Hochschule Neubrandenburg
University of Applied Sciences

Hochschule Neubrandenburg
Studiengang Geoinformatik

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Engineering (B.Eng.)

Entwicklung einer Web Applikation in der Gastronomie

Name: Karsten Prehn
URN: urn:nbn:de:gbv:519-thesis2012-0642-8

1. Betreuer: Prof. Dr. Andreas Wehrenpfennig
2. Betreuer: Dr. Rüdiger Lincke

Datum: 31.08.2012

Danksagung

Ich möchte mich zuerst bei meiner Familie, insbesondere bei meinen Eltern, für die Unterstützung (und auch mittlerweile langjährige Geduld mit mir) bedanken – sowohl die Unterstützung während des Studiums und während des Schreibens der Arbeit, als auch dafür, dass sie mir den Schwedenaufenthalt, und damit auch diese Arbeit, mit ermöglicht haben.
Mein weiterer Dank gilt Rüdiger und Welf, die mir die Chance gegeben haben, ein Praktikum in ihrer Firma zu absolvieren, bei dem ich eine Menge lernen konnte, die es über die tägliche Zusammenarbeit hinaus bei diversen geselligen Abenden aber auch geschafft, dass ich mich heimisch gefühlt hab.
Dann gilt mein Dank noch, and here is, were I continue in English, every employee of Softwerk for every help during daily work and for the good cooperation we had: Therése, Tobi, Hanna, Stani (especially for answering all my Android related questions), Erik (same for iOS and for the crash course in Objective C), Andriy, Viktor, Maksym, Vladimir and everyone I forgot to mention. I hope I didn't, though.

Eidesstattliche Erklärung

Ich versichere, die Bachelorarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Neubrandenburg, _____

_____

Karsten Prehn

Abstract/Zusammenfassung

This bachelor thesis describes the development of a smart phone app for a restaurant on the campus of Linnæus University in the Swedish town of Växjö. The app was created as a commission for the software company ARiSA AB / Softwerk that also can be found in Växjö. Today the app is available as an iPhone and Android OS download. The thesis discusses the development of the app in terms of exchange of information between users and a restaurant and describes the creation of the app with the fully implemented functionality of information flow in one direction, from the restaurant to the user.

Diese Arbeit behandelt die Entwicklung einer Smartphone App für ein Restaurant auf dem Campus der Linnæus Universität in Växjö, Schweden. Die App entstand als Auftrag für das, sich ebenfalls in Växjö befindende, Softwareunternehmen ARiSA AB / Softwerk und ist heute als iPhone und Android OS Download erhältlich. In der Arbeit wird die Entwicklung der App unter dem Gesichtspunkt des Informationsaustauschs zwischen Nutzer und Restaurant diskutiert und die App mit der vollen Funktionalität des Informationsflusses in die eine Richtung, vom Restaurant zum Nutzer, entwickelt.

Inhalt

1 Einleitung

Eine *App*, Kurzform für Applikation (oder englisch *application*), ist ein, zumeist aus einer einzigen Datei bestehendes Programm, das speziell für den Einsatz auf Smartphones entwickelt wurde und aus, eigens für den Vertrieb von Apps entwickelten, Online-Märkten (oft kostenpflichtig) herunterladbar ist. Die Online-Märkte sind für gewöhnlich an das Betriebssystem des Smartphones geknüpft und - ebenfalls als App - bereits vorinstallierter Bestandteil des Gerätes. Smartphone bezeichnet all jene Telefongeräte, die in ihrem Funktionsumfang Computern ähneln und wie diese die Installation von Software von Drittanbietern, eben jenen Apps, erlauben.
In diesem Kapitel folgt eine Beschreibung der Problemstellung sowie von Zielsetzung und Aufbau der vorliegenden Arbeit. Auf die Unterscheidung zwischen einer App für Smartphones und einer Web Applikation wird im Verlauf der Arbeit eingegangen (vgl. 3.1.3)

1.1 Problemstellung

Der Smartphone-Markt wächst in den letzten Jahren rasant. Mittlerweile besitzen 34% und damit mehr als jeder dritte Deutsche ein Smartphone [Hei12]. Die Verbreitung von Smartphones in Deutschland erfuhr vom ersten Quartal 2011 zum ersten Quartal 2012 eine Steigerung um elf Prozentpunkte von 18% auf 29%. In Ländern wie Schweden, Norwegen oder Großbritannien haben jetzt bereits mehr als 50% der Bevölkerung eines [Goo121]. Und der Markt wächst weiter. Innerhalb der verschiedenen Anbieter von Smartphone-Betriebssystemen haben in Deutschland Android-Systeme (Google) den höchsten Marktanteil (40%), gefolgt von Symbian (Nokia) und Apples iOS (jeweils 24% bzw. 22%) [BIT12].
Weltweit betrachtet hatten Android-Geräte im Jahr 2011 einen Anteil von rund 48% an den Gesamtverkäufen an Smartphones, iOS hatte einen Anteil von rund 19% und Symbian von rund 16% und es wurden im Jahr 2011 erstmals mehr Smartphones als herkömmliche PCs (Pads, Net- und Notebooks und Desktop PCs) verkauft [can12].

Einer Google-Studie zufolge, nutzen in Deutschland 40% der Smartphone-Besitzer ihr Gerät täglich zur Suche im Internet - hauptsächlich, um sich über Produktangebote, Restaurants und Gaststätten oder touristische Angebote zu informieren [Goo122]. In Schweden sind es 48% der Smartphone-Besitzer, die ihr Gerät dazu verwenden, sich über Produkte, Restaurants oder Reiseangebote zu informieren [Goo123].

Ein wesentliches Feature von Smartphones sind Apps. Apps existieren in allen erdenklichen Größen und zu allen Thematiken: von Gimmicks ohne näheren Sinn, über Unterhaltungsprogramme und Spiele bis zu ernsthaften Anwendungen, die bspw. Patienten bei Krankheiten unterstützen. So ermöglichen es Apps u.a. auch, Informationen, wie lokale Angebote oder Werbung, in kompakter und für den Nutzer leicht zugänglicher Form

anzubieten; und erlauben es dem Benutzer zusätzlich, diese Informationen zu filtern, zu bewerten oder zu kaufen. Für einen Unternehmer wird es somit auch zunehmend wichtiger, neben einem Webauftritt, auch in Form einer App online und mobil mit dem Kunden in Kontakt zu treten.

Der folgende Graph (Abbildung 1) zeigt, beispielhaft für Schweden und Deutschland, wie Smartphones den Nutzer bei Informationssuche und Entscheidungsfindung unterstützen.

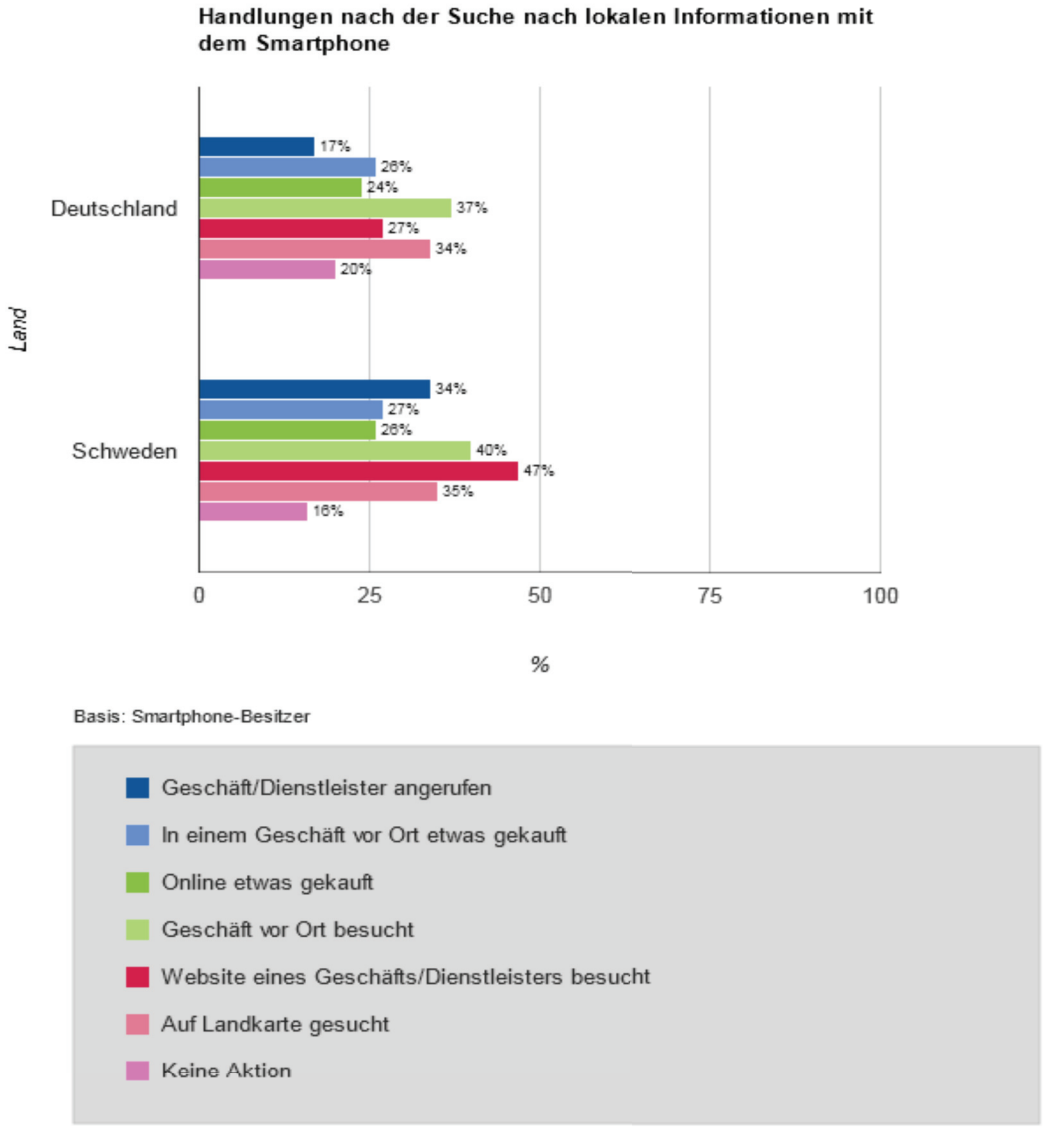

Abbildung 1: Smartphones als Entscheidungshilfe (entnommen von [Goo124])

Aus Entwicklersicht stellt sich auch immer die Frage, für welche Plattform bzw. für welche Plattformen entwickelt man, um möglichst viele Nutzer zu erreichen – wenn dies nicht vom Kunden vorgegeben ist.

1.2 Zielsetzung und Aufbau der Arbeit

Ziel der Arbeit ist es, eine App für zwei Plattformen zu erstellen, die sich in eine angemessene Systemarchitektur einfügt, welche es ermöglicht, veränderliche Daten so zu pflegen, dass der Nutzer die Änderungen als nicht mehr als einen Hintergrundprozess wahrnimmt. Die App wird, beispielhaft für ein beliebiges Unternehmen, das sein Angebot oder seine Dienstleistung in strukturierter und geordneter Form (z.B. Tabelle, Liste, Menü) anbietet, den Zweck haben, den Nutzer bei der Entscheidungsfindung hinsichtlich des Angebots dieses Unternehmens zu unterstützen.
Die Wahl hinsichtlich der Branche fiel auf den Gastronomie-Bereich. Restaurants stellen ihr Speiseangebot bereits als Liste aus; bzw. sind gesetzlich sogar dazu verpflichtet, den Kunden über ihre Preise zu informieren. Eine App kann an dieser Stelle den Aspekt des Informierens als digitale Speisekarte übernehmen und um weitere nützliche Aspekte, wie Angebotspräsentation auch außerhalb der Lokalität, Reaktion auf Angebotsänderungen in einer Form, die mit gedruckten Speisekarten nicht möglich ist oder Filtern und damit Reduzieren des Angebots auf spezielle Wünsche des Kunden, erweitern.
Als Unternehmen konnte die „Pizzeria & Restaurant Maestro“ gewonnen werden, die sich auf dem Campus der Universität Växjö in Schweden befindet.

Die gewählten Plattformen sind Android OS und iOS. Die Wahl auf diese beiden Systeme ist zum einen darin begründet, dass sie die beiden größten Vertreter unter den verwendeten Smartphone Betriebssystemen sind [can12], sie andererseits auf bekannten Technologien (Java und Objective C) aufbauen und die beiden Systeme, neben ihren Gemeinsamkeiten, auch teilweise komplett gegensätzliche Unterschiede haben – sowohl aus technischer Sicht, als auch hinsichtlich der Philosophie der Plattformbetreiber und der Paradigmen, welche die Betreiber ihrem System zugrunde gelegt haben.
So ist es bspw. eine „Eigenart“ des hardwareunabhängigen Android, dass es viele Versionen des Systems aufgrund einer schnellen Weiterentwicklung der Plattform gibt; während iOS viel „ruhiger“, beinahe im Jahresrhythmus und oft zeitglich mit neuer Hardware, seine großen Updates erfährt. Auf solche Unterschiede, aber auch auf die Gemeinsamkeiten, wird im Verlauf der Arbeit noch näher eingegangen. Android und iOS sind darüber hinaus, um das auch hervorzuheben, technisch komplett inkompatibel zueinander.

Andererseits spielen bei der Wahl der Systeme aber auch rein marktwirtschaftliche Aspekte eine Rolle. Mit diesen beiden Systemen sind bereits eine sehr große Menge an Smartphones und damit potentiellen Kunden des Unternehmens abgedeckt.
Solche eher betriebswirtschaftlichen Überlegungen seien hier aber nur deshalb erwähnt, weil für ein reales Unternehmen entwickelt wird und dieses damit als Kunde der Softwareentwicklung auftritt. Sie sollen im Verlauf der Arbeit aber keine weitere Rolle spielen, da sie zur Problemlösung nichts beitragen. Das Problem ist rein technischer Natur. Die Inkompatibilität der

verschiedenen Betriebssysteme bleibt bestehen, wenn Android und iOS gegen andere Plattformen ausgetauscht werden.

Es ist ein zweites Ziel, dass die beiden Apps, die Gegenstand dieser Arbeit sind, in den jeweiligen Verkaufsplattformen - d.h. Google Play (ehemals Android Market) für die Android und Apples *App Store* für die iPhone Version der App - zum kostenlosen Download bereit gestellt werden.

Die Arbeit wird analysieren, welche Technologien zur Verwirklichung des Ziels überhaupt nötig sind und in welche Systemlandschaft diese Technologien eingebettet sein müssen. Die beiden Zielplattformen werden untersucht und gegenübergestellt. Es wird sowohl auf Unterschiede zwischen den beiden Plattformen eingegangen, als auch auf Unterschiede innerhalb der jeweiligen Plattform und es wird erörtert, wie sich diese Unterschiede auf den praktischen Teil des Programmierens auswirken. Es werden aber auch die Gemeinsamkeiten berücksichtigt und darauf aufbauend Schnittstellen und Datenaustauschmöglichkeiten analysiert, die einen von beiden Systemen gemeinsam nutzbaren Serverteil vom jeweiligen Client trennen bzw. verbinden.

Eingangs erfolgt ein Einblick in den Stand der Technik (Kapitel 2), die beiden Systeme Android OS und iOS betreffend. Diesem folgt eine Analyse der Anforderungen an das Projekt (Kapitel 3), unterteilt in nichtfunktionale (u.a. Design, Interaktion, Skalierbarkeit, Erweiterungsmöglichkeiten) und funktionale Anforderungen (Daten, Schnittstellen, Internationalisierung). Als drittes (Kapitel 4) wird - aufbauend auf der Analyse - ein System mit Modellierung der Daten, Entwurf der äußeren Architektur sowie der App-internen Logik, des Aufbaus der App selbst und des GUIs, entworfen. Dem folgt (Kapitel 5) eine Beschreibung der konkreten Umsetzung beider Projektversionen. Zum Abschluss (Kapitel 6) wird eine Zusammenfassung der Arbeit geliefert und ein Ausblick auf zukünftiges gegeben.

2 Stand der Technik

Im diesem Abschnitt folgt eine kurze Einführung in die beiden Plattformen Android OS und iOS und eine Beschreibung der entsprechenden SDKs sowie der IDEs, die für die Programmierung verwendet wurden.

2.1 Android OS, SDK und IDE

Android ist eine Open Source Software Plattform und gleichzeitig ein Betriebssystem für Mobilgeräte (Smartphones, Netbooks, Tablets, etc.), das - anders als beispielsweise Apples proprietäres iPhone mit dem iOS BS - hardwareunabhängig ist. Android wird seit November 2007 von der Open Handset Alliance (http://www.openhandsetalliance.com/), einem Zusammenschluss mehrerer Firmen, darunter Netzbetreiber, Chip- und Endgerätehersteller, Software- und Vermarktungsfirmen, unter der Leitung von Google entwickelt und gepflegt (näheres siehe u.a. [Mos09], S.10 ff).

Das Android Betriebssystem sitzt auf einem Open Source Linux Kernel der Version 2.6 auf, der dem Android Software Stack eine Hardware Abstraktionsschicht hinzufügt (Abbildung 2).

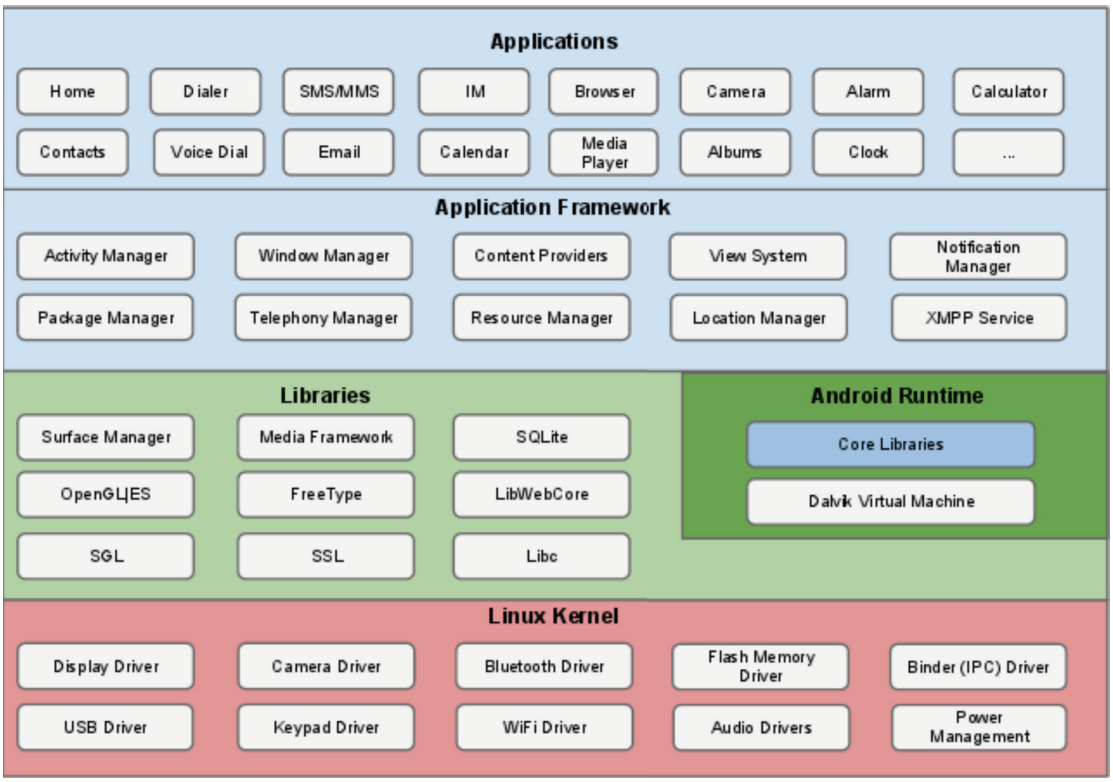

Abbildung 2: Android Software Stack (Quelle: http://source.android.com/tech/security/index.html)

Die Elemente des Stacks sind über Schnittstellen (APIs) frei ansprechbar. Die Verwendung des Linux Kernels ermöglicht UNIX-typische Features, wie ein UNIX Filesystem, Speicherverwaltung, Sicherheitsaspekte wie sichere Interprozess-Kommunikation und Prozessisolation sowie Portabilität durch Abstraktion der unterliegenden Hardware.

Für Entwickler bietet Android die notwendigen Toolkits als Bestandteil des Android Software Development Kits (SDK): bspw. einen Compiler, einen Debugger, Beispiel Code und eine Javadoc konforme Dokumentation sowie Frameworks (Application, UI), Open Source Bibliotheken (bspw. SQLite, WebKit, OpenGL) und eine Runtime - die Dalvik Virtual Machine (DVM) - die die Java Virtual Machine (JVM) ersetzt.
Als Entwicklungsumgebung (IDE) kommt Eclipse zum Einsatz. Für Eclipse wurde das Android Development Tools (ADT) Plugin veröffentlicht, das unter anderem einen Simulator und Debugging Tools beinhaltet. Programmiert wird in Java, wobei zu beachten ist, dass Android seine eigenen Bibliotheken mitbringt. Es entspricht zwar größtenteils der Java Standard Edition; deren Bibliotheken für Benutzerschnittstellen (AWT und Swing) wurden allerdings durch Android-eigene ersetzt (u.a. [Mar11], S.1 ff) und sind auch nicht verwendbar.

Wer für Android entwickeln oder das Programm dieser Arbeit selbst kompilieren möchte, benötigt:

- das Java Development Kit (JDK) 5.0 oder 6.0
- das Android SDK
- Eclipse Classic oder Eclipse für Java Entwickler
- das Android Development Tools (ADT) Plugin für Eclipse (siehe unter Weblinks)
- von hier an können über den Android SDK Manager die gewünschten APIs installiert und mit dem Android Virtual Device Manager ein Simulator erstellt werden

Hilfe bei den ersten Schritten bieten neben viel guter Literatur zwei Quellen im Internet (siehe unter Weblinks),

- die Android Developer Seiten
- und StackOverflow

die auch während der Arbeit von immenser Hilfe waren.

Ein erwähnenswertes Merkmal von Android ist die schnelle Weiterentwicklung des Systems. Während die „Maestro App“, die Gegenstand dieser Arbeit ist, noch für API Level 15, d.h. Android 4.0 ([AndNA]), entwickelt wurde, ist bereits Android 4.1 ([AndNA1]) draußen.
Für den Entwickler kann dieser Umstand durchaus schwierig sein, da hier zwei entgegengesetzte Ziele erfüllt werden müssen. Zum einen möchte man eine möglichst große Zielgruppe erreichen und deshalb den API Level so klein wie möglich halten, zum anderen möchte man aber auch den Funktionsumfang der neuen APIs voll ausnutzen; denn der API Level bestimmt, auf wie vielen Geräten die App letztendlich läuft.
Tabelle 1, ein Schnappschuss des Android Developer Dashboards, listet die verschiedenen Android Versionen mit ihren Codenamen, API Levels und prozentualen Verteilungen auf. Die Daten stammen vom 2. Juli 2012 und wurden während der 14 Tage vor diesem Datum erhoben. Noch nicht aufgeführt ist die neue Version 4.1. Abbildung 3 ist derselben Webseite entnommen und verdeutlicht noch einmal grafisch die Verteilung der Android Versionen. Auffällig ist, dass fast zwei Drittel aller Geräte noch mit Version 2.3 bis 2.3.7 laufen. Android 2.3.3, die häufigste Version, erschien im Februar

2011; 4.0 erschien im Oktober des selben Jahres. Es lässt sich sagen, dass die Weiterentwicklung der Software schneller voran schreitet, als die Nutzer ihre Geräte anpassen. Als Entwickler lässt sich dem durch Einbindung von Backward-Compatibility- oder Support-Libraries (Link unter Weblinks) oder Open Source Extensions wie *ActionBarSherlock* (Link unter Weblinks) begegnen. Letztere kam auch im Rahmen dieser Arbeit zum Einsatz.

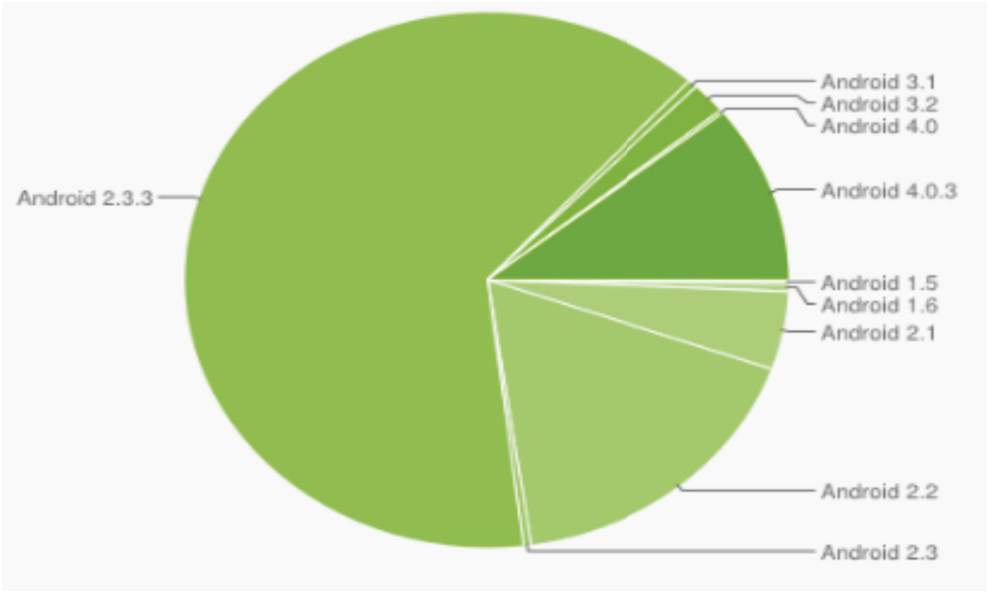

Abbildung 3: Verteilung Android Versionen im Juli 2012 (Quelle: [And12])

Version	Codename	API Level	Distribution
1.5	Cupcake	3	0.2%
1.6	Donut	4	0.5%
2.1	Eclair	7	4.7%
2.2	Froyo	8	17.3%
2.3 - 2.3.2	Gingerbread	9	0.4%
2.3.3 - 2.3.7		10	63.6%
3.1	Honeycomb	12	0.5%
3.2		13	1.9%
4.0 - 4.0.2	Ice Cream Sandwich	14	0.2%
4.0.3 - 4.0.4		15	10.7%

Tabelle 1: Android Versionen, Bez., Verteilung im Juli 2012 (Quelle: [And12])

In der Praxis macht sich diese Versionsvielfalt dadurch bemerkbar, dass Programme erst auf verschiedenen Geräten getestet werden müssen, um ihre korrekte Funktionsweise sicher zu stellen oder Fehler zu entdecken, die sich nur auf einigen der Geräte bemerkbar machen. Dies stellt auch einen der

großen Unterschiede zwischen der Entwicklung für Android zu der für iOS dar. Zwar existieren auch Unterschiede zwischen den Modellen des iPhones oder iPod Touch, aber die grundlegende Nutzerinteraktion über den Touchscreen und den einen Hardware-Button sind immer gleich. So besitzt beispielsweise jedes iPhone eine Kamera, das iPod Touch [1] nicht. Als Entwickler muss man das bedenken und mit einkalkulieren, auf welchen Produktreihen die fertige App fehlerfrei laufen soll, wenn man bspw. die Kamera bezogenen Bibliotheken und Schnittstellen programmieren möchte. Andererseits weiß man in dem Fall dann aber auch genau, welches Kameramodell verbaut ist und welche optischen Parameter diese Kamera hat und könnte bspw., darauf aufbauend, Berechnungen durchführen. Im Gegensatz dazu besitzt ein Gerät mit Android OS zwar mit sehr hoher Wahrscheinlichkeit eine Kamera - diese kann aber, je nach Hersteller, komplett unterschiedlich ausfallen. Eine per se Annahme zu den Kameraparametern wäre in diesem Fall ohne weiteres nicht möglich. (weiter ausführendes zu Unterschieden zwischen iOS Modellen siehe [Sad10], S.48 ff)

[1] Anm.: Die neueren Modellreihen des iPod Touch haben mittlerweile eine Kamera. Dies war zur Zeit des referenzierten Buches noch nicht der Fall.

2.2 iOS, SDK und IDE

iOS (vormals iPhone OS) ist das für den Einsatz mit dem iPhone, iPad und iPod Touch ab dem 9. Januar 2007 entwickelte Betriebssystem der Firma Apple. Im Gegensatz zu Android ist iOS somit für eine konkrete Hardware entwickelt. Es basiert auf dem, u.a. auch in MacBooks verwendeten, Mac OS X Betriebssystem, angepasst an den ARM Prozessor, der in den meisten Smartphones, so auch dem iPhone, verbaut ist [Net11].

Das iPhone SDK von Apple ist für Mitglieder des freien Entwicklerprogramms kostenlos. Bestandteil der SDK ist Apples integrierte Entwicklungsumgebung XCode. XCode ist die IDE mit der für gewöhnlich für iOS entwickelt wird, da XCode - ähnlich Eclipse mit ADT Plugin - alles Erforderliche für die iOS Entwicklung liefert: Compiler, Debugger, Dokumentation, Code-Autovervollständigung, etc.
Ebenfalls Bestandteil der SDK ist ein Simulator, der die iPhone API simuliert. Ein aus Entwicklersicht dritter wesentlicher Bestandteil der SDK ist der Interface Builder, der es, wie der Name andeutet, erlaubt, GUIs und Layouts grafisch aus vordefinierten Interfaces zusammen zu setzen und mit dem Code zu verlinken. iOS Entwicklung folgt dem MVC Ansatz. Zu erwähnen ist noch, dass das SDK keinen Garbage Collector für iOS Programme beinhaltet. Speicherverwaltung geschieht entweder manuell (MRC), indem der Programmierer sich um die Lebensdauer seiner Objekte durch Aufruf der *retain*, *release* oder *autorelease* Methoden selbst kümmert oder, seit XCode 4.2, automatisch (ARC) durch den Compiler (näheres [App111], [App112] und Links).

Kompiliert wird durch Verlinkung mit den verwendeten Frameworks. Die Frameworks (z.B. CoreData, UIKit, Foundation) sind die von Apple zur Verfügung gestellten Softwarebibliotheken, die ihrerseits Klassendefinitionen für Cocoa Touch bereitstellen. Aus Quellcode, verlinkten Frameworks sowie Medien (Bild, Audio) kompiliert XCode ein Anwendungspaket, das auf dem Endgerät installiert werden kann. Programmiert wird in Objective C. (ausführlicheres z.B. unter [App113] und ebenda verlinkte Kapitel)

Objective C 2.0 ist eine objektorientierte Programmiersprache mit Mehrfachvererbung und eine Obermenge von C, die aus C und Smalltalk entwickelt wurde. Objective C erweitert ANSI C um Elemente aus Smalltalk; die Syntax ist an Smalltalk angelehnt. Objective C Klassen bestehen typischerweise aus einer Headerdatei (.h) in der das Interface definiert wird - d.h. die Abstammung (Vererbung) und Protokolle, Instanzvariablen, Konstanten, die Attribute (Properties) und die Methoden der Klasse beschrieben werden - und einer .m Datei, die das Interface implementiert. Die .m Datei entspricht der .c Datei in C.

Wer für iOS entwickeln oder die App dieser Arbeit kompilieren möchte, benötigt

- einen Intel basierten Macintosh Computer mit OS X Betriebssystem

- die iOS SDK und einen Developer Account, der für die reine Entwicklung im Online Developer Program aber kostenlos ist (Link unter Weblinks)[2]
- die XCode IDE aus dem iOS Developer Program (Link unter Weblinks)
- unter Umständen etwas Einarbeitungszeit in die Programmierung mit Objective C

[Sad10]

Hilfe beim Einstieg in die oder zu Problemen während der iOS Programmierung bietet neben dem bereits für Android erwähnten Stackoverflow die hervorragende Mac OS X Developer Library ([Dev127]).

Abbildung 4 zeigt die verschiedenen Abstraktionsschichten des iOS. Als Entwickler nutzt man die Dienste, die jede Schicht anbietet, über Frameworks. Diese Frameworks sind Bibliotheken und liefern die Schnittstellen zu den Schichten. Eine Auflistung aller Frameworks findet sich unter [App11].

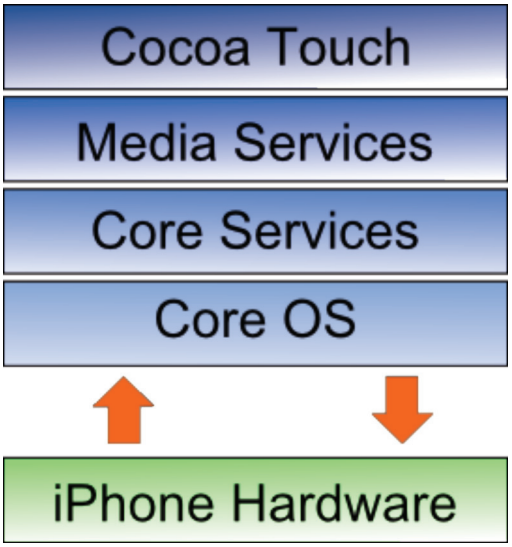

Abbildung 4: iPhone / iOS Architektur (Quelle: [tec11])

[2] Das Online Developer Program beschränkt das Programmieren auf den Mac und das Testen der Anwendungen auf den XCode Simulator. Will man seine Apps im App Store verkaufen oder auf einem echten Gerät testen, benötigt man einen Account für eines der kostenpflichtigen Programme.

Das *Core OS* ist zuständig für die Speicherverwaltung, das Dateisystem, die Netzwerkverwaltung und es ist die einzige Schicht, die direkt mit der darunter liegenden iPhone Hardware kommuniziert. Die Core OS Schicht besteht u.a. aus folgenden Komponenten:

- OS X Kernel
- Mach 3.0 (Kernel)
- Sockel
- Keychain
- Dateisystem

Die *Core Services* Schicht abstrahiert die Dienste der Core OS Schicht und stellt so direkten Zugang zu iOS Diensten dar, wie:

- das Adressbuch
- SQLite
- Netzwerkdienste
- Threading

Die *Media Services* Schicht stellt Multimedia Dienste für iPhone und iPad Anwendungen bereit. U.a.:

- OpenGL ES (Grafik API)
- Audioaufnahme
- Videowiedergabe
- Dateiformate (JPG, PNG, TIFF, PDF)

Die oberste Schicht, *Cocoa Touch*, ist Bestandteil der SDK. Sie stellt eine Abstraktionsschicht zum Programmieren mit den verschiedenen iOS Bibliotheken bereit und ist die für den Entwickler wohl wichtigste Klassenbibliothek. Sie ist zuständig für Ereignisbehandlung und Animation der Benutzeroberfläche und stellt u.a. folgende Schnittstellen zur Verfügung:

- Touch Events
- Beschleunigungsmesser
- Web Views
- Lokalisation
- Controller [Lee12]

Eine weitere wichtige Funktion von Cocoa Touch ist es, den Entwickler dabei zu unterstützen, Anwendungen zu erstellen, die den Design Vorgaben von Apple entsprechen. Im Gegensatz zum eher liberal gehaltenen Google Play, das zwar eine Meldefunktion für auffällige Apps besitzt, den Entwickler beim ersten Upload aber nicht einschränkt, werden iOS Apps nach Upload in den App Store von Apple vor der Freigabe auf Erscheinung, Betrieb, Inhalt und Funktionsumfang überprüft. Die App kann durch diese Prüfung zurückgewiesen und muss ggf. überarbeitet werden.

2.3 Reaktion auf Nutzereingaben & Ereignisse

Jede Interaktion über das GUI erzeugt eine neue Sicht oder modifiziert die bestehende. Die Registrierung von User Events am Touchscreen (vgl. Tabelle 2) oder an Buttons der Hardware und deren Weiterleitung an die App sind Aufgabe des jeweiligen BS.

Bei iOS ist die „Main Run Loop“ der App verantwortlich für die Verarbeitung aller Nutzereingaben. Nutzereingaben, -gesten und -ereignisse werden der Reihe nach abgearbeitet; so wie sie auf dem Ereignisstapel (Event Queue) liegen. Ein spezieller Port liefert die Ereignisse, die vom Betriebssystem generiert werden, an die App weiter, wo sie intern von der Main Run Loop abgearbeitet werden. [Dev125]

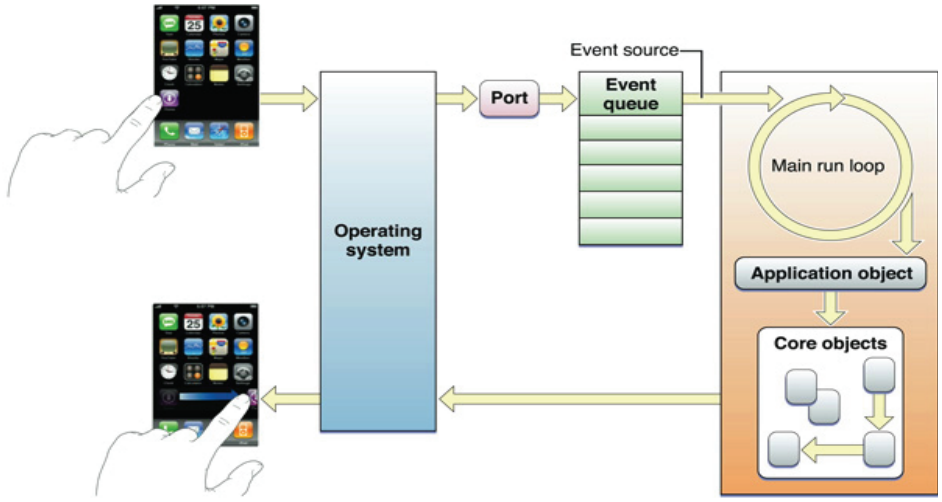

Abbildung 5: Ereignisbehandlung iOS (übernommen von [Dev125], ‚Figure 3-7’)

Bei Android reagieren die Event Listener, die in der momentanen View registriert sind, auf die Callback Methoden, die das Android Framework aufruft, wenn interaktive UI Elemente in der View durch den Nutzer betätigt werden [Dev126].

3 Anforderungsanalyse

3.1 Nichtfunktionale Anforderungen

Wenn ein Kunde (hier ein bspw. Unternehmer als Auftraggeber[3] eines Softwareherstellers) den Wunsch äußert, zu einer bestimmten Thematik eine App auf dem Markt anzubieten, hat er, über diesen Wunsch hinaus, selten eine Vorstellung von der tatsächlich vorherrschenden Systemlandschaft. iPhone und Android sind den meisten zwar ein Begriff, wie aber diese Geräte auf dem Markt zwischen sämtlichen Plattformen verteilt sind und welche Systemplattformen dort noch vorhanden sind, jedoch nicht.
Wie eingangs erwähnt, ist die App dieser Arbeit gleichzeitig ein Auftrag eines realen Unternehmens, des „Restaurant und Pizzeria Maestro“ in Växjö, Schweden. Es war der Wunsch von Maestro, eine App auf dem Markt zu haben, die hauptsächlich ihr Speiseangebot darstellt; zuzüglich aber noch Kontakt- und allgemeine Informationen (Öffnungszeiten, Lieferdienst) sowie ortsbezogene Informationen (Karte, Navigation). Ziel ist demnach eine App, mit welcher der Nutzer sich mobil für ein Essen entscheiden kann und mit der er bei der Navigation zu der Lokalität unterstützt wird, wenn er es wünscht.

3.1.1 Systemverteilung, Zielsysteme

Die erste Frage, die sich stellt, ist, auf welchen Plattformen soll die App erhältlich sein? Das bedeutet, für welche Smartphone-Betriebssysteme soll sie entwickelt werden. Dies ist aus zwei Gründen von erheblichem Interesse. Einerseits möchte der Auftraggeber eine möglichst große Schnittmenge an Smartphone-Plattformen seiner Kunden erreichen. Auf der anderen Seite spielt die nichtvorhandene Portabilität und Kompatibilität zwischen den Plattformen eine Rolle. Je mehr Plattformen abgedeckt werden sollen, desto größer ist zwar die erreichte Klientel, desto teurer wird es im Umkehrschluss aber auch für den Auftraggeber, da jede zusätzliche Plattform eine separate Entwicklung benötigt. So ist es nicht unüblich, dass bspw. zuerst für iOS entwickelt wird und während der Zeit, in der die App für den App Store überprüft wird, für eine zweite oder dritte Plattform zu entwickeln. So muss für gewöhnlich dann nur noch der Code geschrieben werden, da Design, Font, Icons, Layout, etc. bereits anhand der iPhone Version mit dem Auftraggeber abgestimmt sind. Im Idealfall ist so die App auf allen Zielplattformen gleichzeitig erhältlich. Umso wichtiger ist es demnach aus Entwicklersicht, solche gemeinsam genutzten Komponenten zu erkennen und separat zu haushalten, so dass Kosten und Entwicklungszeit minimiert werden.

[3] Ab jetzt soll *Auftraggeber* das Auftrag gebende Unternehmen, im Fall der Arbeit also Maestro, als Kunden der Softwareentwicklung bezeichnen und *Kunde* einen Gast des Unternehmens. Diese Unterscheidung ist sinnvoll, da sowohl der Gast ggü. dem Unternehmen, als auch das Unternehmen ggü. der Softwareentwicklung als Kunde auftritt.

Da das Restaurant Maestro ein schwedisches Unternehmen ist und die Kunden von Maestro größtenteils aus der Nähe des Campus der Universität Växjö, auf dem sich ebenfalls das Restaurant befindet, stammen, ist es zuerst von Interesse, welche Plattformverteilung in Schweden vorzufinden ist. Ideal wäre eine Befragung der direkten Klientel, also der Studenten und Lehrkräfte, gewesen. Dies war im Rahmen der Arbeit allerdings nicht machbar. Einer von Google in Auftrag gegebenen globalen Studie ([Goo124]) zwischen März und Juli 2011 sowie Januar und März 2012 zufolge, bei der in Schweden 1000 Menschen befragt wurden, ergab sich für Schweden folgende Verteilung (Abbildung 6):

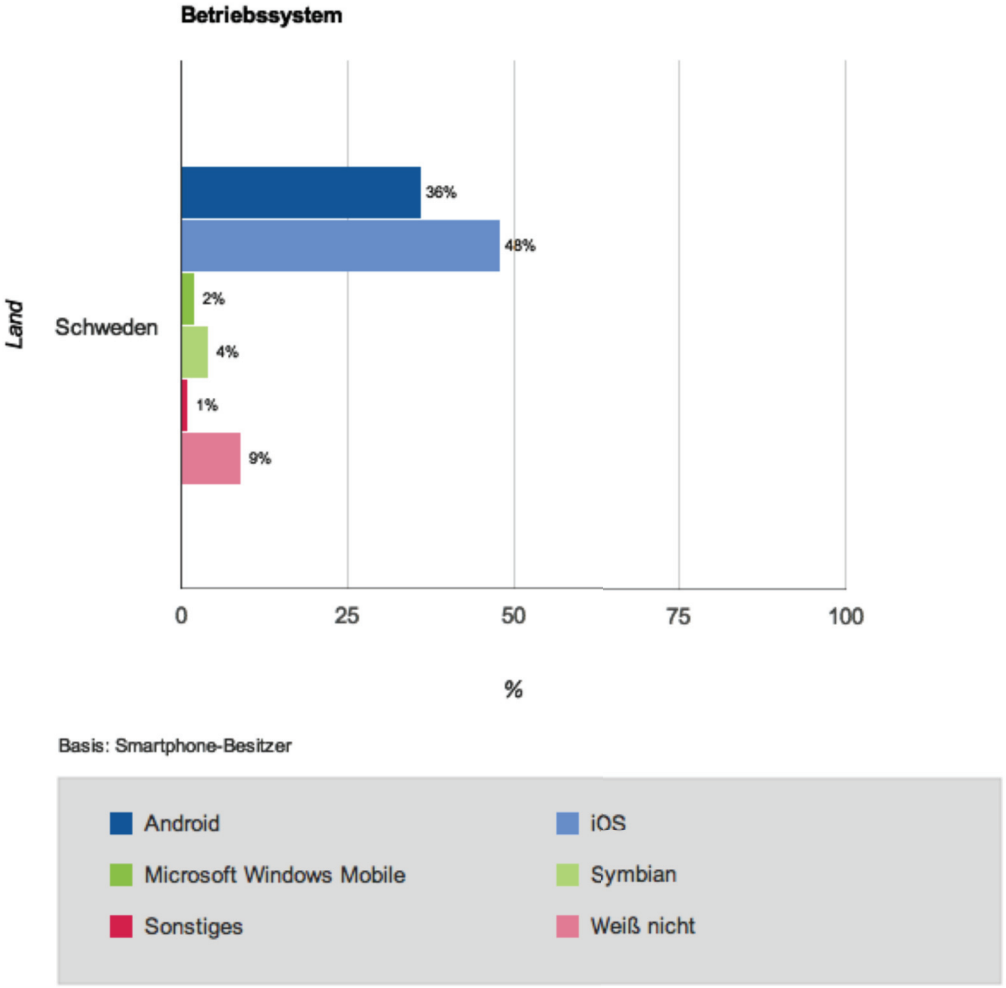

Abbildung 6: Verteilung Smartphones in Schweden entspr. BS ([Goo124])

Annähernd die Hälfte aller Befragten nutzt ein iOS Gerät, etwas mehr als ein weiteres Drittel eines mit Android OS. Anzumerken ist, dass zu Blackberry OS in dieser Studie keine Daten vorliegen. Dennoch ist eine klare Präferenz für iOS und Android OS ersichtlich. Dies sind auch die gewünschten Plattformen des Auftraggebers.

3.1.2 Design Guidelines

Die App muss den jeweiligen Richtlinien der Storebetreiber hinsichtlich Design, Bedienbarkeit, Verhalten, das vor allem der Nutzer erwartet - insbesondere Nutzerfreundlichkeit, Speicherverwaltung und allgemein Effizienz und Verlässlichkeit, etc. - entsprechen. Insbesondere Apple überprüft die Qualität jeder neuen App für den App Store genau. Richtlinien zum Design können den jeweiligen Design Guidelines entnommen werden ([DevNA] für Android OS, [Dev12] für iOS).

3.1.3 Unterschied zwischen nativer App und Webseite / Webapp

Der Unterschied zwischen einer nativen App und einer Webapp mag auf den ersten Blick nicht auffallen; und manch einer, wie Nokia-Präsident Chris Weber, beschwor bereits vor einem Jahr das Ende des App-Konzepts, wie iOS und Android es betreiben, herauf ([Adr11]) - lassen sich doch mit einer Webapp ähnliche Interface-basierte Anwendung erschaffen, wie sie native Apps sind (inklusive Icon zum Start der Webapp vom Homescreen). Und so ist der größte Unterschied auch der, welcher dem Nutzer wahrscheinlich gar nicht bewusst ist: eine Webapp läuft im Browser; eine native App benötigt eine Laufzeitumgebung und muss vorher aus einem Online-Shop (oft kostenpflichtig) heruntergeladen werden - sie kann dafür aber, im Gegensatz zur Webapp, auch offline genutzt werden, da sie auf dem Smartphone installiert wird. Möchte man Funktionen des Smartphones, wie das Gyroskop, Bluetooth oder spieletaugliche Graphik nutzen, führt auch kein Weg an einer nativen App vorbei.
Der größte Unterschied aus Entwicklersicht ist sicherlich der Mehraufwand (Zeit und Kosten) in der Entwicklung nativer Apps: es muss für jede Plattform separat in einer eigenen Sprache entwickelt werden, während Webapps auf HTML5, CSS3, JavaScript und server-seitige Sprachen (PHP, Python) zurückgreifen und sich letztendlich nur durch den darstellenden Browser unterscheiden. (weiterführendes zu Unterschieden zwischen nativer und Webapp unter [JTM12])

Eine Webseite ist für große Bildschirme konzipiert und bündelt eine Fülle an heterogenen Informationen und Optionen oft bereits auf der ersten Seite. Der Besucher könnte die Seite ja verlassen und nie wieder kommen. Diese Gefahr ist in der Form bei einer App[4] nicht gegeben. Der Nutzer hat für Gewöhnlich bereits eine Vorstellung von dem, was die App kann und er erwartet, dass er sich durch Menüs navigieren und Listen scrollen muss. Wichtige Informationen und Kernaussagen müssen bei einer App nicht gebündelt werden, da die App selbst bereits thematisch fokussiert ist. Die beschränkte Größe des Bildschirms eines Smartphones, der als Arbeitsfläche zur Verfügung steht, lassen diese Strategie auch komplett unbrauchbar und ineffizient werden. Bei der Entwicklung einer Smartphone App, rückt das

[4] Wenn nicht extra angegeben, meint App in dieser Arbeit immer eine native App

Design eines intuitiven GUIs und die Strukturierung und damit die Ordnung von Informationen in den Vordergrund.
Letztendlich liegt der vordergründige Unterschied zwischen einer nativen App und einer Webapp bzw. Webseite im Zweck und damit auch in der Erwartung des Nutzers: die native App sollte den Nutzer immer etwas tun lassen – in dem Mindestmaß, dass sie nicht mehr durch eine Webapp ausgetauscht werden kann. [Dev12] [Dev11]
Dies gewinnt hinsichtlich der Entwicklung für iOS an Bedeutung, da Apple bei der Kontrolle von Apps, die für den App Store bestimmt sind, auf diesen - kleinen aber feinen - Unterschied genau achtet.

Diesbezüglich ist der Titel der Arbeit auch etwas irreführend. Es wird, genau genommen, keine Web Applikation entwickelt, auch wenn Browsertechnologien wie HTML und CSS zum Einsatz kommen, sondern in der Tat eine native App.

3.1.4 Display und View Stack

Das Display ist das wichtigste Element eines Smartphones. Es dient sowohl der Darstellung des Inhalts, als auch - als Touchscreen - der Interaktion des Nutzers mit der Anwendung. Das Display kann sowohl horizontal (Landscape Modus) als auch vertikal (Portrait Modus) ausgerichtet werden; die Ausrichtung der Darstellung des App-Inhalts ändert sich dementsprechend - wenn diese Funktionalität nicht Software seitig deaktiviert ist. Die Abmaße des Bildschirms betragen 3,5 Zoll Bildschirmdiagonale beim iPhone ([App12]). Das betrifft die drei Modelle, die noch ausgeliefert werden (3GS, 4, 4S). Bei Android kann kein konkreter Wert festgehalten werden, da die Gerätelandschaft viel größer und mehr Änderungen unterworfen ist. Die Bildschirmdiagonale liegt aber in einem ähnlichen Bereich wie bei den genannten iPhones. Ein ungefährer Eindruck dazu kann anhand der Abbildungen Abbildung 8 und Abbildung 7 gewonnen werden. Größere Abmessungen, z.B. die eines iPad, müssen nicht berücksichtigt werden. Eine genaue Auflistung von Bildschirmwerten nach Gerät kann unter [Wik12] abgerufen werden. Die Bildschirmgröße ist aus folgen Gründen von Bedeutung: Im Gegensatz zu Desktop Computern oder Notebooks sind Smartphones klein und der darstellbare Bereich ist begrenzt. Es ist immer nur eine App gleichzeitig auf und innerhalb der nur eine Sicht (View), also nur eine Liste, nur eine Webview, nur eine Karte, etc.

Abbildung 8: iPhone (Quelle: Screenshot vom XCode iPhone SImulator)

Abbildung 7: Android Gerät ohne Hardware Navigationsknöpfe (Quelle: [DevNA], UI Overview)

Es ist nicht nötig und auch nicht sinnvoll, das Aussehen und Verhalten von iOS Apps auf Android Geräten zu simulieren oder umgekehrt. Es ist durchaus legitim, dass bspw. Tab Bar Widgets, die sich in iOS Anwendungen für gewöhnlich am unteren Ende des Bildschirms befinden, auf Android Apps hingegen am oberen Ende, auch entsprechend der systemspezifischen Richtlinien umgesetzt werden.

Während der Nutzer durch das UI navigiert, landen neue Views auf einem Navigation Stack (push), der den Verlauf der Navigation oder den Pfad, den der Nutzer entlang navigiert ist, beinhaltet. Der Stack kann rückwärts über einen Back-Button wieder abgearbeitet werden. Dabei erscheint die jeweils vorherige View vom Stack auf dem Bildschirm (pop) bis root erreicht ist. Nach root geht es nicht mehr weiter zurück. [Eri12]
Das Android OS nutzt für seine Activities ebenfalls einen LIFO Stack.

3.1.5 Nutzerinteraktion

Die Interaktion mit dem GUI erfolgt über den Touchscreen. Der Nutzer der App wird bereits mit Apps in Berührung gekommen sein und intuitiv eine Erwartung haben, welche Geste welche Aktion in der App auslöst. Touchscreen Gesten müssen sich an den allgemein gültigen Konventionen anpassen: einmaliges Antippen entspricht einem einzelnen Mouse-Klick, Wischen entspricht Scrollen, etc. Eine Übersicht über mögliche Gesten kann Tabelle 2 (übernommen von *Table 1-1* der iOS Human Interface Guidelines (

[Dev12])) entnommen werden. Die App sollte mindestens Tap, Drag und Flick (letzteres beides in der Darstellung des Speisemenüs) unterstützen.

Geste	Aktion
Tap (Antippen)	Ein Kontrollelement auswählen oder drücken (entspr. einfachem Mouseklick)
Drag (Ziehen)	Scrollen oder pan (seitliches scrollen) oder bewegen eines Elements
Flick (Schnipsen)	Schnelles scroll oder pan
Swipe (Wischen)	1 Finger: Delete-Button 2 Finger: Zwischen Apps wechseln
Double Tab (zweimaliges Antippen)	Reinzoomen und zentrieren
Pinch (Kneifen, Zwicken)	Rein- und Rauszoomen
Touch and hold (anhaltende Berührung)	Vergrößerung in editier- oder selektierbarem Text
Shake (Schütteln, Rütteln)	Undo, rückgängig machen

Tabelle 2: iPhone Gesten

iPhones (und auch das iPod Touch) haben nur einen Hardware-Button, den Home-Button am unteren Ende des Gehäuserahmens. Der Home-Button ist ein Multifunktionsbutton. Je nach Art und Dauer der Betätigung übernimmt er eine andere Funktion. Back-Buttons und andere Navigationselemente sind in iOS seit jeher Teil des Betriebssystems und werden je nach Anwendung und Implementierung als Teil der Action Bar, in einer Zelle einer Tabelle, etc. eingeblendet.

Bei Android Geräten ist zwischen zwei Ausführungen zu unterscheiden. Android OS Versionen ab 3.0 haben eine Software Actionbar am oberen Ende des Bildschirms. Die Actionbar beinhaltet wichtige Nutzerinteraktionen als Menüpunkte und weniger wichtige als Dropdown Menü. Ältere Geräte mit Pre 3.0 Android OS Versionen haben diese Actionbar nicht. Dort wird das Dropdown Menü über einen Hardware-Button am unteren Ende des Telefons erreicht. Neben dem Menü-Button gibt es noch drei standardisierte Buttons (Back, Home, Recent). Auf älteren Geräten befinden sich diese, als Teil der Hardware, am unteren Ende des Gehäuses, auf neuen Geräten sind sie Teil der Software. So ist ab Android 4.0 kein Hardware-Button mehr nötig. Back, Home und Recent sind als Teil der Navigation Bar integraler Bestandteil des Betriebssystems und ersetzen so komplett die vormalige Hardwarelösung (Abbildung 9).

Abbildung 9: Navigation Bar mit Back-, Home- und Recents-Button ([DevNA])

So ruft bspw. der (sich links befindende) Back-Button immer die Methode *onBackPressed()* auf, unabhängig ob als Hard- oder Software-Button. Diese Methode lässt sich dann, wie jede nicht *final* Methode in Java, überschreiben.

Die Android Version der App muss auf mindestens Version 2.3.3 des BS laufen. Das ist die Version, die auf fast zwei Drittel aller Android Geräte vorhanden ist (vgl. Abbildung 3). Da aber neue Geräte nicht ausgeschlossen werden dürfen, muss sie rückwärtskompatibel sein aus Sicht von Android 4.x.

3.1.6 Betriebsdauer und Wartung

Die App wird nicht lange in Betrieb sein. Deshalb muss sie schnell starten und schnell schließen. Von Datenabgleich, -update und -speicherung darf der Nutzer nichts mitbekommen. Gegenüber häufigem Gebrauch muss sie robust sein.

Im Sinne einer möglichst angenehmen Nutzererfahrung sollten Änderungen am darstellenden Client (an der App) gering gehalten oder, wenn möglich. komplett vermieden werden. Ausnahmen bilden Bug Fixes und größere Updates, die neue Features liefern. Der sich ändernde Teil der Anwendung, d.h. hauptsächlich der Datenbestand, sollte komplett serverseitig gehalten werden und für den Nutzer sollte der Prozess der Änderung verborgen bleiben.

3.1.7 Design Pattern

iOS Entwicklung folgt dem Model-View-Controller Design Pattern. Das iOS SDK unterstützt das MVC Architekturmuster voll; durch beispielsweise den separaten Interface Builder, der die Erstellung von GUIs und die Verlinkung der GUI Elemente mit Methoden im Code ermöglicht. MVC bedeutet, dass das Datenmodell (M), die Sicht (V) und die Steuerung (C) voneinander unabhängige Einheiten sind. Unabhängig heißt, dass jede Einheit austauschbar ist und dabei keine der beiden anderen Einheiten beeinflusst wird. Es heißt aber auch, dass ein UI Element, wie ein Schalter, für sich genommen nur ein Schalter ist – ohne direkte Funktion. Die Funktion stellt der Controller her. MVC auf dem iOS ist wie folgt umgesetzt:

- Model Methoden stellen über Protokolle Daten bereit
- View Elemente erben von der *UIView* Klasse
- Zu jeder View gehört ein *UIViewController* oder eines seiner Kinder. Die Controller Einheit des iOS MVC Paradigmas kann ihrerseits wieder über verschiedene Konzepte umgesetzt werden:
    - *Delegation*: Weitergabe (Delegation) jedweder Nachricht oder Information einer Nutzerinteraktion an eine Controller Klasse
    - *Target Action*: es ist vorher festgelegt, welche Information an welches Zielobjekt gesendet wird, wenn ein UI Element betätigt wird.
    - *Notification*: Eine Nachricht, die über ein Notification Center an ein oder mehrere lauschende Objekte (Observer) gesendet wird, die individuell auf die Nachricht reagieren können.

([Dev121])

Das Android OS und das Android SDK unterstützen kein Design Pattern, so wie es iOS strikt mit MVC tut. Das zentrale Element einer Android Application ist die *Activity*. Mit Activities wird sowohl die Darstellung des UIs, als auch die Behandlung von Ereignissen in einer Klasse realisiert. Damit übernehmen Activities, im Vergleich mit der iOS MVC Architektur, sowohl Aufgaben des Controller als auch der View. Ausführliches zur *Activity.class* findet sich unter [Dev122].

Beide Systeme werden darüber hinaus objektorientiert programmiert.

3.1.8 Sichten

Die App benötigt mindestens folgende Sichten, die der Nutzer auf den Inhalt hat:

- Darstellung des eigentlichen Zwecks der App: eine Speisekarte
- Informationen die den Auftraggeber, das Restaurant Maestro, betreffen
- Informationen in Form eines Impressums zu Hersteller, Kontaktdaten, Build Version, etc

Kernstück der App ist die Sicht auf die Speisekarte.
Da es der Nutzer intuitiv erwartet, sollten gewisse Informationen interaktiv sein, z.B. Telefonnummern (Callback der Dialer API bei Klick), Links zu Webseiten (Aufruf des internen Browsers), E-Mail Adressen (Aufruf des E-Mail Clients). Außerdem sollte die Menü Darstellung filterbar sein.

3.1.9 Skalierbarkeit

Software ist laufenden Änderungen unterworfen. Sie muss in der Lage sein, auf neue Anforderungen, die Änderungen mit sich bringen, zu reagieren oder darf Anpassungen am Code oder am System durch einen schlechten Entwurf nicht erschweren. Änderungen können äußerer Herkunft sein, z.B. ein Anstieg der Nutzerzahl, oder am System selbst auftreten. Letzteres könnten Änderungen an den Daten der App sein (neues Speise, Anpassung des Preises bestehender Speisen, etc.) oder das Hinzufügen neuer Funktionen am darstellenden Client der Nutzer. Wie eingangs erwähnt, ist das Android OS einer steten und häufigen Weiterentwicklung und damit Änderungen unterworfen und mit einer heterogenen und teilweise unübersichtlichen Gerätelandschaft ausgestattet; aber auch Apple entwickelt sein iOS und iPhone ständig weiter.

Hier ist es sinnvoll und wichtig, von vornherein und unter besonderer Berücksichtigung von iOS' MVC Paradigma, die Daten - das schließt Datenerfassung, -pflege und -aktualisierung mit ein - strikt von der Client-seitigen Darstellung zu trennen. Änderungen am Client dürfen die Daten nicht beeinflussen und umgekehrt. Das stärkt einerseits die Robustheit der App gegen solche Änderungen und erhöht die Portierbarkeit – nicht nur die

Portabilität auf weitere Smartphone Systeme als iOS und Android OS, sondern allgemein auf weitere Systeme.
Andererseits ermöglicht die strikte Trennung von Datenpflege und Datendarstellung Wiederverwendbarkeit. Die App dieser Arbeit ist speziell für das Restaurant Maestro entwickelt. Unter Austausch des Datensatzes ließe sich - ohne größeren Mehraufwand - das Angebot irgendeiner anderen Gastronomie darstellen.
Portabilität und Wiederverwendbarkeit der Daten benötigen jedoch noch eine konkrete Schnittstellendefinition.

Zusammengefasst ergeben sich daraus zwei Dimensionen der Skalierbarkeit:
Horizontal, d.h.

- es können Daten auf der Speisekarte verändert, welche von ihr entfernt und auch Daten zur Karte hinzugefügt werden.
- es kann in Zukunft für weitere Zielsysteme entwickelt werden, ohne dass sich jedes Mal neu um Herkunft und Pflege der Daten gekümmert werden muss.

Vertikal: mehr Speisekarten, d.h.

- es können besondere Angebote, z.B. Tageskarten oder Damenkarten (Karten ohne Preisangaben), dynamisch dargestellt werden.
- es kann der ganze Datensatz ausgetauscht werden, also das Angebot eines gänzlich anderen Restaurants dargestellt werden (vorausgesetzt es existiert eine einheitliche Schnittstelle)

Die App besteht in dieser Phase der Analyse aus zwei unabhängigen abgekapselten Systemen: der Datenhaushaltung und der Datendarstellung. Abbildung 10 illustriert diese räumliche Trennung graphisch. Datenhaushaltung in Form bspw. einer Datenbank links und die Darstellung auf einem Endgerät rechts.

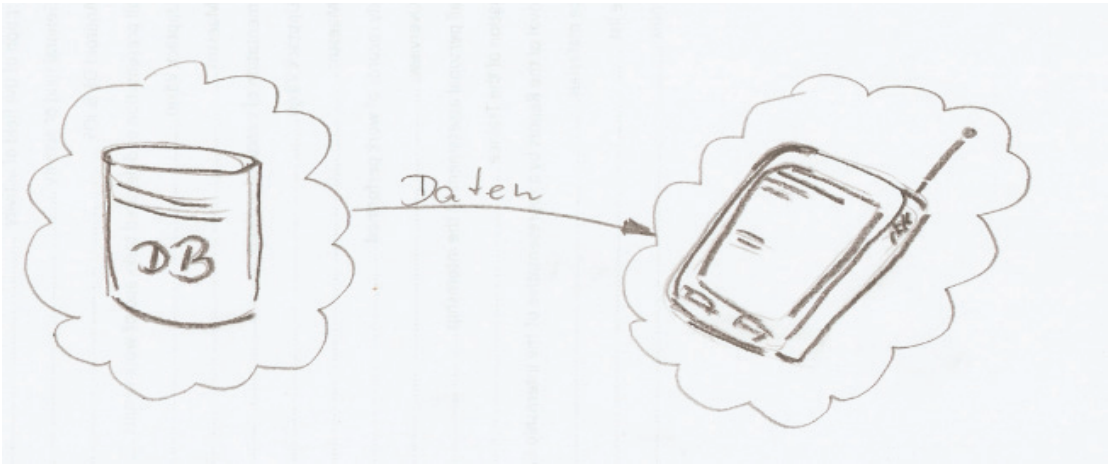

Abbildung 10: Verteilung von Datenhaltung und –darstellung

Als abschließender Gedanke stellt sich an der Stelle die Frage, welche Systemvoraussetzungen geschaffen werden müssen, damit die App nicht nur im Gastronomiesektor wiederverwendbar ist, sondern auch in anderen Branchen – also ob sie, um das Bild der Dimensionalität weiter zu denken, in die Tiefe skalierbar ist?

3.1.10 Möglichkeiten der Erweiterung

Bisher wurde gezeigt, wie das System aufgebaut sein muss, damit Informationen, die das Restaurant betreffen, vom Restaurant zum Kunden gehen. Präziser: ein Bündel von Information von einem Restaurant zu beliebig vielen Kunden. Die Informationen selbst sind dabei wertungsfrei. Ihre Aus- oder Bewertung ist dem Kunden überlassen. Er wird allenfalls durch Features der App (Suchfilter, Favoriten) dabei unterstützt (vgl. 3.1.3).
Doch was ist mit dem Rückweg und welche Informationen könnten vom Kunden oder von den Kunden zum Restaurant fließen? Der nächste logische Schritt, der allerdings im Rahmen der Arbeit nicht umsetzbar war, ist die Antwort des Kunden auf die Informationen, die ihm das Restaurant sendet. Informationen, die vom Kunden zum Restaurant fließen, könnten sein:

- Bestellung
- Bezahlung
- Bewertung

Bestellung wäre die erste und natürliche Handlung des Kunden (die Alternative ist, er beendet die App). Das könnte so aussehen, dass er ein oder mehrere Produkte (Speisen, Getränke) auswählt und über einen Bestätigungs-Button die Bestellung absendet. Das Restaurant empfängt die Bestellung und setzt sie um. Die Umsetzung kann dann ein Lieferdienst sein oder das Aufbewahren der Bestellung, damit der Kunde es sich abholen kann.

Auf die Bestellung folgt die Bezahlung. Hier wären zwei Fälle zu unterscheiden: Bezahlen mit der Bestellung oder nach Erhalt der Ware. Das kann je nach Unternehmen unterschiedlich sein. Eine online bestellte Lieferpizza bspw. wird für gewöhnlich, auch bei Barzahlung, erst bei Erhalt der Ware bezahlt; wohin gegen das Risiko für das Unternehmen bedeutend größer ist, wenn ein anonymer Kunde sein online bestelltes Essen auch im Restaurant zu sich nehmen oder abholen und auch erst vor Ort zahlen möchte.
Mobile- oder Handypayment ist bereits möglich und wird auch genutzt und ist für den Kunden unkompliziert. Der Kunde fordert mit seiner Mobilfunknummer einen eindeutigen Code an und gibt beides in ein Online-Formular ein. Ein Server seitens des Unternehmers überprüft beide Daten und schaltet den Zugang oder die Bestellung frei oder blockiert sie.
Für beides, Bestellung und Bezahlung, müsste die Seite der Datenhaltung nicht nur in der Lage sein, Informationen bereit zu stellen, sondern auch welche zu empfangen. Hier bliebe zu diskutieren, ob die Daten aller beteiligten Gastronomien auf einem zentralen Server liegen und auch dort verwaltet werden oder ob jedes Restaurant (bzw. allgemein jedes Unternehmen, sollte das System über die Grenzen der Gastronomie erweitert und verallgemeinert werden) seinen eigenen Server betreibt. Dies ist sicherlich auch eine Kostenfrage bei der auch beide Möglichkeiten parallel denkbar sind.

Als drittes sei die Möglichkeit der Bewertung des Angebots durch den Kunden diskutiert. Dieses Feature ist u.a. von Online-Shops bekannt, bei denen es

den Kunden möglich ist, auf einer Skala von 1 bis 5, symbolisiert durch 5 Sterne, oder 1 bis 10 (können auch 5 Sterne mit halber Füllung sein) einzelne Angebote selbst zu bewerten und statistisch aufgearbeitete Informationen aus der Summe aller Bewertungen zu gewinnen. Beispielhaft ist solch ein Favoriten Feature in der iPhone Version der App bereits umgesetzt (siehe Abbildung 37 und Abbildung 38). Der Nutzer hat die Möglichkeit, sein favorisiertes Essen zu markieren (drücken des Sterns in der Tabelle) und alle seine Favoriten in einer eigenen Sicht zu betrachten. Da es in der vorliegenden Version der App (noch) nicht möglich ist, auch online eine Rückmeldung an das Restaurant zu senden, ist es auch hinreichend, die Auswahlmöglichkeit auf Favorit und nicht Favorit, d.h. auf einen entweder leeren oder vollen Stern, zu beschränken. Von hier an ist es aber leicht vorstellbar, dass die Favoriten nicht nur eine lokale Bewertung sind, sondern dass die Bewertung der Durchschnitt der Bewertungen aller Nutzer ist, die sich beteilig haben. Dieser Durchschnitt würde serverseitig ermittelt werden und wäre Teil der Information, die dem Nutzer übermittelt wird. Der Nutzer würde in dieser Realisierung seine Bewertung entweder lokal speichern und zu festgelegten Zeitpunkten (z.B. beim Start der App) mit dem Server abgleichen oder bei jeder Aktualisierung seiner Favoriten dem Server direkt eine Nachricht senden und die lokale Sicht mit der Rückantwort des Servers aktualisieren.

Noch einen Schritt weiter ließe sich aus den Bewertungen aller Speisen eine Bewertung des Restaurants ermitteln oder alternativ das Restaurant selbst Teil der Favoriten und damit direkt bewertbar sein. Unter Verweis auf die in 3.1.9 diskutierte Skalierbarkeit, könnte der Nutzer so die Restaurants selbst, welche die App nutzen, bewerten und vergleichen; oder er könnte ähnliche Speisen restaurantübergreifend miteinander vergleichen.
Voraussetzung für solche Informationen ist eine Infrastruktur, die Kommunikation in beide Richtungen zulässt: von der Datenspeicherung zum Nutzer und umgekehrt (siehe Abbildung 11). Jeder einzelne Nutzer könnte so Informationen von jedem beteiligten Restaurant empfangen und wiederum über jedes einzelne Restaurant Informationen zurücksenden. Da in dieser Architektur die Informationen aller Restaurants zentral gespeichert sind, sind sie untereinander vergleichbar. Die Nutzer stehen so indirekt auch in Beziehung zueinander und könnten sogar über bspw. ebenfalls zentral verwaltete Foren direkt miteinander kommunizieren.

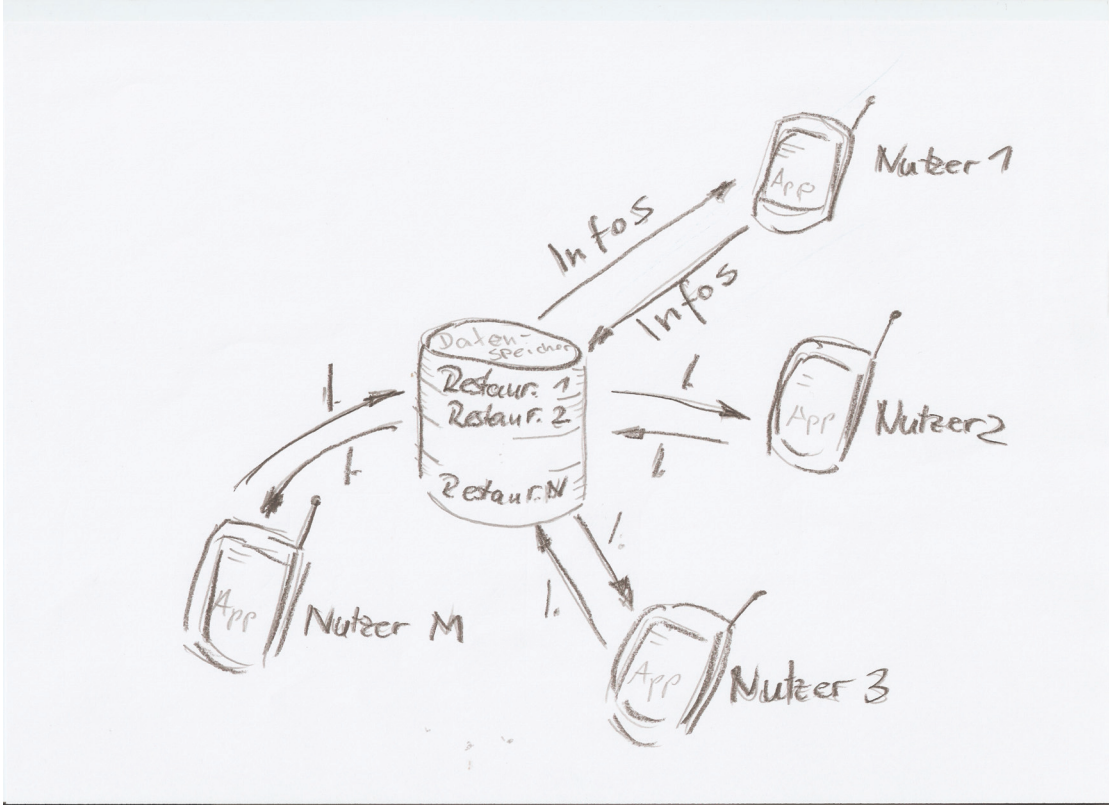

Abbildung 11: System mit Kommunikation in zwei Richtungen

Ein Restaurant, das die Infrastruktur besitzt, Bestellung, Bezahlung und Bewertung (BBB) oder auch nur einen Teil davon mit einer App durchzuführen ist, aus Sicht der internen Geschäftsprozesse des Restaurants, mit Abbildung 12 illustriert. Die Infrastruktur des Restaurants ist in der Lage, die Bestellung, die ein Kunde mit der App durchführt, zu empfangen, sie zuzubereiten und dem Kunden wieder zu übergeben. Das Restaurant ist auch in der Lage, die Bestellung dem Kunden zuzuordnen (durch Codes, Kundenaccounts, etc.). Die Übergabe der Bestellung kann entweder im Restaurant selbst stattfinden oder als Lieferdienst.
Darüber hinaus besitzt das Restaurant entweder die Technik, die Bezahlung der Bestellung direkt mit der App umzusetzen oder alternativ herkömmlich im Restaurant an einer Kasse bzw. an der Wohnungstür im Fall eines Vor-Ort-Lieferservices.
Sollte die App auch die Möglichkeit der Bewertung bieten, werden diese Informationen seitens des Restaurants in der Geschäftsführung oder im Management ausgewertet.

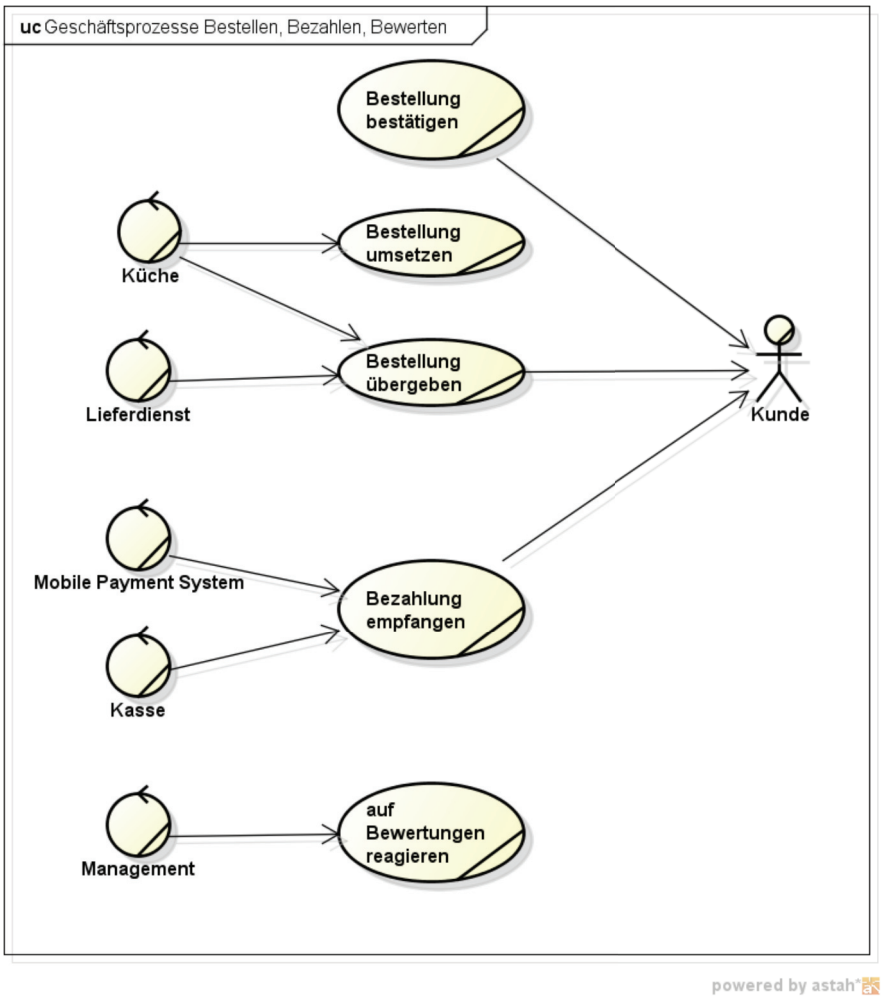

Abbildung 12: Geschäftsprozessdiagramm BBB (erstellt mit astah community)

Alles zusammengefasst lässt sich die App als Modell mit drei Ausbaustufen beschreiben. Jede Ausbaustufe erweitert die vorherige um Meilenstein-gleiche Funktionalität. Mit jeder zusätzlichen Funktion ist auch ein Anstieg der Kosten für den Entwickler und damit auch für den Auftraggeber verbunden (z.B. Bereitstellung der Server zur Datenspeicherung, -auswertung, etc.)

1. Darstellung des spezifischen Angebots eines Unternehmens. Kommunikation in eine Richtung (von der Datenspeicherung zum Endgerät). Das ist in dieser Arbeit umgesetzt.
2. Ein Unternehmen aber Antwort vom Endgerät zur Datenspeicherung möglich (Bestellen, Bezahlen, Bewerten, ...)
3. Verallgemeinerung von 1. und 2. zu einem abgekapselten Modul, das Daten in gewisser Form erwartet, dem es aber egal ist, für welches Unternehmen oder welche Branche die Daten spezifisch sind.

(3.) wäre der Übergang zur Schaffung von Zielsystem-spezifischen wiederverwendbaren Softwaremodulen. Zielsystem-spezifisch bedeutet, dass die Grenzen, die vorher zwischen den Plattformen bestanden, auch bei Modularisierung der App weiterbestehen und für jede anvisierte Zielplattform ein separates Modul entwickelt werden muss.

Dafür braucht es ein möglichst allgemein gehaltenes und gleichzeitig präzise formuliertes Datenschema, da bspw. ein Attribut *Zutaten* schwer auf das Angebotsmenü eines Bekleidungsgeschäfts anwendbar ist, wohingegen ein Attribut *Einzelheiten* viel allgemeiner gehalten und auch branchenübergreifend belegbar ist. Einzelheiten in der Gastronomie können so immer noch Zutaten sein, auf Kleidung bezogen jedoch Stoffart, Farbe, etc. Abbildung 13 zeigt beispielhaft, wie solch ein Schema aussehen könnte, wobei dort noch keine Attribute aufgeführt sind.

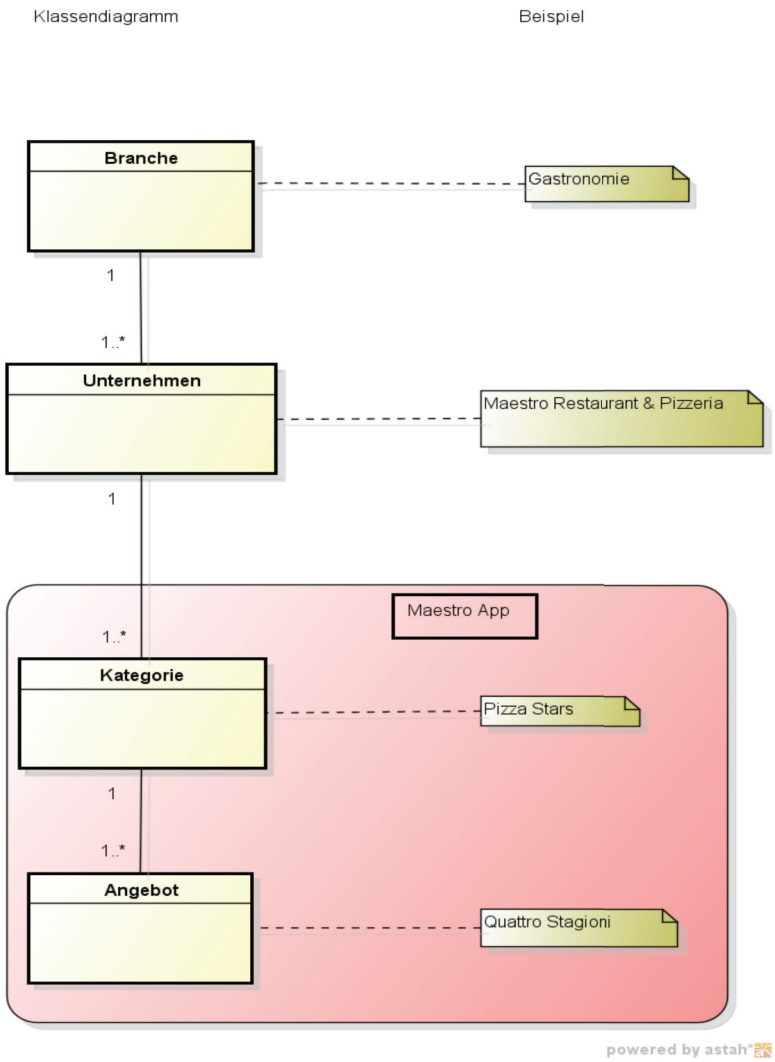

Abbildung 13: UML Klassendiagramm eines Datenschema für vorher nicht spezifizierte Unternehmen (erstellt mit astah community)

Eine Branche hat ein oder mehrere Unternehmen, jedes Unternehmen stellt ein Angebotssortiment zur Verfügung, das aus einem oder mehreren Kategorien besteht und jede Kategorie beinhaltet ein oder mehrere Angebote. Der untere rot gefüllte umrandete Teil des Schemas ist die zuvor, als erste Ausbaustufe diskutierte, Darstellung des Angebots eines spezifischen Unternehmens. Ab hier ist das weitere Verfahren leicht ersichtlich. Möchte man als Entwickler dieselbe App modular für weitere Unternehmen derselben Branche anbieten, schließt man die Klasse Unternehmen mit in sein Schema

ein. Möchte man die App branchenübergreifend anbieten, erweitert man das Schema noch um die Klasse Branche.

Natürlich bleibt auch mit diesem Modell das Problem, dass für jedes Zielsystem (Android OS, iOS, etc.) ein separates Modul entwickelt werden muss, bestehen. Die Inkompatibilitäten zwischen den Betriebssystemen lassen sich so auch nicht aufheben. Die Maestro App befindet sich in den unteren beiden Ebenen der Klassenhierarchie. Das ist im Diagramm auch graphisch nochmal vermerkt.

3.1.11 Vorteile gegenüber einer herkömmlichen Speisekarte

Was bisher noch nicht geklärt wurde, ist die Frage, welche Vorteile denn nun eigentlich eine Speisekarte als App ggü. einer herkömmlichen Speisekarte bietet. Der auffälligste Vorteil ist der in den vorherigen Kapiteln diskutierte: es lassen sich die 3 „B“s, Bestellen, Bezahlen Bewerten, in einem Alles-in-Einem-Produkt vereinen. Setzt man eine entsprechend Smartphone-affine Zielgruppe voraus, ist dies u.U. bereits der Grund, das Restaurant einmal zu besuchen. Dazu kommen, aus Restaurantsicht, die Aspekte der Werbung und Vermarktung über einen App-Store; bzw. aus Kundensicht eine als, umschrieben vielleicht, „zeitgemäß“ empfundene Gesamterfahrung, in der es der Kunde ist, der sein Essen selbst aussucht, der selbst zahlt und der selbst durch seine Bewertung auf die Entwicklung des Restaurants Einfluss nehmen kann. Hinzu kommt, dass die Entscheidung des Kunden bzgl. des Essens außerhalb der Gastronomie stattfindet, was Schlangenbildung (in entsprechenden Restaurants) und so die Wartezeit aller Kunden reduziert.
Ein anderer Aspekt ist der der Dynamik. Die Kunden lassen sich bei Programmänderungen, Sonderangeboten, etc. direkt erreichen, bzw. ist mit einer App solch eine direkte Form der Werbung erst möglich. Ein Nebeneffekt ist auch, dass bei Änderungen an der Speisekarte Druckkosten vermieden werden können, bzw. solch Änderungen finanziell überhaupt erst möglich werden, da keine Karten, Aushänge, Flyer, etc. neu bedruckt werden müssen.

3.2 Funktionale Anforderungen

In diesem Kapitel folgt eine Ausarbeitung der funktionalen Anforderungen an das System. Der Schwerpunkt liegt auf einer App der ersten Ausbaustufe (wie in Kapitel 3.1.10 vorgestellt)

3.2.1 Use Cases

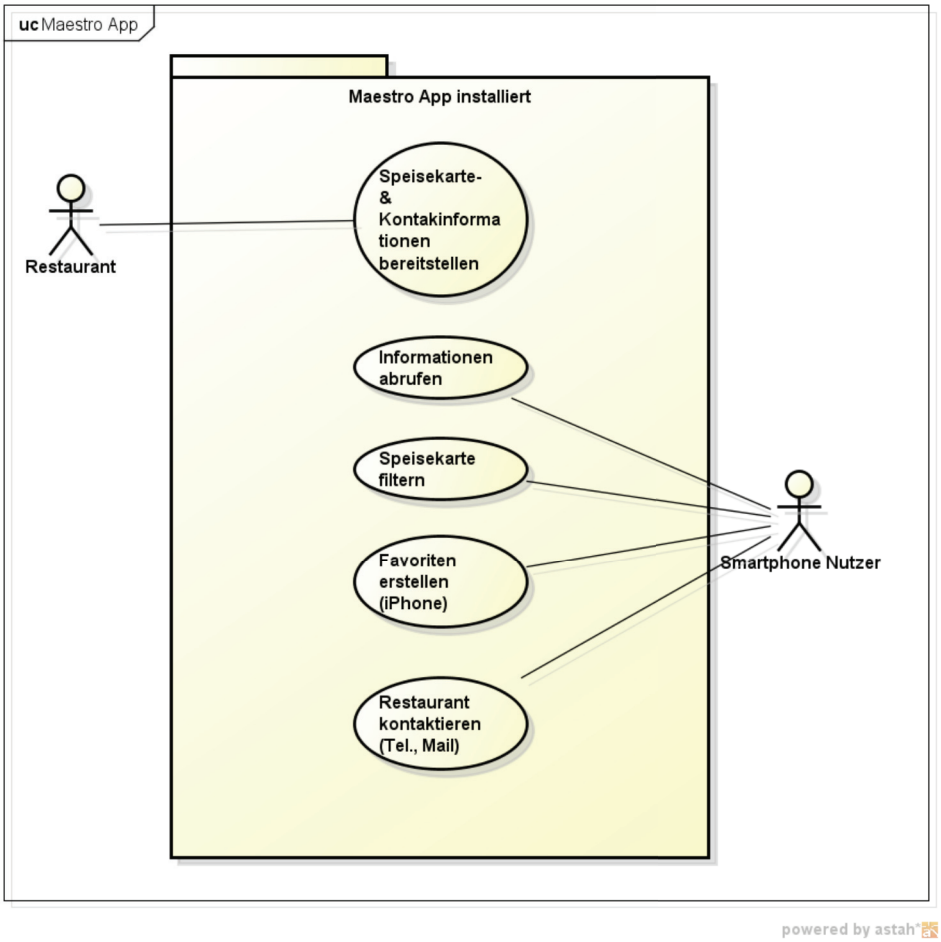

Abbildung 14: Use Cases Maestro App (erstellt mit astah community)

Es existieren zwei Akteure: der Nutzer, der sich die App auf sein Smartphone lädt und das Restaurant, das sich und sein Angebot über die App darstellt. Der Nutzer nutzt die App, um sich zu informieren über das Angebot des Restaurants. Er kann vergleichen, filtern, eine Vorauswahl treffen und ist mobil jederzeit über aktuelle Angebotsänderungen informiert. Außerdem erhält er dem Restaurant und auch dem Hersteller zugehörige Informationen in Form von Kontaktdaten sowie eine Hilfe bei der Wegfindung.
Für das Restaurant ist die App in der vorliegenden Version eine Möglichkeit zur Werbung und zur Präsentation. Erweitert um eine Bestell- und Bewertungsmöglichkeit (vgl. 3.1.10), könnte das Restaurant die App zur Bestellungsaufnahme und zum Gewinn von Feedback nutzen.

3.2.2 Benötigte Daten

Da die App für den Gastronomiesektor erstellt wird, sind die aus Nutzersicht wichtigsten Daten das Speiseangebot des Restaurants betreffend. Der Nutzer erwartet demnach mindestens dieselben Informationen, die er auch beim Betrachten einer Speisekarte erhält: die Zutaten des Essens, Preis, eine Vergleichsmöglichkeit mit anderen Speisen, Extras sowie organisatorische Informationen wie Bestellname, Bestellnummer oder Verfügbarkeit des Angebots (Frühstück, Tageskarte, etc.).
Unter Beachtung dessen, dass der Nutzer sich unterwegs und nicht im Restaurant befinden könnte, denn das ist letztendlich der Zweck der App, benötigt er außerdem Informationen das Restaurant selbst betreffend: Kontaktinformationen (Telefonnummer, E-Mail-Adresse, Adresse), Öffnungszeiten, Sonderangebote, Wegfindung zum Restaurant, etc.
Eine dritte Kategorie sind Informationen die App selbst betreffend: Version der App, Hersteller, Kontaktinformationen über den Hersteller, etc.

Der Umfang dieser Daten ist im vorliegenden Fall vergleichsweise gering. Zum Vergleich, das Angebot des Restaurants Maestro (siehe Link zur Homepage unter Weblinks) ließe sich manuell in weniger als einer Stunde in eine Datenbank eingeben. Auch die Pflege der Daten ist durch den geringen Umfang unkompliziert. Änderungen an den Daten selbst sind nicht sehr häufig, wobei sich am Speiseangebot noch eher etwas ändert (Preisänderungen, neue Gerichte, Tageskarten) als an den Informationen über das Restaurant selbst (es kann erwartet werden, dass Adresse oder Telefonnummer über längeren Zeitraum gleich bleiben). D.h., die Speiseangebotsdaten sollten in einem dynamischen Austauschformat bereit gestellt werden, das es ermöglicht, bei punktuellen Änderungen auf Seiten der Datenhaltung (Server), client-seitig auch nur punktuell diese Daten zu aktualisieren. Dieses Format darf auch nur der Informationsübermittlung dienen und sollte keine Ordnung, Wertung oder Darstellungsform vorgeben. Dieses ist komplett dem Client bzw. dem Nutzer überlassen, wie er die Informationen filtert, vergleicht und auswertet.
Im Gegensatz dazu können die Restaurant- und Herstellerinformationen bereits vorformatiert sein und als statisches HTML Dokument oder gar Bild übermittelt werden. Einerseits werden diese Informationen sehr selten bis gar nicht geändert werden, andererseits beinhalten sie alleinstehende Elemente (Kontaktdaten) und Designvorgaben (Corporate Design), bei denen es nicht sinnvoll oder gar nicht erwünscht ist, dass der Nutzer sie ändert.

Um die Vorteile einer App ggü. einer Webseite auszuspielen (vgl. 3.1.3), ist es wichtig, dass die Sicht auf die Speisedaten manipulierbar ist (Suche, Filter, Favoriten, Kategorien, etc.). Das wiederum setzt voraus, dass das Schema der Daten einheitlich ist, um eine Such- oder Filteranfrage über dem kompletten Datenbestand ausführen zu können. Ein einheitliches Schema wiederum ermöglicht es, die Daten mit vorhandenen Technologien (z.B. in einer relationalen Datenbank) zu speichern, sowohl auf Seiten des Servers, als auch Client seitig, und zum Austausch zwischen Server und Client auf ebensolche strukturierten Austauschformate zurückzugreifen. Das Austauschdokument, das die Datenbeschreibung enthält, lässt sich

clientseitig ebenfalls mit vorhandenen Technologien auslesen (parsen) und dort speichern oder anders weiterverwerten.
Ein dabei zu beachtender Punkt ist, dass die Daten, so wie sie bspw. von der Speisekarte kommen, u.U. nicht diese einheitliche Ordnung aufweisen und ggf. noch umstrukturiert oder komplett neu angeordnet werden müssen, was zu Unterschieden zwischen realer Speisekarte und jener der App führt. Hier ist es besonders wichtig, jede solcher Änderungen mit dem Auftraggeber abzusprechen und ggf. Kompromisse zwischen den Anforderungen an ein DBMS und den Wünschen des Auftraggebers zu finden.

Eine vierte, bisher nicht erwähnte Information, sind Icons (App Icon, Tab- und Button Icons) und das Splashscreen-Bild. Dies sind .PNG Dateien, die je nach Zielsystem, eigenen Design Richtlinien (vgl. 3.1.2) unterliegen und als fester Bestandteil der App lokal gespeichert sind und nur über eine neue Build Version geändert werden können.

3.2.3 Datenabgleich

In einer räumlich getrennten Client Server Architektur ist eine zentrale Frage, wie die Daten vom Server zum Client gelangen und unter welchen Sicherheitskriterien der Datentransfer erfolgen soll. Die Daten der Maestro App bestehen aus Angebotsinformationen und Designspezifikationen. Ein Ausfall oder Fehler bei der Datenübertragung ist nicht kritisch. Eine nicht sofort stattfindende Aktualisierung des Datenbestandes des Clients bei Änderungen am Server liegt im Bereich des Akzeptablen. Es muss auch das Medium Smartphone bedacht werden und die Möglichkeit mit in Betracht gezogen werden, dass nicht immer und überall Online Empfang möglich ist, bzw. dieser auch jederzeit unterbrochen werden kann. Kommunikation erfolgt über W-LAN bzw. WiFi und das Nutzerverhalten sieht, wie vorher bereits erwähnt, für gewöhnlich so aus, dass die App nur kurz zum Informationsgewinn genutzt und dann wieder beendet wird. Es ist somit hinreichend wenn der Datenbestand nur bei jedem Start abgeglichen und ggf. aktualisiert wird. Dennoch sollte die Transaktion der Daten korrekt ablaufen und bei Scheitern, aus welchem Grund auch immer, muss der lokale Datenbestand in den Zustand vor Beginn der Transaktion zurückkehren. Aus diesem Grund ist es auch wichtig, dass beim ersten Start der App bereits ein Basisdatenbestand vorliegt der mit der App ausgeliefert wird.

Die Daten der App bleiben solange gespeichert, bis der Nutzer die App von seinem Telefon löscht (in dem Fall sorgt das jeweilige BS für die Entfernung aller zugehöriger Daten) oder er die Daten über die Settings des Telefons löscht (in dem Fall wird die App beim nächsten Start die Daten wieder herstellen, d.h. sie werden erneut heruntergeladen) oder sich der Datenbestand Server-seitig ändert (in dem Fall werden, wie oben beschrieben, die Client Daten denen des Servers angepasst).

Einen Sonderfall stellen die gespeicherten Favoriten in der iPhone-Version der App dar. Diese Informationen können nicht als Teil der Daten, die vom

Server kommen - bspw. als zusätzliches boolesches Attribut in einem ER-Modell, dass die Daten beschreibt - abgespeichert werden, sondern müssen anderweitig vorgehalten werden. Der Grund dafür ist, eine Änderung oder Entfernung der Daten, hätte auch eine Entfernung der Favoriten zur Folge, was nicht gewünscht ist. Bei Entfernung der App werden die Favoriten allerdings auch mit entfernt.

3.2.4 Schnittstellen

Eine Änderung der Daten durch den Nutzer ist nicht vorgesehen, obwohl er in der iPhone-Version der App mit den Favoriten die Möglichkeit hat, seinen eigenen Datenbestand auf Basis der Serverdaten aufzubauen. Er kann diesem neue Elemente, die allerdings Teil der regulären Daten sein müssen, hinzufügen und auch wieder Elemente aus diesem Favoriten-Datenbestand entfernen.

Die Schnittstellen zwischen Nutzer und App sind der Touchscreen und die Geräte-spezifischen Hardware-Buttons (bzw. im Fall von Android 4.x Geräten sind diese auch als Software Lösung implementiert). Über den Touchscreen interagiert der Nutzer mit den Sichten der App. Die Interaktion erfolgt mit den Fingern der Hand über Gesten (vgl. Tabelle 2). Die Interpretation der Gesten und ihre Umsetzung in sichtbare Animationen und Effekte sind dem jeweiligen Betriebssystem überlassen. Die konkrete Umsetzung des GUIs jedoch in Gestaltung und Verhalten sind dem Entwickler überlassen. Der Entwickler definiert, von welchen Klassen die View (Sicht) und ihre Elemente erben und überschreibt ggf. entsprechende Listener und Klassen- (oder Superklassen-) Funktionen, er legt Animationen mit denen Elemente erscheinen oder verschwinden fest, etc.. So bietet es sich bspw. bei der Maestro App an, die Sicht auf das Speisemenü als geordnete Liste zu implementieren, das Selektieren der Listenelemente (also der einzelnen Speisen) allerdings zu deaktivieren bzw. von Beginn nicht zu implementieren, da es in der App nicht vorgesehen ist, dass der Nutzer typische Interaktionen (z.B. Wechseln in eine Detailansicht, Neuanordnen oder Löschen der Einträge, etc.) mit den Listenelementen durchführt. Als Analogie, die Darstellung der Daten der App dient in gleicher Weise der alleinigen Information wie eine Speisekarte im Restaurant und ist nicht offen für Veränderung oder Manipulation wie es bspw. ein Rezeptbuch ist. Was der Nutzer jedoch zur Verfügung gestellt bekommt - und was die App damit auch von der Speisekarte im Restaurant unterscheidet - sind Werkzeuge zur Filterung der Daten.

Gefiltert werden kann nach mehreren Stichworten getrennt durch gewisse Trennzeichen. Der Filter betrachtet die Stichworte als eine Oder-verknüpfte logische Aussage. Es wären auch andere logische Operatoren möglich (z.B. Suche nach „Salami und kein Käse“), das ist bei dem Umfang der App aber nicht zwingend nötig, da ein einzelnes Filterwort die Suche bereits auf eine überschaubare Anzahl begrenzt. Die Suche ist „Case Insensitive“.

Interaktion mit Telefonnummern, E-Mail Adressen, Navigations-Links oder Web Adressen veranlassen die App über das jeweilige BS entsprechende System Applications - andere installierte Apps - zu rufen. Dies ist bei iOS und Android OS gleich, wenn auch jeweils anders implementiert. So werden, der Reihe nach, die internen Dialer, Mail Application, Google Maps App oder Browser gestartet. (vgl. Intents in Android: [Dev124], URL Links in iOS: [Dev123])

Eine dritte sehr wichtige Schnittstelle ist die zwischen der App selbst und dem Ort, an dem die Daten bereitgestellt werden. Datenhaltung und Datenpräsentation (die App) sind ein verteiltes System. Als räumlich von diesem Ort getrenntes und in sich selbst abgeschlossenes Subsystem (Android / iOS Endgerät + Software Application) muss die App in der Lage sein, sich mit der Datenbereitstellung zu verbinden, zu entscheiden, ob und welche Daten übertragen werden müssen, ob diese Daten noch transformiert werden müssen und wie diese Daten auf dem Endgerät persistent gespeichert werden sollen. Die Transaktion der Daten sollte unter Einhaltung der ACID Bedingungen erfolgen.

3.2.5 Sprache

Die App muss zweisprachig sein. Die Standardsprache ist Schwedisch, die Zweitsprache ist Englisch. Grund dafür ist, neben dem geringen Mehraufwand, den das aus Entwicklersicht bedeutet, die internationale Anforderung und die besondere Situation, dass die Kundschaft von Maestro, die hauptsächlich aus den Studenten und Lehrkräften der ansässigen englischsprachigen Universität besteht, wenn nicht Schwedisch, dann Englisch spricht.

4 Entwurf

Für den Entwurf müssen zwei Ansätze der Präsentation des Angebots strikt unterschieden werden: die Präsentation in einer Speisekarte, die an kein Schema gebunden ist und einzelne Angebote durch bspw. einen Wechsel des Stils hervorheben kann und die Präsentation in einer App, die ihre Daten aus bspw. einer Datenbank bezieht und jedes einzelne Angebot nach demselben Schema darstellt. Für den Nutzer oder Kunden ist die Information aber immer dieselbe, egal ob er sie aus der Speisekarte oder einer App gewinnt: das Speiseangebot des Restaurants.
In diesem Kapitel wird geklärt, welche Daten in welchem Format überhaupt benötigt werden, es wird ein ER-Modell zu diesen Daten entworfen und es wird ein System beschrieben, das die Daten, beginnend an einer Quelle, bereitstellt und durch die App abruft, mit dem Datenbestand der App abgleicht, und Unterschiede herunterlädt. Außerdem wird für die App ein GUI entworfen, in dem der Nutzer letztendlich die Daten präsentiert bekommt und über das er Interaktionsmöglichkeiten mit den Daten hat.

4.1 Transformation der Daten von der Speisekarte in ein Austauschformat

Die Speisekarte des Restaurants kann als Grundlage des Entwurfs genommen werden; aus ihr ist ersichtlich, welche Daten überhaupt benötigt werden. Es kann angenommen werden, dass die Speisekarte vollständig ist. Bevor die App mit diesen Daten arbeiten kann, müssen die Daten in ein einheitliches Modell überführt (4.5) und in einem noch zu vereinbarenden Austauschformat bereitgestellt werden.
Das System, das die Transformation der Daten von der Speisekarte in ein Austauschformat vornimmt, kann als Blackbox betrachtet werden. Für das Thema der Arbeit ist es nicht wichtig, wie dieses System implementiert ist und wie die Daten zur Quelle gelangt sind. Es muss sich nicht einmal um ein aufgesetztes System handeln; die Datentransformation kann auch manuell stattfinden. Es wird somit einfach angenommen, dass es ein abgekapseltes System gibt, das die Daten von der Speisekarte in IT-verarbeitbare Daten transformiert. Für die hiesige Betrachtung ist die Quelle eine URL, die die Daten bereitstellt.

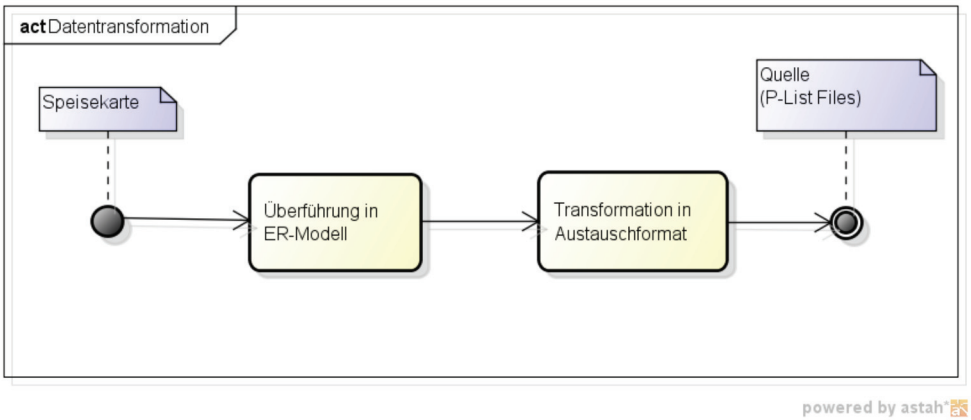

Abbildung 15: Datentransformation (erstellt mit astah community)

4.2 P-List als Datenaustauschformat

An der Quelle stehen Daten im Format P-List bereit. Property List (.plist) ist ein auf XML basierendes Format, das unter Mac OS X und iOS Anwendung findet. Das Format wurde von dem Vorgänger beider BS, NeXTSTEP, übernommen. P-list Dateien speichern hierarchisch organisierte Information und Daten als *dictionary*. Dictionaries sind assoziative Datenfelder, also Arrays mit nichtnumerischen Schlüsseln (keys). P-Lists unterstützen eine begrenzte Anzahl an Datentypen, die sich nach simplen Typen (String, Integer, Float, Date und Bool) und Arrays unterscheiden. Die Arrays können andere Arrays und die simplen Typen enthalten. Die Daten werden als key-value-Paare gespeichert, wobei auf jeden *<key>Bezeichner</key>* Tag ein *<valuetype>Value</valuetype>* Tag folgt. Da es sich um hierarchisch geordnete Daten handelt, existiert außerdem noch ein root Knoten. Ein P-List Eintrag könnte beispielhaft folgendermaßen aussehen:

```
<root>
	<dict>
		<key>Pizza Kategorie</key>
		<array>
			<key>Pizza Name</key>
			<string>Pizza 1 Name</string>
			<key>Pizza Name</key>
			<string>Pizza 2 Name</string>
		</array>
	</dict>
</root>
```

P-Listen können in drei verschieden Formaten gespeichert werden: als XML Repräsentation, als Binärdatei oder in dem alten ASCII Format, das von NeXTSTEP herrührt. Während binäre P-Listen weniger Speicher beanspruchen, sind P-listen im XML Format plattformübergreifend viel portabler und können auch manuell bearbeitet sowie durch einen Menschen gelesen werden. P-List ist in die iOS SDK voll integriert. Da auch Java

assoziative Datenfelder unterstützt und die Android SDK mehrere XML-Parser mitliefert, können P-List Dateien unter Android OS als XML Dateien betrachtet und wie solche geparst werden. (näheres zu Property Lists findet sich unter [Dev10] und den weiterführenden Links)

4.3 Prozesse

Die Daten kommen von einer externen Quelle. Da sie durch einen Server bereitgestellt werden, ist dies eine URL. Das datenverarbeitende System ist die Maestro App. Eine Weiterleitung der Daten findet nicht statt. Die interne App Logik speichert die Daten (lokaler Telefonspeicher, Datenbank), ruft sie ab und präsentiert sie dem Nutzer. So gesehen sind die Ausgangsdaten der App formatierte Informationen, die der Nutzer visuell aufnimmt (Abbildung 16).

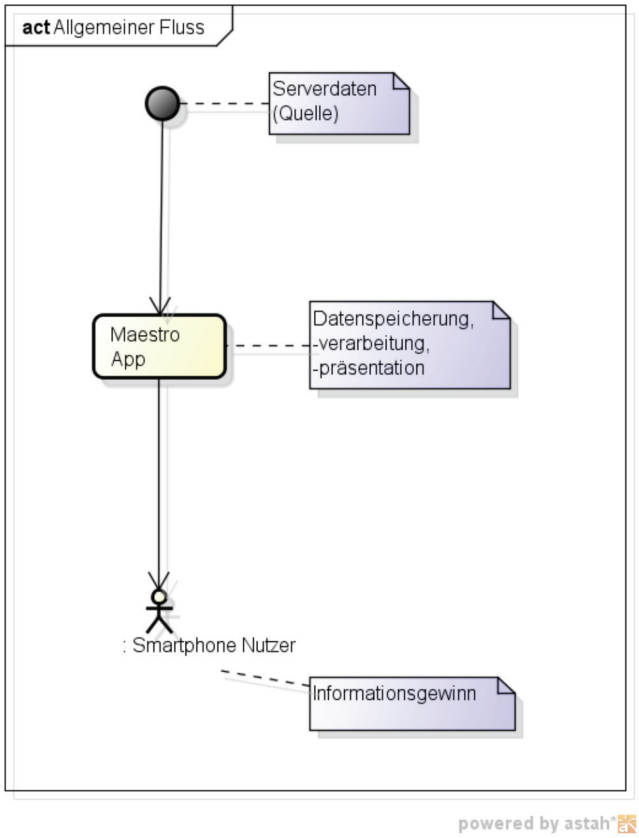

Abbildung 16: Datenfluss (erstellt mit astah community)

4.4 Module der App

In der App selbst gibt es verschiede abgekapselte Einheiten, denen jeweils eine Aufgabe bei der Datenverarbeitung zufällt. Es gibt zum einen ein Framework, das für das Synchronisieren und Aktualisieren des internen Datenbestandes mit dem externen auf dem Server verantwortlich ist sowie

auf derselben Betrachtungsebene wie dieses Framework die App spezifische Logik. In der App existiert wiederum eine Art der Datenspeicherung und Objekte, die diese Daten repräsentieren sowie Methoden, die dafür zuständig sind, die Daten auf diese Objekte abzubilden.
Alles, was die Sicht (View) des Nutzers betrifft ist auch eine funktionelle Einheit. Das beinhaltet den Aufbau des GUI, die Art der Navigation zwischen den Views und die Abbildung der Daten in den jeweiligen Views.
Außerdem gibt es noch die verschiedenen Betriebssystem spezifischen Frameworks (iOS) oder Bibliotheken (Android OS), denen Navigation in der App, Speicher Management, Systemrufe, Graphik und alles weitere, wo auf BS-interne APIs oder andere Apps, die vorinstallierter Bestandteil des Betriebssystems sind, zugegriffen wird, zufällt. Sie sind an dieser Stelle in einem Modul zusammengefasst, da sie die Gemeinsamkeit verbindet, dass sich der Entwickler um diese Funktionalitäten nicht ganz so intensiv kümmern muss.
Ein fünftes, in iOS eher vernachlässigbares, in Android dafür umso wichtigeres Feature, ist die Einbindung externer Libraries, wie z.B. Kompatibilitätsbibliotheken. Da eine solcher Bibliotheken in der Android Version der App Anwendung findet, ist es hier als eigenständiges Modul mit aufgeführt. Alle Module zusammen erzeugen die Nutzersicht auf die App.

Der Nutzer selbst wiederum bekommt von dem internen Zusammenspiel dieser Module, die seine Gesamtsicht auf die Daten formen, nichts mit. (Abbildung 17)

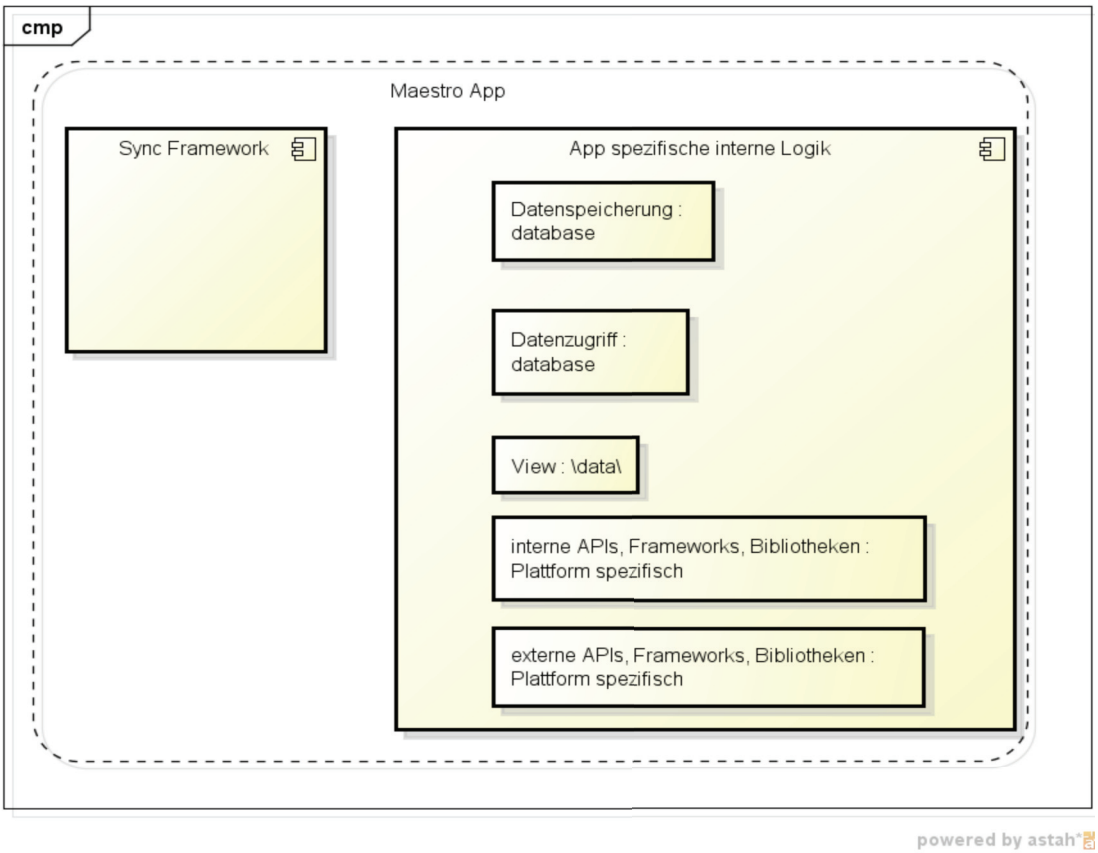

Abbildung 17: Modularisierung der App (erstellt mit astah community)

4.4.1 Datensynchronisation

Das Einstiegsmodul der App bildet das Framework, das für die Aktualität des Datenbestandes verantwortlich ist. Es synchronisiert sich vor dem Start der internen App Logik mit dem externen Datenbestand und es sorgt dafür, dass nach dem App-Start die Daten aktuell sind. Es ist in der Lage, selbstständig zu entscheiden, ob die Daten aktuell sind und aktualisiert dementsprechend. (Abbildung 18)

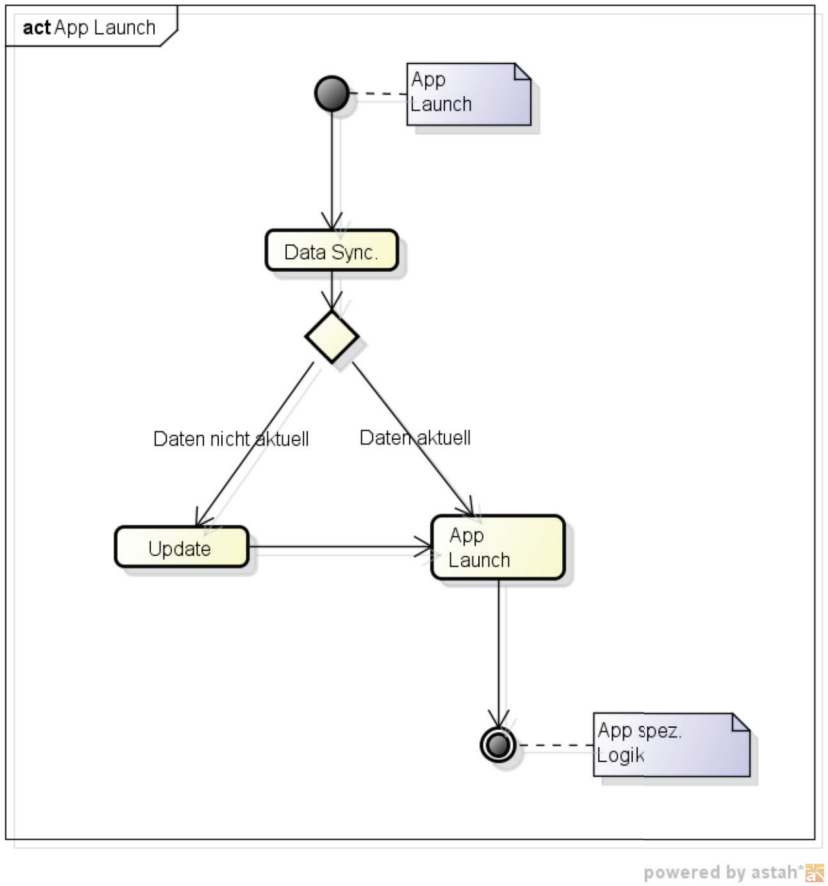

Abbildung 18: Aktivitätsdiagramm "App Launch" (erstellt mit astah community)

4.4.2 App spezifische interne Logik

Zwei Fälle sind zu unterscheiden: Der Nutzer kann zum einen eine komplett neue Sicht erzeugen. Dies ist der Fall, wenn er zentrale GUI Elemente wie Tabs oder auch den Back-Button bedient oder wenn er in einer Sicht Interaktionselemente wie Schalter, Hyperlinks, Detailsichten, etc. bedient und so bspw. von einer Web View in eine geographische Karte wechselt. Zum anderen kann er eine bestehende Sicht verändern durch Angabe einer Bedingung für die dargestellten Daten. Dies geschieht immer dann, wenn bspw. eine Tabelle durch Eingabe eines Stichwortes in ein entsprechend

dafür vorgesehenes Formfeld (*EditBox*) gefiltert wird und auf Basis dieses Filters eine neue Tabelle erzeugt wird. Die neue Tabelle kann auch eine leere Tabelle sein. Der Fall, dass der Nutzer nur in bspw. besagter Tabelle scrollt oder in einer View zoomt o.ä. sei ebenfalls nicht betrachtet, da das entsprechende Scroll- oder Zoomverhalten ebenso dem BS zugerechnet werden kann.

Die App startet mit einem Splashscreen. Der Splashscreen ist eine Graphik mit dem Logo des Restaurants und überbrückt für einen kurzen Moment (ca. 1 sec.) die Arbeit des Synchronisations-Frameworks. Nach dem Splashscreen stellt die App die erste View dar. Welche das ist, ist vordefiniert (Abbildung 19).

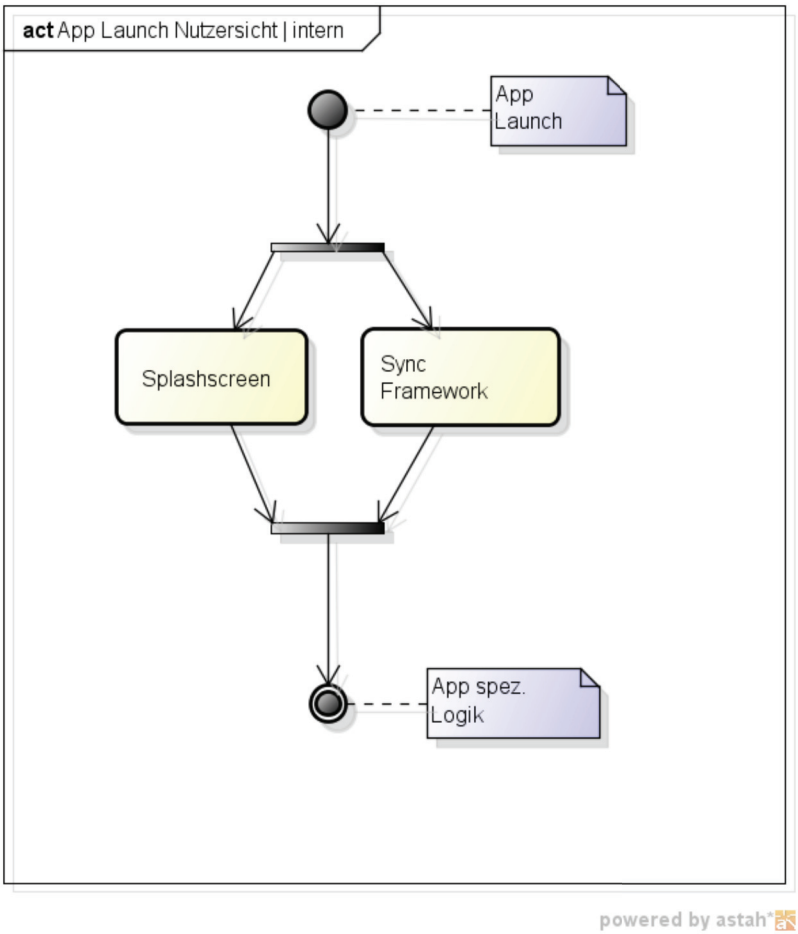

Abbildung 19: App Launch aus Nutzersicht und intern (erstellt mit astah community)

Wenn der Nutzer ein GUI Element bedient, unterscheidet die App, ob es eigener Code der App ist (intern), der das weitere Verhalten beschreibt oder ob eine andere App (extern) dafür gerufen wird. Sowohl Android (*Intents*) als auch iOS (*URL Schemes*) unterstützen die Ausführung anderer Apps aus der aktuellen heraus. So könnte der Nutzer beispielsweise auf eine Telefonnummer drücken. Das ruft die Telefon App in den Vordergrund, die aktuell noch offene App wird der Verwaltung des BS übergeben und der Nutzer kann von jetzt an mit der Telefon App weiterarbeiten. Über den Back-Button könnte er die letzte Sicht auf die vorherige App wieder in den Vordergrund holen.

Anders verhält es sich, wenn mit dem Datenbestand der App gearbeitet wird, also wenn bspw. der Nutzer die Speisen nach einem Stichwort filtert oder in eine Informationssicht über das Unternehmen wechselt. Die Informationssicht des Unternehmens ist als selten veränderliche Sicht von Beginn an nicht filterbar und wird immer als monolithisches Objekt geladen. Eine nicht gefilterte Liste der Speisen ist der komplette Datensatz aller Speisen. Eine gefilterte Speisenliste beinhaltet auch nur die Speisen, für die die Filterbedingung wahr ist. Sie kann demnach auch leer sein.

Aus dem gewonnenen Datensatz wird dann eine View geformt. Im Fall der Informationssicht ist dies natürlich nur der Fall, wenn der Nutzer von einer komplett anderen Sicht (bspw. von der Speisenliste) in eben diese Informationssicht wechselt. Auf der Speisenliste kann dem gegenüber durch Hinzufügen, Löschen und Verändern der Suchfilter mit dem Datensatz gearbeitet werden ohne die Sicht selbst zu verlassen. Die erstellte View ist eine Unterklasse der allgemeinsten View Klasse des jeweiligen BS von der alle Views erben. Im Fall der Speisekarte bietet sich hierfür eine Tabellensicht an (z.B. *UITableView* in iOS), für statische Informationen hingegen eher eine WebView oder ImageView.

Die neu erstellte View ist dann wieder die Ausgangsview für die nächste GUI Interaktion. So kann bspw. die Informationssicht, zu der sich der Nutzer gerade navigiert hat, einen interaktiven Link zur Homepage des Unternehmens das Gegenstand dieser Informationssicht ist, beinhalten. Eine Betätigen des Links würde dann die Browser Application des Smartphones starten und automatisch die Homepage laden. (Abbildung 20)

Drei Sichten als Ergebnis von Nutzerinteraktion mit GUI Elementen seien unterschieden: die Sicht auf eine externe App auf der einen Seite und innerhalb der App zwei Sichten auf den Datenbestand der App, die sich unterscheiden nach dem Kriterium, ob der Nutzer über dem Datenbestand der Sicht arbeiten kann (die Sicht auf das Angebot) oder ob die Sicht starr ist (Informationssicht auf das Unternehmen).

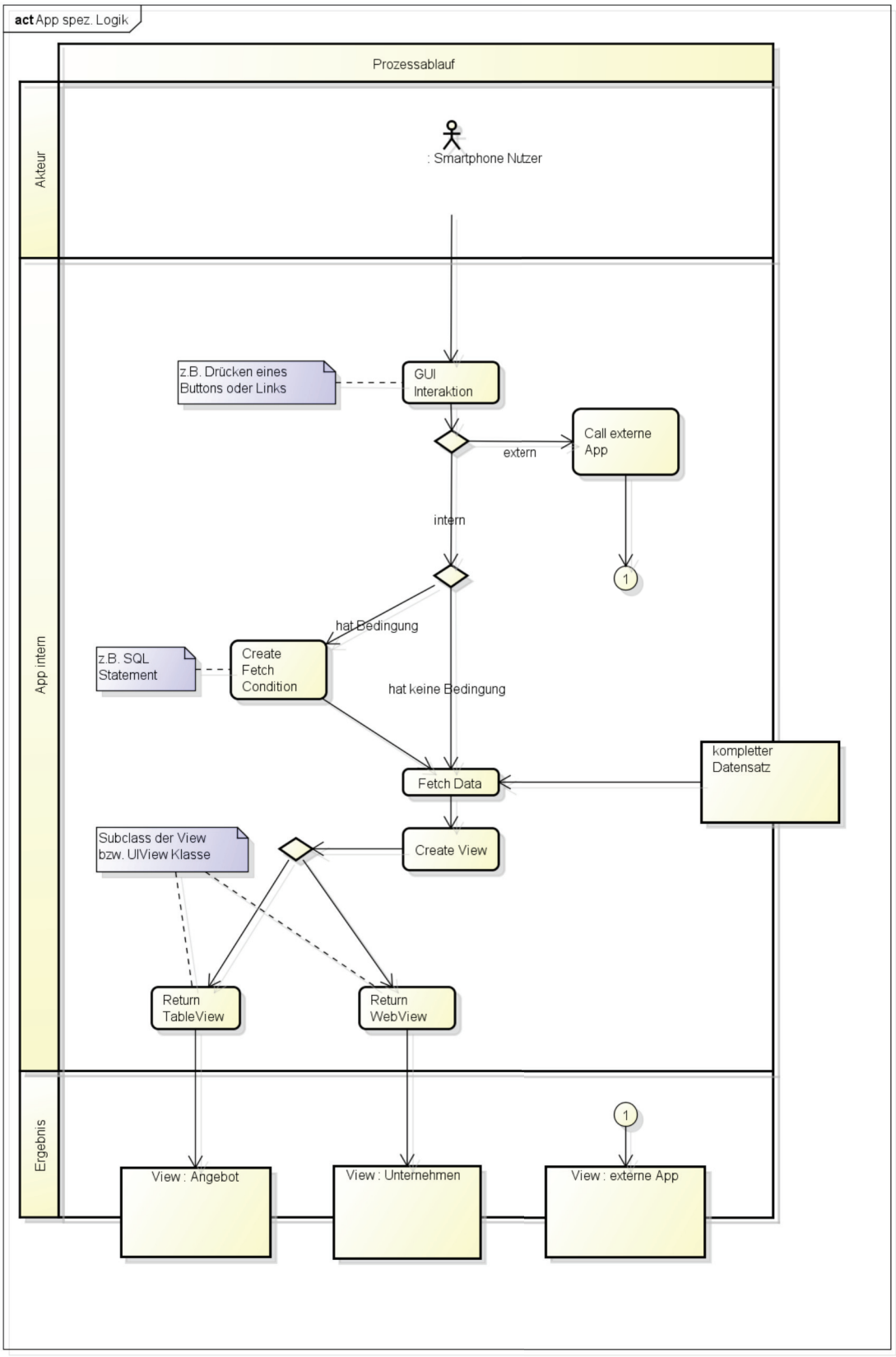

Abbildung 20: App spezifische Logik bei Nutzerinteraktion (erstellt mit astah community)

4.5 Datenmodellierung

4.5.1 Organisation der Daten in Verzeichnissen

Das Datenaustauschformat ist, wie zuvor erwähnt, P-List. Die App empfängt neben P-List Dateien aber noch andere Formate, wie HTML-, Bild- oder CSS-Dateien.
Der gesamte Informationsgehalt der Speisekarte lässt sich direkt in P-List-Dateien festhalten, wenn ein entsprechendes hierarchisches Schema existiert. Das Schema erlaubt es, die Daten sowohl extern auf Seiten der Datenhaltung wie auch intern auf Seiten der App verlustfrei auf eine Tabelle, eine Liste oder ein ER-Modell abzubilden und von und zu P-List im- bzw. exportieren.
Starre Informationen, die selten verändert oder nicht auf ein Schema abgebildet werden müssen bzw. gar nicht können, können als eine Datei (HTML-, CSS- oder Bildformat) übermittelt werden. Aus dieser Datei entsteht in der App bereits eine komplette Sicht. Der Pfad zu dieser Datei, einschließlich Dateiname, steht wiederum in einer P-List Datei.

Die Dateien (P-List und andere) sind an der Quelle (vgl. Abbildung 16) in einem Verzeichnisbaum mit fester Struktur und mit festen Bezeichnungen für die Unterverzeichnisse abgelegt. Die App bildet exakt diese Struktur im internen Dateisystem des Smartphones ab. Pro Verzeichnisknoten existiert als Blatt maximal genau eine P-List-Datei. So kann jede P-List-Datei denselben Namen (*name* + Erweiterung *.plist*) haben, da beides - Verzeichnis und Dateiname - eindeutig sind. Innerhalb der App ist mit Konstanten und auch durch die App-Logik festgelegt, in welchen Verzeichnissen welche Informationen zu finden sind.

Abbildung 21: Verzeichnisbaum Maestro App

Bei schwedischer Spracheinstellung greift die App auf den Inhalt der mit *_sv* endenden Verzeichnisse zu, andernfalls auf den der mit *_en* endenden. In *shared/* befinden sich keine P-List-Dateien, sondern Bild- und CSS-Dateien.

Zusammengefasst: sowohl die Daten, welche die App vom Server als auch die Verzeichnisstruktur sind seitens der App bekannt. Die App implementiert auch bereits den Rahmen, um diese Daten speichern, verarbeiten und darstellen zu können. Zudem ist die App-Logik in der Lage, auf leere Datenfelder oder vorher vereinbarte Sonderwerte (Preis von -1?) sinnvoll zu reagieren. Die App weiß auch, wo sie entweder die Daten selbst herbekommt, oder den Zielort an dem diese zu finden sind. Dies ist immer eine P-List Datei. Auf Basis all dessen, bleibt die Logik der App immer unberührt, wenn Seitens der Datenhaltung Änderungen am Datenbestand vorgenommen werden, da diese Änderungen immer in eine oder mehrere P-List Datei(en) exportiert werden, die die App lesen kann.
Wird natürlich das Modell selbst geändert, d.h. ändert sich die Struktur innerhalb der P-List, muss dies auch seitens der App Logik berücksichtigt und ein Update der ganzen App ausgeliefert werden.

Im Folgenden wird das Datenmodell entworfen das für den zentralen Zweck der App - die Darstellung des Angebots eines Restaurants - das Rückgrat bildet.

4.5.2 ERM Entwurf

Wenn ein Gast sein Essen bestellt, benötigt er zur Bestellung eine Bezeichnung des Essens. Oft existiert zu der Bezeichnung auch eine eindeutige Nummer. Er möchte wissen, was in dem Essen enthalten ist, da dies selten aus der Bezeichnung allein hervorgeht und er möchte wissen, wie teuer das Essen ist. Zu Vergleichszwecken, oder weil er nur eine grobe Vorstellung davon hat, was genau er essen möchte, benötigt er die Art der Speise. Da zu einem Essen auch Trinken gehören kann, wird eine Mengenangabe (0,3 oder 0,5 L) benötigt und eine Einheit der Menge (Stück, Liter, etc.). Trinken lässt sich genau wie Essen kategorisieren (alkoholisch, nicht alkoholisch, verschiedene Weine, Säfte, etc.). Das Essen ließe sich analog in vegetarisch und nicht vegetarisch unterteilen. Manche Speisen werden nur zu bestimmten Tageszeiten serviert (morgens, mittags, abends) oder in einem bestimmten Behältnis (Flasche oder Glas, Teller oder in Folie eingepackt). Außerdem können sowohl die Speisen als auch die Kategorie in der sich die Speise befindet einen erklärenden Hinweis enthalten.

Eine Klasse mit diesen Attributen könnte in der ersten Normalform aussehen, wie in Abbildung 22 dargestellt. Diese Klasse hat nur Attribute und noch keine Operationen dargestellt.

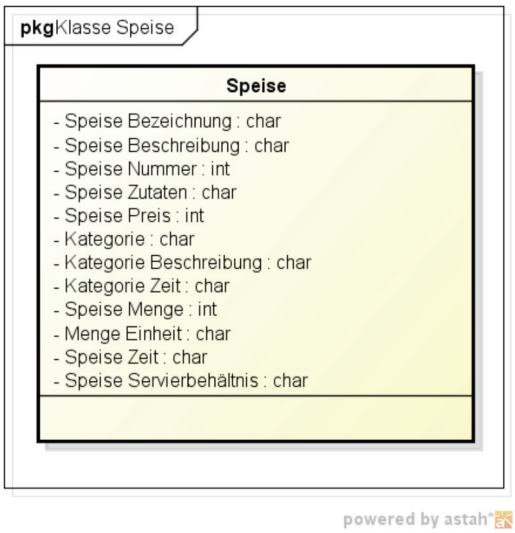

Abbildung 22: Klasse Speise (erstellt mit astah community)

Um die Redundanzen bereinigt würde dieses Modell gemäß Abbildung 23 wie folgt aussehen:

Jede Speise hat ein oder mehrere Zutaten und jede einzelne Zutat befindet sich in einer beliebigen Menge an Speisen.
Jede Speise befindet sich in einer Kategorie und jede Kategorie beherbergt eine beliebige Menge an Speisen. Da Essen und Trinken zwei grundlegend verschiedene Tätigkeiten sind, lässt sich die Speise in entsprechende generelle Klassen aufteilen. Ein Attribut *vegetarisch* ist für Trinken nicht wichtig, genauso wie der Alkoholgehalt beim Essen nicht interessiert.
Die Speise wird in einer Menge verkauft, wobei die Menge eine Mengeneinheit hat (Stück, Liter, ...). Für jede Mengeneinheit wiederum gibt es keine oder eine endliche Anzahl an Speisen, die mit dieser Mengeneinheit verkauft werden.
Manche Speisen sind auf spezielle Tageszeiten begrenzt. Diese Begrenzung kann natürlich auch für alle Speisen einer Kategorie gelten (bspw. nur mittags oder nur morgens und abends). Das Speiseangebot kann demnach begrenzt sein oder nicht. In letzterem Fall ist es ganztägig erhältlich.
Das Servierbehältnis lässt sich analog der Kategorie auch in Behältnisse speziell für Essen und welche für Trinken spezialisieren und so getrennt Attribute definieren, die nur für eines der Beiden Sinn ergeben - z.B. Pfand (wobei sich an dieser Stelle mit Recht argumentieren ließe, dass auch ein Essensbehältnis mit Pfand ausgestattet sein kann).

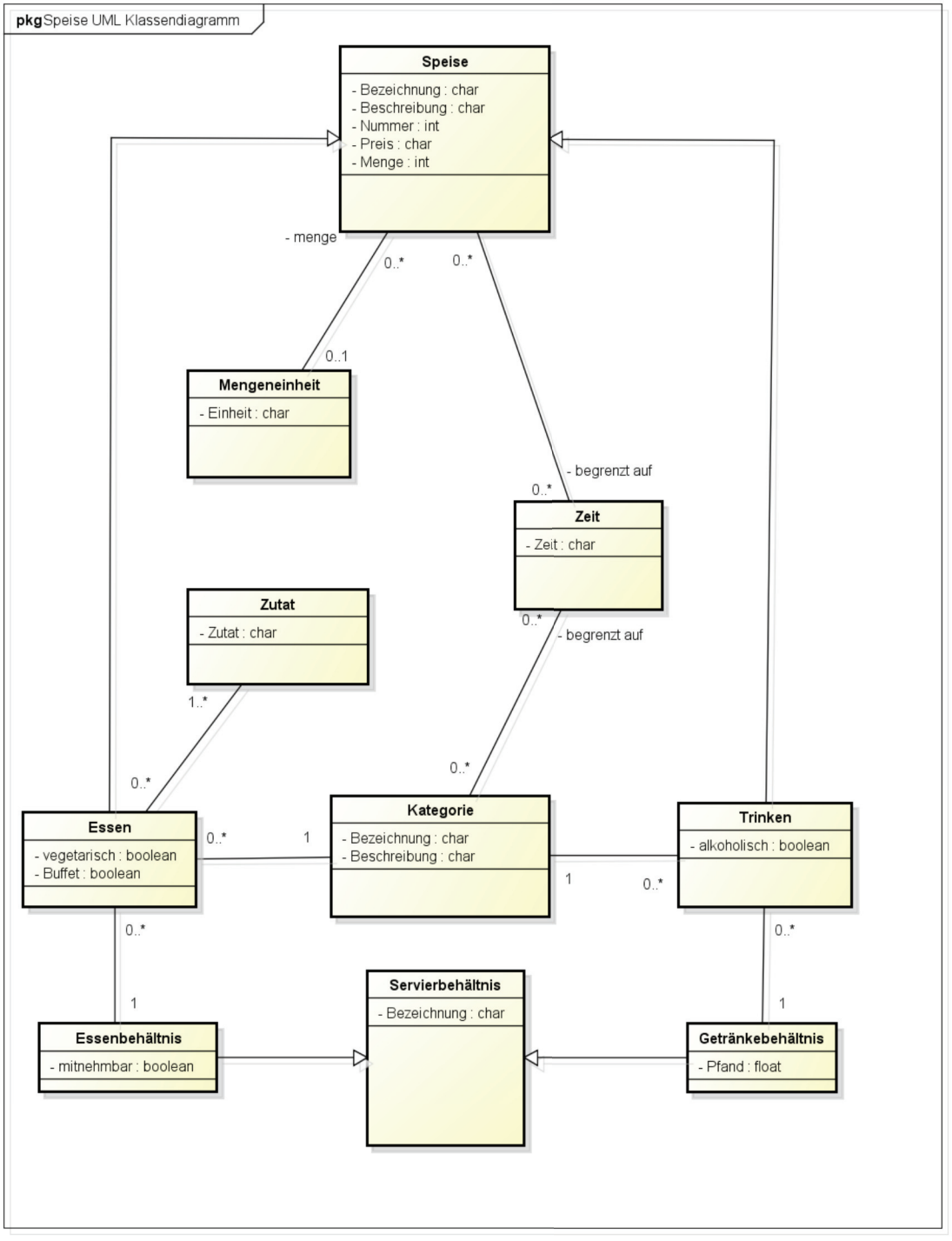

Abbildung 23: UML Klassendiagramm einer Speise (erstellt mit astah community)

Es lassen sich sicherlich noch weitere Klassen und Attribute finden. Für nur ein spezifisches Unternehmen jedoch, ist solch eine feine Granularität im Rahmen der Arbeit nicht sinnvoll. Würde die App hingegen allgemein für ein Unternehmen der Branche Gastronomie entwickelt werden, müssten auch die verschiedenen Ausprägungen eines gastronomischen Unternehmens berücksichtigt werden und sich das Datenmodell an Abbildung 23 orientieren.

Für das Unternehmen, das Gegenstand der Arbeit ist, lässt sich das Modell auf ein einfacheres Modell reduzieren. Zum Vergleich kann das Angebot, wie es auf der Maestro Homepage zu finden ist, dienen (siehe unter Weblinks). Ziel der Vereinfachung ist es, einerseits die Menge der zu übertragenden Daten klein zu halten und so sowohl die serverseitige Datenpflege als auch die clientseitige Auswertelogik unkompliziert zu halten und auch eventuelle Fehlerquellen im Modell zu reduzieren. Andererseits ist dem Medium Smartphone Rechnung zu tragen, das auf seinem ca. 3,5'' Bildschirm nur begrenzten Platz zur Darstellung bietet und bei einem komplizierteren Datenmodell die Verwendung von Detailsichten nötig werden lässt. Hier ist es besser, zugunsten einer angenehmen Nutzererfahrung, auf Detailreichtum zu verzichten und das Modell so einfach wie möglich zu halten *(*in Anlehnung an *„Ockhams Rasiermesser“*, [wik12]).

Da das Modell letztendlich immer noch eine Speisekarte darstellen soll, werden die Attribute *Name*, *Nummer*, *Zutaten* und *Preis* sowie die sie beinhaltende Klasse weiterhin benötigt. Auch benötigt wird weiterhin ein Attribut, dass die Art der Speise näher kennzeichnet.
Angesichts des geringen Umfangs der Speisekarte, lassen sich die Klassen *Zutaten*, *Zeit*, *Mengeneinheit* und *Servierbehältnis* sowie jede spezialisierte Klasse streichen. Es ist nicht nötig, eine eigene Klasse *Zutaten* und davon abgeleitete Objekte zu verwalten, da jedes erzeugte Objekt auf Seiten der App nur der Erzeugung einer darstellenden Sicht dient und es nicht geplant ist, dass der Nutzer mit den Objekten auch arbeitet (sich bspw. eine eigene Mahlzeit aus einer Zutatentabelle zusammenstellt). Das macht es hinreichend, dass die Zutaten als ein *String* im Attribut Zutaten gespeichert werden können. Sollten Mengen- Zeit- oder anderweitige Zusatzangaben zu einer Speise nötig sein, lassen sich diese ebenfalls im Zutaten-String festhalten.
Die *Kategorie* Klasse wird weiterhin benötigt, da sie zwei Zwecken dient: zum einen die Speise einer Kategorie zuzuweisen und damit das geforderte Attribut *„Art der Speise“* umzusetzen (die Art einer Speise ist gleichbedeutend mit ihrer Kategorie) sowie allgemeine Informationen oder Notizen, die für alle Speisen einer Kategorie gelten zu speichern; zum anderen die Filterfunktion der App (und damit die Datenabfrage) zu unterstützen; wenn der Nutzer bspw. nach eben jeder Speise einer Kategorie oder Art suchen möchte.

Dies führt zu einem vereinfachten Modell mit zwei Klassen, *Speise* und *Kategorie*, wobei jede Speise sich in genau einer Kategorie befindet und jede Kategorie hat eine oder mehrere Speisen. Es ist demnach eine „1 zu N“-Beziehung zwischen Kategorie und Speise. Die Informationen, die ein einzelnes Speise-Objekt beinhaltet, sind - aus den Attributen der Speise-Klasse - der Name der Speise, die (Bestell-) Nummer, der Preis und die Zutaten und - von der Kategorie geerbt - der Name der Kategorie (das ist gleichbedeutend mit der Art der Speise) sowie eine Beschreibung der Kategorie. (Abbildung 24)

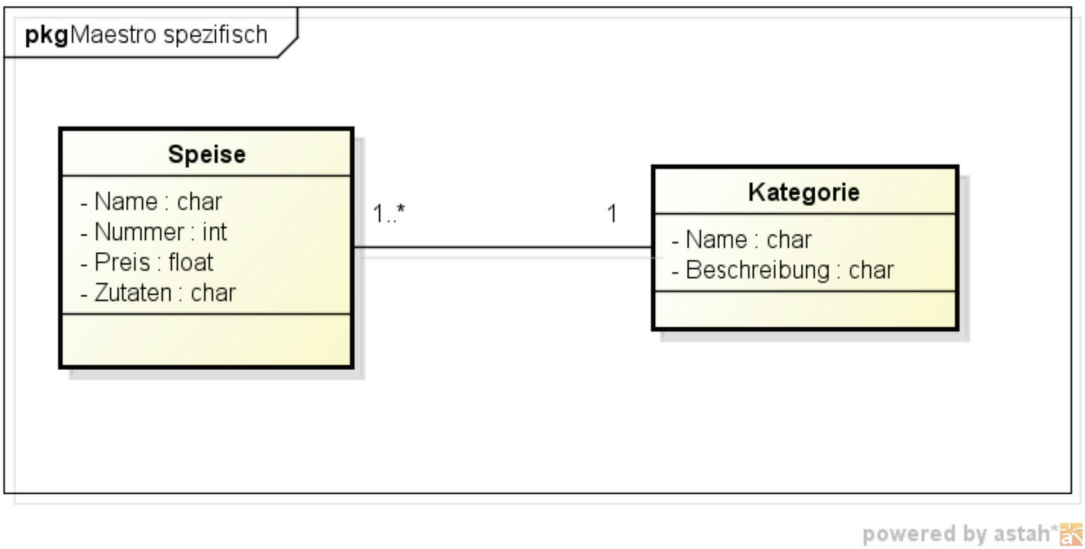

Abbildung 24: UML des Maestro Datenmodells (erstellt mit astah community)

4.6 Architektur

Die Architektur, in die sich die App einbindet, sei hier nur beschrieben. Sie wurde für die App nicht extra entworfen. Zur Realisierung der App konnte ein Unternehmen gefunden werden, das über eine funktionierende Infrastruktur und die nötige Expertise zur Erstellung, zum Vertrieb und zur Wartung von Apps, die im laufenden Betrieb ihren Inhalt dynamisch aktualisieren müssen, verfügt.
Das verwendete Modell ist, grob beschrieben, eine Client-Server Architektur, auf die zwei verschiedene Sichten möglich sind: eine Architektursicht und eine Algorithmensicht. Ein Verweis auf eine Dokumentation der Architektur kann nicht gegeben werden, da diese schlicht nicht existiert. Es wird deshalb versucht, die Architektur zum Verständnis so präzise wie möglich und ohne Referenz zu beschreiben – ohne dabei zu sehr ins Detail zu gehen, da dies nicht Thema der Arbeit ist.

Den Client in der Architektur bildet das Synchronisations-Framework der App, das für das Synchronisieren des lokalen Datenbestandes der App mit dem des Servers zuständig ist und auf das in Kapitel 4.7 noch näher eingegangen wird. Das Synchronisieren mit dem Server besteht hauptsächlich aus Algorithmen. Auf dem Server befinden sich, abrufbar über eine URL, Daten (als komplettes Dateisystem) und Hashwerte über die Daten. Sowohl das Dateisystem als auch die Hashwerte, die das Dateisystem abbilden, werden mit PHP-Scripten erzeugt. Der Client verarbeitet die Hashwerte, vergleicht sie mit dem Hash seines eigenen Dateisystems und lädt die Unterschiede herunter.
Das Synchronisations-Script des Clients bildet die Verzeichnisstruktur des Servers im Filesystem des Clients komplett ab. So ist die Synchronisation auch nicht an ein bestimmtes Austauschformat, wie P-List gebunden, da das Synchronisations-Script den Verzeichnisbaum unabhängig der sich darin befindenden Daten kopiert. (Abbildung 25)

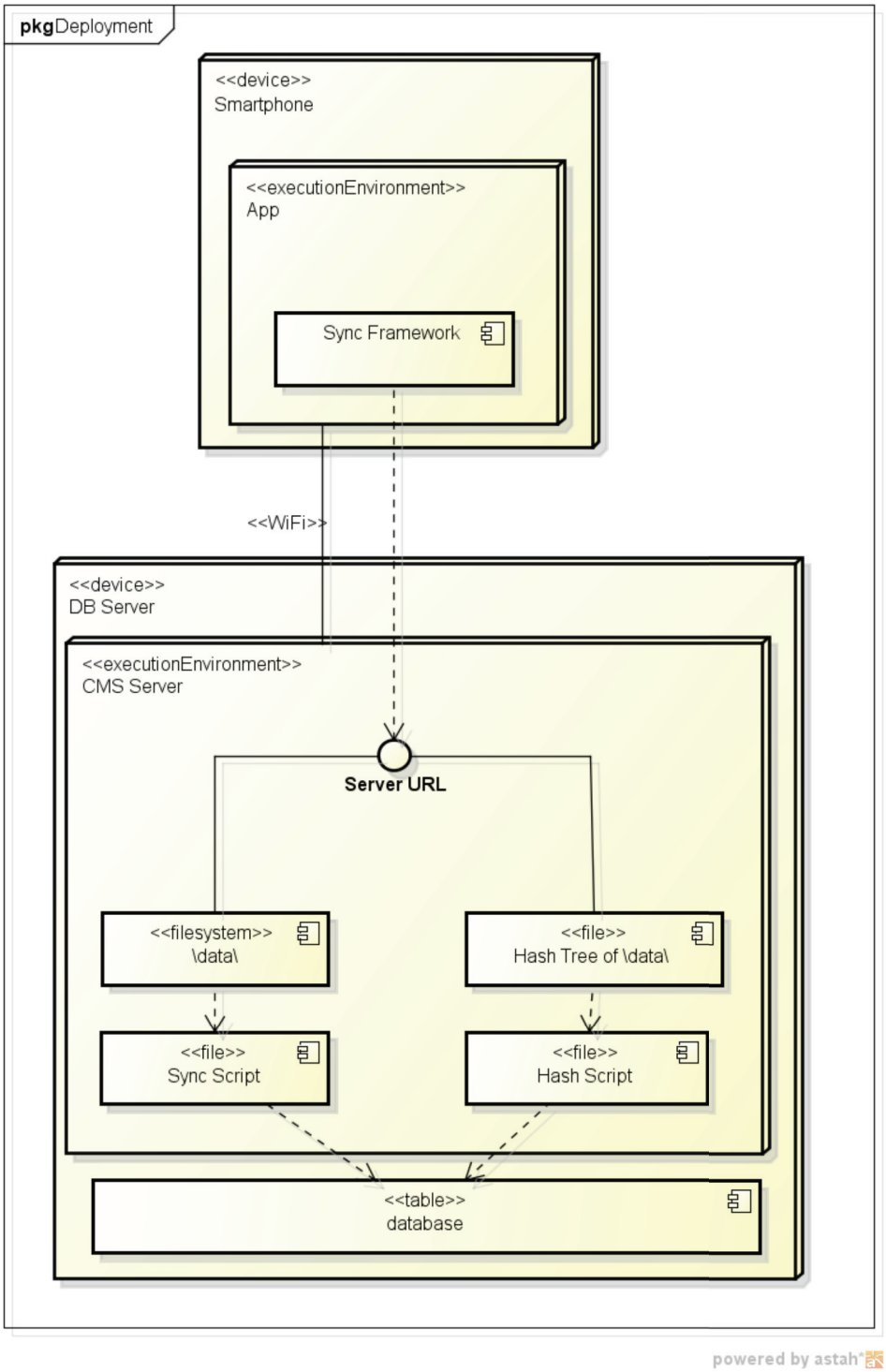

Abbildung 25: System Architektur Verteilung Maestro App (erstellt mit astah community)

4.7 Sync Framework

Das Synchronisations-Framework (Sync Framework) ist Teil der Architektur, in die sich die App einfügt und sei hier, genau wie die Architektur selbst, nur kurz beschrieben, aber nicht extra entworfen. Ebenfalls existiert zum Sync Framework keine Dokumentation, auf die sich an dieser Stelle verweisen ließe.
Das Sync Framework ist in beiden Fällen Bestandteil des Codes der App - im Android Projekt als Java-Bibliothek und im iOS Projekt als eine Sammlung von Header- und Classdateien. Während die iOS Version des Sync Frameworks bereits in anderen Apps Anwendung fand, ist die Android Version die erste App, welche die Android Version des Frameworks implementiert. Firmenintern war die Android Version somit auch ein – erfolgreicher – Testfall des Sync Frameworks.

Aufgabe des Sync Frameworks ist es, den Datenbestand der App mit den aktuellen Daten auf dem Server zu vergleichen und Unterschiede seitens der App neu zu laden bzw. zu löschen. Zum Verständnis, wie das Framework in die Architektur eingebunden ist, dient Abbildung 25. Der Server erzeugt zwei Dateien mit SHA-1 Hashwerten, die den Inhalt des Datenverzeichnisses des Servers abbilden. Die eine Datei beinhaltet den Hashwert vom gesamten Verzeichnisbaum (root hash), die andere die Hashwerte zu jedem Knoten des Verzeichnisbaums (node hash). Das Sync Framework erzeugt die gleichen Hashwerte des lokalen Verzeichnisbaums im Dateisystem des Smartphones. Da das komplette Dateiverzeichnis mit den Daten der App vom Server lokal auf dem Smartphone abgebildet wird, sind beide über die Hashwerte vergleichbar.

Als erstes vergleicht das Script die root hash-Werte vom Client mit dem Server. Sind diese gleich, so ist der gesamte Inhalt gleich und ein Update findet nicht statt. Sind die root hash Werte unterschiedlich, werden die Unterverzeichnisse verglichen. Dafür werden die Hashwerte der Knoten des Verzeichnisbaums verglichen, beginnend an der Wurzel, und jeweils nur die Knoten traversiert, die verschiedene Hashwerte aufweisen.
Bei unterschiedlichen Hashwerten werden die Blätter des Knotens (die Inhalte des Unterverzeichnisses) aktualisiert. Fehlt ein Knoten des Server-Baums im Client-Baum, wird das komplette Unterverzeichnis heruntergeladen. Hat der Client einen Knoten, den es auf dem Server nicht (mehr) gibt, wird er gelöscht.

4.8 GUI Modellierung

Bei der Modellierung des Interfaces (oder GUI bzw. UI) sollten einige Vorgaben beachtet werden, die sowohl allgemeiner Natur (u.a. nachzulesen bei [Sta10]) oder Richtlinien bzw. konkrete Vorgaben des Anbieters der Plattform sind ([DevNA], [Dev12]). Das Interface sollte dem Nutzer nicht den Eindruck eines Werkzeuges geben, sondern mehr die Illusion vermitteln, selbst Teil der Handlung zu sein. Es soll den Anwender auch vor redundanten

und zu vielen Informationen schützen. Die Bedienbarkeit des GUIs sollte so einfach wie möglich gehalten werden. Die Qualität des Interfaces bestimmt die Deutung. Eine Information, die bspw. in einer Web View als wichtig hervorgehoben werden soll, sollte nicht unterstrichen werden, da sie sonst mit einem Hyperlink - der für gewöhnlich unterstrichen ist - verwechselt werden kann. Auch sollte bedacht werden, dass die App kein alleinstehendes Produkt ist. Der Nutzer wird mit vielen anderen Apps und der Interaktion mit diesen vertraut sein und erwarten, dass diese App sich genau so verhält, wie er es von anderen gewohnt ist.
Im Vorfeld sollte auch der Einfluss der geringen Größe des Gerätes bedacht werden und dass die Interaktion mit der Software fast ausschließlich über den Touchscreen bzw. Multitouchscreen stattfindet. Die Interaktion mit dem Touchscreen ist der mit einer Mouse zwar ähnlich (vgl. Tabelle 2), es existieren dennoch vorhandene Unterschiede. Im Ganzen ist die Interaktion weniger filigran und pixelgenau und Informationen können durch die Hand des Nutzers schnell verdeckt werden. Dies hat wiederum einen Einfluss auf die Größe, die Anzahl und das Design von Icons, Bedienelementen, etc.

Bei Nutzereingaben in Formfelder zur bspw. Suche in einer List wird eine virtuelle Software-Tastatur vom BS eingeblendet. Dies hat den Vorteil, dass Tastaturlayouts anderer Sprachen genutzt werden können – der zur Informationswiedergabe verfügbare Teil des Bildschirms wird dadurch aber noch weiter eingeschränkt.

Bei der Entwicklung eines GUIs können sechs wesentliche Aspekte, wie in Tabelle 3 dargestellt, unterschieden werden ([Sta10])

a) Gestaltung von Standbildern; Typo/Layout; Illustration; Metaphern; Icons	**Screendesign**
b) Gestaltung von Daten zu Informationen; Visualisierung	**Informationdesign**
c) Gestaltung von Funktionselementen	**Interfacedesign**
d) Inszenierung von Interaktion. Dynamik und Entwicklung, Struktur und Gestaltung der Repräsentanz von Interaktion	**Interactiondesign**
e) Gestaltung von Bewegtbildern	**Film-/Video-/Animationsdesign**
f) Musik, Tongestaltung	**Ton/Musik/Sounddesign**

Tabelle 3: Aspekte bei der Gestaltung interaktiver Produkte (übernommen von [Sta10], S.13)

(a) bis *(f)* wirken direkt aufeinander und stärken bzw. schwächen sich gegenseitig. *(e)* und *(f)* kommen in der App nicht vor, mit der Ausnahme von vielleicht Animationen beim Wechseln der Seiten, was allerdings durch das jeweilige BS übernommen wird.
Screendesign bezeichnet das Erstellung eines Layouts. Das *Screendesign* bildet sich aus Gestaltungsvorgaben, welche größtenteils durch die Anbieter der jeweiligen Plattformen vorgegeben sind. Screen- und Interfacedesign stellen sicher, ob eine Interaktionsmöglichkeit erkannt wird. Das

Interfacedesign ist die visuelle Repräsentation der Funktionalität. Dessen Gestaltung hängt von *Screen*-, *Information*- und *Interactiondesign* ab. *Informationdesign* unterscheidet sich vom *Screendesign* dadurch, dass es mehr Ordnung und Struktur vorgibt. *Interactiondesign* definiert die Inszenierung und das Nutzerempfinden von Interaktionen. ([Sta10], S.14 ff)
In vielerlei Hinsicht ist die App bezüglich Screen- und Interfacedesign weitestgehend den Designvorgaben der Plattformanbieter unterworfen; bzgl. *Interactiondesign* zusätzlich noch den Bibliotheken, Frameworks oder APIs, welche die jeweiligen Plattformen anbieten und mit welchen man als Entwickler arbeitet.

Eine Mindmap ist eine Gedankenkarte zur Beschreibung von Schlüsselwörtern, Tätigkeiten, Eigenschaften und Assoziationen ([Pau12], S.46). Folgende Mindmap (Abbildung 26) soll illustrieren, welche verschiedenen und thematisch trennbaren Informationen sich in eigenen getrennten Bildschirmseiten darstellen lassen; also zu welchem Informationspaket der Nutzer navigieren kann und welche weiteren Informationen sich so untergruppieren lassen. Die Mindmap erhebt keinen Anspruch auf Vollständigkeit hinsichtlich des tatsächlichen Informations- und Funktionsumfangs der fertigen App. Sie ist bewusst kurz und knapp gehalten und nicht in die letzten Blätter der Baumstruktur ausformuliert. Der Hauptzweck dieser Mindmap ist es, Informationen zu bündeln und daraus (die geringst mögliche Menge an) Informationskategorien zu gewinnen, die in den obersten Navigationselementen im GUI der App Entsprechung finden.

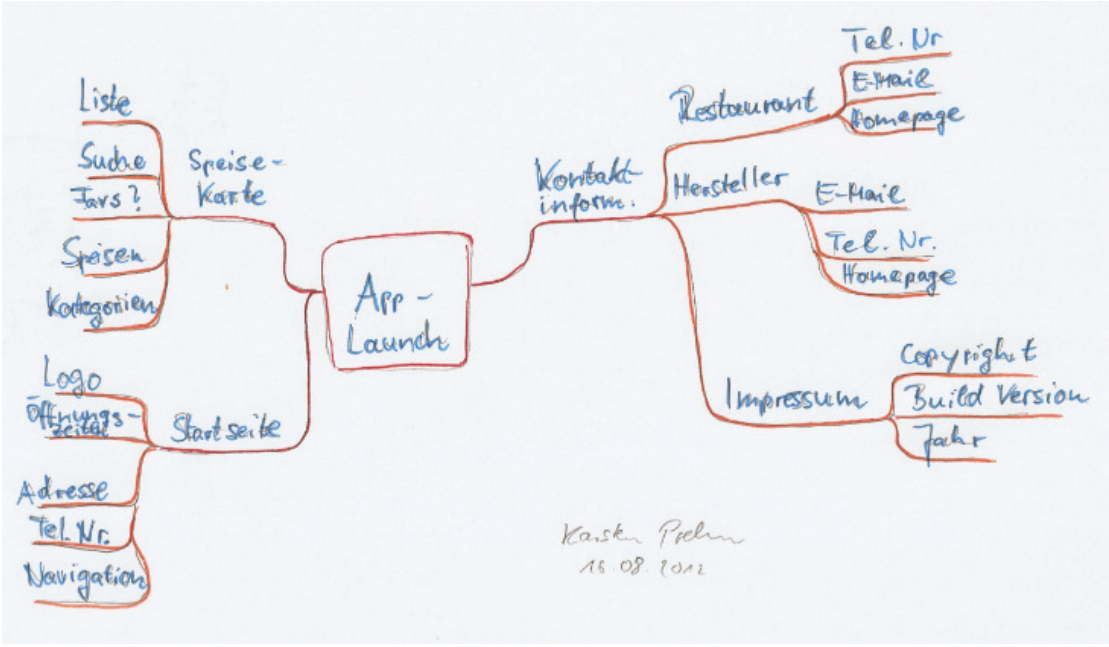

Abbildung 26: Mindmap “Navigation und Informationsaufteilung”

Als Ausgangspunkt dient der Start der App (App Launch). Ein wesentliches Feature der App ist die Speisekarte. Die Informationen, die der Nutzer der Speisekarte erhält, betreffen die Speisen und deren Attribute. Diese können z.B. in einer Liste mit Suchfunktion angeordnet und in Kategorien gruppiert sein und die Möglichkeit der Kennzeichnung als Favorit bieten.

Ein anderes Bündel an Informationen betrifft solche, die vom Typ her Kontaktinformationen oder einem Impressum zugeordnet werden können. Dies sind u.a. Telefonnummern, E-Mail-Adressen, Internetauftritte und Anschriften von Beteiligten – Beteiligte sind das Restaurant selber, der Hersteller und Dritte.

Ein drittes Informationsbündel ist mit „Startseite“ angegeben. Dies meint technisch tatsächlich die Startseite der App und im übertragenen Sinne die Wichtigkeit der allerersten Seite, die der Nutzer nach Start der App sieht. Die Startseite sollte etwas über das Restaurant selbst aussagen – in Information und vermitteltem Eindruck vergleichbar mit der Eingangsseite des realen Restaurants. Und gleich dem realen Restaurant, vor dem der Gast steht, Interesse gewinnt und nähertritt, um den Aushang der Speisekarte zu studieren, startet der Nutzer der App selbige, gewinnt Interesse und navigiert erst dann (digital) zur Seite mit der Speisekarte. Dies beeinflusst auch direkt die Informationen, welche die Startseite vermittelt: ein graphisches Logo des Restaurants, die Öffnungszeiten und Adresse, Telefonnummer und, wenn möglich, eine Navigationshilfe – denn schließlich steht der Nutzer, höchstwahrscheinlich, nicht persönlich vor dem Restaurant.

Zusammengefasst ergibt das drei Sichten, d.h. drei eigenständige und unabhängige Bildschirmseiten, die nach der Thematik und auch nach der Wichtigkeit des darzustellenden Inhalts unterschieden werden können. Naheliegend ist ein Design des Interfaces mit Registerkarten (siehe Abbildung 27).

Abbildung 27: Screenskizze des GUI

Screenskizzen zeigen das gedankliche Modell, das zu einem bestimmten Zeitpunkt von Inhalt und Handhabung einer Applikation bei den Entwicklern besteht. Sie konkretisieren nur Wichtiges und lassen Unbestimmtes schemenhaft. ([Pau12], S.48)
Abbildung 27 illustriert den Blick des Nutzers auf die App nach deren Start. Er hat die Wahl zwischen der Navigation zu einer anderen Sicht mit einer anderen Kategorie an Informationen (Essen und Kontakt) oder zum Verbleiben in der aktuellen Sicht. Durch das Design des Interfaces mit

Registerkarten, nimmt er automatisch die Registerkarten selbst als hierarchisch gleichranging wahr und den Inhalt der Registerkarten als untergeordnet. Die Punkte 1 bis 5 sind bisher noch nicht spezifiziert, aber *1.* könnte ein graphisches Logo sein, *2.* und *3.* rein textliche Informationen und *4.* und *5.* weiterführende Navigationselemente sein. „5. Navigation“ meint auch eine Form der geographischen Navigation zum Restaurant, nicht die Navigation im GUI.

Ein Unterschied bei der Verwendung von Registerkarten zwischen iOS und Android sei an dieser Stelle erwähnt: das UI-Element, das die Reiter (Tabs) beinhaltet, befindet sich auf iPhones für Gewöhnlich am unteren Ende des Bildschirms, auf Android Geräten am oberen Ende. Dies hat prinzipiell keine Auswirkungen auf den zur Verfügung stehenden Platz (Abbildung 28), ist optisch aber ein Unterschied zwischen beiden Plattformen.

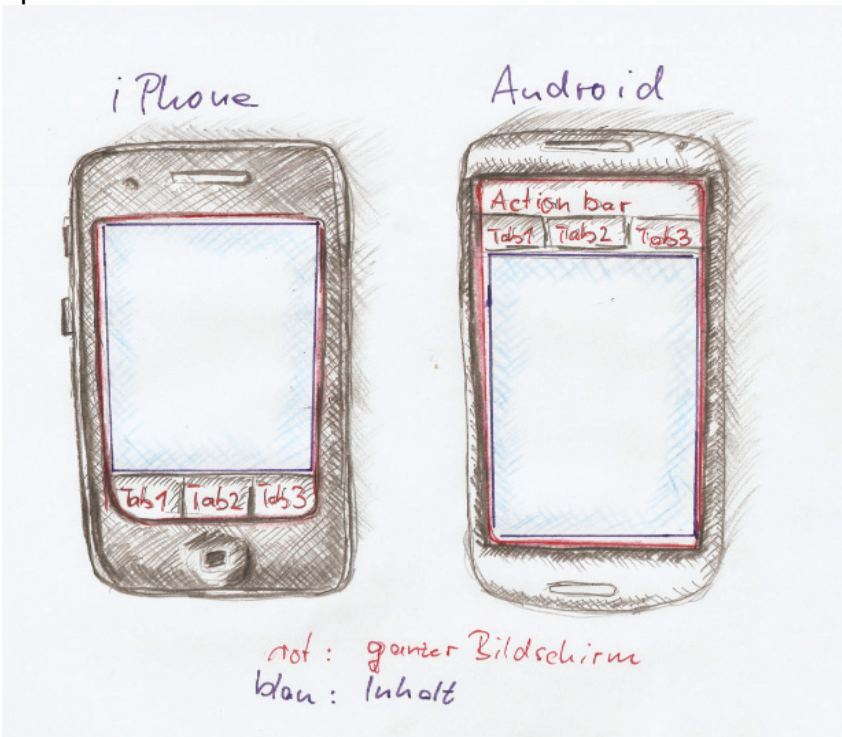

Abbildung 28: GUI Modell

Die Zeichen auf den Tabs sind Metaphern. In Android Anwendungen ist es üblich, die Tabs nur zu beschriften, in iOS Anwendungen ein sinnbildliches Icon zur Schrift hinzuzufügen (Abbildung 29). Die Wahl des passenden Icons kann u.U. schwierig sein und sollte gut durchdacht werden.

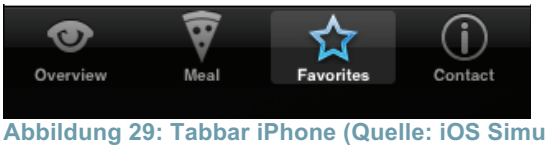

Abbildung 29: Tabbar iPhone (Quelle: iOS Simulator)

5 Umsetzung

In diesem Kapitel folgt eine Beschreibung der Umsetzung des Projekts.

5.1 Kompatibilität

Kompatibilität[5] ist für die iOS Version der App eher nebensächlich, dafür für die Android Version von zentraler Bedeutung. Zur Erinnerung, die Mehrzahl der Android Smartphones betreibt eine OS Version 2.3.x (vgl. Abbildung 3), während mit Version 4 sowohl einige Paradigmenwechsel im Android BS stattfanden, als auch eine Menge neuer Funktionalitäten hinzugekommen sind.
Zur Gewährleistung einer Rückwärtskompatibilität kommen zwei Bibliotheken zum Einsatz: die *Support v4 Library* (android-support-v4-r7-googlemaps.jar) zur Unterstützung der neuen Android APIs auf Geräten mit API Level 4 (Android Version 1.6) oder mehr und *ActionBarSherlock*.

Die Support Library ermöglicht es der Android Version, Fragmente (seit Android 3.0, API Level 11) und ViewPager auch in älteren BS Versionen zu verwenden. Während Fragmente ein Feature sind, das eher auf größeren Bildschirmen, wie dem von Tablets zur Unterteilung des Bildschirms in eben jene Fragmente, zum Einsatz kommt, ermöglicht es die *ViewPager.class* eine interessante GUI Erfahrung zu gestalten, bei der der Nutzer durch eine Wischgeste zwischen den Seiten wechseln kann.

ActionBarSherlock (ABS) (siehe unter Weblinks) ist eine Open Source Erweiterung der Kompatibilitätsbibliothek. ABS vereint die Kompatibilitäts-Bibliothek mit einer rückwärts kompatiblen *ActionBar* (seit API Level 11) unter einer API. ABS delegiert zur *ActionBar.class* auf OS Versionen, die diese unterstützen und simuliert eine ActionBar ansonsten.

5.2 GUI

Die Seiten sind in einen Tab-basierten GUI-Rahmen integriert. Das Tabwidget befindet sich in der iPhone Version der App standardmäßig am unteren Ende des Bildschirms, in der Android Version standardmäßig am oberen Ende. Die Navigation zwischen den Seiten erfolgt durch Antippen der Tabs (Reiter). Die Android App unterstützt zudem die Navigation durch „Viewpaging“, d.h. Seitenwechsel durch Antippen, „Festhalten“ und Wischen der Seite nach links oder rechts. Außerdem beinhaltet die Android Version eine dauerhafte Actionbar oberhalb des Tabwidgets, die Teil der verwendeten Rückwärts-

[5] Kompatibilität meint hier die Kompatibilität innerhalb einer Plattform zwischen unterschiedlichen Versionen des BS der Plattform, nicht die Kompatibilität zwischen verschiedenen Plattformen.

Kompatibilität-Bibliothek ist (genaugenommen ist das Tabwidget Teil der Actionbar).

Die der Actionbar entsprechende Navigationbar in der iOS Version fällt aus hauptsächlich Platzgründen und zugunsten eines einheitlichen Designs weg. Eine oben eingeblendete Navigationbar und ein unten eingeblendetes Software Keyboard verengen den Blick auf die Speisekarte zu sehr. Ein Verzicht auf die Navigationbar nur in der einen Seite, welche die Karte darstellt, trübt jedoch möglicherweise den subjektiven Gesamteindruck der App durch plötzliche Änderung der Weite des Sichtbereichs beim Wechsel der Seiten. Ein Verzicht auf die Navigationbar ist die beste Lösung. In der Android Version kann dieser Kompromiss aus besagten Kompatibilitätsgründen nicht eingegangen werden.

Eine „halbe“ Sicht ist zudem der Splashscreen, den der Nutzer beim ersten Start der App sieht, der aber aus GUI Sicht keine Interaktionsmöglichkeit bietet (vgl. 4.4.1).

Die erste Seite, die der Nutzer nach Start der App und nach dem Splashscreen sieht, dient der Übersicht. Sie fasst alle wichtigen Informationen das Restaurant betreffend zusammen: Öffnungszeiten, Mittagsangebot, Sonderangebote und Daten, die der schnellen Kontaktaufnahme dienen (Adresse, Telefonnummer, Homepage, geographische Karte der Lage).

Die zweite Seite beinhaltet des Speiseangebot als scrollbare Liste und ein Formfeld zur Eingabe von Stichworten zum Filtern der Liste. Dafür wird eine virtuelle Software-Tastatur vom Betriebssystem eingeblendet, deren Layout (QWERTZ, QWERTY, etc.) sich an den Ländereinstellungen des Smartphones orientiert – inklusive sprachtypischer Sonderzeichen.

Die letzte Seite dient als Impressum und beinhaltet noch einmal die Adresse des Restaurants sowie Telefonnummer, E-Mail Adresse, Homepage URL und Öffnungszeiten. Außerdem befinden sich auf dieser Seite dieselben Kontaktinformationen zu Unternehmen, die mit dem Restaurant verbunden sind (Sponsoren) und Informationen zur App selbst (Hersteller, Build Version).

Basierend auf dem Datenmodell, wird die Seite, welche die Speisen darstellt, eine Liste sein. Die Liste wird Sektionen haben für die Kategorie Einträge und normale Zellen für die Speisen und so das Datenmodell 1:1 abbilden. Pro Sektion gibt es demnach so viele Zellen in der Liste wie die der Sektion entsprechende Kategorie Speisen hat. Die Zelle stellt am linken Rand die Nummer dar und am rechten Rand den Preis. Im oberen Drittel zwischen Nummer und Preis befindet sich der Name der Speise; darunter, mit dem größten Platzanteil, befinden sich die Zutaten. (Abbildung 30)
Die Sektionen passen sich dieser Aufteilung an, mit dem Unterschied, dass sie nur ein Feld für den Namen und eines darunter befindende Feld für die Beschreibung der Kategorie haben.

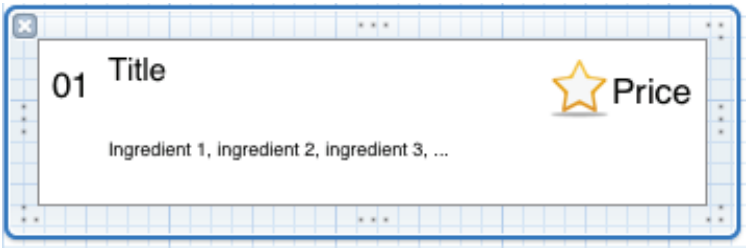

Abbildung 30: eine Zelle der Speisekartenliste (Quelle: XCode Interface Builder)

Am oberen Ende der Seite befindet sich eine Eingabemaske zur Filterung der Angebotsliste nach Stichwörtern. Die Stichwörter werden über ein Software-Keyboard eingegeben, das durch das jeweilige BS automatisch als Reaktion auf ein Tipp-Event in die Eingabemaske eingeblendet und durch andere Events (Back-Button, Cancel-Button) auch wieder ausgeblendet wird. Der Standardwert des Suchfeldes ist ein ausgegrautes *„Sök“* bei schwedischer Spracheinstellung oder *„Search“* bei jeder anderen. Der Standardwert wird bei Nutzereingaben überschrieben und erscheint nur bei einem leeren Suchfeld.

Eine Besonderheit stellt die iPhone Version der App dar. Es ist bereits auf Abbildung 30 zu erkennen, dass die Zelle noch einen Stern links des Preises hat. Dieser Stern ist interaktiv und markiert eine Speise als Favorit (voller Stern) oder nicht (leerer Stern wie in Abbildung 30; Standardwert). Die Favoriten sind eine Untermenge der Speisen der Speisekarte. Sie sind als eigenständige Seite über einen zusätzlichen Tab abrufbar (Abbildung 29). Auf dieser Seite wird eine im Design identische Speisekartenliste der Favoriten dargestellt oder eine Leere Seite für den Fall, dass es keine Favoriten gibt.

5.3 Datenmodellierung auf Basis der Speisekarte

Als Grundlage bzw. Vergleichsobjekt zur Umsetzung des Datenmodells (vgl. 4.5.2) kann die Homepage des Restaurants Maestro (siehe unter Weblinks) herangezogen werden. Hier fällt recht schnell auf, dass sich die dort präsentierte Hierarchie nur in Teilen und damit eigentlich gar nicht auf das zuvor erarbeitete Datenmodell abbilden lässt. Für das Datenmodell ist es wichtig, dass es so allgemein aber immer noch präzise formuliert gehalten ist, dass sich jede einzelne Speise auf das Modell abbilden lässt, ohne die Notwendigkeit zu provozieren, für manche Kategorien oder Speisen eine eigene Darstellung entwerfen zu müssen. D.h. es müssen Anpassungen am Datenmodell des Restaurants getroffen werden und Anpassungen bedeuten auch immer, einen Kompromiss einzugehen, der im Vorfeld mit dem Geschäftspartner (in diesem Fall ebendem Restaurant) kommuniziert werden sollte.

Abbildung 31: Auszüge der Speisekarte der Maestro Homepage

Abbildung 32: Auszüge der Speisekarte der Maestro Homepage

Die Speisekarte des Restaurants (vgl. Auszüge in Abbildung 31 & Abbildung 32) wird im Sinne der App wie folgt abgeändert (zur Erinnerung, es wird eine hierarchische Darstellung mit Kategorien und untergeordneten Speisen benötigt):

- *Pizza* als Knotenpunkt der Kategorien *Classics*, *Stars* (65 und 70), *Cults* und *Icons* fällt weg
- *Pizza* ist stattdessen ein Suffix besagter Kategorien (*Pizza Classics*, ...). Damit gliedert sich auch die, zuvor alleinstehende, Kategorie *Mexican Pizza* in dieses Schema ein.
- Da der Preis ein Attribut der Speise ist, rückt er als Information von den entsprechenden Kategorien, die einen Preis aufführen in die untergeordneten Speisen.
- Damit werden auch *Pizza Stars* für 65:- und 70:- einer Kategorie *Pizza Star*s zusammengefasst.
- Die weiteren Kategorien sind *Sallads*, *Burger*, *Kebab*, *Other* und *Extra*. Das ergibt 10 Kategorien insgesamt.
- Die Informationen zur Familienpizza rücken in das *Beschreibung*-Attribut der jeweiligen Pizzakategorie für die sie gelten. Dies ist auch ein hervorragendes Beispiel für einen Kompromiss, der im Sinne des Datenmodells getroffen werden muss. *Familienpizza* hätte als eigene

Kategorie mit vier Eintragungen umgesetzt werden können. Dies hätte aber redundante Informationswiedergabe zur Folge. Die Nummern, die eine Familienpizza umfasst (1-15, 15-26, ...), sind alle Pizzen einer Kategorie. Damit ist es eine viel bessere Lösung, wenn Familienpizza - einschließlich Preis unter Wegfall der Angabe der abgedeckten Nummern - eine Information des *Beschreibung*-Attribut der Kategorie ist.

Eine Repräsentation der Daten als P-List Datei kann Abbildung 33 entnommen werden.

```
▼<plist version="1.0">
  ▼<dict>
     <key>index</key>
    ▼<array>
      ▼<dict>
         <key>category</key>
         <string>Pizza Classics</string>
         <key>ingredients</key>
         <string>skinka</string>
         <key>name</key>
         <string>Vesuvio</string>
         <key>number</key>
         <string>1</string>
         <key>price</key>
         <string>60</string>
       </dict>
      ▼<dict>
         <key>category</key>
         <string>Pizza Classics</string>
         <key>ingredients</key>
         <string>skinka, ananas</string>
         <key>name</key>
         <string>Hawaii</string>
         <key>number</key>
         <string>2</string>
         <key>price</key>
         <string>60</string>
       </dict>
```

Abbildung 33: Auszug einer P-List Datei mit Speiseeinträgen

5.4 Datenspeicherung und Parsen der Daten

5.4.1 Android

Die Android Version nutzt keine Datenbank zur persistenten Speicherung der Daten, stattdessen werden die Daten bei jedem Start der App aus den P-Lists vom lokalen Dateisystem des Smartphones in eine Datenstruktur *ArrayList* geparst. Dies ist dem Umstand geschuldet, dass ein erster Prototyp der App für Android entwickelt wurde, dieser aber zu diesem Zeitpunkt weder eine Anbindung an einen Datenserver besaß, noch existierten Server-seitig irgendwelche Daten, mit denen gearbeitet hätte werden können. Stattdessen nutzte die App direkt in der *MainActivity* einen Stub, der die Datenstruktur *ArrayList*, aus der die Speisekartenliste generiert wurde, mit hard-codierten String Werten füllte. So konnte zum einen angenommen werden, dass das Synchronisations-Framework Daten liefert, obwohl es noch gar nicht

Bestandteil der App war und zum anderen die App Logik unabhängig dieses Fehlens bereits implementiert werden. Der frühe Prototyp ermöglichte es außerdem, früh die konkrete Umsetzung der App in Layout und Design mit Maestro abzustimmen.

Zu einem späteren Zeitpunkt wurde die App um das Framework und um die Kompatibilitätsbibliotheken (5.1) erweitert. Die Entscheidung, eine interne Datenspeicherung in der finalen Version vorerst wegzulassen, war letztendlich ein Kompromiss zugunsten eines früheren Uploads in den Market. Eine Datenbank kann noch nachgeliefert werden. Dies ist allerdings aufgrund des geringen Umfangs der Daten nicht kritisch. Im Vergleich mit der iOS Version, die eine Datenbank nutzt, können keine Performance Einbußen beim Start oder während des Betriebs der App ausgemacht werden.

Der verwendete Parser ist ein XML Pull Parser, der die P-List Dateien als normale XML Dateien betrachtet. Die Parser Logik konnte erst nach Kenntnisstand des zu verwendenden Datenmodells und damit der zu erwartenden XML Tags umgesetzt werden. Der Inhalt der P-Listen wird in Datenstrukturen des Typs *ArrayList<EntityTyp>* geparst, wobei *EntityTyp* eine Klasse mit den Attributen sowie entsprechenden Settern und Gettern von Kategorie oder Speise (Abbildung 24) ist. Der Parser hat Methoden zum Parsen von Kategorien, Speisen sowie von URL Strings und kann damit jede Information, die in P-Listen übermittelt wird, verarbeiten.

5.4.2 iOS

In der iOS Version ist ein Data Access Object verantwortlich für das Einlesen der P-List Einträge in eine SQLite DB und für das Auslesen (Fetch Requests) der DB mit SQL Statements (Predicate). Das Core Data Framework, das mit dem Projekt verknüpft ist, übernimmt die Verwaltung von Objekten des Datenmodells und die Verbindung zwischen diesen Objekten und der graphischen Oberfläche. Das Datenmodell (Abbildung 24) ist durch eine *.xcdatamdodel* Modelldatei realisiert (Abbildung 34).

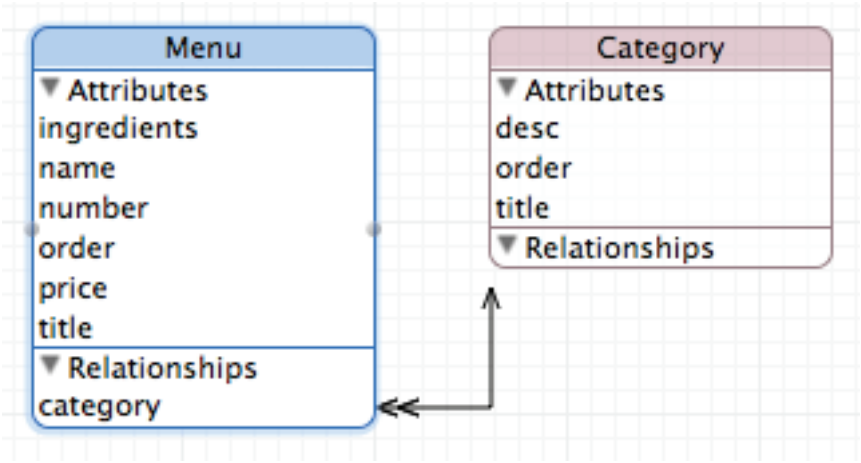

Abbildung 34: Core Data Datenmodell (Quelle: XCode)

Zu sehen sind zwei Entitäten. *Menu* entspricht Speise, *Category* ist die Kategorie. Beide Entitäten haben dieselben Attribute wie in Abbildung 24, mit dem Unterschied, dass noch ein Attribut *order* zusätzlich definiert ist. *Order* ist ein inkrementeller Wert und entspricht der Reihenfolge in der die Einträge eingelesen werden. Dieses zusätzliche Attribut wird benötigt, um die beim Einlesen (parsen der P-List) in ein *NSDictionary* nicht zwangsläufig sortierten Einträge bei der Ausgabe in der immer gleichen Reihenfolge sortiert vorzufinden. Jeder Fetch Request benötigt die Angabe einer Sortierreihenfolge. Dafür dient das *order* Attribut. Die Attribute sind bis auf *order* alle vom Typ String.
Außerdem ist bereits eine 1:N Beziehung zwischen beiden Entitäten zu sehen. Die Beziehung ist invers.

Im Fall, dass sich der Datenbestand des Server geändert hat und neue Daten durch das Sync Framework heruntergeladen werden, wird die DB gelöscht und neu gefüllt.

5.4.3 Favoriten

Die iOS Version die App erlaubt es dem Nutzer, Favoriten zu speichern. Die Favoriten sind nicht Teil des Datenschemas (was prinzipiell auch möglich gewesen wäre), da die Datenbank bei jeder Synchronisation gelöscht wird, was auch immer die Favoriten mit löschen würde. Dies ist nicht gewollt. Stattdessen werden die Favoriten in den User Defaults der App, die persistent sind, gespeichert. Die User Defaults werden beim Start der App geladen oder neu erstellt, wenn sie nicht existieren (dies ist der Fall beim ersten Ausführen der App).
Die Logik des Favoriten-Buttons ist in der .m Datei der *TableView* Zelle (vgl. Abbildung 30) implementiert. Da angeraten wird, nur einfache Datentypen in den User Defaults zu speichern, wird der Wert des *order* Attributs der favorisierten Speisen gespeichert. Wichtig ist, dass dieser Wert eindeutig einer Speise zugeordnet werden kann. Außerdem ist der Text des Titel Attributs des Favoriten-Buttons gleich dem *order* Wert. Über dem Titel liegt noch das Icon. Der Text ist somit zwar nicht sichtbar, was auch so gewollt ist, der Button kann aber mit dem *order* Wert in den User Defaults verglichen werden. So kann dem Icon beim Erstellen der Zelle in der Speiseliste korrekt der volle oder leere Stern zugeordnet werden.
Ein Drücken des Favoriten-Buttons schaltet im Event Handler des Buttons das Icon um und auch der User Defaults Eintrag wird auf den nicht vorhandenen Wert gesetzt (d.h. gelöscht, wenn vorhanden, ansonsten hinzugefügt). Gleichzeitig wird eine Nachricht an einen Listener in dem View Controller des separaten Favoriten Tabs (vgl. Abbildung 29) gesendet, die veranlasst, dass die Favoriten Liste aktualisiert wird, während die Speisekartenliste offen ist. Andersherum ist dieselbe Logik implementiert. In der Favoritenliste können auch nur Einträge, wenn vorhanden, gelöscht werden. Wird der letzte Eintrag gelöscht oder ist sie bereits bei Betätigung des Tabs leer, zeigt sie an, dass keine Favoriten vorhanden sind. Ein Vergleich zwischen Speisekartenliste mit markierten Favoriten und

Favoritenliste kann den Abbildungen Abbildung 38 und Abbildung 37 entnommen werden.

5.5 Datenwiedergabe

Die Wiedergabe der Daten ist ähnlich zwischen beiden Versionen. Je eine Web View zur Darstellung des HTML basierten Information- und Kontaktinhalts in der ersten und der letzten Seite und eine Liste bzw. Tabelle zur Darstellung der Speisekarte in der Android Version bzw. von Speisekarte und Favoriten in der iPhone Version. Die Web Views in beiden Plattformen erlauben das Betätigen von Hyperlinks. Das Drücken einer Telefonnummer startet die Telefon App, einer Homepage den Browser, eine E-Mail Adresse startet den E-Mail Client und der Navigation Link startet die Google Maps App.
List Views in Android und Table Views in iOS haben eine bemerkenswerte Gemeinsamkeit. Wenn eine Zelle außerhalb des sichtbaren Bereichs scrollt, wird ihre Instanz vom *ArrayAdapter* (Android) bzw. von der *UITableView* Klasse (iOS) für die neu erscheinende Zelle wiederverwendet. So muss das Zell Objekt nicht neu erstellt werden und im Idealfall nicht mehr Zell Objekte beim ersten Laden der Liste erstellet und im weiteren Betrieb im Speicher gehalten werden, wie Zellen auf dem Bildschirm sichtbar sind.

5.5.1 Android

Die Android Version der App nutzt die ActionBarSherlock API. Die darstellenden Activities sind von der Klasse SherlockFragmentActivity. Der eigentliche Inhalt wird, wie es der Name andeutet, als Fragment dargestellt. In der MainActivity wird der Inhalt der P-Listen in Datenstrukturen vom Typ *ArrayList* geparst. Die URL der HTML Templates der beiden äußeren Tabs (1 und 3) ist als String Wert gespeichert. Eine Lokalisierungs-Methode ist zuständig für das Erkennen der Spracheinstellung und das parsen der richtigen P-Listen. Die gewonnenen Datenstrukturen (ArrayList und URL String) werden den Tab Activities als Bundle übergeben und in den jeweiligen Activities aus dem Bundle wieder gewonnen, so dass mit den Daten gearbeitet werden kann. Eine Custom Tab Manager Klasse verwaltet die Tabs und die reagiert auf Tab-Ereignisse (Selektieren eines Tabs) und Wischgesten zum Wechseln der Seiten (präziser: der Fragmente).

Die äußeren beiden Tabs (*Tab1Activity.java* & *Tab3Activity.java*) implementieren jeweils' ein Fragment mit einer Web View. Eine Helferklasse (HtmlWorker.java) parst den kompletten Inhalt des darzustellenden HTML Templates in einen String, den die Web View als HTML Inhalt darstellt. Die String-Umwandlung ist notwendig, da WebViews in Android keine andere Methode und damit Möglichkeit haben, HTML Seiten, die relative Verweise beinhalten darzustellen. Die einzige Methode die das kann, *loadDataWithBaseURL()*, erwartet mehrere Strings als Übergabeparameter:

wovon der erste Parameter die Basis URL angibt und der zweite den Inhalt als eben String.
Ein Custom *WebViewClient* ist zuständig für die korrekte Delegierung von Touch Ereignissen auf Telefonnummern, Homepages, E-Mail Adressen und Google Maps Links zu den entsprechenden Apps.

Etwas komplizierter ist die Umsetzung der Speisekarte im zweiten Tab (*Tab2Activity.java*). Die Speisekarte wird als scrollbare Liste in einem *ListFragment* (genauer: in einem *SherlockListFragment*) dargestellt. Die Liste ist die *ArrayList* aus dem Bundle.
In einem Custom *ArrayAdapter*, der das *Filterable* Interface implementiert, wird die gesamte Liste mit der Speisekarte erstellt. Dabei wird zwischen Separator-Zelle für die Kategorie Einträge und eigentlicher Zelle, die eine Speise darstellt, unterschieden. Separator-Zelle und Speise-Zelle sind eigens definierte XML Templates, die an den entsprechenden Zellpositionen in der Liste eingesetzt (= *inflate*) werden.

In der *ActionBar* der Activity befindet sich eine *EditBox* in die der Nutzer Suchbegriffe eingeben kann. Ein *TextWatcher* lauscht nach Änderungen in dem Suchfeld und ruft bei jeder Änderung eine *Filter getFilter()* Methode auf. In dieser Methode befindet sich ein relativ komplizierter Algorithmus, der die komplette Speisekarte nach Treffern des Suchbegriffs filtert und auch in der Lage ist, mehrere Suchbegriffe zu unterscheiden. Die Unterscheidung von Suchbegriffen wird entsprechend vorher vereinbarter Trennzeichen unternommen. Diese Zeichen sind aus dem Set [„.,; “]. Der Filter „*tokenize*t“ also den gesamten Suchbegriff und iteriert über jedes Token. Untersucht werden Kategorie-Name und -Beschreibung in einer Separator-Zelle und Speise-Name und Speise-Zutaten in einer Speise-Zelle. Bei einem Treffer in einer Kategorie, werden alle Speisen dieser Kategorie in die Filterergebnis-Liste übernommen. Die Filterergebnis-Liste ist Rückgabewert der *performFiltering()* Methode.
TextWatcher und Filter sind Teil der *ArrayAdapter* Klasse. Jede Texteingabe in die *EditBox* erzeugt eine neue Filterergebnis-Liste, die maximal der gesamten Speisekarte entspricht (wenn die *EditBox* leer ist) und minimal leer ist (wenn ein Suchbegriff nicht gefunden werden kann). Aufbauend auf der Filterergebnis-Liste generiert der *ArrayAdapter* die Speisekarte. Hier greift wieder ein relativ komplizierter Algorithmus, der die Zelleinträge farblich markiert, wenn ein Suchbegriff auf einen Eintrag zutrifft. Die Logik des Filterns und des Nachkolorierens kann aus den mit Javadoc-Kommentaren versehen Klassen und Methoden im Quelltext viel leichter verstanden werden, als es an dieser Stelle ohne das Einfügen von Quelltext möglich ist. Das Android Projekt ist durchgängig mit Javadoc Kommentaren erklärend begleitet.

Abbildung 35: Tab 1 Android Projekt (Quelle: AVDM Eclipse)

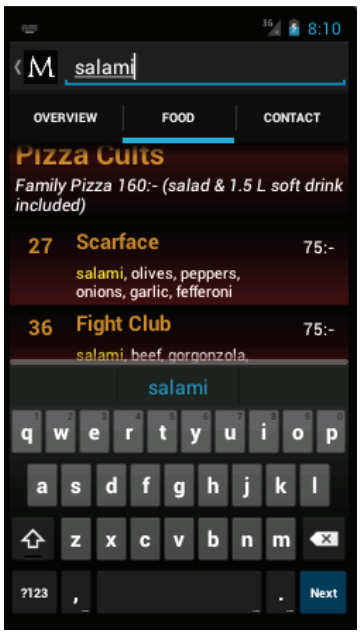

Abbildung 36: Tab 2 Android Projekt (Quelle: AVDM Eclipse)

5.5.2 iOS

Das Analogon zur *MainActivity* in Android ist die *main.m* in iOS – ein Einstiegspunkt der App, nachdem diese über das App Icon gestartet wurde. In der *main.m* befindet sich die main()-Methode, die die *UIApplicationMain* Funktion aufruft. Diese Funktion erstellt ein Objekt der Application und ein Objekt der App Delegate (*SWAppDelegate*). In der App Delegate wird das Fenster, in dem sich der Inhalt der App befindet instantiiert und einige Konfigurationen vorgenommen (laden/erstellen der User Defaults, Synchronisieren mit dem Server und ggf. das Data Access Object aktualisieren).

Die SWAppDelegate alloziert vier View Controller – einen für jeden Tab. Analog zum Android Projekt sind die beiden äußeren Tabs (1 und 4) *UIWebViews*, der zweite und dritte Tab sind als *UITableView* umgesetzt. Die Views sind jeweils im Interface Builder (IB) erstellt worden. Ebenfalls im IB ist das Layout einer einzelnen Zelle (*SWTableViewCell*) beider Table Views entworfen worden (vgl. Abbildung 30). Auf diese Zelle wird in der *cellForRowAtIndexPath* Methode als zu verwendende Zell verwiesen.
Das Layout einer Header-Zelle (analog dem Separator im Android Projekt) wird dynamisch in der *viewForHeaderInSection* Methode berechnet, je nachdem ob der Header (also die Kategorie) eine Beschreibung hat und wie lang diese Beschreibung hinsichtlich der Anzahl der Zeilen ist. Eine separate Liste mit Header Objekten an genau den Indizes, an denen die Kategorie wechselt, sowie komplizierte Logik zur Bestimmung, wann eine Zelle in der Liste eine Header-Zelle ist, müssen, so wie im Android Projekt, nicht implementiert werden. Der Core Data *FetchResultsController* beinhaltet die Information bereits, nach welchem Attribut gruppiert (= Name der Kategorie)

wird. TableView Methoden wie *viewForHeaderInSection* arbeiten dann bereits mit dem korrekten View Objekt.

Die *SearchBar* zum filtern der Tabelle ist am oberen Ende der TableView arretiert. D.h. beim Scrollen scrollt sie mit der Tabelle mit aus der Sicht heraus. So wird im Fenster mehr Platz für die Liste geschaffen. Bei einem Text-Änderungs-Ereignis in der *SearchBar* wird ein neuer *FetchController* vom Data Access Object (*SWDataAccessObject*) angefordert mit dem SearchBar-Text als Übergabeparameter.
Im DAO wird ein *Predicate*, eine SQL-ähnliche Anfrage, an die Datenbank gestellt und eine nach dem *order* Attribut sortierte sowie nach der Kategorie gruppierte Ergebnisliste zurückgegeben.
Darauf wird die Tabelle neu geladen. Abbildung 37 zeigt die Tabelle mit *SearchBar* und zwei als Favoriten markierten Speisen.

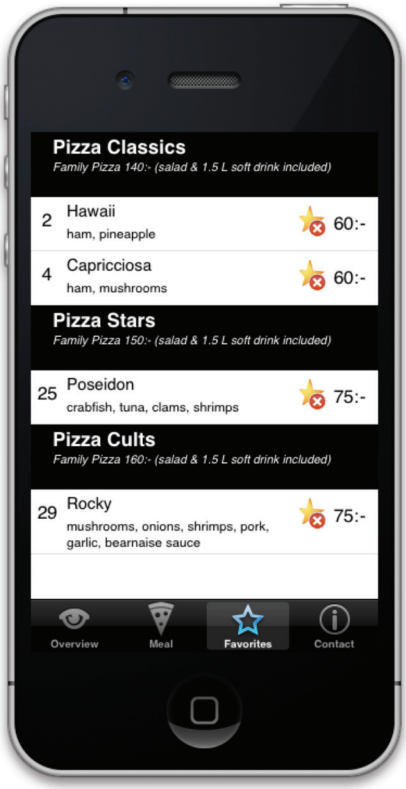

Abbildung 38: Favoriten Tab, iPhone (Quelle: XCode Simulator)

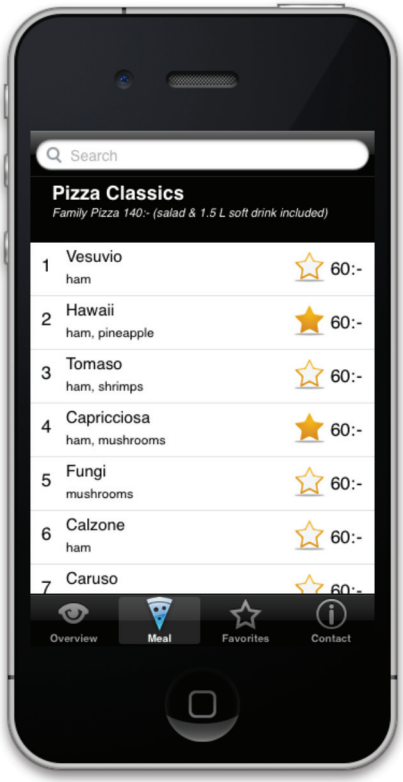

Abbildung 37: Speisekarte mit markierten Favoriten, iPhone (Quelle: XCode Simulator)

6 Zusammenfassung und Ausblick

Abbildung 39: QR Code Maestro Android

Abbildung 40: QR Code Maestro iPhone

Das Ziel der Arbeit war es, eine App für den Gastronomiesektor für die beiden Plattformen Android OS und iOS zu entwickeln, die den Austausch zwischen Gastronomie und Gast als Smartphone-Appplication übernimmt. Es wurde erörtert, welches Interesse ein Restaurant an solch einer App haben kann und welche Informationen das Restaurant betreffend für den Gast wichtig sind, bzw. welche Informationen der Gast wiederum dem Restaurant zukommen lassen kann. Als anschauliches Beispiel wurde die App für beide Systeme mit allen nötigen Funktionalitäten, die der Informationshinweg – vom Restaurant zum Gast – beinhalten muss, entwickelt. Ab dieser Entwicklungsstufe ist es zukünftig möglich, die App auch um einen Informationsfluss vom Gast zum Restaurant zu erweitern. Die nächste logische Funktionalität wäre eine Bestellmöglichkeit. Beide Versionen der App können über die oben abgebildeten QR Codes (Abbildung 39, Abbildung 40) kostenlos aus den jeweiligen Download-Plattformen heruntergeladen werden.
Es wurden darüber hinaus Unterschiede sowie Gemeinsamkeiten zwischen beiden Plattformen illustriert und auf Besonderheiten bei der Entwicklung für die jeweilige Plattform eingegangen.

Abschließend sei noch einmal hervorgehoben, dass sowohl die Arbeit als Ganzes - und unter besonderer Berücksichtigung, dass beide Apps als vollwertige Software auf dem Markt erhältlich sind - als erfolgreich zu Ende geführt, als auch die eingangs gestellten Ziele als erreicht betrachtet werden können.

Quellen

[AndNA1] Android. (NA. NA NA). *Android 4.1, Jelly Bean*. Abgerufen am 10. 8 2012 von Android about Jelly Bean: http://www.android.com/about/jelly-bean/

[And12] Android Developer. (July 2012). *Dashboards.* Von Android Developer: http://developer.android.com/about/dashboards/index.html abgerufen

[AndNA] Android. (NA. NA NA). *Introducing Android 4.0*. Abgerufen am 10. 8 2012 von Android ICS: http://www.android.com/about/ice-cream-sandwich/

[App111] Apple. (7. 6 2011). *Apple Mailing Lists.* Abgerufen am 18. 7 2012 von Apple Lists: http://lists.apple.com/archives/Objc-language/2011/Jun/msg00013.html

[App12] Apple. (NA. NA 2012). *Compare iPhone Models.* Abgerufen am 21. 7 2012 von Compare iPhone: http://www.apple.com/iphone/compare-iphones/

[App11] Apple Developer. (12. 10 2011). *iOS Frameworks*. Abgerufen am 18. 07 2012 von iOS Technology Overvire: http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html

[App112] Apple. (3. 10 2011). *Introduction to Garbage Collection.* Abgerufen am 18. 7 2012 von OS X Developer Library: https://developer.apple.com/library/mac/#documentation/cocoa/conceptual/GarbageCollection/Introduction.html#//apple_ref/doc/uid/TP40002431

[App113] Apple. (12. 10 2011). *The Objective-C Programming Language.* Abgerufen am 18. 7 2012 von OS X Developer Library: https://developer.apple.com/library/mac/#documentation/cocoa/conceptual/objectivec/introduction/introobjectivec.html

[BIT12] BITKOM. (29. Mai 2012). *Wettkampf der Smartphone-Plattformen.* Von BITKOM: http://www.bitkom.org/de/presse/30739_72316.aspx abgerufen

[can12] canalys. (3. Februar 2012). *Smart phones overtake client PCs in 2011.* Von canalys: http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011 abgerufen

[Pau12] Chlebek, P. (2012). *Praxis der User Interface-Entwicklung - Informationsstrukturen, Designpatterns, Vorgehensmuster.* Wiesbaden: Vieweg+Teubner Verlag / Springer.

[Dev122] Developer Android. (7. 18 2012). *Activity Class Overview*. Abgerufen am 7. 23 2012 von Android Develop Reference: http://developer.android.com/reference/android/app/Activity.html

[DevNA] Developer Android. (NA. NA NA). *Android Design Pattern.* Abgerufen am 21. 7 2012 von Android Developer Guidelines: http://developer.android.com/design/patterns/index.html

[Dev126] Developer Android. (26. 07 2012). *Input Events*. Abgerufen am 30. 07 2012 von Android Developer API Guides: http://developer.android.com/guide/topics/ui/ui-events.html

[Dev124] Developer Android. (26. 7 2012). *Intents and Intent Filters*. Abgerufen am 28. 7 2012 von Androdi Developer API Guides: http://developer.android.com/guide/components/intents-filters.html
[Dev125] Developer Apple. (07. 03 2012). *App States and Multitasking*. Abgerufen am 30. 07 2012 von iOS APp Programming Guide: http://developer.apple.com/library/ios/#DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html
[Dev123] Developer Apple. (15. 12 2012). *Apple URL Scheme Reference*. Abgerufen am 28. 7 2012 von iOS Developer Library: http://developer.apple.com/library/ios/#featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html
[Dev12] Developer Apple. (07. 03 2012). *iOS Human Interface Guidelines*. Abgerufen am 23. 7 2012 von iOS Developer Library: http://developer.apple.com/library/ios/#DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html
[Dev127] Developer Apple. (NA. NA 2012). *Mac Developer Library*. Abgerufen am 18. 8 2012 von Apple Developer: http://developer.apple.com/library/mac/navigation/
[Dev121] Developer Apple. (16. 02 2012). *Model-View-Controller*. Abgerufen am 23. 07 2012 von Cocoa Core Competencies: https://developer.apple.com/library/mac/#documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html
[Dev10] Developer Apple. (24. 3 2010). *Property List Programming Guide*. Abgerufen am 29. 7 2012 von Mac Developer Library: https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html
[Dev11] Developer Apple. (12. 10 2011). *Safari Web Content Guide - Configuring Web Applications*. Abgerufen am 17. 8 2012 von iOS Developer Library: http://developer.apple.com/library/ios/#documentation/AppleApplications/Reference/SafariWebContent/ConfiguringWebApplications/ConfiguringWebApplications.html
[Mar11] Gargenta, M. (2011). *Einführung in die Android Entwicklung* (1. Auflage 2011 Ausg., Bd. 1). (L. Schulten, Übers.) Köln, Deutschland: O'Reilly Verlag.
[Gar12] Gartner. (15. 02 2012). *Gartner Newsroom - Press Resleases.* Von Gartner: http://www.gartner.com/it/page.jsp?id=1924314 abgerufen
[Goo121] Google. (15. Mai 2012). *Google Mobile Ads.* Von Google Mobile Ads Blog: http://googlemobileads.blogspot.de/2012/05/new-research-shows-6-countries-are.html abgerufen
[Goo12] Google. (May 2012). *Our Mobile Planet.* Abgerufen am 10. July 2012 von Thinkwithgoogle: http://www.thinkwithgoogle.com/mobileplanet/de/
[Goo124] Google. (1-3 2012). *Our Mobile Planet Graph.* Abgerufen am 11. 7 2012 von Our Mobile Planet: http://www.thinkwithgoogle.com/mobileplanet/de/graph/
[Goo123] Google. (May 2012). *Our Mobile Planet: Sweden (PDF).* Von Our Mobile Planet: http://services.google.com/fh/files/blogs/our_mobile_planet_sweden_en.pdf abgerufen

[Goo122] Google. (2012). *Unser mobiler Planet: Deutschland Bericht (PDF).* Von Thinkwithgoogle Downloads: http://services.google.com/fh/files/blogs/our_mobile_planet_germany_de.pdf abgerufen
[Hei12] Heise. (16. 04 2012). *Heise Newsticker.* Von Heise: http://www.heise.de/newsticker/meldung/Bitkom-Jeder-dritte-Deutsche-hat-ein-Smartphone-1526048.html abgerufen
[Adr11] Kalenda, A. K.-H. (12. 8 2011). *ZDNet / News*. Abgerufen am 17. 8 2012 von ZDNet: http://www.zdnet.de/41555646/nokia-manager-das-app-konzept-ist-veraltet/
[Lee12] Lee, W.-M. (2012). *Beginning iOS 5 Application Development.* Indianapolis, USA: John Wiley & Sons Inc.
[Mos09] Mosemann, H., & Kose, M. (2009). *Android - Anwendungen für das Handy-Betriebssystem erfolgreich programmieren.* München Wien: Carl Hanser Verlag München Wien.
[JTM12] Mudge, J. (11. 7 2012). *Native App vs. Mobile Web App: A Quick Comparison*. Abgerufen am 17. 8 2012 von sixrevisions: http://sixrevisions.com/mobile/native-app-vs-mobile-web-app-comparison/
[Net11] Netzwelt. (24. 12 2011). *Hintergrund zur ARM-Architektur*. Abgerufen am 18. 7 2012 von Netzwelt: http://www.netzwelt.de/news/90025-netzwelt-wissen-prozessoren-arm-familie.html
[Sad10] Sadun, E. (2010). *Das große iPhone Entwicklerbuch.* München, Germany: Addison-Wesley Verlag.
[Eri12] Sadun, E. (2012). *The iOS 5 developer's cook: core concepts and essential recipes for iOS programmers* (3rd Edition Ausg., Bd. 2ns Printing). Bosten, MA 02116, USA: Pearson Education, Inc.
[Sta10] Stapelkamp, T. (2010). *Interaction- und Interfacedesign - Web-, Game-, Produkt- und Servicedesign Usability und Interface als Corporate Identity.* Berlin Heidelberg: Springer-Verlag .
[tec11] techotopia. (11. 2 2011). *The iPhone OS Architecture and Frameworks*. Abgerufen am 11. 8 2012 von techotopia: http://www.techotopia.com/index.php/The_iPhone_OS_Architecture_and_Frameworks
[Wik12] Wikipedia. (16. 7 2012). *List of displays by pixel density*. Abgerufen am 23. 7 2012 von Wikipedia: http://en.wikipedia.org/wiki/List_of_displays_by_pixel_density
[wik12] wikipedia. (28. 7 2012). *Ockhams Rasiermesser*. Abgerufen am 13. 8 2012 von wikipedia: http://de.wikipedia.org/wiki/Ockhams_Rasiermesser

Abkürzungsverzeichnis

Abkürzung	Erklärung
ACID	Atomicity, Consistency, Isolation, Durability'
ADT	Android Development Tool
API	Application Programming Interface
App	Application
ARC	Automatic Reference Counting
ARM	Advanced RISC Machines
AVDM	Android Virtual Device Manager
Bez.	Bezeichnung
BS	Betriebssystem
bspw.	beispielsweise
CSS	Cascading Style Sheets
DBMS	Datenbankmanagementsystem
entspr.	entsprechend
ER (-Modell)	Entity Relationship (Modell)
Etc	et cetera, und so weiter
ff.	folgende
ggü.	gegenüber
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
iOS	iPhone Operating System
JVM	Java Virtual Machine
LIFO	Last In First Out
Linux	ein unix-ähnliches Betriebssystem
MRC	Manual Reference Counting
MVC	Model View Controller
n.a., NA	nicht angegeben
OS	Operating System, Betriebssystem
PHP	Hypertext Preprocessor
PNG	Portable Network Graphics
RISC	Reduced Instruction Set Computer
SDK	Software Development Kit
SHA	Secure Hash Algorithm
u.a.	unter anderem
UI	User Interface
UML	Unified Modeling Language
UNIX	ein Betriebssystem
URL	Uniform Resource Locator
vgl.	vergleiche
XML	Extensible Markup Language

Tabellenverzeichnis

Abbildungsverzeichnis

Verwendete Software

- XCode 4.3.3
- iPhone 4.3 Simulator
- Eclipse IDE for Java Developers (Juno Release)
- Eclipse IDE for Java Developers (Indigo Service Release 2)
- Android DDMS
- ADT
- Android Virtual Device Manager
- astah community 6.5

Weblinks

- ActionBarSherlock (http://actionbarsherlock.com/)
- android.support.v4.app (http://developer.android.com/reference/android/support/v4/app/package-summary.html)
- ADT Pluging (http://developer.android.com/tools/sdk/eclipse-adt.html)
- Eclipse IDE (http://www.eclipse.org/downloads/)
- iPhone SDK (http://developer.apple.com/iphone)
- XCode IDE (https://developer.apple.com/xcode/)
- Android Developer Pages (http://developer.android.com/index.html)
- Apple Developer Pages (https://developer.apple.com/)
- Stackoverflow (http://stackoverflow.com)
- Android Developer Guidelines (http://developer.android.com/design/index.html)
- iOS Developer Guidelines (http://developer.apple.com/library/ios/#DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html)
- Maestro Homepage (http://www.maestrovaxjo.se/index_en.html)