausgabe der oben definierten View dargestellt.

4.2.3 Ansätze für visuelle Programmierung

Im Zuge der Entwicklung des komponentenbasierenden Systems wurde auch mit der Konzeptionierung eines visuellen Editors experimentiert. Dieser sollte dazu dienen, die auf JSON basierende Beschreibung der Anwendung automatisiert aufgrund einer visuellen Repräsentation der App zu erstellen. Der Vorteil eines solchen Editors ist, dass Benutzer ohne Programmierkenntnisse Smartphoneanwendungen ohne großen Lernaufwand erstellen könnten.

Um den Editor möglichst plattformunabhängig zu gestalten, wurde dieser mit HTML5 und JavaScript umgesetzt. Die Basisbausteine können per „Drag and Drop“ frei auf der Arbeitsfläche des Editors angeordnet und miteinander verknüpft werden. Durch einen Klick auf „Save“ erstellt der Editor automatisch die Beschreibung für die Anwendung, welche vom komponentenbasierten System geladen und interpretiert werden kann. In Abb. 4.17 ist ein Screenshot von dem Editor dargstellt. Es befinden sich eine *Component*, ein *Event* und ein *Model* auf der Arbeitsfläche, welche miteinander zu einer lauffähigen Anwendung verknüpft wurden. Als Einstiegspunkt dienen die schwarz-grauen Kästchen am linken Rand, welche die Referenzen auf die gewünschte *Component* halten.

Da die Implementierung eines solchen Editors viel Zeit in Anspruch nimmt, konnte dieser im Zuge dieser Arbeit nicht zur Gänze vervollständigt

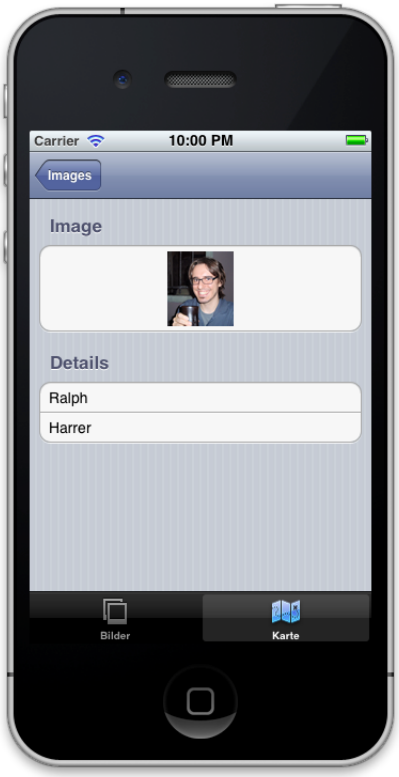

Abbildung 4.16: Darstellung eines *Models* mit Hilfe einer *Table-View*.

werden. Das Grundprinzip geht jedoch klar hervor und einfache Anwendungen lassen sich damit bereits gestalten.

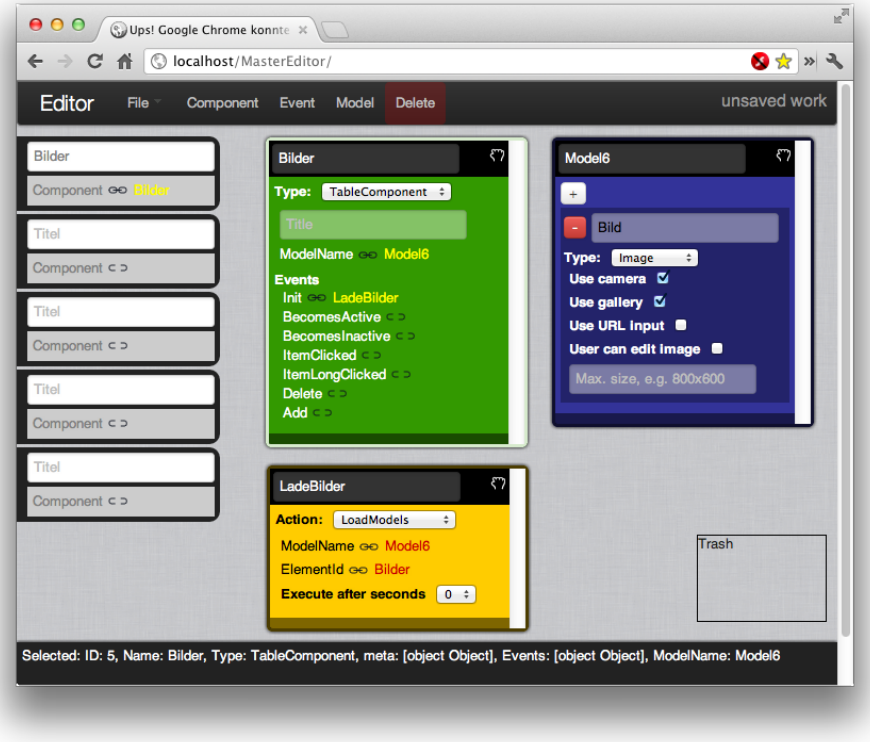

Abbildung 4.17: Der visuelle Editor für die Anwendungsbeschreibungssprache.

Kapitel 5

Umsetzung

Im Zuge dieser Arbeit wurde ein Prototyp zur Erstellung von Smartphoneanwendungen mit Hilfe einer einfachen Beschreibungssprache entwickelt. Wie das System aufgebaut wurde und wie es im Detail funktioniert wird im folgenden Kapitel genauer erläutert.

5.1 Allgemein

Das komponentenbasierte System wurde für Apples iPhone entwickelt, weswegen Objective C (mehr dazu unter [9]) als Programmiersprache zum Einsatz kam. Diese ist für dieses Anwendungsgebiet sehr gut geeignet, da sie trotz ihres objektorientieren Ansatzes sehr flexible Codekonstrukte zulässt, die teilweise einer Scriptsprache ähneln.

In Apples Cocoa-Framework, welches bei der iOS Entwicklung zum Einsatz kommt, sind alle Klassen mit dem Präfix „NS“ (was für „NeXTStep“ steht) versehen. Es wird zwar nicht vorgeschrieben, aber für iOS Entwickler gehört es zum guten Ton eigene Klassen ebenfalls mit einem Präfix zu versehen. Aus diesem Grund wurden allen Klassennamen die Initialen des Autors „RH“ vorangestellt. Einen ausführlicheren Einblick in das Thema Cocoa-Framework zur Entwicklung von Mac OS X und iPhone Software geben Hillegass in [6] bzw. Sadun in [11].

Nähere Informationen zum Thema Umsetzung und Konzeptionierung von komponentanbasierter Systeme sind in [24] zu finden.

5.1.1 Programmablauf

Der Ablauf des Systems wurde bewusst sehr simpel gestaltet, um den Entwicklungsaufwand in Grenzen zu halten. Der erste Schritt der Anwendung ist das Laden der auf JSON basierenden Beschreibung (siehe Abschnitt 4.2). Zuerst wird überprüft, ob es sich bei der Beschreibung um valides JSON handelt. Danach wird begonnen die wichtigesten Teile für den Start auszulesen.

Das sind einerseits die `Root`-Objekte mit den dazugehörigen *Components*, welche die erste Navigationsebene der Anwendung bilden, und deren *Events*, welche beim Initialisieren (Verknüpfung mit dem `Init-`*Triggernamen*) der jeweiligen *Component* ausgeführt werden sollen. Andererseits gibt das `Root`-Objekt bekannt, welches *Model* im jeweiligen Programmabschnitt verwendet werden soll. Jedes nachfolgende *Event* und jede nachfolgende *Component* erwartet das angegebene *Model* oder eine Liste davon, um damit zu arbeiten oder diese darzustellen. Für letzteres wird die Klasse `RHViewProvider` herangezogen. Diese erstellt die angeforderten *Views* bzw. lädt diese aus dem Cache, falls diese bereits erstellt wurden, und weist ihnen die Werte der geladenen *Models* zu. Die *Components* verwenden diese *Views* für die Darstellung der *Models* im User-Interface.

Als Schnittstelle zur Anwendungsbeschreibung dient die Klasse `RHDataProvider`. Diese hilft beim Auffinden von Bausteinbeschreibungen und liefert sie als `NSDictionary` zurück. Diese Beschreibung wird vom `RHElementBuilder` und von der `RHFactory` verwendet, um die gewünschten Bausteine zu initialisieren und zu konfigurieren.

5.2 Architektur

Der Aufbau des Systems lässt sich gut anhand von Abb. 5.1 beschreiben. Die zentrale Anlaufstelle im Code stellt die Klasse `RHElementBuilder` dar. Diese dient dazu, neue Instanzen von Bausteinen aufgrund deren eindeutigen Namen zu erzeugen. Wird ein Baustein angefordert, wird der Name an den `RHDataProvider` weitergeleitet, der die Anwendungsbeschreibung nach dem gewünschten Objekt durchsucht. Wird diese gefunden, wird sie der `RHFactory` übergeben. Hier wird die Beschreibung des Bausteins in eine konkrete Instanz von `RHComponent`, `RHEvent` oder `RHModel` umgesetzt. Genau an dieser Stelle kann auch eingegriffen werden, um Erweiterungen am System vorzunehmen.

Details zu den Klassen und dem Programmablauf wird in den nächsten Abschnitten genauer beschrieben.

5.2.1 Schnittstelle zur Beschreibungssprache

Die Schnittstelle zur Beschreibungssprache wird von der Klasse `RHDataProvider` zur Verfügung gestellt. Da die Beschreibung zu den jeweiligen Bausteinen nicht auf einmal eingelesen, sondern erst nach Bedarf angefordert wird, benötigt diese Klasse viele unterschiedliche Zugriffsmöglichkeiten, um diese Aufgabe effizient zu erledigen.

Die Hauptaufgabe von der Klasse `RHDataProvider` ist das Laden der Beschreibungssprache. Diese kann entweder lokal aus dem Anwendungsbundle der App oder aus dem Internet geladen werden. Danach wird diese im Hauptspeicher der App gehalten, bis diese beendet wird.

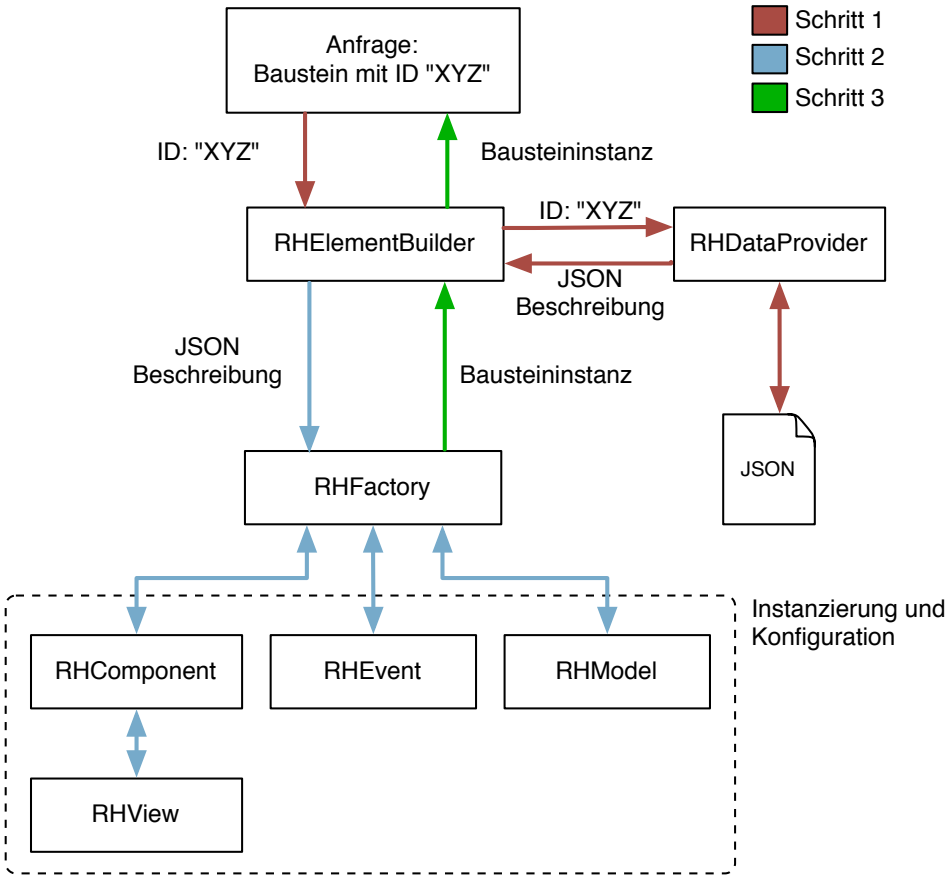

Abbildung 5.1: Architektur des komponentenbasierten Systems.

Die wichtigsten Zugriffsmethoden erlauben das Abrufen der Beschreibung der Bausteine über deren Namen. Zusätzlich gibt es Methoden, die z. B. die *View-Field-Mappings* der *Models* laden oder die Anordung der Felder einer *View* zurückliefern.

5.2.2 RHComponent

Die Klasse *RHComponent* ist eine Subklasse eines `UIViewControllers` und dient zur Anzeige des User-Interface' so wie zur Verarbeitung von Benutzereingaben. Sie sollte als abstrakte Basisklasse verwendet werden, um neue *Components* für das System zu erstellen. In Abb. 5.2 ist die Vererbungshirarchie der bestehenden *Components* dargestellt. Ein wichtiger Sachverhalt ist, das die `RHComponent` Klasse immer eine Referenz auf eine Liste von *Models* oder auf ein spezielles *Model* hält, welche den auszulösenden *Events* indirekt weitergereicht werden.

Damit auf Benutzerinteraktionen reagiert werden kann, muss sich die

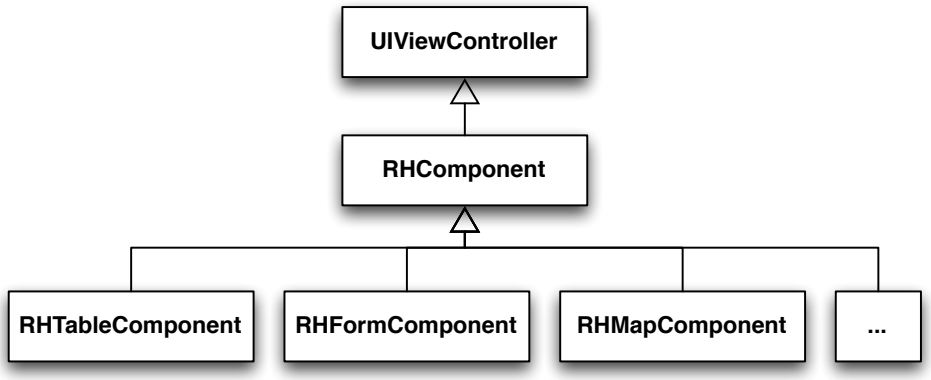

Abbildung 5.2: Vererbungshirarchie der *Components*.

Component darum kümmern, *Events* auszulösen. In der Beschreibungssprache werden *Event*-Namen so genannten *Triggern* zugewiesen. Jede `RHComponent` Subklasse kann beliebig viele dieser *Trigger* zur Verfügung stellen und mit gewissen Benutzerinterationen verknüpfen.

Angenommen ein Entwickler möchte in einer `RHComponent`-Subklasse auf eine Zwick-Geste reagieren, könnte dieser einen Trigger mit dem Namen `UserDidPinch` erstellen. Die Super-Klasse stellt zum Feuern eines Triggers die Methode `performEventForTrigger:(NSString*)trigger;` zur Verfügung, über die das zugeordnete *Event* ausgelöst werden kann. Führt der Benutzer nun die Zwick-Geste im User-Interface aus, kann mit der erwähnten Methode der *Trigger* `UserDidPinch` ausgelöst werden, was zum Start des zugewiesenen *Events* führt. In der Beschreibunggssprache könnte diese Zuordnung so aussehen:

```
1 {
2   "Components:" {
3     "PinchTestComponent": {
4       "Type": "PinchComponent",
5       "Title": "Geste ausführen",
6       "Events": {
7         "UserDidPinch": "MyPinchEvent"
8       }
9     }
10  },
11  "Events": {
12    "MyPinchEvent": {
13      ...
14    }
15  }
16 }
```

Sobald ein *Trigger* ausgelöst wurde, wird über den `RHDataProvider` die angeforderte *Event*-Beschreibung geladen und der `RHFactory` übergeben, um

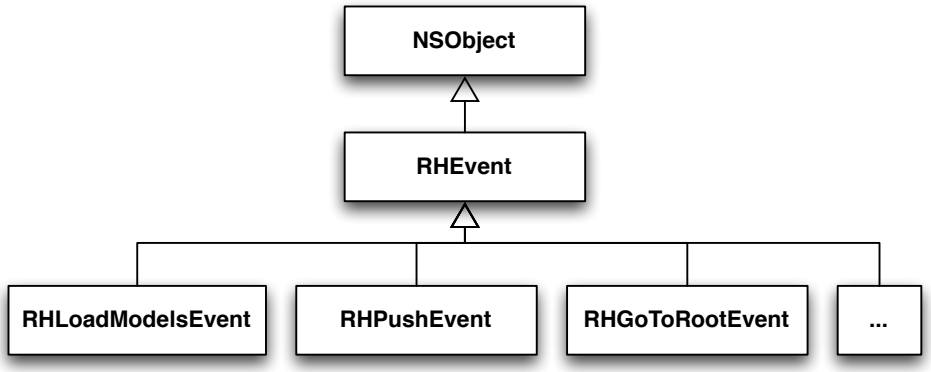

Abbildung 5.3: Vererbungshirarchie der *Events*.

ein neues *Event* zu initialisieren. Dieses wird daraufhin der `RHComponent`-Instanz, welche den *Trigger* ausgelöst hat, übergeben. Gleichzeitig wird dem *Event* die Referenz der *Component* als `sender` mitübergeben. Danach wird die Methode `- (void)performEvent;` des *Events* ausgelöst, um die gewünschte Aktion auszuführen. Dabei greift das *Event* auf sein `sender` Property zu, um Zugriff auf die *Component* und somit auf die referenzierten *Models* bzw. referenzierte *Model* Instanz zu bekommen.

5.2.3 RHEvent

Wie auch `RHComponent` ist die `RHEvent` Klasse ebenfalls abstrakt und dient als Basis für alle *Events* welche im System verwendet werden sollen. Als Superklasse kommt beim `RHEvent` die `NSObject` Klasse vom *FoundationFramework* von *Apple* zum Einsatz (siehe Abb. 5.3).

Die drei wichtigsten Methoden der `RHEvent`-Klasse sind in Tabelle 5.1 beschrieben. Als Entwickler muss lediglich die Methode `-(void)eventImplementation;` überschrieben werden. Darin werden die gewünschten Aktionen an der *Component*, welche im `sender`-Property referenziert wird oder an dem *Model* bzw. an den *Models* ausgeführt. Diese Methode wird automatisch bei der Ausführung des *Events* aufgerufen.

Das *Event* benötigt in der Beschreibungssprache die Referenz auf ein weiteres *Event* oder auf eine *Component*, welche nach der Ausführung instanziert werden soll. Wird auf ein *Event* referenziert, wird dieses ebenfalls sofort ausgeführt. Dadurch wird ermöglicht, mehrere Aktionen hintereinander auszuführen, wie z. B. das Löschen und neu Laden von *Model*-Instanzen.

5.2.4 RHModel und die Datenhaltung

Die `RHModel`-Klasse dient zur Speicherung von Instanzen der in der Beschreibungssprache definierten Datentypen. Um die genaue Arbeitsweise der Da-

Methode	Beschreibung
`- (void)performEvent;`	Startet die Abarbeitung des *Events*
`- (void)eventImplementation;`	Führt die eigentliche Aktion des *Events* aus
`- (void)performFollowingEvent;`	Nach jedem *Event* kann entweder erneut ein *Event* ausgeführt werden, oder eine *Component* auf den internen Navigationsstabel gelegt werden. Ist ersteres der Fall, wird auch dieses *Event* sofort ausgeführt. Dieser Vorgang wird so lange wiederholt, bis die Abarbeitung der *Events* zu einer *Component* führt.

Tabelle 5.1: Die drei wichtigsten Methoden der `RHEvent`-Klasse.

tenhaltung zu beschreiben, muss zuerst ein kurzer Überblick über *Apples CoreData-Framework* gegeben werden, welches in diesem System zum Einsatz kam.

CoreData-Framework

Mit dem *CoreData-Framework* stellt *Apple* ein mächtiges Werkzeug für Datenbanken und Datenbankoperationen zur Verfügung, welches im Hintergrund mit SQLite Datenbanken arbeiten.

Anstatt Operationen direkt an der Datenbank auszuführen, werden alle Aktionen an einen *Kontext* weitergeleitet der die Datenbank repräsentiert. Der Vorteil dieser Methode ist, dass die Daten in der Datenbank so lange unverändert bleiben, bis der Programmierer die Änderungen im *Kontext* durch einen Methodenaufruf mit der Datenbank abgleicht. Dadurch hat man die Möglichkeit, Änderungen an Datensätzen jederzeit zu verwerfen und erst wenn der gewünschte Zustand erreicht ist, endgültig zu speichern.

Detaillierte Informationen zum *CoreData-Framework* können in *Apples Developer Library*[1] gefunden werden.

Umsetzung

Da die Datenhaltung auf *CoreData* basiert, musste Rücksicht auf die Arbeitsweise dieser Datenbankabstraktionschicht genommen werden. Aus diesem

[1] http://developer.apple.com/library/mac/#documentation/cocoa/conceptual/coredata/cdprogrammingguide.html

Grund arbeitet das komponentenbasierende System intern mit zwei unterschiedlichen *Model* Klassen. Das ist einerseits die bereits erwähnte `RHModel` und andererseits die `RHTemporaryModel` Klasse.

`RHModel` repräsentiert ein abgespeichertes *Model* in *CoreData*. Diese werden beim Abruf der Daten aus der Datenbank instanziert und dem System als Array übergeben. Um diese bestehenden Objekte um weitere Funktionalitäten und Schnittstellen zu erweitern, ohne diese selber verändern zu müssen, wurde die Klasse `RHTemporaryModel` eingeführt. Letztere hält intern eine Referenz auf die jeweilige `RHModel`-Instanz, stellt aber zusätzliche Methoden zur Bearbeitung sowie zur Verwaltung der Datenbankobjekte zur Verfügung. Das komponentenbasierte System arbeitet somit intern mit `RHTemporaryModel`-Objekten und wandelt diese vor dem Schreiben in die Datenbank in kompatible `RHModel`-Instanzen um.

Ähnliches betrifft auch die Felder der *Models*. Jedes *Model* hält ein Array mit Feldern vom Typ `RHTemporaryField`, welche für Datenbankoperationen auf `RHField`-Objekte umgewandelt werden. Diese Felder speichern die eigentlichen Daten des *Models*. Diese bestehen aus einem Typ (wie z. B. Text, Nummern, Koordinaten usw.), einem dazu passenden Wert und einem einmaligen Namen.

Die eigentlichen Datenbankoperationen werde von den modellmodifizierenden *Events* (siehe Abschnitt *Typen*, 4.1.6) vorgenommen. Diese nehmen bestehende `RHTemporaryModel`-Instanzen entgegen und führen damit die gewünschte Operation aus.

5.2.5 RHField

Die Klasse `RHField` dient zur Repräsentation der Daten eines *Models*. Darüber hinaus muss jedes Feld eine Subklasse von `RHInputComponent` zur Verfügung stellen, welche von der *FormComponent* über die Methode `- (RHInputComponent*)inputComponent` angefordert werden kann. Wie der Name schon sagt, stellt die Subklasse der *RHInputComponent* ein User-Interface für die Dateneingabe zur Verfügung. So kann ein Feld, das Textwerte speichert, eine Subklasse von *RHInputComponent* zur Verfügung stellen, welche nur ein Textfeld als Eingabemöglichkeit hat. Diese wird von der *FormComponent* angefordert, falls der Benutzer den Wert dieses Feldes ändern möchte.

Eine weitere wichtige öffentliche Schnittstelle der `RHField`-Klasse stellt die Methode `- (void)configureViewForUI:(UIView*)view` dar. Diese wird vom `RHViewProvider` verwendet, um den Wert des Feldes in einer View darzustellen. Um das zu erreichen, kann mit dieser Methode ein beliebiges Objekt vom Typ `UIView` an das Feld übergeben werden. Diese Klasse wird von *Apple* zur Verfügung gestellt und dient zur Darstellung von beliebigen User-Interface-Elementen, wie Text Boxen, Buttons, Labels und vieles mehr. Nun kann der Programmierer selber entscheiden, wie der jeweilige Wert in dem übergebenen UI-Element dargestellt wird. Wird einem Feld, was zur

Speicherung eines Bilder verwendet wird, ein Label übergeben, kann dort z. B. der Name des Bildes eingetragen werden. Sinnvollerweise sollten diesem Feld aber Elemente zur Darstellung von Bildern übergeben werden.

5.2.6 RHView

Die Klasse `RHView` dient zur Darstellung der in der Anwendungsbeschreibung definierten *Views*. Diese Klasse ist eine direkte Subklasse von `UIView`, welche aus *Apples* iOS Framework stammt. Diese lässt sich am besten mit einem `div`-Tag in HTML vergleichen: eine `UIView` Klasse kann beliebig viele weitere Instanzen von `UIView` enthalten, welche immer relativ zum Elternelement ausgerichtet werden können. Außerdem ist nahezu jedes User-Interface-Element in Apples Framework (wie bereits im Abschnitt 5.2.5 erwähnt) eine Subklasse von `UIView`.

Die Generierung der *Views* wird vom `RHViewProvider` übernommen. Dieser stellt Schnittstellen zur Verfügung, um Beschreibungen der `Views` aus der Anwendungsbeschreibung über den Namen und dem aktuellen Kontext zu laden und diese daraufhin zu generieren und zu cachen. Bei der Erstellung der *Views* wird wie folgt vorgegangen: Die Felder die in der Beschreibungssprache definiert wurden, werden der Reihe nach durchlaufen und aufgrund deren Styleinformationen visuell aufbereitet und in einem Container (eine `UIView-Instanz`) platziert. Danach wird die erzeugte *View*, welcher noch keine Daten zugewiesen wurde im Cache des *RHViewProviders* zwischengespeichert. Von nun an wird die *View*, falls sie erneut angefordert wird, aus dem Cache geladen und nicht neu generiert.

Im nächsten Schritt wird die *View* mit den Daten der *Fields* eines *Models* befüllt. Über die *View-Field-Mappings* wird ermittelt, welche *Fields* welchen visuellen Repräsentationen innerhalb der *View* (das können nun Labels, Buttons, Text Boxen usw. sein) zugewiesen werden sollen. Den *Fields* werden diese über die bereits in Abschnitt 5.2.5 vorgestellte Methode `-(void)configureViewForUI:(UIView*)view` übergeben und dieses kann nun selbst entscheiden, wie die Daten in dem übergebenen UI-Element dargestellt werden sollen.

5.2.7 Die RHElementBuilder-Klasse

Das Herzstück des Systems bildet die `RHElementBuilder`-Klasse. Diese stellt elementare Schnittstellen zur Bausteingenerierung zur Verfügung. Die wichtigesten Methoden der Klasse sind in Tabelle 5.2 aufgeführt. Fordert das System eine Baustein-Instanz an, läuft dies immer über die `RHElementBuilder`-Klasse. Diese leitet den Prozess intern weiter und liefert als Resultat eine Instanz des angeforderten Bausteins zurück. Schlägt der Ablauf fehl, wird `nil` zurückgeliefert, was in *Objective C* so viel wie `null` bedeutet.

Methode	Beschreibung
`-(NSArray*) tabBarItems;`	Erstellt die Hauptnavigation über die definierten Einstiegspunkte und instanziert die dazugehörigen `RHComponent` und `RHEvent` Klassen.
`-(RHEvent*) eventById:(NSString*)_id sender:(id)object;`	Nimmt den Namen eines *Events* entgegen und lädt dessen Beschreibung und instanziert es. Optional kann der Sender angegeben werden, welcher das *Event* angefordert hat.
`-(RHComponent*) componentById:(NSString*)_id sender:(id)object;`	Nimmt den Namen einer *Component* entgegen und lädt derem Beschreibung und instanziert es. Optional kann der Sender angegeben werden, welcher die *Component* angefordert hat.
`-(RHTemporaryModel*) temporaryModelById:(NSString*)_id;`	Erstellt ein leeres `RHTemporaryModel` und fügt die Felder hinzu, welche in der Beschreibung angegeben wurden.
`-(RHTemporaryModel*) temporaryModelById:(NSString*)_id andModel:(RHModel*)model;`	Erstellt ein `RHTemporaryModel`, fügt die Felder hinzu, welche in der Beschreibung angegeben wurden und befüllt diese mit den Daten einer bestehenden `RHModel`-Instanz.

Tabelle 5.2: Wichtige Schnittstellen der `RHElementBuilder`-Klasse.

5.2.8 Kommunikation der Bausteine untereinander

Die interne Kommunikation zwischen den Bausteinen stellt einen elementaren Bestandteil des Systems dar, weshalb dieser hier noch einmal beschrieben wird. Wie in Abschitt 4.1.4 bereits erwähnt, werden zwischen den `RHComponent`- und `RHEvent`-Instanzen `RHModels` weitergereicht. Abb. 5.4 stellt diesen Ablauf noch einmal detailliert dar: Jede *Component* und jedes *Event* verfügt über zwei Properties namens `model` und `models`. Ersteres ist

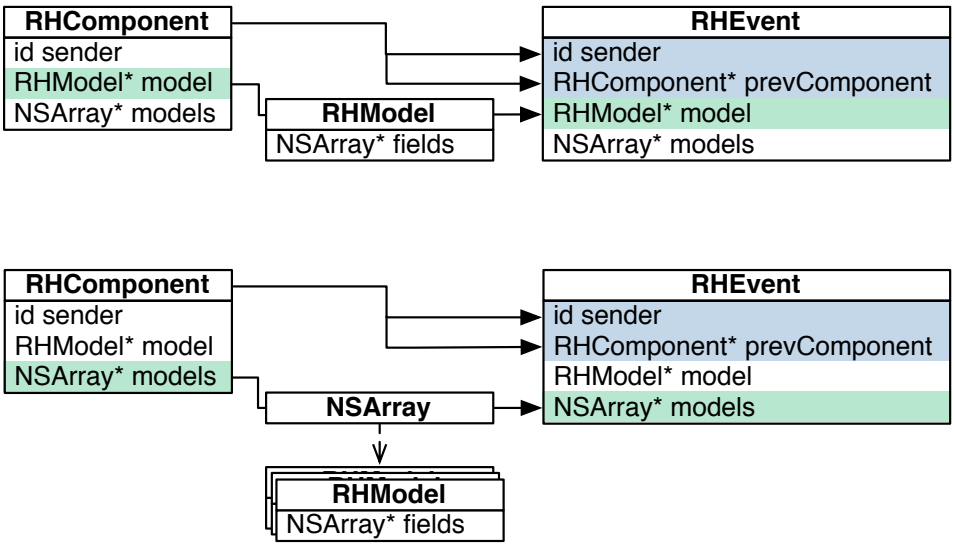

Abbildung 5.4: Übergabe der `RHModel`-Instanzen.

eine direkte Referenz auf ein einzelnes *Model*. Das Property `models` ist eine Referenz auf ein Array, welches eine Liste von *Models* enthält. Ob eine Liste oder ein einzelnes *Model* übergeben wird, hängt vom verwendeten Baustein-Typ ab. Die *Component* vom Typ `DetailComponent` greift auf das Property `model` zu, um die Details eines einzelnen *Models* anzuzeigen. Die *Component* zum Darstellen von Listen verwendet hingegen das Array von *Models*.

Zusätzlich wird immer die Quelle der *Models* als `sender` mitübergeben. Dies benötigen vor allem die *Events*, welche einerseits die Instanz des Senders im Property `sender` abspeichern und andererseits immer die Referenz auf die letzte `RHComponent`-Instanz im Property `prevComponent` halten. Das ist notwendig, um User-Interface-*Events* den Zugriff auf die *Component* und deren *View* zu ermöglichen.

5.3 Erweiterung des Systems

Das komponentenbasierte System stellt schon einige Bausteine zur Verfügung, mit deren Hilfe man einfache Anwendungen erstellen kann. Der Erfolg eines solchen Systems steht oder fällt jedoch mit der Erweiterbarkeit und Flexibilität. Deshalb wurde bei der Entwicklung stets darauf geachtet, Entwicklern die Möglichkeit zu geben, eigene Bausteine ins System integrieren zu können.

Das gesamte System arbeitet intern über eine vorgegebene Namenskonvention über welche Klassen automatisch instanziert werden. Wie dieses für die Erweiterung verwendet werden kann, wird im folgenden Abschnitt ge-

nauer erklärt.

5.3.1 Erstellung eigener *Components*

Um neue *Components* zum System hinzuzufügen, reicht es, eine Subklasse von `RHComponent` zu erstellen und diese im Projektordner abzuspeichern. Dabei spielt die Namensgebung aber eine wichtige Rolle. Möchte man das System z. B. um eine neue *Component* namens „AccordionComponent“ erweitern, muss die Subklasse von `RHComponent` den Namen „RHAccordionComponent“ tragen. Ab sofort können davon Instanzen über die Beschreibungssprache erzeugt werden, indem diese wie folgt angegeben werden:

```
1 {
2   ...
3   "Components": {
4     "MyAccordionComponentInstance": {
5       "Type": "AccordionComponent",
6       "Title": "Accordion Example",
7       "Events": {
8         "Init": "...",
9         "AccordionEvent": "PerformSomethingEvent"
10      }
11    }
12  }
13 }
```

Es wurde in der Beschreibung auch ein neuer *Trigger* mit dem Namen „AccordionEvent“ angelegt. Dieser kann nun im *AccordionComponent* von einem beliebigen Systemevent erzeugt und an das Komponentensystem über die Methode `- (void)performEventForTrigger:(NSString*)trigger` weitergereicht werden. Falls damit ein *Event* verknüpft wurde, wird dieses daraufhin ausgeführt.

Die Arbeitsweise der *Component* kann nun wie bei einem `UIViewController` implementiert und das User-Interface über den *Interface Builder* von Apple erzeugt und gestaltet werden. Auf die übergebenen *Models* kann über die Properties `model` bzw. `models` zugegriffen werden.

Um Zugriff auf die Beschreibung der *Component* in Form eines *NSDictionary* zu bekommen, muss das Property *data* herangezogen werden.

5.3.2 Erstellung eigener *Events*

Neue *Events* müssen zwei Bedingungen erfüllen: Einerseits müssen diese eine Subklasse von `RHEvent` sein und andererseits müssen sie die Methode `- (void)eventImplementation;` überschreiben. Letztere wird vom System aufgerufen, wenn das `RHEvent` ausgelöst wird. Zugriff auf die vorherige `RHComponent`-Subklasse bekommt man über das Feld `prevComponent`, die Beschreibung ist über `data` und die *Models* sind wie bei der *Compenent* über die Properties `model` bzw. `models` erreichbar.

Auch bei den *Events* spielt die Namensgebung eine große Rolle, damit diese dynamisch vom System instanziert werden können. Angenommen man möchte ein neues *Event* vom Typ „ChangeBackgroundColor“ erstellen, so muss die Klasse den Namen „RHChangeBackgroundColorEvent“ tragen. In der Beschreibungssprache kann dieses *Event* nun so erzeugt werden:

```
{
  ...
  "Events": {
    "MyColorChanger": {
      "Type": "ChangeBackgroundColor"
    }
  }
}
```

Die Implementierung des *Events* könnte wie folgt aussehen:

```
- (void)eventImplementation
{
  if (self.prevComponent) {
    self.prevComponent.view.backgroundColor = [UIColor redColor];
  }
}
```

5.3.3 Erstellung eigener *Fields*

Models können ebenfalls ohne großen Aufwand um neue *Fields* erweitert werden. Dazu muss nur eine Subklasse von `RHTemporaryField` erstellt werden. Die neu erstellte Klasse muss ebenfalls wieder einer Namenskonvention folgen, um dynamisch instanziert werden zu können: Angenommen der Typ das neuen *Fields* soll den Namen „Color“ tragen, so muss die Klasse „RHColorField“ genannt werden:

```
{
  "MyModel": {
    "Fields": [{
      "Name": "MyColor",
      "Type": "Color",
      "Options": {...}
    }]
  }
}
```

Die Methoden in Tabelle 5.3 sollten überschrieben werden, damit das neue Feld vollständig ins System integriert werden kann.

5.3.4 Erstellung eigener *View*-Felder

Mit dem Ausdruck „*View*-Felder“ sind die UI-Elemente gemeint, welche in der *View* platziert werden und die Werte der *Fields* darstellen. Um das Repertoire an vorgegebenen UI-Elementen zu erweitern, muss eine Subklasse

Methode	Beschreibung
`-(RHInputComponent*)inputComponent;`	Diese Methode muss eine `RHInputComponent`-Subklasse zurückliefern, welche vom Formular zur Befüllung des Feldes herangezogen wird.
`-(NSString*)formLabelValue;`	Liefert den Wert des Feldes als String zurück, damit dieser im Formular angezeigt werden kann.
`-(BOOL)isValid;`	Wird vom Formular herangezogen, um zu überprüfen, ob die eingetragenen Werte auch gültig sind. Dabei sollte der Wert gegen die angegebenen Validierungsoptionen in der Beschreibungssprache geprüft werden, falls diese angegeben wurden.
`-(BOOL)valueIsEqual:(id)val;`	Sollte den internen mit dem übergebenen Wert auf Gleichheit prüfen und `YES` zurückgeben, falls diese identisch sind.
`-(void)configureViewForUI: (UIView*)view;`	Versucht den Wert des Feldes im übergebenen UI-Element darzustellen.

Tabelle 5.3: Überschreibbare RHField-Methoden.

von *RHUIViewGenerator* erstellt werden. Der Typ des neuen Elements bestimmt auch wieder die Namensgebung der Klasse. Angenommen das neue UI-Element soll vom Typ „Slider“ sein, so muss die Klasse „RHUISliderViewGenerator“ genannt werden.

Die neuen Klassen müssen eine Methode namens `- (UIView*)viewWithStyle:(NSDictionary*)style;` überschreiben. Diese nimmt ein `NSDictionary` mit den in der Beschreibunggssprache angegebenen Styleinformationen entgegen. Diese sollen auf eine neue UIView-Instanz angewendet werden, welche danach zurückgegeben wird.

Kapitel 6

Evaluierung und Vergleich

Das komponentenbasierte System, welches im Zuge dieser Arbeit entwickelt wurde, sollte vor allem die einfache Erstellung von Smartphoneanwendungen für moderne Plattformen ermöglichen. Durch den Einsatz einer simplen Beschreibungssprache bzw. einer grafischen Programmierumgebung bleibt des Erlernen einer komplexen Programmiersprache und plattformspezifischer Frameworks aus. Wie sinnvoll dieser experimentelle Ansatz ist und wie vielfältig sich das System einsätzen lässt, soll die kritische Betrachtung in diesem Kapitel klären.

6.1 Einsatzbereiche

Wie im Kapitel 4, Abschnitt 4.1.2 bereits erwähnt, ist das System zur Verwaltung von benutzerdefinierten Datensätzen vorgesehen. Diese Einschränkung wurde bewusst gewählt, um die Komplexität des Systems in Grenzen zu halten. Jedoch lassen sich über einfache Schnittstellen und über die dynamische Instanzierung von Klassen (was über eine Namenskonvention ermöglicht wird) Erweiterungen zum System hinzufügen. Je nach Anwendungsfall können dadurch fehlende Funktionen hinzugefügt werden, wodurch aber wiederum Programmierkenntnisse notwendig sind.

Durch die Evaluierung und Findung oft verwendeter Komponenten, lassen sich übliche Programmabläufe damit nachstellen. Werden jedoch spezielle Funktionalitäten für eine App benötigt, stößt man mit dem beschriebenen System schnell an die Grenzen des Machbaren. Zum Beispiel lagern moderne Applikationen die Datenhaltung und sogar die Programmlogik oft auf Server aus, wodurch die Anwendung lediglich als Schnittstelle zur „Cloud“ fungiert. Für solche Einsatzzwecke ist das System nicht vorgesehen worden. Jedoch könnte es um neue *Components* und *Events* zur Serverinteraktion erweitert werden.

Ein passendes Einsatzgebiet für das beschriebene komponentenbasierte System stellt das Prototyping dar. Lauffähige Anwendungen lassen sich in

wenigen Minuten erstellen und können somit als Prototypen herangezogen werden, welche den Entwicklungsprozess von „echten“ Anwendungen unterstützen und beschleunigen können. Dadurch ist es z. B. möglich, potenziellen Kunden den Programmablauf mit so genannten „Clickdummies“ näher zu bringen oder diverse Funktionen im Vorhinein auf deren Sinnhaftigkeit und/oder Usability prüfen.

6.2 Flexibilität

Durch die leicht zu erlernende Beschreibungssprache ist es möglich, die Anwendung zu jedem Zeitpunkt ändern zu können, ohne diese neu kompilieren zu müssen. Das bringt diverse Vorteile mit sich: Neue Funktionen lassen sich sofort ausliefern, ohne dass die App erneut in einen der Vertriebswege hochgeladen werden muss. Leider verbietet der Smartphonehersteller *Apple* das dynamische Nachladen von Programmcode, was in diesem Fall durchaus als solches interpretiert werden könnte. Das würde bedeuten, dass Anwendungen, welche mit dem System erstellt wurden, eventuell erst gar nicht in den *AppStore* hochgeladen werden dürfen bzw. die Prüfung durch *Apple* nicht bestehen würden.

Die Bausteine dieses Systems sind im Vergleich zu manch anderen Ansätzen sehr „grob“ gehalten. Diese erfüllen genau einen gewissen Zweck. Mehrere Aktionen in einer *Component* zu vereinen ist theoretisch über *Events* möglich, jedoch wurden diese Funktionalität nicht in die bestehende Version integriert. Je flexibler ein Baukastensystem gestaltet wird, desto höher ist die Komplexität und desto schwieriger ist der Einstieg für Neulinge. Der in dieser Arbeit vorgestellte Ansatz versucht Letzteres zu minimieren, um möglichst einfach Anwendungen erstellen zu können und trotzdem ein gewisses Maß an Flexibilität zu wahren.

Werden spezielle Anforderungen an das System gestellt, kann dieses über die bereitgestellten Schnittstellen um neue *Components* und *Events* erweitert werden. Welche Aufgaben die neuen *Components* erfüllen hängt ganz alleine von der Implementierung ab. Dabei werden keinerlei Grenzen vorgegeben. Daten können jedoch nur über die *Models* zwischen den Bausteinen ausgetauscht werden. Dieser Sachverhalt muss bei der Implementierung berücksichtigt werden.

6.3 Vergleich

Zwei ähnliche Vertreter wurden bereits im Kapitel 3 vorgestellt. Es folgt nun ein direkter Verlgeich zwischen dem in dieser Arbeit vorgestellten Ansatz mit den beiden Vertretern *APPlause* und *Tersus*.

6.3.1 APPlause

APPlause verfolgt einen sehr interessanten Ansatz, der mit Hilfe von domänenspezifischer Programmiersprachen arbeitet. Die Anwendungen werden mit einer sehr generisch gehaltenen domänenspezifischen Programmiersprache entwickelt. Diese wird nach dem Erstellen in direkt ausführbaren Programmcode für die jeweilige Plattform umgewandelt. Dabei werden vordefinierte Programmtemplates herangezogen, um den gewünschten Code dynamisch zu generieren. Dieser muss vom Compiler der jeweiligen Plattform kompiliert werden, um daraus eine lauffähige Anwendung erstellen zu können.

Vergleich

APPlause basiert zwar nicht auf Komponenten, der Ansatz mit der sehr generisch gehaltenen Beschreibungssprache ähnelt dem in dieser Arbeit vorgestellten Konzept. Trotzdem gleicht diese zum Großteil einer Programmiersprache, was die Einstiegshürde für unerfahrene Entwickler erschwert. Andererseits können durch diese Flexibilität komplett frei gestaltete Anwendungen erstellt werden, was mit dem vorgestellten Ansatz nur sehr eingeschränkt möglich ist.

Verwendet man *APPlause* zum Erstellen von Anwendungen, kommt man um die Kompilierung nicht herum, da lediglich der Quellcode dynamisch generiert wird. Erst danach steht eine ausführbare Anwendung zur Verfügung, welche ohne Bedenken in den *AppStore* eingereicht werden kann, falls diese sich an die Richtlinien[1] von *Apple* hält. Nachträgliche Änderungen können nicht direkt eingepflegt werden, sondern müssen wiederum „ausprogrammiert“ und kompiliert werden.

6.3.2 Tersus

Tersus bietet eine eigenständige Entwicklungsumgebung an, mit der *Native-Webapps* erstellt werden können (mehr über *Native-Webapps* in Abschnitt 3.1.2). Die Gemeinsamkeit mit dem hier vorgestellten komponentenbasierenden System, ist die Möglichkeit, Anwendungen in einem grafischen Editor über die Kombination von Komponenten zu erstellen.

Vergleich

Wie bereits erwähnt, arbeitet *Tersus* mit einer grafischen Programmiersprache, mit deren Hilfe plattformunabhängige Anwendungen erstellt werden können. Diese Anwendungen, sind aber im Gegensatz zu dem in dieser Arbeit vorgestellten System, nicht nativ. Es werden lediglich Webanwendungen generiert, welche in einer nativen App in einem Browser angezeigt werden.

[1] https://developer.apple.com/appstore/resources/approval/guidelines.html

Durch CSS Anpassungen und JavaScript werden das Aussehen und das Verhalten von einer Smartphone-Anwendung nachgebildet. Das führt mitunter zu einer schlechteren Performance und der Zugriff auf die Hardware unterliegt ebenfalls diverser Einschränkungen.

Die Komponenten mit denen *Tersus* arbeitet sind höchst flexibel. So können beliebige Bausteine miteinander kombiniert werden, um komplexe Arbeitsabläufe abzubilden. Der große Vorteil ist, dass Komponenten innerhalb von anderen Komponenten platziert werden können, um deren Arbeitsweise exakt auf die Wünsche des Entwicklers anzupassen. Um das zu bewerkstelligen, lässt sich die Arbeitsfläche von der Tersus eigenen Entwicklungsumgebung beliebig vergrößern, um das Innenleben von sich auf der Arbeitsfläche befindlichen Bausteinen zu verändern. Die Kommunikation läuft über Ein- und Ausgabebausteine ab, welche innerhalb der bestehenden Komponenten plaziert werden und die Daten weiterreichen. Durch dieses Konzept lassen sich zum Beispiel komplexe Datenbankabfragen gestalten, indem innerhalb einer Datenbankkomponente Datenbankabfragen hinzugefügt werden. Das Resultat wird über eine Output-Komponente an beliebige Bausteine weitergereicht.

Tersus ist ein gutes jedoch sehr komplexes Werkzeug zum Erstellen von Webbasierten-Anwendungen. Der Einstieg gestaltet sich deshalb als nicht sehr einfach und benötigt einiges an Einarbeitungszeit.

6.3.3 App Inventor

Ein weiterer sehr ähnlicher Vertreter der Gattung der komponentenbasierten Systeme zum Erstellen von nativen Anwendungen für Android-Smartphones stellt Googles *App Inventor* dar. Dieser wurde in der Arbeit noch nicht erwähnt, soll aber für den Vergleich des in dieser Schrift behandelten Ansatzes dienen.

Ansatz

Der *App Inventor*[2] ist eine auf Webtechnologien und Java basierende grafische Entwicklungsumgebung zum Entwickeln von Android Apps. Diese können in den beiden Editoren aus bestehenden Komponenten visuell zusammengebaut und direkt auf einem angeschlossenes Android Smartphone oder im Emulator getestet werden. Dabei werden Änderungen, die in den Editoren vorgenommen werden in Echtzeit auf das Testgerät bzw. auf den Emulator übertragen und erfordert keine Kompilierung. In [7, Seite 269] beschreibt die Autorin Googles *App Inventor* mit folgenden Worten:

> Googles App Inventor allows non-programmers to create genuine Android apps by using a visual block-based programming inter-

[2] http://www.appinventor.mit.edu/

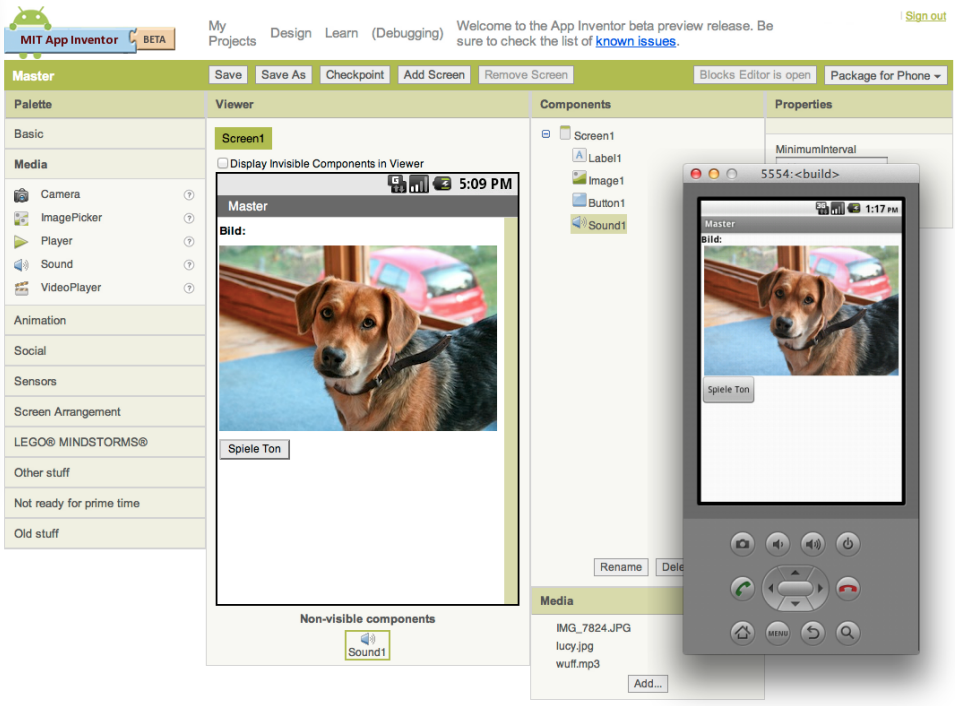

Abbildung 6.1: Der *Designer* des *App Inventors* und die Ansicht im Android Emulator.

> face based on MIT's OpenBlocks Java library. It's similar to the children's programming language Scratch. [...]
>
> While App Inventor seems to do for Android development what WYSIWYG (what you see is what you get) HTML editors did for web design, it's not going to end traditional app development, and it's still a good idea to understand the foundations of Android development, just as most web designers understand HTML basics.

Der Vergleich mit einem WYSIWYG-Editor ist sehr treffend, da das Erscheinungsbild der Anwendung in einer Weboberfläche namens *Designer* frei gestaltet und sofort am Testgerät oder im Emulator betrachtet werden kann, wie in Abb. 6.1 dargestellt. Das User Interface kann mit üblichen Elementen, wie Schaltflächen oder Textboxen, aber auch unsichtbaren Bausteinen, wie Datenkbanken, Social-Media- oder Sensorfunkionen ausgestattet werden. Letztere bleiben dem Benutzer verborgen, verfügen aber über teilweise sehr mächtige Funktionen. So kann z. B. der Twitter-Baustein einen Benutzer authentifizieren, dessen Timeline abrufen oder neue Kurznachrichten versenden, ohne dass der Entwickler eine Zeile Code verfassen muss.

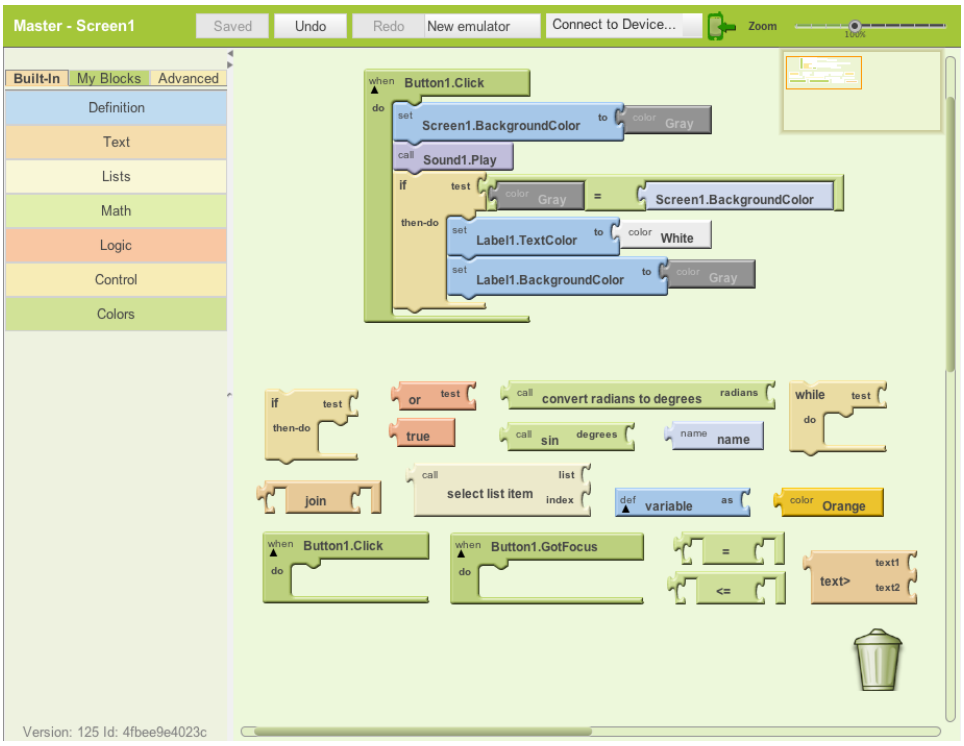

Abbildung 6.2: Übersicht und Verschachtelung der Bausteine im Blocks Editor des *App Inventors*.

Die Verknüpfung mit der Programmlogik findet im *Blocks Editor* statt. Dieser ist eine Java-Applikation, welche direkt mit dem angeschlossenen Testgerät oder dem Emulator kommuniziert. Das System stellt eine große Vielfalt an unterschiedlichsten Blöcken zur Verfügung, die logischen Operatoren, Methodenaufrufe, Zuweisungen oder Zugriffe auf diverse Objekteigenschaften repräsentieren (siehe Abb. 6.2). Durch diese feine Granularität können durchaus komplexe Programmabläufe abgebildet werden. Die Grenzen des Systems werden auf einem höheren Anwendungslevel erreicht: Das System stellt viele Blöcke für die Interaktion mit dem Gerät an sich zur Verfügung. Fehlt jedoch eine gewünschte Funktionalität, besteht zur Zeit keine Möglichkeit, diese in das bestehende System zu integrieren.

Die fertigen Anwendungen können per Klick in eine Android Anwendungsdatei (APK Datei) umgewandelt und auf das Testgerät übertragen oder in den *Google Play Store* gestellt werden. Der Vorteil des *App Inventors* liegt definitiv in der Einfachheit, mit der die Anwendungsentwicklung Einsteigern nähergebracht wird. Um diesen verwenden zu können, werden ein Google Konto und die Installation[3] einer *App Inventor* Software, welche

[3] http://beta.appinventor.mit.edu/learn/setup/index.html

auf der offiziellen Website heruntergeladen werden kann, vorausgesetzt.

Vergleich

Google hat mit dem *App Inventor* eine gute Kombination aus Komplexität und Flexibilität für die App Entwicklung gefunden. Im Gegensatz zu dem in dieser Arbeit behandelten Komponentensystem, verwendet der *App Inventor* feinere Bausteine, die sich vielfältiger einsetzen lassen. Jedoch erfordert die Kommunikation zwischen den Bausteinen geringe Programmierkenntnisse, um deren Kompatibilität untereinander nachvollziehen zu können. Fehlen diese, artet die Anwendungserstellung in „Trial-And-Error“ aus, da erst bei falschen Kombinationen von Bausteinen Hinweise zur Verwendung ausgegeben werden.

Die Speicherung der Daten erfolgt im *App Inventor* nicht automatisch, was aber kein Nachteil ist. Dem Programmierer bleibt es selber überlassen, ob Daten in eine Datenbank (welche sich lokal oder auf einem entfernten Server befinden kann) geschrieben werden sollen oder nicht. Damit auch dieses Unterfangen möglichst einfach vonstatten geht, beschränkt sich *App Inventor* auf einen simplen Key-Value Speicher, der jedoch keine komplexen Datenbankanfragen zulässt.

Die Gestaltung des User Interfaces geht hingegen bei *App Inventor* wesentlich einfacher von der Hand, dank des WYSIWYG-Prinzips. Jede Änderung an der Oberfläche wird dem Entwickler sofort am Testgerät bzw. Emulator angezeigt. Der in dieser Arbeit vorgestellte Ansatz erfordert jedoch Erfahrung im Bereich der User Interface Gestaltung, da dieses in der JSON-Beschreibungssprache mit einer CSS ähnlichen Syntax geschieht und erst nach einem Neustart der App angezeigt wird. Bestehende *Components* können nur sehr grob den gewünschten Anforderungen angepasst werden, die Datenausgabe ist hingegen wieder frei konfigurierbar.

Ein wichtiger Punkt ist die Erweiterbarkeit des Systems. Bei der Entwicklung des hier vorgestellten Systems wurde großer Wert auf diese gelegt (siehe Abschnitt 5.3). Fehlende Funktionen sollen von Entwicklern möglichst einfach ins System intergriert werden können. Um Selbiges bei *App Inventor* zu erreichen, muss der Weg über das Internet genommen werden. Die gewünschte Logik wird serverseitig mit beliebiger Technologie implementiert, auf die über die *Web*- oder *TinyWebDB*-Komponente (letztere setzt eine API[4] voraus) über HTTP-Requests zugegriffen wird. Dieser Ansatz setzt eine permanente Internetverbindung voraus damit die App verlässlich arbeiten kann. In beiden Fällen sind für die Erweiterung vom System Programmierkenntnisse notwendig.

[4] http://appinventorapi.com/using-tinywebdb-to-talk-to-an-api/

Eigenschaft	APPlause	Tersus	App Inventor	Eigener Ansatz
Flexibilität	Hoch	Hoch	Hoch	Mittel
Komplexität	Hoch	Hoch	Mittel	Niedrig
Grafischer Editor	–	✓	✓	Liegt als Prototyp vor
Eigene IDE	Eclipse-Plugin	✓	✓	nicht notwendig
App-Typ	Nativ	Native Web-App	Nativ	Nativ

Tabelle 6.1: Übersicht über die Resultate der Vergleiche.

Zusammenfassung

Die Ergebnisse der Vergleiche mit den vorgestellten Ansätzen wird in Tabelle 6.1 noch einmal zusammengefasst.

6.3.4 Verbesserung

Der in dieser Arbeit beschiebene Prototyp zur Erstellung von komponentenbasierten Smartphoneanwendungen lässt sich in vielen Bereichen noch verbessern, um das System noch vielfältiger und flexibler zu machen. Im folgenden Abschnitt werden potentielle Punkte zur Verbesserung des Ansatzes aufgeführt.

Datenbankabfragen

Der vorgestellte Ansatz verfügt über eine tief ins System integrierte Datenbank (siehe 5.2.4) die sich um die persistente Datenhaltung kümmert. Diese kann beliebige vom Benutzer gestaltete *Models* abspeichern und abrufen. Letzteres unterliegt jedoch noch einer großen Einschränkung. Über die Beschreibungssprache ist es bis jetzt nicht möglich, spezielle Datenbankabfragen zu erstellen. Die eingetragenen Modelle werden zur Zeit noch anhand der *Models*-Namen sortiert und an den Empfänger weitergereicht.

Anzudenken wäre, das Datenbankabfragen in der Beschreibungssprache definiert und mit Datenbank-*Events* (siehe Abschnitt 4.1.6) verknüpft werden können, um dadurch *Models* gezielter abrufbar zu machen.

Erhöhen der Flexibilität

Das vorgestellte System lässt sich am besten mit einem Baukastensystem vergleichen: Jeder Baustein hat vordefinierte Anschlüsse, über die Daten an verbundene Bausteinelemente weitergereicht oder empfangen werden können. Dieser Ansatz ist vor allem für Einsteiger gut geeignet, da zueinander inkompatible Verbindungen untereinander (im Idealfall) nicht möglich sind und somit der reibungslose Ablauf gewährleistet wird. Für manche Anwendungszwecke ist jedoch eine gewisse Flexibilität von Vorteil. Mit dem angesprochenen System ist es zum Beispiel nicht möglich mit zwei Bausteinen auf ein und dasselbe Datum zuzugreifen. Die einzige Möglichkeit dies zu erreichen, ist, die Bausteine in Serie anzuordnen (falls keiner davon die empfangenen Daten verändert), oder den selben Arbeitsablauf zu wiederholen.

Eine Lösung, um die Kommunikation zwischen den Bausteinen zu "lockern", wäre der Einsatz von Systemevents. In der Spieleentwicklung wird dieses System verwendet, um eine lose Kopplung der Entitäten untereinander zu erreichen und um dadurch die Flexibilität zu erhöhen. In [10, Seite 224] wird „Loose Coupling“ wie folgt erläutert:

> Coupling is a measure of how much two classes know about and depend on each other. You should always strive to have very loose coupling between classes, which means they're very modular and easy to change, and modifications to one don't affect the other.

Das Ziel dieses Ansatzes ist, die Abhängigkeit von Klassen (oder später von Bausteinen) auf ein Minimum zu reduzieren. Daraus resultiert eine flexiblere und weniger fehleranfällige Architektur. Ein bekannter Vertreter, der stark auf diesem Ansatz beruht ist *Unity*[5]. Dabei handelt es sich um eine Entwicklungsumgebung für 3D-Spiele, welches hauptsächlich auf Komponenten basiert. Diese können frei miteinander kombiniert werden, um damit neue Spielelemente (wie Spieler, Kameras usw.) zu kreieren.

Würde man diesen Ansatz auf das in dieser Arbeit vorgestellte komponentenbasierte System anwenden, würde das zur Folge haben, dass die Bausteine weniger voneinander abhängig wären und trotzdem frei miteinander kommunizieren könnten. Dabei würden die Daten nicht mehr über die vorgegebenen Schnittstellen getauscht, sondern über Systemevents im System freigegeben und an alle Bausteine, die sich für einen gewissen Eventtyp registriert haben, weitergereicht werden. Dieser Ansatz hätte den Vorteil, dass Daten gleichzeitig an mehreren Stellen verarbeitet werden können. Außerdem ließe sich die Granularität der Komponenten verringern, wodurch es theoretisch möglich wäre, eigene Bausteine, den eigenen Wünschen nach gestalten zu können. Andererseits muss einiges an Aufwand betrieben werden, um die Abarbeitung der Bausteine zu koordinieren, um Konflikte untereinander zu vermeiden und den Programmfluss zu koordinieren. Dies würde sich

[5] http://unity3d.com/

ebenfalls auf die Anwendung des Systems für den Entwickler auswirken, da auch hier mit einer komplexeren Bedienung so wie mit einer umfangreicheren Beschreibungssprache und somit einer größeren Einstiegshürde gerechnet werden müsste.

Visueller Editor

Wie in Abschnitt 4.2.3 bereits erwähnt, wurde im Zuge dieser Arbeit ein Prototyp eines webbasierenden visuellen Editors für die Erstellung von Anwendungen entwickelt. Dieser ermöglicht es Bausteine auf der Arbeitsfläche zu arrangieren und miteinander zu verknüpfen. Dieses Konzept ist zwar einfach zu verstehen, es stößt jedoch schnell an seine Grenzen, wenn sich viele Bausteine auf der Arbeitsfläche befinden, was bei großeren Programmen durchaus der Fall sein kann. Die Übersicht ginge sehr schnell verloren wodurch die Fehleranfälligkeit steigen würde. Besser wäre hier ein anderer Ansatz, der bei steigender Komplexität trotzdem einen übersichtlichen Arbeitsbereich ermöglicht. Anzudenken wäre, voneinander unabhängige Programmabschnitte in eigene Arbeitsbereiche auszulagern und diese über Reiter einblendbar zu machen. Das wäre nur eine von den vielen Verbesserungsmöglichkeiten des Editors, auf welche hier nicht mehr weiter eingegangen wird, da dies den Rahmen dieser Arbeit sprengen würden. Der Editor stellt trotzdem einen sehr wichtigen Bestandteil des Systems dar, da es sich dabei um die direkte Schnittstelle zum Anwender handelt. Diese sollte möglichst übersichtlich, leicht zu bedienen und optisch ansprechend sein, damit dieser für Anfänger einen schnellen Einstieg ermöglicht.

Kapitel 7

Schlussbemerkungen

Die mobilen Betriebssysteme *Android* und *iOS* haben gezeigt, wie wichtig eine Entwicklergemeinschaft für den Erfolg der Plattformen sein kann. Letztere kann sich nur dann bilden, wenn euch die richtigen Tools für die Entwicklung zur Verfügung gestellt werden. Mittlerweile gibt es neben den offiziellen SDKs eine große Anzahl alternativer Ansätze und jeder bringt seine eigenen Vor- und Nachteile mit sich. Die Entwicklung des in dieser Arbeit vorgestellten komponentenbasierten Systems war mit sehr viel Recherche im Bereich der bestehenden SDKs verbunden und veranschaulichte, mit welchen interessanten Methoden versucht wird, Anwendungen schnell und einfach umzusetzen.

Aus eigener Erfahrung zeigt sich, dass der native Weg doch den vorgestellten Ansätzen vorzuziehen ist. Die Zeit- und Kostenersparnis einer „Ein Code für alle Plattformen"-Lösung wirken Anfangs sehr verlockend, bei größeren Projekten, benötigt man oft mehr Zeit um Workarounds für diverse Limitierungen zu finden und schlussendlich werden dadurch die anfangs erwähnten Vorteile wieder wett gemacht. Bei kleineren Projekten kann sich der Einsatz einer solchen Lösung jedoch durchaus lohnen. Vor dem Projektstart sollte trotzdem der Einsatz eines solchen Systems wohl überlegt und die Risiken abgewogen werden.

7.1 Ansatz

Der in dieser Arbeit vorgestellte Ansatz basiert auf Komponenten, die sich auf mobilen Betriebssystemen finden lassen. Diese wurden durch Analyse von bestehenden *iOS*-Anwendungen identifiziert und isoliert als Baustein abgebildet. Die für die Beschreibungssprache wurde bewusst auf JSON zurückgegriffen, da sie leicht verständlich ist und die Definition von Arrays und Objekten erlaubt, welche in *iOS* direkt verwendet werden können. Durch die Kombination dieser beiden Bestandteile, dem Baukastensystem und der Beschreibungssprache, ließ sich ein System konstruieren, welches ohne Neu-

kompilierung Anwendungen erstellen kann.

Die Einsatzbereiche liegen vor allem im Prototyping, um die Entwicklung von „echten“ Anwendungen zu unterstützen. Außerdem ermöglicht das System den schnellen Einstieg für Anfänger in die App Entwicklung.

Ein Prototyp eines browserbasierten visuellen Editors wurde ebenfalls entwickelt, der die Erstellung der Beschreibungssprache automatisch im Hintergrund übernimmt. Aus Zeitgründen konnte dieser leider nicht fertiggestellt werden. Trotzdem gewährt er einen guten Einblick, wie ein solcher Editor umgesetzt werden könnte.

7.2 Ausblick

Der mobile Markt wurde durch *iOS* und *Android* revolutioniert und führte das Konzept der Apps ein. Mitbewerber wir *Microsoft* versuchen bereits an das Erfolgskonzept mit eigenen Systemen anzuknüpfen. Leider hat *Microsoft* mit *Windows Phone 7* ein Nischendasein geführt, was der Softwaregigant aber mit der Einführung von *Windows 8* ändern möchte. Es wird spannend, ob sich das Desktop-Tablet-Hybridkonzept von der neuesten *Windows*-Generation durchsetzen wird. Durch den verspäteten Einstieg in das neue Mobile Business hat *Microsoft* jedoch viel Zeit gewonnen um aus den Fehlern der Konkurenz zu lernen und stellte in diversen Vorschau-Versionen von *Windows 8* einige Interessante Konzepte vor, die das neue System von der Konkurrenz abheben soll.

Selbst im Web zeigt sich ein neuer Trend was die mobile Gestaltung von Websites ansgeht. Der Ansatz des „Mobile First“[1] schlägt vor, sich zuerst auf die mobile Version von Web-Seiten zu konzentrieren, bevor man die Desktop Variante erstellt, um den Fokus der Seite gezielt zu setzen. Dazu entstehen neue Technologien wie RESS[2], die nicht nur Client-seitig die Webauftritte aufbereiten, sondern den Server miteinbeziehen.

Der Trend der mobilen Technologie schreitet fort und ist spannender denn je. Die neuen Bedienkonzepte haben frischen Wind in das IT Leben gebracht, welche sich Schritt für Schritt den Weg in Desktop-Betriebssysteme suchen. So setzt *Microsoft* die Metro UI unter *Windows 8* ein und *Mac OS X* greift immer mehr Konzepte vom moderneren *iOS* auf.

7.3 Fazit

Die Entwicklung des vorgestellten Systems war mit sehr viel spannender Recherche verknüpft. Der Einblick in die bestehenden Konzepte und deren Funktionsweise, brachten interessante Erkenntnisse mit sich, die das vorgestellte komponentenbasierte System oft eine andere Richtung einschlagen

[1] http://www.lukew.com/ff/entry.asp?933
[2] http://www.lukew.com/ff/entry.asp?1392

haben lassen. Der Mut zum Neuanfang und bestehende Ansätze neu zu überdenken oder zu verwerfen spielten bei der Entwicklung eine große Rolle. Es ist wichtig den Fokus zu bewahren und sich nicht in Details zu verlaufen, was bei einem solchen Konzept durchaus passieren kann. Durch die gezielte Einschränkung nur benutzerdefinierte Datensätze zu verwalten, minimierte den Entwicklungsaufwand so, dass das System in der vorgegebenen Zeit umgesetzt werden konnte.

Das vorgegebene Ziel wurde erreicht: Es wurde eine native Anwendung erstellt, die sich selbstständig aufgrund einer Beschreibungssprache generiert. Dafür werden vier Bausteintypen verwendet (*Components*, *Events*, *Models* und *Views*), welche sich zu lauffähigen Programmen kombinieren lassen. Durch die sehr grob gehaltenen Komponenten wurde zwar Flexibilität eingebüßt aber andererseits ein schneller Einstieg, geringe Fehleranfälligkeit und Erweiterbarkeit gewonnen. Der visuelle Editor konnte leider aus Zeitgründen nicht komplett umgesetzt werden, aber er bietet einen schönen Einblick in das, was auf diesem Gebiet noch möglich wäre.

Anhang A

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Dateien

Pfad: /

_DaBa.pdf	Diplom- oder Bachelorarbeit mit Instruktionen (Gesamtdokument)

A.2 Online-Literatur

Pfad: /web-pdfs

phonegap-com.pdf . . .	Auflistung der Features von PhoneGap
stackoverflow-com.pdf .	Detaillierte Beschreibung zur Arbeitsweise von Appcelerator
www-gartner-com . . .	Marktverteilung der mobilen Betriebssysteme
www-microsoft-com.pdf	Newsmeldung zum Mango-Update
NierstraszLumpe.pdf . .	Oscar Nierstrasz und Markus Lumpe. *Komponenten, Komponentenframeworks und Gluing.*
xTextFramework.pdf . .	Sven Efftinge und Markus Völter. *oAW xText: A framework for textual DSLs.*
Applause.pdf	Informationen über Applause
AppleMobileHIG.pdf . .	*Apples* Human Interface Guidelines
MSGuidelines.pdf	*Microsofts* Application Certification Requirements

A.3 Dokumentation

Pfad: /project/documentation/

Installation.pdf	Installationsanleitung
Structure.pdf	Beschreibung der Grundstruktur
Components.pdf	Beschreibung der vorhandenen *Components*
Events.pdf	Beschreibung der vorhandenen *Events*
Models.pdf	Beschreibung der *Models*

A.4 Projektdateien

Pfad: /project

sourcecode	Quellcode des Projekts
sample/Master.json	Codebeispiel

Quellenverzeichnis

Literatur

[1] Kerstin Blumenstein und Grischa Schmiedl. *Die vier Kernprobleme der mobilen Webentwicklung*. Techn. Ber. St. Pölten, Niederösterreich: Fachhochschule St. Pölten, Mobile Forschungsgruppe, Nov. 2011. URL: http://fmt2011.fhstp.ac.at/wp-content/uploads/2011/06/Die4KernproblemeDerMobilenWebentwicklung.pdf.

[2] Andre Charland und Brian Leroux. „Mobile Application Development: Web vs. Native“. In: *Communications of the ACM* 5 (Mai 2011), S. 49–53.

[3] Christiane M. Eckl. „Cross-Platform-Development für Smartphones und dessen Einsatzmöglichkeiten im Einzelhandel“. Diplomarbeit. Hagenberg, Austria: Upper Austria University of Applied Sciences, Digitale Medien, Juni 2010. URL: http://theses.fh-hagenberg.at/thesis/Eckl10.

[4] Brian Fling. *Mobile Design and Development*. O'Reilly Media, Inc., 2009.

[5] Frank Hartmann. *Multimedia*. Facultas Verlag- und Buchhandels AG, 2008.

[6] Aaron Hillegass. *Cocoa – Programmierung für Mac OS X*. 3. Aufl. mitp, 2008.

[7] Marziah Karch. *Android for Work: Productivity for Professionals*. Apress, 2010.

[8] Ansgar Mayer. *App-Economy – Milliardenmarkt Mobile Business*. mi-Wirtschaftsbuch, 2012.

[9] Sebastian Meyer und Torben Wichers. *Objective-C 2.0: Programmierung für Mac OS X und iPhone*. Verlagsgruppe Hüthig-Jehle-Rehm, Aug. 2009.

[10] Steve Rabin. *Introduction to Game Development*. 2. Aufl. Charles River Media, 2009.

[11] Erica Sadun. *Das große iPhone Entwicklerbuch: Rezepte für Anwendungsprogrammierung mit dem iPhone SDK*. 1. Aufl. Addison-Wesley, 2010.

[12] Jonathan Stark. *Building iPhone Apps with HTML, CSS, and JavaScript*. 1. Aufl. O'Reilly Media, 2010.

[13] Christof Weinhardt u. a. „Cloud-Computing – Eine Abgrenzung, Geschäftsmodelle und Forschungsgebiete". In: *WI – STATE OF THE ART* (Mai 2009), S. 453–462.

Online-Quellen

[14] Android.com. *User Interface Guidelines*. URL: http://developer.android.com/guide/practices/ui_guidelines/index.html.

[15] Appcelerator.com. *Build Native Apps*. URL: http://www.appcelerator.com/products/native-iphone-android-development/.

[16] Apple.com. *iOS Human Interface Guidelines*. Kopie auf CD-ROM (Datei web - pdfs / AppleMobileHIG . pdf). URL: http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html.

[17] Sven Efftinge und Markus Völter. *oAW xText: A framework for textual DSLs*. Version 1.1. Kopie auf CD-ROM (Datei web-pdfs/xTextFramework.pdf). Sep. 2006. URL: http://eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf.

[18] Peter Friese und Heiko Behrens. *Applause*. Kopie auf CD-ROM (Datei web-pdfs/Applause.pdf). URL: https://github.com/applause/applause#readme.

[19] Laurence Goasduff und Christy Pettey. *Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth*. Kopie auf CD-ROM (Datei web-pdfs/www-gartner-com.pdf). Feb. 2012. URL: http://www.gartner.com/it/page.jsp?id=1924314.

[20] Jeff Haynie. *How Does Appcelerator Titanium Mobile Work?* Kopie auf CD-ROM (Datei web-pdfs/stackoverflow-com.pdf). URL: http://stackoverflow.com/questions/2444001/how-does-appcelerator-titanium-mobile-work.

[21] Microsoft. *Codename Mango: Windows Phone mit vielen neuen Funktionen*. Kopie auf CD-ROM (Datei web-pdfs/www-microsoft-com.pdf). Mai 2011. URL: http://www.microsoft.com/germany/newsroom/pressemitteilung.mspx?id=533371.

[22] Microsoft.com. *Application Certification Requirements for Windows Phone*. Kopie auf CD-ROM (Datei web-pdfs/MSGuidelines.pdf). URL: http://msdn.microsoft.com/en-us/library/hh184843(v=vs.92).aspx.

[23] Microsoft.com. *Windows Phone 7: Start Screen (Blue)*. Okt. 2012. URL: http://www.microsoft.com/presspass/ImageGallery/ImageDetails.mspx?id=A2964CEA-FC7B-4BF2-BE42-1D1004214216.

[24] Oscar Nierstrasz und Markus Lumpe. *Komponenten, Komponentenframeworks und Gluing*. Kopie auf CD-ROM (Datei web-pdfs/Nierstrasz Lumpe.pdf). 1997. URL: http://scg.unibe.ch/archive/papers/Nier97aKomponentenUndGluing.pdf.

[25] PhoneGap.com. *Supported Features*. Kopie auf CD-ROM (Datei web-pdfs/phonegap-com.pdf). URL: http://phonegap.com/about/features.