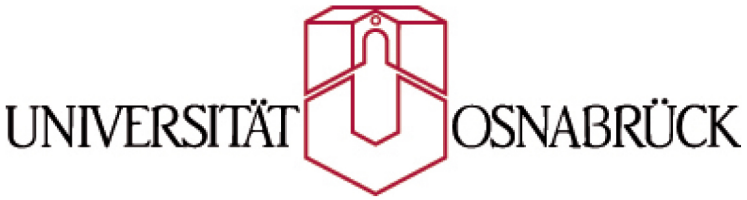

INSTITUT FÜR INFORMATIK
AG MEDIENINFORMATIK

Masterarbeit

Eine iPhone-Applikation für ein mobiles soziales Netzwerk mit Augmented-Reality Routing

Phillipp Bertram

Osnabrück, November 2012

Erstgutachter : Prof. Dr. Oliver Vornberger

Zweitgutachter : Jun.-Prof. Dr.-Ing. Elke Pulvermüller

Zusammenfassung

In dieser Arbeit wird die Konzeption und Implementierung von *iLocator* vorgestellt. Hierbei handelt es sich um eine in Objective-C für das iPhone geschriebene Applikation, welche die Eigenschaften eines sozialen Netzwerkes, sowie der *Augmented Reality* Technologie und *Routing* vereinigt. Sie verwaltet eine Liste mit Freunden, die anhand der Kontakte aus dem iPhone ermittelt werden und kann deren Standorte auf einer Karte anzeigen, sofern diese von den Freunden nicht verborgen werden. Außerdem können Treffpunkte mit Freunden geteilt werden, sodass diese ebenfalls auf der Karte sichtbar sind. Über die Routing-Funktion lässt sich der Anwender dann zu den gewünschten Orten navigieren. Eine Besonderheit stellt zudem die spezielle Kameraansicht dar, die im Sinne der *Augmented Reality* Technologie auch hier die Standorte der Freunde oder Treffpunke anzeigt.

Da letztendlich eine Client-Server-Architektur vorliegt, werden neben den zugrunde liegenden Technologien und Konzepten der App auch die des Servers behandelt.

Inhaltsverzeichnis

1 Einleitung

Im Zeitalter der Smartphones gibt es zahlreiche Applikationen, die auf die jeweiligen Bedürfnisse der Benutzer zugeschnitten sind. In dieser Arbeit wird die im Rahmen der vorliegenden These entwickelte iPhone Applikation (App) *iLocator* vorgestellt, die den aufsteigenden Trend der *Augmented Reality* Technologie mit den Eigenschaften eines sozialen Netzwerkes verbindet. Sie erlaubt dem Anwender sowohl interaktiv mit dessen Freunden zu kommunizieren, als auch deren geographische Position, sowie eigens gesetzte Standorte bzw. Treffpunkte – so *genannte Points of Interests* (POI)– auf einer Karte oder in einer speziellen Kamera-Perspektive in Echtzeit zu sehen und sich zu den jeweiligen Punkten navigieren zu lassen.

In dieser Einleitung werden zunächst für die App wichtige Begriffe definiert und schließlich das Ziel und der Aufbau dieser Arbeit vorgetellt.

1.1 Der Markt der iPhone Apps

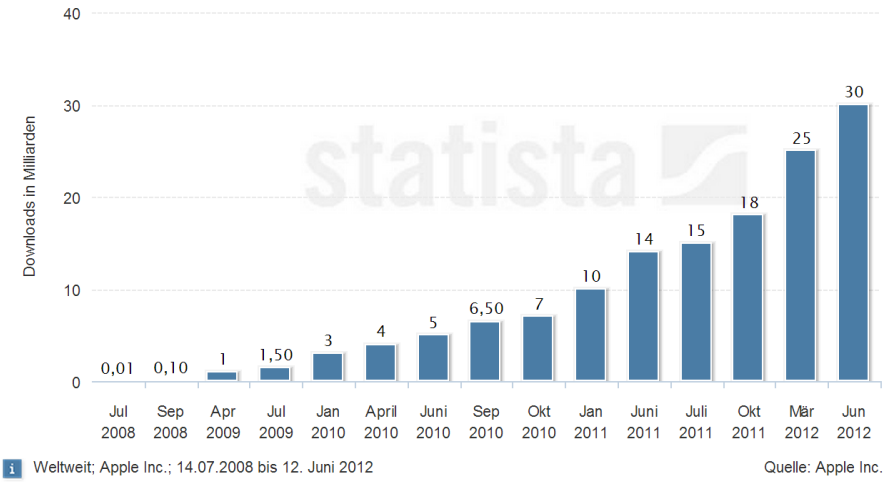

Abbildung 1.1: Kumulierte Anzahl der weltweit heruntergeladenen Anwendungen aus dem Apple App Store. Stand; 12. Juni 2012

Seit der Inbetriebnahme des *iTunes App Store* im Jahre 2008 wurden von Apple mehr als 700.000 Apps zum Download freigegeben[1]. Mit 30 Milliarden App Downloads kann

[1] techcrunch.com/2012/09/12/ios-app-store-boasts-700k-apps-90-downloaded-every-month/

bislang kein vergleichbarer App Store mithalten, wie das Diagramm in Abbildung 1.1 veranschaulicht. Sich dem aufstrebenden Markt der App-Entwicklung anzuschließen scheint nunmehr als logische Schlussfolgerung für viele Unternehmer und Entwickler.

Apple's mobile Geräte wie das iPhone, iPad oder der iPod Touch verwenden alle das gleiche Betriebssystem *iOS*. Es liegt seit dem September 2012 mit der Veröffentlichung des neuen iPhone 5 in der Version 6.0 vor und ist die mobile Variante seines bereits existierenden Cocoa-Frameworks namens Cocoa Touch.

1.2 Mobiles soziales Netzwerk

Abbildung 1.2: **Soziales Netzwerk**: „Eine abgegrenzte Menge von Personen, die über (soziale) Beziehungen miteinander verbunden sind."

Sehr allgemein definiert ist ein soziales Netzwerk „eine abgegrenzte Menge von Personen, die über (soziale) Beziehungen miteinander verbunden sind"[2]. Eine etwas spezifischere Definition liefern Boyd und Ellison [8]. Sie beschreiben ein soziales Netzwerk als webbasierten Service, der über drei wesentliche Merkmale definiert ist: Zum einen soll der Anwender die Möglichkeit haben, ein Profil (öffentlich oder halböffentlich) innerhalb eines abgegrenzten Systems anlegen zu können. Zum anderen muss er in der Lage sein, eine Liste von anderen Nuzern, mit denen er in diesem System verbunden ist, zu pflegen. Diese Listen müssen zu guter Letzt ebenfalls betrachtet und durlaufen werden können. Die Art und Weise, diese Merkmale und Verbindungen umzusetzen, ist dabei nicht festgelegt.

[2] `http://www.informatik.uni-oldenburg.de/~iug10/sn/html/content/definition.html`

Mittlerweile gibt es zahlreiche Plattformen, die diese Funktionalität bereitstellen. Die derzeitig wohl populärsten Beispiele sozialer Netzwerke sind *Facebook*[3], *MySpace*[4] und *Google+*[5]. Da die Definition dennoch sehr allgemein gehalten ist, daher auch keine konkrete Aussage über das Ziel oder den Zweck eines sozialen Netzwerks getroffen werden kann, handelt es sich bei den meisten derartiger Plattformen um reine Kummunikationsplattformen.

Der Begriff Mobilität in diesem Zusammenhang erweitert die obige Definition lediglich um die Anforderung, die Eigenschaften des sozialen Netzwerkes von unterwegs – zum Beispiel mit einem Smartphone – verwenden zu können. Laut Lee Humphreys stellen Mobiltelefone einen zunehmend wichtiger werdenden Kommunikationskanal dar, soziale Kontakte zu knüpfen [9]. Sie sind allgegenwärtig und in vielen Teilen der Welt vertreten. Im Jahre 2006 besaßen sogar mehr Amerikaner ein Mobiltelefon, als einen Internetanschluss. Mit dem immer größer werdenden Trend zum Smartphone und den einhergehenden neuen Möglichkeiten, wie der Internetanbindung oder einer Kamera, ist es nur eine logische Entwicklung, dass fast alle Dienste der Plattformen sozialer Netzwerke nun auch als App unterwegs nutzbar geworden sind.

1.3 Augmented Reality

Augmented Reality[6] – oder nur AR – ist die Bezeichnung einer Technologie, die reale Umgebung durch virtuelle, computergenerierte Objekte so zu erweitern, dass der Anscheind erweckt wird, sie seien Bestandteil der realen Welt [10], [14]. Die reale Welt wird dabei nicht vollständig ersetzt, wie es bei *Virtual Reality* der Fall ist, sondern nur ergänzt. Sportübertragungen im Fernsehen nutzen schon seit Jahren diese Technologie und haben erst kürzlich damit begonnen, in Echtzeit analytische Informationen zur selben Zeit dem normalen Bild zu überlagen, um dem Zuschauer somit mehr aktuelle Auskünfte zu geben.

Allerdings erfordert der Einsatz dieser Technologie leistungsstarke Systeme. Auch wenn die Anfänge von *Augmented Reality* bis in die fünfziger Jahre zurückgehen, hat sie erst seit kurzem ihren Durchbruch in die heutige Computerwelt geschafft. Das mag vor allem an dem enormen Fortschritt der mobilen Geräte wie Smartphones und Tablets mit integrierter Videokamera liegen: Apple hat mit seiner Veröffentlichung des iPhones, sowie Google mit seinem Betriebssystem Android einen großen Beitrag für diese Entwicklung

[3] `http://www.facebook.com`
[4] `http://de.myspace.com/`
[5] `https://plus.google.com/`
[6] Die Namenskreation ist im Allgemeinen auf Thomas Caudell (1990) zurückzuführen.

beigetragen. Die Anwender müssen nicht wie im Fernsehen auf die AR-Effekte warten, sondern diese können direkt in deren realer Umgebung eingesetzt werden (vgl. Abbildung 1.3).

Quelle: http://www.guardian.co.uk/technology/2010/mar/21/augmented-reality-iphone-advertising

Abbildung 1.3: Beispiel einer *location based* AR-Applikation auf dem iPhone.

Zur Abgrenzung zwischen Realität und Virtualität führte Paul Milgram (1994) das in Abbildung 1.4 dargestellte *Mixed-Reality-Continuum* ein, welches einen Bereich aufspannt, der von der realen Umgebung bis zur virtuellen Umgebung reicht [10]. Während auf der linken Hälfte des Kontinuums die reale Welt überwiegt, nimmt der Anteil der Virtualität auf der rechten Hälte einen größeren Stellenwert ein. Daher wird bei *Augmented Reality* im Gegensatz zur *Augmented Virtuality* nicht die virtuelle Welt durch reale Elemente erweitert, sondern die reale Umgebung durch virtuelle Objekte.

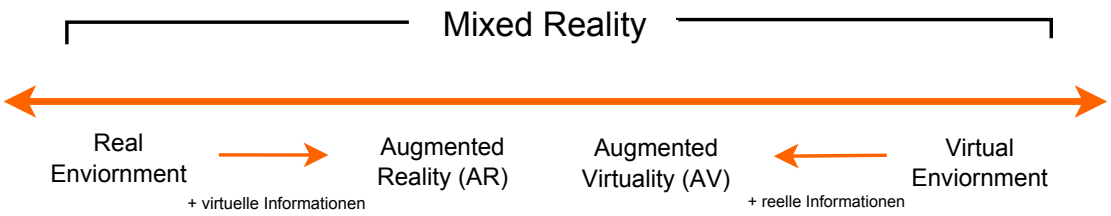

Abbildung 1.4: Milgram's *Reality-Virtuality-Continuum.*

Inzwischen bedienen sich viele Apps dieser Technologie. Eines der bekanntesten und am meist verbreiteten Apps ist *WikiTude*[7] – eine der ersten Mobile-Apps, die auf einem standortbasierten Ansatz der Augmented Reality basiert. Mittlerweile bietet WikiTude SDK an, welches für seine eigene App gegen bestimmte Auflagen eingesetzt werden kann.

An dieser Stelle soll kurz erwähnt werden, dass neben stanortbasierten Ansätzen auch noch weitere Ansätze existieren. Während alle virtuellen Inhalte physische Umgebungen oder Objekte erweitern und demnach standortbasierte Informationen bereitstellen, sind 3D-Objekte im Hinblick auf ihre Geometrie (Position und Ausrichtung) und in idealer Weise auch auf ihre Photometrie genau zu ihrer physischen Umgebung registriert. Diese Registrierung erfordert zusätzlich adäquate Technologien zum Tracken (z.B. Marker) derartiger Objekte. Da diese Arbeit allerdings die Technologie der standortbasierten Augmented Reality verwendet, wird an dieser Stelle nur erwähnt, dass auch Ansätze in dieser Richtung existieren. Ein bekanntes Beispiel für ein Marker-basiertes Tracking Verfahren verwendet beispielsweise das von Sony entwickelte Spiel *EyePet*[8].

1.4 Routing und Navigation

Ein weiterer Begriff soll für das Verständnis des Themas dieser Arbeit in diesem Abschnitt näher erklärt werden.

Unter *Routing* wird die Berechnung von Wegstrecken auf einem logischen Netzwerk zur Lösung von räumlichen Fragestellungen verstanden. Dieses Netzwerk umfasst in der Regel sämtliche Geodaten wie Straßennetze, Fußwege, etc, eines Bereichs. Außerdem verfügen alle Wegsegmente im Allgemeinen über zusätzliche Kostenfaktoren (Länge, Zeit) oder andere Informationenm die diese näher beschreiben (z.B. Einbahnstraße), die bei der Berechnung der Route mit berücksichtigt werden.

Im Zusammenhang mit Routing wird oft auch der Begriff Navigation verwendet. Navigation impliziert zum einen die genaue Ermittlung der eigenen Position und zum anderen eine Routenplanung auf dem direktesten Weg zum Ziel. Hierzu werden die Verkehrsbestimmenden Regeln, wie Einbahnstraßen und Abbiegevorschriften, der Wegsegmente herangezogen, sodass die geplante Route auch wirklich abgefahren oder abgelaufen werden kann. Navigationssysteme geben dazu in der Regel zusätzlich noch Instruktionen und leiten den Anwender zum gewünschten Ziel.

[7] `http://www.wikitude.com/`
[8] `http://www.eyepet.com/home.cfm?lang=de_DE`

1.5 Ziel und Aufbau der Arbeit

Smartphones nehmen einen immer größeren Stellenwert in unserer Gesellschaft ein. Das liegt zum einen an ihrer leistungsstarken Hardware, aber auch an den vielfältigen Funktionen der Geräte, wie Internetanbindung, GPS für Navigation und die Videokamera. Aber auch neue Technologien wie Augmented Reality reizen mit ihren Anwendungsmöglichkeiten. Die in Abschnitt 1.1 dargestellte Statistik über den Verlauf der Anzahl der heruntergeladenen Apps verdeutlicht diesen aufsteigenden Trend. Die Tatsache, dass mit der heutigen Mobilfunkstandards wie 3G oder neuerdings auch LTE das mobile Internet immer besser nutzbar wird, stellt sicherlich ebenfalls eine Ursache für diese Tendenz dar.

Zudem zeigen soziale Plattformen wie Facebook, dass die Bedeutung sozialer Kontakte bzw. Netzwerke immer größer wird und deren Dienste von einer Vielzahl an Menschen auf der Welt tagtäglich von zu Hause aus – oder zunehmend von unterwegs aus – genutzt werden. Alleine die Herausforderung, all diese Eigenschaften mit den einhergehenden Restriktionen der Mobilfunkgeräte zu vereinen, ist Motivation genug, eine derartige App zu entwickeln. Daher soll das Ziel dieser Arbeit eine iPhone App sein, die die Eigenschaften eines sozialen Netzwerks in Verbindung mit der Augmented Reality Technologie und einem Routing bzw. Navigationsmechanismus vereint.

Das hierauf folgende Kapitel 2 behandelt zunächst allgemeine Grundlagen zum Verständnis der Entwicklung der iPhone App *iLocator*. Es werden alle Komponenten und verwendete Technologien beschrieben, die für die Umsetzung dieses Projekts benötigt wurden.

Das Hauptaugenmerk liegt allerdings im darauffolgenden Kapitel 3. Hier werden die Konzepte und die Umsetzung anhand zahlreicher Code-Ausschnitte näher erläutert. Dazu gehört zum einen die Implementierung der eigentlichen iPhone App mit den dort verwendeten Komponenten, sowie auch der Entwurf einer geeigneten Serverschnittstelle, mit der die Kommunikation zwischen den Clients erfolgen kann.

Abschließend werden noch weiterführende Überlegungen zu der App sowie das Fazit dieser Arbeit vorgestellt.

2 Allgemeine Grundlagen

Die vorliegende Arbeit verwendet Elemente aus diversen Bereichen der Informatik. Diese erstrecken sich von der Geoinformatik (Rechnen mit und Visualisierung geographischer Objekte) über webbasierte Dienste (Kommunikationsschnittstelle zwischen iPhone und Server), Datenbanken (Speicherung und Verwaltung von Benutzerdaten), Computergrafik (Umsetzung von *Augmented Reality*), sowie der Entwicklung einer ganzen Applikation auf einem Smartphone.

Dieses Kapitel soll die grundlegenden Informationen und Technologien zur Entwicklung von *iLocator* vermitteln, bevor in Kapitel 3 die konkrete Implementierung der Applikation diskutiert wird. Zunächst erfolgt eine Einführung in die Entwicklung von Apps auf dem iPhone (Abschnitt 2.1). Die darauffolgenden Abschnitte befassen sich mit den Grundlagen aus der Geographie (Abschnitt 2.3), Computergrafik (Abschnitt 2.4), der Datenhaltung auf dem iPhone (Abschnitt 2.6), sowie Webservices und PHP (Abschnitt 2.7).

2.1 App-Entwicklung auf dem iPhone

Das Entwickeln von iOS-Applikationen unterscheidet sich in vielen Punkten von Desktop-Anwendungen. Eines der Unterschiede ist, dass auf iOS immer nur eine Applikation gleichzeitig aktiv und auf dem Display angezeigt werden kann. Seit iOS 4 gibt es allerdings die Möglichkeit, diese auch im Hintergrund, verbunden mit extra Code und begrenzten Ressourcen, weiterlaufen zu lassen. Ferner grenzt iOS die Zugriffsmöglichkeiten der Anwendung streng ein. Lediglich auf das eigene Dateisystem der Anwendung ist der Zugriff gestattet. Dieser Bereich wird auch *Sandbox* genannt. Schließlich stellt auch die begrenzte Bildschirmgröße und Auflösung einen großen Unterschied zur Entwicklung von Desktop Applikationen dar, die derzeit bei dem iPhone 4 Retina Display eine Auflösung von 640×960 Pixeln und bei dem neuesten Modell iPhone 5 eine Auflösung von 1136×640 Pixeln misst. Der größte Unterschied liegt allerdings in den begrenzten System Ressourcen.

Erst nach erfolgreicher Registrierung als *Developer* bei Apple[9] kann das dazugehörige iOS SDK zusammen mit der äußerst umfangreichen Entwicklungsumgebung *XCode* heruntergeladen werden. Hierbei stehen unterdessen verschiedene Developer-Programme zur Auswahl. Das kostenfreie Programm ist für Einsteiger gut geeignet. Es bietet allerdings zum Testen des Codes lediglich einen Simulator, der wiederum keine hardwareabhängi-

[9] `developer.apple.com/devcenter/ios/index.action`

gen Features wie den Beschleunigungssensor oder die Kamera unterstützt. Außerdem ist der Vertrieb von Applikationen im App Store nicht erlaubt. Das Standard-Programm hingegen erlaubt den Vertrieb, kostet dafür $99 im Jahr. Mit diesem Programm ist der Entwickler außerdem in der Lage, seinen Code auch auf einem iOS Gerät zu testen. Das größte Enterprise-Programm ist für Unternehmen entworfen, die proprietäre iOS Applikationen anfertigen, an denen mehr als nur ein Entwickler arbeitet. Dieses kostet jährlich $299. Obendrein wird für die Entwicklung ein Intel-basierter Macintosh mit dem derzeitigen OS X 10.8 Mountain Lion Betriebssystem benötigt.

Dieser Abschnitt gibt einen groben Einblick in die Entwicklung von Applikationen auf dem iPhone. Es werden nur die grundlegenden Elemente erläutert, da dies sonst den Rahmen dieser Arbeit überschreiten würde. Für einen tieferen Einblick bietet die Entwickler-Website von Apple[10] einen guten Einstieg.

2.1.1 Objective-C

Die Sprache *Objective-C*, oder einfach nur *ObjC* genannt, ist eine reflexive, objektorientierte Programmiersprache, die entwickelt worden ist, um anspruchsvolle, objektorientierte Programmierung zu ermöglichen. Laut dem *Objective-C Programming Guide*[11] ist sie als kleine, aber leistungsstarke Menge von Erweiterungen der Standart ANSI Programmiersprache *C* definiert. Ihre Erweiterungen bezüglich *C* basieren hauptsächlich auf *Smalltalk*, einer der ersten objektorientierten Programmiersprachen. Objective-C bildet somit eine Obermenge von *C*, es kann also jedes in *C* geschriebenes Programm mit einem ObjC-*Compiler* übersetzt werden. Heute wird es in erster Linie auf Apple's Mac OS X und iOS verwendet: beide Umgebungen basieren auf dem *OpenStep*-Standard, sind mit ihm allerdings nicht konform. ObjC ist die primäre Sprache der Apple Cocoa-API und war ursprünglich die Hauptsprache auf *NeXTStep OS*.

Die folgenden Abschnitte greifen wichtige Eigenschaften dieser Sprache auf und geben anhand einiger Beispiele einen Einblick in die Programmiersprache Objective-C.

Objekt Zugriffe Anders als in *Java* werden die Methoden der Objekte nicht über eine Punkt-Notation angesprochen, sondern als Message an das Objekt geschickt. Folgender Java-Code:

[10] `http://developer.apple.com`
[11] Quelle siehe [7]

```
Class anObject = new Class();
anObject.methodWithoutParameter();
anObject.methodWithParameter(5, 10);
```

sieht in ObjC also entsprechend so aus:

```
Class *anObject = [[Class alloc] init]; // alternative [Class new];
[anObject methodWithoutParameter];
[anObject methodWithParameterA:5 andParameterB:10];
```

Speicherverwaltung Wie bereits erwähnt, muss man sich bei der Programmierung auf dem iPhone selber um die Speicherverwaltung kümmern, da es – anders als in Java – keinen Garbage Collector gibt. Daher müssen alle allokierten Objekte auch wieder freigegeben werden. Eine Faustregel besagt, dass zu jedem `alloc` ein `release` folgen muss, wie in dem folgenden Beispiel gezeigt ist.

```
String *myString = [[NSString alloc] initWithFormat:@"a String"];
// ... some code ...
[myString release];
```

Objekte können in ObjC über `autorelease` automatisch wieder freigegeben werden. Allerdings sollte das nur in wenigen Fällen angewendet werden, da es hier leicht zu Fehlern kommen kann. Der folgende Code zeigt, wie `autorelease` eingesetzt wird.

```
String *myString = [[NSString alloc] initWithFormat:@"a String"];
[myString autorelease];
```

Strings in Objective-C müssen immer mit einem `@` versehen werden. So wird zwischen ObjC- und C-Strings unterschieden, bei denen wie gewohnt das Zeichen entfällt.

```
NSLog(@"Objc String");
printf("C String");
```

Mit dem Realease von iOS 5.0 wurde das optionale Feature *Automatic Reference Counting* (ARC) eingeführt, welches die Speicherverwaltung automatisch verwaltet[12].

[12] `http://developer.apple.com/library/mac/#releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html`

Properties und der Aufbau einer Klasse Eine noch zu erwähnende Eigenschaft von ObjC sind die sogenannten Properties. Guter Programmierstil ist, wenn Attribute nicht direkt, sondern über *Accessoren* modifiziert und abgerufen werden. Um nicht für jedes Attribut einen solchen *Getter* und *Setter* schreiben zu müssen, gibt es die *Properties*.

Das folgende Beispiel zeigt, wie in etwa eine Pseudo-Klasse aufgebaut ist. Eine Klasse besteht immer aus einem *Interface* (`Class.h`) und einer *Implementation* (`Class.m`). Für die meisten Anwendung, die für das iPhone programmiert werden, ist es sinnvoll, die Klasse von `NSObject` erben zu lassen. `NSObject` ist wie `Object` in Java eine Basisklasse, von der alle Klassen abstammen sollten, die iPhone-spezifisch sind. Sie enthält die essenziellen Methoden zur Speicherverwaltung (wie `dealloc`) und noch weitere Eigenschaften.

```
1  // Class.h
2  @interface Class : NSObject {
3     String *aString;
4  }
5
6  @property(retain) String *aString;
7
8  + (NSString*)classMethod;
9
10 - (void)methodWithoutParameter;
11 - (void)methodWithParameterA:(String*)str andParameterB:(int)integer;
12
13 @end
```

Listing 2.1: Beispiel einer Header-Datei der Klasse `Class` mit einem Attribut, einer Property, einer Klassenmethode, sowie zwei Objektmethoden

Attribute werden in der *Header*-Datei innerhalb der geschweiften Klammern deklariert. Danach besteht die Möglichkeit, zum einen Properties anzulegen und zum anderen Methoden-Signaturen aufzuführen, wie es schon aus *C* bekannt ist. Properties werden mit dem Schlüsselwort `@property` eingeleitet, gefolgt von wenigen Einstellungsmöglichkeiten innerhalb der Klammer (z.B. `retain`). Anschließend muss der Datentyp und der Name des Attributs aufgeführt werden, der der Property zugewiesen wird. Eine weitere Eigenschaft der Properties ist, dass diese Attribute - wie in Java - auch per Punkt-Notation angesprochen werden können.

Die Implementierung geschieht wie in *C* in einer seperaten `.m` Datei (vgl. Listing 2.2). Hier müssen die Properties noch synthetisiert werden (Zeile 5). Das bedeutet nur, dass dem Compiler mit dem Schlüsselwort `@synthesize` kenntlich gemacht wird, dass er die Getter- und Setter-Methoden beim Kompilieren einfügen soll.

Die Methode `dealloc` wird immer dann aufgerufen, wenn ein Objekt dieser Klasse zerstört wird. Hier sollte der gesamte Speicher freigegeben werden, den das Objekt allokiert hat.

```
#import Class.h

@implementation Class
@synthesize aString;

+ (void)classMethod {
  return @"Hello World!";
}

- (void)methodWithoutParameter {
   // do something
}

- (void)methodWithParameterA:(String*)string andParameterB:(int)integer {
  // do simething with the parameters
}

- (void)dealloc {
   [aString release];
   [super dealloc];
}

@end
```

Listing 2.2: Die zur Klasse `Class` gehörende Implementierung

Protokolle Was Interfaces in Java sind, stellen Protokolle in Objective-C dar. Sie werden mit dem Schlüsselwort `@protocol` gefolgt von dem Namen des Protokolls eingeleitet und mit dem bekannten `@end` abgeschlossen. Dazwischen können sämtliche Methoden-Deklarationen, sowie Attribute oder Properties aufgelistet werden, wie das nachfolgende Beispiel zeigt.

```
@protocol TheProtocoll <NSObject>
- (void)methodToImplement;
@end
```

Soll eine Klasse bestimmten Protokollen konform sein, so müssen diese beim Klasseninterface in spitzen Klammern hinter den Namen aufgelistet werden.

```
1 @interface Class : NSObject <TheProtocol1>
2 // ...
3 @end
```

2.1.2 View Controller

Ein *View Controller* ist ein wichtiges Bindeglied zwischen den Daten der App und ihrer visuellen Darstellung. Traditionell nimmt er die Position des *Controllers* im *Model-View-Controller* (MVC) Paradigma ein, übernimmt in der Regel aber noch weitere Aufgaben. *View Controller* können eigene Klassen (`NSObject` Unterklassen) sein, werden jedeoch meistens von diversen generischen *Controller* Klassen des `UIKit` Frameworks abgeleitet – wie `UIViewController` oder `UINavigationViewController`. Jedes Mal, wenn eine iOS-Applikation eine Benutzeroberfläche anzeigt, wird dessen Inhalt von einem dieser *View Controller* verwaltet.

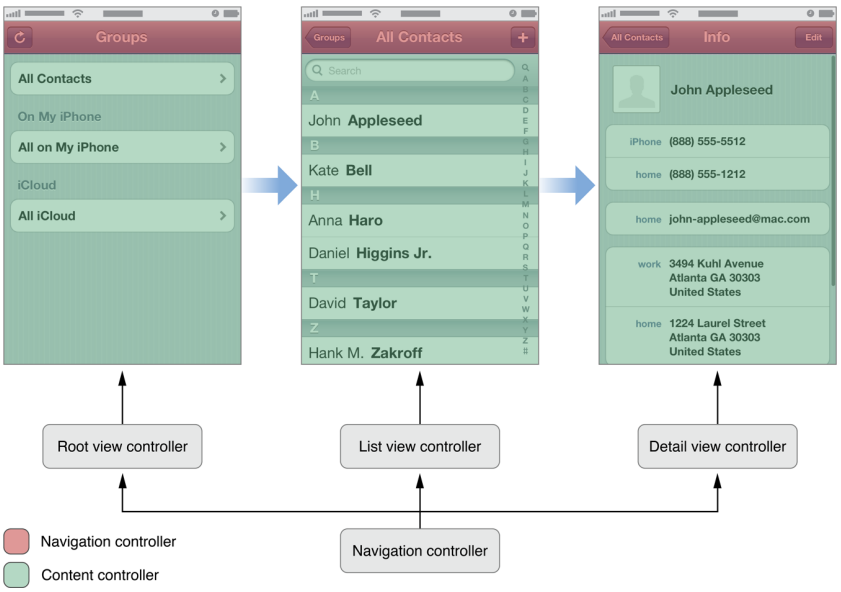

Abbildung 2.1: Mit einem *Navigation Controller* durch hierarchische Daten navigieren. Quelle: [6]

2.1.3 Komponenten einer Applikation

Kompilierte iPhone-Applikationen „leben“ in sogenannten Application-Bundles. Dies sind nichts weiter als Ordner mit einer `.app` Endung, die eben alle Ressourcen eines

Programms enthalten. Die Hierarchie dieser Bundles ist einfach: alle Materialien befinden sich in der obersten Ebene des Ordners. Es können auch Unterordner erzeugt werden, die das Projekt besser strukturieren und organisieren, allerdings folgen diesen eigenen Unterordnern keine Standards. Das iPhone SDK unterstützt hierfür die `NSBundle` Klasse. Diese Klasse stellt unter anderem Methoden für den Zugriff auf diese Dateien bereit.

Die *Executable* Datei befindet sich ebenfalls in der obersten Ebene des *Bundles*. Sie enthält Ausführungsrechte, sodass eine Applikation nur dann geladen und gestartet werden kann, wenn es mit einem offiziellen Entwickler-Zertifikat ausgewiesen wurde.

Wichtiger Bestandteil einer Applikation ist die *Property List* (`Info.plist`). Sie beinhaltet *Key-Value* Daten für viele unterschiedliche Zwecke und kann diese entweder in einem lesbaren, textbasierten oder komprimierten, binären Format abspeichern. Hier wird z.B. spezifiziert, welches die auszuführende Datei ist oder wie der Name des *Bundles* ist.

2.1.4 Application Life Cycle

Der Lebenszyklus konstituiert die Abfolge der Ereignisse einer Applikation, die zwischen Programmstart und -ende auftreten. An bestimmten Punkten der Laufzeit sendet das `UIKit` Framework Nachrichten an das *Application Delegate* Objekt, sodass entsprechend darauf reagiert werden kann (vgl. Abbildung 2.2).

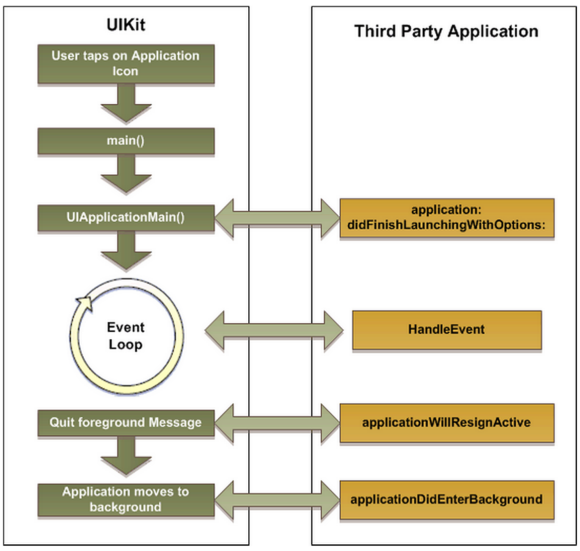

Abbildung 2.2: Der *Life Cycle* einer iOS Applikation.

2.2 Routing Web-Service von CloudMade

CloudMade ist ein im Jahr 2007 gegründetes Unternehmen, das APIs, gerenderte Karten, positionsbasierte Web-Services auf Basis von *OpenStreetMap*[13], sowie diverse Mobile und Desktop-Bibliotheken anbietet. Einer dieser Web-Services führt eine Routenberechnung zwischen zwei Geo-Koordinaten durch[14]. Der Dienst ist kostenlos; allerdings ist eine Registrierung notwendig, um einen API-Key für die jeweilige Applikation zu erhalten, der bei jeder Anfrage mitgesendet werden muss.

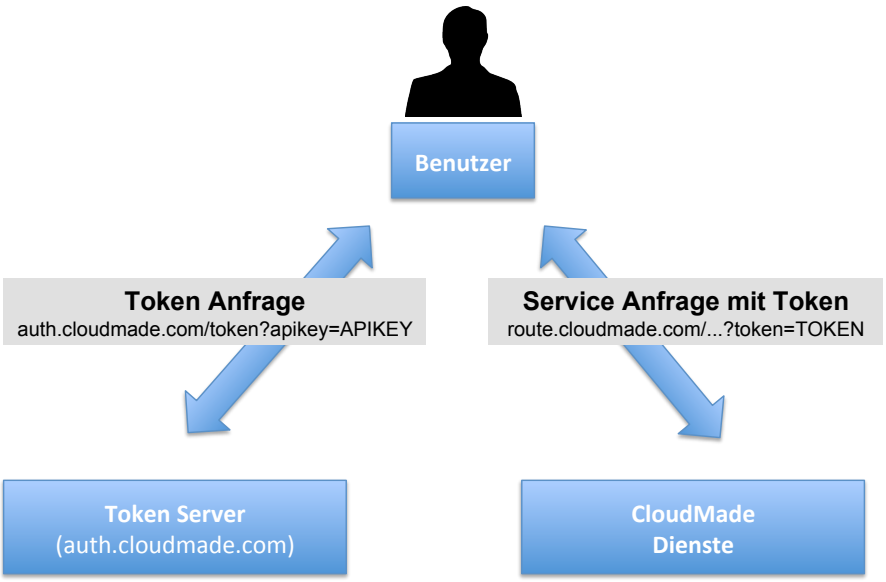

Abbildung 2.3: Schema für eine Anfrage mit der Token basierten Autorisierung für CloudMade's Webservices

Seit Februar 2010 müssen alle mobilen Applikationen die *Token*-basierte Autorisation verwenden. Das neue Verfahren ist einfach implementiert und lässt *CloudMade* die Anzahl der Anwender, sowie den Gebrauch der Dienste für jede der eigenen Applikation messen. Abbildung 2.3 zeigt den Ablauf einer Dienst Anfrage. Diese lässt sich in vier Schritte einteilen, wobei die ersten zwei wegfallen, wenn bereits ein *Token* vorhanden ist.

[13] OpenStreetMap Homepage: `http://www.openstreetmap.org/`
[14] Details siehe `http://developers.cloudmade.com/projects/show/routing-http-api`

1. Token Anfrage Besitzt der *Client* noch keinen gültigen *Token*, so muss dieser zunächst vom *Token*-Server generiert werden. Ein *Token* ist eine reine Zeichenkette, die zur eindeutigen Identifizierung eines *Clients* mit dem dazugehörigen API-Key verwendet wird. Sie sind global eindeutig und werden bei der ersten Verwendung des *CloudMade* Dienste vergeben. Wann immer der *Client* eines der Dienste von *CloudMade* nutzt, muss dieser Token mitgesendet werden, um eine gültige Antwort zu bekommen. Um einen solchen *Token* zu bekommen, muss per POST eine HTTP Anfrage an folgende URL übermittelt werden:

```
http://auth.cloudmade.com/token/APIKEY?userid=UserID&deviceid=DeviceID
```

Der *APIKEY* ist derjenige Key, den man über die Website für die jeweilige Applikation generieren lassen hat.

UserID steht für eine eindeutige Kennung des Benutzers. Dies kann zum Beispiel ein Hash der Userkennung in Verbindung einer eindeutigen Device-ID sein.

Die *DeviceID* muss eine eindeutige Kennung des Mobilen Gerätes, wie zum Beispiel die Mac-Adresse, sein.

2. Token Antwort Bei erfolgreicher Antwort erhält man einen *Token*, der in etwa so aussieht:

```
71c411efgc0668bd45baafcdb7d7258e1db1
```

Dieser *Token* muss nun im weiteren Verlauf bei jeder Dienst-Anfrage von *CloudMade* von dem mobilen Gerät mitgesendet werden, wie der kommende Abschnitt zeigt.

3. Service Anfrage Um eine gültige Antwort von einem Dienst zu erhalten, muss nun zusätzlich zum API-Key der Token an die Anfrage gehängt werden. Für das Beispiel einer *Routing*-Anfrage muss die URL wie folgt aussehen:

```
http://routes.cloudmade.com/APIKEY/api/0.3/[PARAMETERS]&token=TOKEN
```

Hier können zahlreiche Parameter angegeben werden, die auf der Website von *CloudMade*[15] näher beschrieben sind. Die wichtigsten sind der Startpunkt, das Ziel, die Art des Fortbewegungsmittel und das Ausgabeformat. Es ist auch möglich so genannte *Transistion Points* zu setzen, falls noch weitere Punkte angesteuert werden sollen, bevor das Ziel erreicht wird.

[15] `http://developers.cloudmade.com/wiki/routing-http-api/Documentation`

Es können drei Arten von Fortbewegunugsmitteln (`route_type`) gewählt werden: zu Fuß (`foot`), das Fahrrad (`bicycle`), sowie das Auto (`car`). Für das Auto gibt es einen zusätzlichen Parameter (`route_type_modifier`), der bestimmt, ob entweder die schnellste (`fastest`) oder die kürzeste (`shortest`) Route berechnet werden soll.

Als Ausgabeformat steht zum einen eine JSON-Struktur (`js`) zur Verfügung und zum anderen eine XML-Struktur (`gpx`). Das folgende Beispiel zeigt, wie eine Anfrage mit der oben genannten URL und den hier aufgeführten Parametern eine Route vom Rechenzentrum-Westerberg (52.283630, 8.025502) zum Schloss in der Innenstadt Osnabrücks (52.271795, 9.044288) zu Fuß mit JSON als Ausgabeformat aussehen kann.

```
52.283630,8.025502,52.271795,9.044288/foot.js
```

4. Service Antwort Die Antwort kann zum einen in Form von JSON oder als XML erfolgen, je nachdem, welches Ausgabeformat bei der Dienst-Anfrage angegeben wurde. XML ist zwar sehr weit verbreitet, ist aber JSON gegenüber relativ schlecht lesbar und hat viel mehr Overhead. Daher wird die Struktur der Antwort einer *Routing*-Anfrage im JSON Format demonstriert.

```
1  {
2  version:0.3,
3  status: 0 - OK, 1 - Error,
4  status_message: Error message string,
5  route_summary: {
6     total_distance: Distance in meters,
7     total_time: Estimated time in seconds,
8     start_point: Name of the start point of the route,
9     end_point: Name of the end point of the route,
10    transit_points: Transit points if they are present in request.
11 },
12 route_geometry: Array of nodes from start to end
13 route_instructions: [[instruction, length, position, time,
14                       length_caption, earth_direction,
15                       azimuth, turn_type, turn_angle],...,]
16 }
```

Listing 2.3: JSON Struktur einer Route-Response

Die Zeilen 2-4 sind hierbei eher irrelevant: es wird nur die API-Versionsnummer angegeben und ob ein Fehler aufgetreten ist. Zeile 5-10 stellt ein JSON-Dictionary namens `route_summary` dar. Wie der Name vermuten lässt, können hier allgemeine Informationen wie Gesamtdistanz (`total_distance`) und die Gesamtzeit (`total_time`) in Ab-

hängigkeit des gewählten Fortbewegungsmittels über die errechnete Route ausgelesen werden. Wichtig ist das Array `route_geometry` in Zeile 12. In diesem Array befinden sich alle aus *OpenStreetMap* für die gewünschte Route ermittelten Knotenpunkte (*Nodes*) mit ihren Koordinaten. Dabei befindet sich der Startknoten am Anfang und der Endknoten am Ende des Arrays.

Vor allem für Applikationen mit Navigationsfunktion sehr nützliche Informationen befinden sich in den `route_instructions`. Dieses Array enthält eine Menge Informationen für jedes Wegsegment. An erster Stelle (`instruction`) befindet sich eine Anweisung. Diese kann zum Beispiel „Bei Barbarastraße rechts abbiegen“ lauten. Weitere wichtige Informationen sind `length`, welches die Länge des aktuellen Segments in Metern angibt, `position`, mit der die erste Position des Segments im `route_geometry` ermittelt werden kann und schließlich `time`, mit der die benötigte Zeit in Sekunden für diesen Abschnitt enthält. Die Bedeutung dieser und der anderen Parameter kann jederzeit online auf der *CloudMade* Website[16] nachgelesen werden.

2.3 Geographische Grundlagen

Wie bereits erwähnt, flossen bei der Entwicklung dieser App auch Elemente aus der Geoinformatik mit ein. Vor allem bei der Implementierung der standortbezogenen *Augmented Reality* (Abschnitt 3.4.8) ist das Verständnis der in diesem Abschnitt dargelegten Grundlagen von Bedeutung. Zunächst werden wichtige Begriffe aus der Geographie erörtert, woraufhin etwas genauer auf drei bezogen auf diese Arbeit verwendete geographische Koordinatensysteme eingegangen wird.

Bezugssystem / Referenzsystem Ein Bezugs- bzw. Referenzsystem ist ein physikalisch definiertes, grundlegendes Bestimmungssystem. Es wird zur Erfassung, Speicherung, Darstellung und Nutzung von topographischen Sachverhalten mit thematischen Informationen auf, unter oder über der Erdoberfläche als Ordnungssystem benötigt und gestattet die gegenseitige Zuordnung von Informationen zueinander. Die praktische Realisierung erfolgt durch die Festlegung der Koordinaten von Punkten.

Geoid Es ist im Allgemeinen bekannt, dass die Erde keine perfekte Kugel darstellt. Um die Figur der Erde dennoch mathematisch möglichst exakt zu beschreiben, entwickelte Gauß 1828 das Geoid als physikalisches Modell über die Äquipotenzialfläche des

[16] `developers.cloudmade.com/wiki/routing-http-api/Response_structure`

Erdschwerefelds. Es dient zur Definition von Höhen, sowie zur Vermessung und Beschreibung der Erdfigur. Als Näherung repräsentiert das Geoid die Niveaufläche des mittleren Meeresspiegels der Weltmeere.

Koordinatensystem Ein Koordinatensystem ist eine mathematische Abbildungsvorschrift zur Beschreibung der Lage von Punkten im Raum. Das geläufigste Koordinatensystem ist das kartesische Koordinatensystem, bei dem alle Achsen senkrecht zueinander stehen und sich in einem willkürlich festgelegten Ursprung schneiden. Innerhalb eines Bezugssystems kann zwischen verschiedenen Koordinatensystemen beliebig umgerechnet werden (Koordinatentransformation). Punkte eines Festpunktfeldes, die ein bestimmtes Referenzsystem realisieren, werden in einem Koordinatensystem abgebildet.

Koordinatentransformation Bei einer Koordinatentransformation werden Punktkoordinaten eines Bezugssystems in ein anderes umgerechnet.

Ellipsoidisches Koordinatensystem Ellipsoidische[17] (oder geodätische) Koordinaten eines Punktes $P(\phi, \lambda, h)$ werden durch drei Werte festgelegt: Die ellipsoidische Breite ϕ, den Winkel zwischen Ellipsoidnormalen in P und Äquatorebene, die ellipsoidische Länge λ, den Winkel zwischen Nullmeridian und Meridian von P, und die ellipsoidische Höhe, der metrische Abstand des Punktes P von der Ellipsoidoberfläche entlang der Ellipsoidnormalen.

2.3.1 Geographische Koordinatensysteme

Geoobjekte, die sich durch eine Lage auf der Erdoberfläche auszeichnen, können nicht ohne weiteres in kartesischen Koordinatensystemen dargestellt werden, da die Erde keine Kugel ist. Sie wird mathematisch eher durch ein Rotationsellipsoid angepasst, wobei aufgrund der regional verschiedenen Oberflächengestalt der Erde eine Anpassung durch unterschiedliche Ellipsoide erfolgt. Geographische Koordinatensysteme beziehen sich auf die Oberfläche der Erdkugel. Die Koordinaten werden hier zum Beispiel in Breiten- (ϕ) und Längengraden (λ) angegeben [3]. Es existieren aber auch weitere Systeme, von denen drei spezielle im weiteren Verlauf dieses Abschnitts vorgestellt werden, da diese für die Umsetzung der *Augmented Reality* Technologie des *iLocator*s, in Abschnitt 3.4.8 verwendet werden.

[17] Ein Ellipsoid ist die drei- oder mehrdimensionale Entsprechung einer Ellipse.

WGS84 Das *World Geodedic System 1984* bezeichnet ein weltweites geodätisches Referenzsystem, auf dessen Grundlage Positionen auf der Erde im erdnahen Raum bestimmt werden. Entwickelt und veröffentlicht wurde es von der US-amerikanischen *Defense Mapping Agency*, wird unter anderem in Satelliten für GPS-Systeme verwendet und ist eins der am häufigsten verwendeten Systeme der Welt. Die Koordinaten eines gesuchten Ortes werden mit Hilfe eines dreidimensionalen, kartesischen Koordinatensystem festgelegt, welches seinen Ursprung im Massenschwerpunkt der Erde hat. Die x-Achse weist zum Schnittpunkt des Null-Meridians[18] mit dem Äquator, die y-Achse befindet sich im rechten Winkel zur x-Achse, weist nach Osten und befindet sich ebenfalls zur Äquatorebene. Die z-Achse zeigt zum Nordpol, sie überlagert sich mit der Rotationsachse der Erde.

Weitere Bestandteile sind ein Referenzellipsoid[19] zur Vereinfachung der Erdoberfläche und zur Angabe von Längen- und Breitengrade, der Geoid[20] nach dem *EGM96*[21] und die Koordinaten von zwölf auf der Erde verteilten Fundamentalstationen [4].

ECEF *Earth-Centered Earth-Fixed* ist ein Oberbegriff für Koordinatensysteme, die mit der Erde rotieren und ihren Ursprung im Schwerpunkt haben. Das eben erklärte WGS84 ist ein solches Koordinatensystem, wobei hier die Koordinaten nicht in Längen- und Breitengrad, sowie Höhe in Abhängigkeit des Referenzellipsoids, sondern durch ein *[X, Y, Z]* Tripel angegeben werden. Die Ausrichtung der Koordinatenachsen eines ECEF-Systems ist allerdings die selbe wie beim WGS84.

ENU In vielen Ziel- und Tracking-Anwendungen ist das lokale kartesische *East-North-Up* Koordinatensystem weitaus praktischer und intuitiver als ein ECEF oder ein geodätisches Koordinatensystem wie das WGS84. Sein Ursprung befindet sich lokal an einem fixen Punkt, wobei die *North-* und *East-*Achse eine Tangentialebene zur Erdoberfläche bilden. In der Regel wird die East-Achse mit x, die North-Achse mit y und die Up-Achse mit z gekennzeichnet.

[18] Der Null-Meridian ist ein senkrecht zum Erdäquator stehender und von Nord- zu Südpol verlaufender Halbkreis, von dem aus die geographische Länge nach Osten und Westen gezählt wird.

[19] Ein Rotationsellipsoid, dessen Dimension und Lagerung so gewählt wird, dass es sich dem Geoid im Vermessungsgebiet optimal anpasst.

[20] Niveaufläche des von verschiedenen Einflüssen befreiten Erdschwerefeldes in Höhe des mittleren Meeresniveaus.

[21] Earth Gravitation Model 1996. Ein aktuelles Geoidmodell, das für das *WGS84* den Abstand des Geoids von dem Referenzellipsoid, gemessen entlang der Ellipsoidnormalen, liefert.

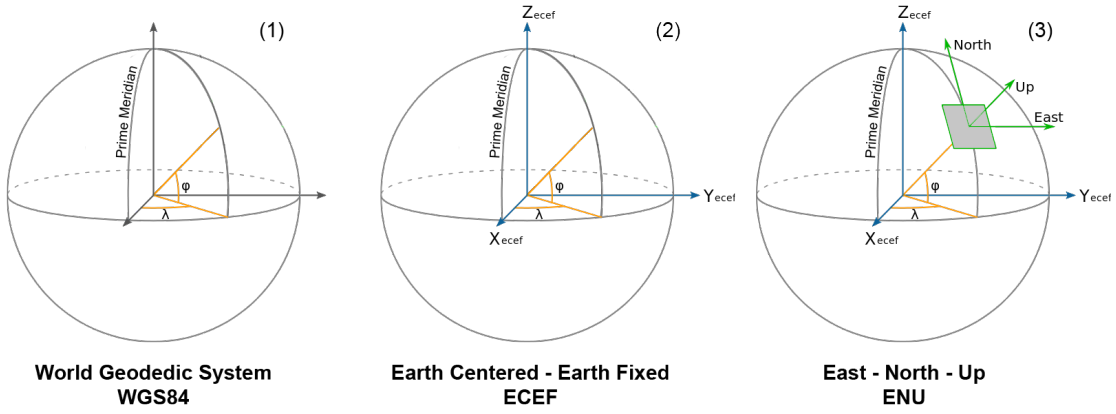

Abbildung 2.4: **Die drei geographischen Koordinatensysteme im Vergleich**. **(1)** Das geodätische Koordinatensystem mit Längen-, Breitengrad, sowie Höhe in Abhängigkeit eines Referenzellipsoids, **(2)** Das kartesische Koordinatensystem mit X,Y,Z Werten, **(3)** Das für Tracking Anwendungen optimale Koordinatensystem, dessen Ursprung sich auf einem lokalen Punkt der Erdoberfläche befindet.

2.4 Grundlagen der Computergrafik

Nachdem nun die geographische Grundlagen besprochen wurden, widmet sich dieser Abschnitt den Grundlagen der Computergrafik. Es werden die wichtigsten Begriffe zum Verständnis der Implementierung von *iLocator* erläutert, sowie Techniken zur Manipulation von 3D-Objekten dargelegt, die vor allem bei der Umsetzung von *Augmented Reality* in dieser APP von großer Bedeutung waren.

Frustum „Frustum (dt. Stumpf) [...], steht im Bereich der 3D-Grafik für einen Körper, dessen Inhalt in irgendeiner Form auf den Bilschirm projiziert wird. Fast immer ist die Projektionsebene (entspricht dem Bildschirm) eine Randebene des Frustums.“[22]

Projektionen Unter einer Projektion versteht man allgemein eine Abbildung aus einem Raum der Dimension n in einen Raum mit einer kleineren Dimension als n. Da ein Bildschirm ein zweidimensionales Ausgabemedium ist, müssen dreidimensionale Objekte in zweidimensionalen Ansichten dargestellt werden. Zu den zwei wesentlichen Klassen

[22] `http://wiki.delphigl.com/index.php/Frustum`

derartiger planarer Projektionen gehört zum einen die *Parallelprojektion* ohne Tiefenwirkung und zum anderen die *perspetivische Projektion* mit einem oder mehreren Fluchtpunkten. Auch sie sind eine Art Tranformation und lassen sich daher ebenfalls als Matrix darstellen.

Transformationen Mit Hilfe von Transformationen ist es möglich, die Position, die Orientierung, die Form und die Größe grafischer Objekte zu manipulieren. Dies geschieht durch mathematische Operationen auf den Definitionspunkten eines Objekts. Bereits in der Algebra wird gelehrt, dass Matrizen Transformationen repräsentieren. Dahier wird durch eine Matrizenmultiplikation ($A \cdot B = C$) der Transformationsmatrix A und dem als einspaltige Matrix interpretierten Koordinatenvektor B die eigentliche Transformation durchgeführt.

Für die Auswertung einer zusammengesetzten Transformation

- gilt das Assoziativgesetz, d.h. $A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$,
- gilt im Allgemeinen nicht das Kommutativgesetz, d.h. $A \cdot B \neq B \cdot A$.

Jede Sequenz von Rotation, Translation und Skalierung erhält dabei die Parallelität von Linien, aber nicht Längen und Winkel. Solche Transformationen heißen auch affine Transformationen oder affine Abbildungen.

Homogene Koordinaten Im Gegensatz zu Translationen lassen sich Rotationen im zweidimensionalen Objektraum mit einer 3×3-Matrix beschreiben. Um dennoch auch eine Translation durch eine Matrixmultiplikation ausdrücken zu können, muss das Konzept der homogenen Koordinaten eingeführt werden. Dabei wird dem Objektraum eine Dimension hinzugefügt.

Ein Punkt $P = (x, y, z)$ hat somit die homogenen Koordinaten $(x_h, y_h, z_h, w)^T$ mit $w \neq 0$, wobei gilt:

$$\begin{aligned} x_h &= x \cdot w \\ y_h &= y \cdot w \\ z_h &= z \cdot w \end{aligned}$$

Die Verschiebung des Punktes $P = (x, y, z)$ um den Vektor $\vec{v} = (x_0, y_0, z_0)^T$ kann demnach wie folgt dargestellt werden:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_0 \\ y + y_0 \\ z + z_0 \\ 1 \end{pmatrix}$$

2.5 Core Motion Framework

Mit eines der vielseitigsten und innovativsten *Features* auf dem iPhone sowie auf den anderen iOS Geräten, ist der eingebaute Beschleunigungssensor (Accelerometer). Durch ihn lässt sich herausfinden, wie ein Gerät ausgerichtet ist, oder wie es sich gerade bewegt. Es misst sowohl die Beschleunigung des Gerätes selber, als auch die Gravitation, die darauf wirkt, in dem die Summe aller wirkenden Kräfte (gemessen in g) in einer gegebenen Richtung berechnet wird. iOS selbst verwendet es unter anderem für die Autorotation der Benutzeroberfläche und viele Spiele als Kontrollmechanismus, mit ihm können aber auch Schüttelgesten oder andere Bewegungen registriert werden. Seit der Einführung des iPhone 4 gibt es darüber hinaus ein Gyroskop, mit dem der Winkel jeder der drei Achsen des iPhones bestimmt werden kann (siehe Abbildung 2.5).

Über das *Core Motion* Framework können diese (rohen) Bewegungsdaten ausgelesen und bearbeitet werden. Es stellt die Klasse `CMMotionManager` zur Verfügung, mit der schließlich die Daten dieser Sensoren entweder periodisch abgefragt oder in bestimmten Intervallen vom Gerät empfangen werden können. Beide Ansätze haben ihre Vor- und Nachteile; welches benutzt wird, hängt von der Art der Verwendung ab. Zusätzlich verarbeitet das Framework die Beschleunigungs- und Gyroskopdaten unter Verwendung spezieller Algorithmen und stellt diese aufbereiteten Daten zur Verfügung. Außerdem, aber für diese Arbeit weniger von Bedeutung, existiert ein Magnetometer, der das umgebene Magnetfeld misst.

Dieser Abschnitt soll die Verwendung dieses Frameworks verdeutlichen und anhand von Beispielen zeigen, wie die oben genannten Sensordaten auf unterschiedliche Weise ausgelesen und verarbeitet werden können.

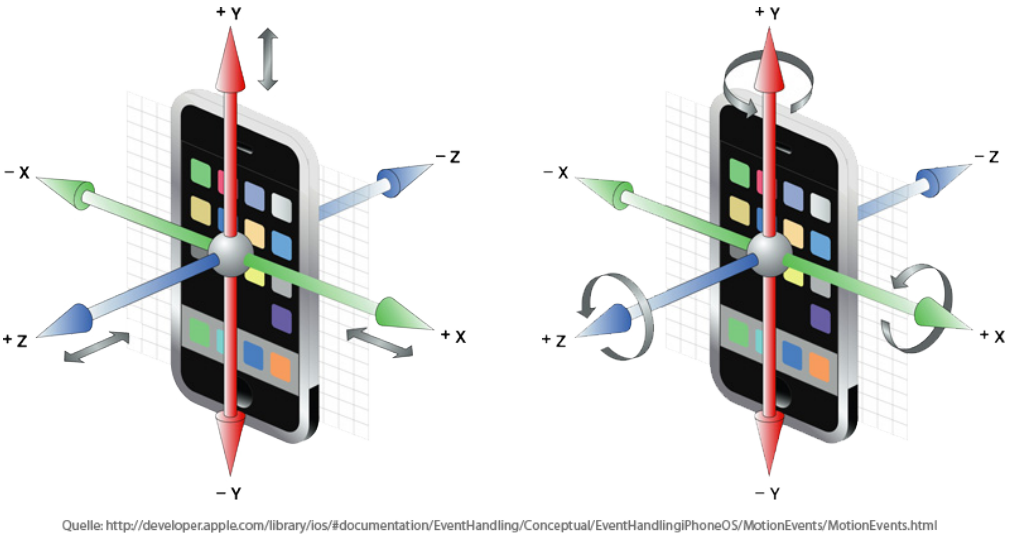

Abbildung 2.5: **Die Achsen der Bewegungssensoren:** Links die Achsen des Beschleunigungssensor, rechts die Rotationsachsen des Gyroskops

Motion Manager

Die Klasse `CMMotionManager` bildet die zentrale Einheit des *Core Motion* Frameworks. Auch wenn diese Klasse nicht als *Singleton* entworfen ist, wird empfohlen, diese wie eine zu verwenden (siehe Abbildung 2.6) [5]. Um schließlich an die Sensordaten zu gelangen, erstellt man also eine Instanz dieser Klasse, setzt ein *Update*-Intervall, startet die entsprechenden *Updates* und verarbeitet die übergebenen *Motion Events*.

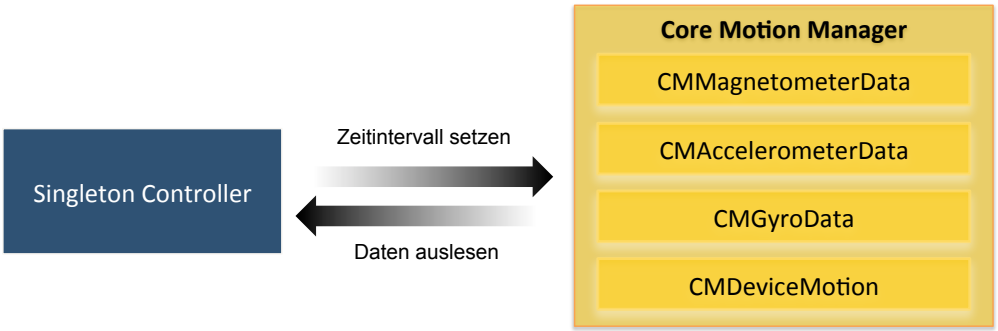

Abbildung 2.6: Verwendung des Core Motion Managers als Singelton

Motion Events werden in *Core Motion* durch drei Datenobjekte beschrieben, die jeweils eine oder mehrere Messungen abkapseln:

`CMAccelerometerData` entält die Beschleunigungsdaten (gemessen in g) entlang jeder der räumlichen Achsen.

`CMGyroData` enthält die Rate der Drehung um jede der drei Raumachsen.

`CMDeviceMotion` entählt diverse Messwerte, einschließlich der Lage und weiteren nützlichen Messungen der Drehrate, sowie Beschleunigung. Als besonderes Highlight stellt dieses Objekt ebenfalls eine Rotationsmatrix zur Verfügung, die die Ausrichtung des Gerätes beschreibt und besonders nützlich bei der Umsetung des *Augmented Reality* Konzeptes in Abschnitt 3.4.8 ist.

Das folgende Beispiel zeigt, wie Beschleunigungsdaten Ereignis-basiert über `CMMotionManager` ausgelesen und verarbeitet werden.

```
1 self.motionManager      = [[CMMotionManager alloc] init];
2 NSOperationQueue *queue = [[NSOperationQueue alloc] init];
3
4 motionManager.accelerometerUpdateInterval = 1.0 / 10.0;
5 [motionManager startAccelerometerUpdatesToQueue:queue withHandler:
6     ^(CMAccelerometerData *accData, NSError *error)
7     {
8       // do something with accData
9     }];
```

Listing 2.4: Beispiel für ein Event-basiertes Auslesen von Beschleunigungsdaten

Zunächst wird eine Instanz von `CMMotionManager` erstellt, über den die *Motion Events* abgefangen und verarbeitet werden. Anschließend wird ein `NSOperationQueue` angelegt, welches lediglich einen Container darstellt, dem eine Menge an Operationen übergeben werden kann, die er im Hintergrund abarbeitet. Anschließend wird die Zeit zwischen den Updates in Sekunden gesetzt, woraufhin der *Motion Manager* veranlasst wird, die Beschleunigungsupdates zu starten. Hierbei muss der Methode ein so genannter *Block* übergeben werden, der den auszuführenden Code enthält.

Blocks sind bereits aus der Sprache *C* bekannt und bilden eigenständige Arbeitseinheiten. Diese können Variablen zugewiesen und als Argument einer Funktion oder Methode übergeben werden. Dieser *Block* wird in diesem Beispiel also etwa zehnmal pro Sekunde mit den aktuellen Sensordaten ausgeführt. Da diese Art der Verwendung nicht zu jeder Applikation passt, soll das nächste Beispiel zeigen, wie zum Beispiel Gyrodaten auch proaktiv ausgelesen werden können.

```
self.motionManager = [[CMMotionManager alloc] init];
motionManager.gyroUpdateInterval = 1.0 / 10.0;
[motionManager startGyroUpdates]:

// somewhere else in code
CMGyroData *gyroData = self.motionManager.gyroData;
```

Listing 2.5: Beispiel eines proaktiven Zugriffs der Gyrodaten

Auch hier wird zunächst eine Instanz der Klasse `CMMotionManager` angelegt. Anders als im Beispiel zuvor wird hingegen kein `NSoperationQueue` oder extra *Block* benötigt, die dem *Motion Manager* übergeben werden müssen. Lediglich das *Update*-Intervall muss gesetzt werden, ehe der *Motion Manager* mit den Updates startet. In der Regel wird zum Auslesen der Daten ein *Timer* angelegt, der in regelmäßigen Zeitabständen auf das `CMGyroData` Attribut des *Motion Managers* zugreift und die aktuellen Daten verarbeitet.

Ein letztes Beispiel soll den Zugriff auf das `CMDeviceMotion` Objekt verdeutlichen.

```
[self.motionManager startDeviceMotionUpdatesUsingReferenceFrame:
  CMAttitudeReferenceFrameXMagneticNorthZVertical];

// somewhere else in code
CMDeviceMotion  *deviceMotion   = self.motionManager.deviceMotion;

CMAttiude       *attitude       = deviceMotion.attitude;
CMRotationrate  *rotationRate   = deviceMotion.rotationRate;
CMAcceleration  *gravity        = deviceMotion.gravity;
CMAcceleration  *userAcc        = deviceMotion.userAcceleration;

CMRotationMatrix *rotMatrix     = attitude.rotationMatrix;
```

Listing 2.6: Beispiel eines proaktiven Zugriffs auf die Daten des `CMDeviceMotion` Objekts

Hier ist es ebenfalls zum einen möglich die Daten Ereignis-basiert oder zum anderen proaktiv auszulesen. In diesem Beispiel wird wiederum die proaktive Variante gezeigt, da sich diese für die Umsetzung von *iLocator* als äußerst nützlich erwiesen hat.

Wie bereits erwähnt, beinhaltet dieses Objekt mehr als nur einen Messwert. Die Lage, also die Winkel zu den jeweiligen räumlichen Achsen, sind in `attitude` gespeichert. Das Attribut `rotationRate` enthält die Rotationsrate, also die Änderung des Winkels pro Sekunde. *Core Motion* misst ebenfalls die Kräfte, die auf das Gerät wirken. Die reine Erdanziehungskraft wird dabei im Objekt `gravity` und die zusätzlichen Kräfte, erzeugt von dem Benutzer im Objekt `userAcceleration` gespeichert. Anhand dieser gesammelten Daten generiert *Core Motion* automatisch die Rotationsmatrix, welche

ein Teil von `CMAttitude` ist (Zeile 12). Es handelt sich dabei, wie in Abschnitt 2.4 beschrieben, um eine 3×3-Matrix

$$\vec{R_{rot}} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix},$$

welche die Lage des iPhones mathematisch beschreibt.

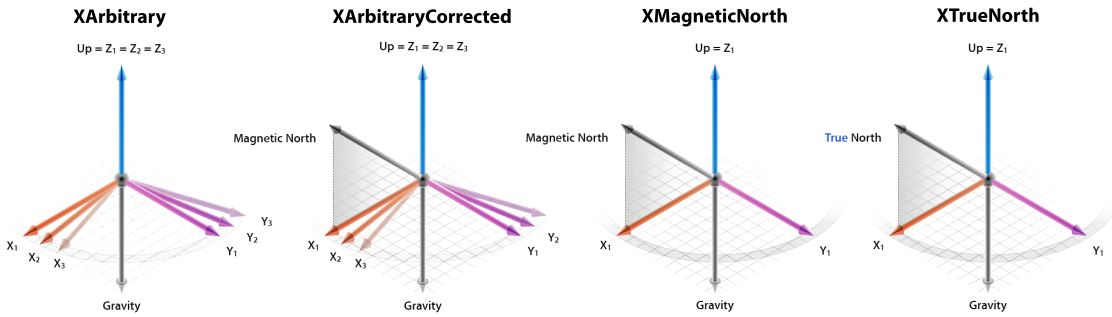

Abbildung 2.7: Die vier *Reference Frames* von Core Motion

Wie vielleicht schon aufgefallen ist, muss dem *Motion Manager* ein so genannter *Reference Frame* übergeben werden. Die `attitude` Daten, also die räumliche Orientierung des iPhones, wird relativ zu einem *Reference Frame* gemessen. *Core Motion*'s *Reference Frame* ist jedes Mal so gewählt, dass die z-Achse stets vertikal, sowie die x-Achse und y-Achse stets orthogonal zur Gravitation gerichtet sind. Es stehen insgesamt vier derartiger *Frames* zur Verfügung, je nachdem was und wie genau gemessen werden soll (vgl. Abbildung 2.7). In diesem Beispiel ist der *Reference Frame* `XMagneticNorth` gewählt worden, was bedeutet, dass die x-Achse immer in Richtung des magnetischen Nordens zeigt. Da der magnetische Norden sich nach dem Magnetfeld der Erde richtet, also nicht mit dem Norden bezogen auf den Nordpol übereinstimmt, gibt es außerdem den `XTrueNorth` Frame (vgl. Abbildung 2.8). Die anderen beiden Frames `XArbitrary` und `XArbitraryCorrected` sind nicht fix. Die x-Achse ist einmalig initial gesetzt, kann sich aber über die Zeit verschieben. `XArbitraryCorrected` wirkt diesem entegen, indem die x-Achse über den Magnetometer opportunistisch korrigiert wird.

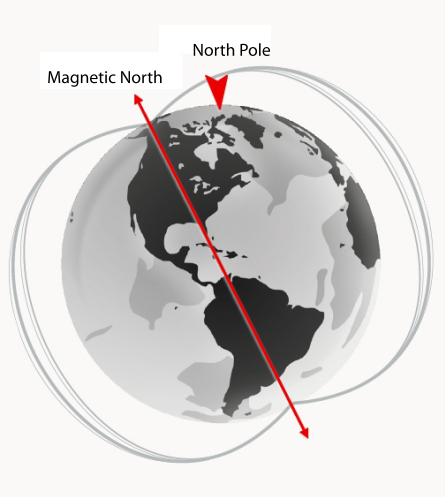

Abbildung 2.8: Unterschied zwischen dem magnetischen Norden und dem Nordpol

2.6 Datenhaltung auf dem iPhone

Beinahe jede Applikation auf dem iPhone muss in der Lage sein, Daten dauerhaft zu speichern und wieder aufzurufen. *iOS* stellt eine umfassende Sammlung an Werkzeugen und Frameworks zur Verfügung, Daten zu speichern, zu laden und zu teilen. *Core Data* ist ein mächtiges und voll funktionsfähiges, datenmodellierungs Framework für objektorientierte *Cocoa Touch* Anwendungen, wobei *SQLite* perfekt für relationale *low-level* Datenbankanwendungen geeignet ist. Es lassen sich auch beliebige Datenobjekte archivieren, wenn sie `NSCopying` konform sind und die nötigen Methoden implementieren. Dazu werden die Helfer Klassen `NSKeyArchiever` und `NSKeyUnarchiever` verwendet, um jeweils die Objekte zu speichern oder wieder zu laden. Da *Objective-C* eine Erweiterung der Programmiersprache `C` ist, gibt es ebenfalls die Möglichkeit, die traditionellen *I/O* Methoden wie `fopen()` zum Öffnen und Schreiben von Dateien zu verwenden [15].

Die Art der Datenhaltung kann demnach sehr gut an die Bedürfnisse der jeweiligen Applikation angepasst werden. Für größere Projekte sticht jedoch *Core Data* ganz klar hervor. Da im Rahmen dieser These hauptsächlich das *Core Data* Framework verwendet wurde, wird im Folgenden näher darauf eingegangen.

2.6.1 Core Data Framework

Core Data bietet seit iOS 3.0 ein flexibles und leistungsstarkes Datenmodellierungs-Framework, zugeschnitten auf das MVC Design. Entgegen allen Vermutungen handelt es sich nicht um eine relationale Datenbank bzw. ein relationales Datenbank Verwaltungs System. Das Framework stellt lediglich eine Infrastruktur zum Manipulieren, Speichern und Laden von Daten zur Verfügung. Von allen Mechanismen ist es am Besten dafür geeignet, nicht-triviale Daten persistent zu halten. Es gibt viele Gründe die dafür sprechen, *Core Data* zu verwenden. Der einfachste ist, dass weniger Code geschrieben werden muss, da viele Features bereits von Haus aus implementiert sind und somit nicht extra getestet und optimiert werden müssen.

Core Data stellt generalisierte und automatisierte Lösungen für die häufigsten Aufgaben zur Verfügung und verwaltet dabei ebenfalls den Lebenszyklus der Objekte. Applikationen mit größeren Texteingaben können die Built-In `Undo` und `Redo` Funktionen verwenden und müssen diese nicht selber implementieren. Zudem ist es möglich, Beziehungen zwischen den einzelnen Datenobjekten herzustellen. Änderungen an diesen Objekten oder Verbindungen verwaltet das Framework selber, sodass die Konsistenz stets gewährleistet ist. Damit bei größeren Datenmengen kein *Memory-Overhead* besteht, ist über einen speziellen *Controller Lazy Loading* der Objekte möglich. Diese Objekte werden erst dann vollständig in den Speicher geladen, wenn sie wirklich gebraucht werden.

Ein weiterer großer Vorteil ist, dass alle erzeugten Datenobjekte *Key Value*-konform (KV) und somit auch *KV-Observing* möglich ist. Es werden dynamische *Properties* erzeugt, die automatisch die Werte und Wertebereiche beim Schreiben der Attribute verifizieren. Alle, die schon einmal mit Datenbanken gearbeitet haben, wissen, dass meist schon bei relativ einfachen Abfragen komplizierte *SQL-Queries* aufgestellt werden müssen. Über ein `NSPredicate` Objekt, wie es auch schon zum Filtern einfacher *Arrays* verwendet wird, können komplexe *Queries* mondän ausgewertet werden. Auch im Zusammenhang mit *Multi-Threading* bietet das Framework Built-In Strategien zum Zusammenfügen und zur parallelen Manipulation der Daten.

Neben all diesen Vorzügen von *Core Data*, integriert sich das Framework außerdem perfekt in die *OS X* Werkzeugpallette. Mit den Modelentwurfswerkzeugen kann das Schema graphisch, schnell und einfach erstellt, sowie verändert werden. Über die *Instruments* Anwendungen kann weiterhin die *Performance* gemessen und diverse Probleme gefunden und korrigiert werden.

Der einzige Nachteil besteht darin, dass *Core Data* ein Werkzeug für eher fortgeschrittene Entwickler ist, die bereits Erfahrung bei der Entwicklung von Applikationen haben. Um

ein Überblick über die Arbeitsweise zu erhalten, werden in den folgenden Abschnitten kurz die Komponenten und deren Zusammenwirken erläutert.

Der Core Data Stack

Der Begriff *Stack* wird verwendet, um alle *Core Data* Objekte zu beschreiben, die beim Laden oder Speichern der Objekte aus dem *Persistent Store* zusammenarbeiten. Wie in Abbildung 2.9 dargestellt ist, besteht der *Stack* aus mehr oder weniger drei Komponenten.

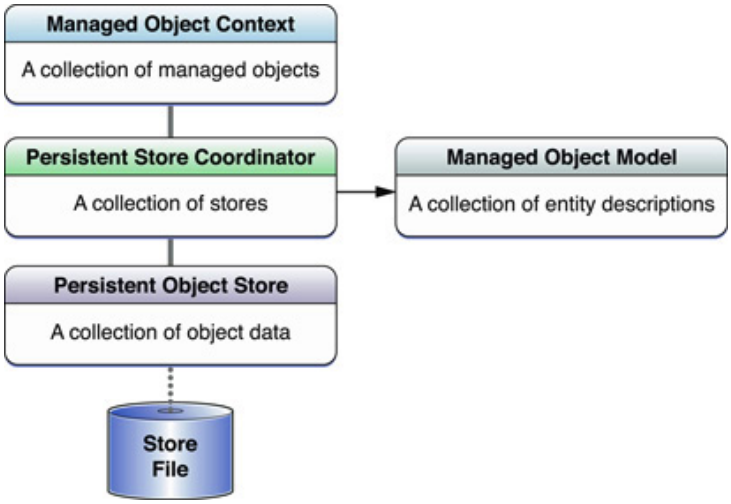

Abbildung 2.9: **Core Data Stack**: Die Bestandteile des *Core Data* Frameworks und deren Beziehung untereinander. Quelle: [15]

An oberster Stelle befindet sich der *Managed Object Context* (oder nur Kontext), der den größten Teil der Funktionalität von *Core Data* zur Verfügung stellt. Er funktioniert als Schnittstelle zu den darunter liegenden Framework Objekten wie der *Persistent Store Coordinator*, der wiederum einen oder mehrere *Persistent Object Stores* verwaltet. Die einzelnen Komponenten werden der Reihe nach in den kommenden drei Abschnitten näher erläutert.

Managed Objects und Managed Object Context

Ein *Managed Object* ist eine Instanz von `NSManagedObject` und repräsentiert einen Eintrag einer Tabelle in einer Datenbank. Es ist also im Sinne des MVC Paradigmas ein

Model Object, das von *Core Data* verwaltet wird und mit dem der Anwender in der Applikation interagiert.

Alle *Managed Objects* sind immer an ein *Managed Object Context* (eine Instanz von `NSManagedObjectContext`) geknüpft. Das geschieht automatisch bei der Erstellung eines neuen Objekts. Er ist ein sehr leistungsstarkes Objekt, hat eine zentrale Rolle in der Applikation und ist vor allem dafür verantwortlich, die *Managed Objects* zu verwalten. Da der Kontext die Änderungen der Daten verfolgt, ist er in der Lage, diese auch wieder Rückgängig zu machen (`Undo - Redo`). Er gewährleistet außerdem die Integrität der Objektgrafen bei Änderungen der Beziehungen der Objekte untereinander.

Der Kontext ist mit einem intelligenten Notizzettel vergleichbar. Werden Objekte aus dem *Persistent Store* geholt, befinden sich temporäre Kopien auf diesen Notizzettel und bilden einen oder mehrere Objektgraphen. Diese können dann beliebig verändert werden. Erst wenn die Änderungen explizit übertragen werden, bleiben sie auch im *Persistent Store* persistent. Dabei wird sichergestellt, dass die Objekte einen gültigen Status haben.

Es besteht die Möglichkeit, mehrere Kontexte in einer Applikation zu haben. Vor allem bei Nebenläufigkeit muss jeder *Thread* seinen eigenen Kontext bereitstellen, in dem dann die Änderungen an den Daten vorgenommen werden können. Die Änderungen können anschließend über den *Main*-Kontext zusammengeführt werden [16]. Sollte ein Objekt aus dem *Persistent Store* gleichzeitig von weiteren Kontexten geladen und manipuliert werden, kann dies zur Inkonsistenz führen. *Core Data* stellt auch hierfür einige Mechanismen bereit, derartige Probleme zu lösen.

Managed Object Model

Um sowohl den Objekt-Graphen zu verwalten, als auch Persistenz zu unterstützen, benötigt *Core Data* eine reichhaltige Beschreibung dieser Objekte. Das *Managed Object Model* ist ein Schema, dass eben diese Beschreibung der *Managed Objects* oder auch *Entitäten* bereitstellt. Typischerweise wird das Schema grafisch über XCode's *Data Model Design Tool* erstellt und ist letztendlich eine Instanz von `NSManagedObjectModel`. Das *Model* setzt sich aus einer Sammlung von Entitätsbeschreibenden Objekten (Instanz von `NSEntityDescription`) zusammen, die für jedes Objekt Metadaten bereitstellt. Dazu gehört unter anderem der Name der Entität, der Name der Klasse, die diese repräsentiert (muss nicht mit dem Namen der Entität übereinstimmen), die Attribute und Beziehungen. Die Attribute und Beziehungen werden wiederum von beschreibenden Objekten repräsentiert, wie in Abbildung 2.10 zu sehen ist.

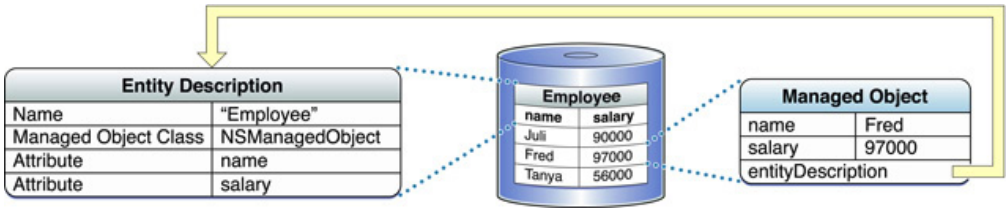

Abbildung 2.10: Die Beziehung zwischen einer Entitätsbeschreibung in einem *Model*, einer Tabelle in der Datenbank und einem *Managaged Object* mit dem dazugehörigen Eintrag in der Tabelle. Quelle: [15]

Core Data verwendet das *Model*, um die Einträge in der Datenbank auf *Managed Objects* in der Applikation abzubilden. Sollten Änderungen am Schema vorgenommen werden, ist zu beachten, dass die Daten aus dem vorherigen Model nicht mehr gelesen werden können.

Persistent Store Coordinator

Der *Persistent Store Coordinator* (eine Instanz von `NSPersistentStoreCoordinator`) spielt eine zentrale Rolle bei der Verwaltung der Daten, auch wenn der Anwender diesen in der Regel selber nicht direkt verwendet. Wie Abbildung 2.9 zeigt, stellt er das Bindeglied zwischen den *Managed Object Contexts* und den *Persistent Object Stores* dar.

Ein *Persistent Object Store* repräsentiert eine Datei mit persistenten Daten und bildet die Objekte der Datenbank auf die Objekte der Applikation ab. In iOS Applikationen wird in der Regel nur ein einziger *Store* verwendet. In komplexeren Applikationen, z.B. auf *Mac OS X*, können jedoch verschiedene *Stores* zum Einsatz kommen, die jeweils unterschiedliche Entitäten enthalten. Dazu müssen die Entitäten des *Models* über die Konfiguration des *Managed Object Models* partitioniert werden.

Der *Persistent Store Coordinator* hat die Aufgabe, diese *Stores* zu verwalten und dem dazugehörigen *Managed Object Context* den Eindruck eines einzelnen vereinigten *Stores* zu vermitteln. Wenn Daten abgerufen werden, holt *Core Data* diese aus allen *Stores*.

Wie bereits im vorherigen Abschnitt erwähnt, können mehrere *Managed Object* Kontexte erstellt werden und sind nützlich bei Nebenläufigkeit. Der Anwender erstellt für jeden *Thread* einen eigenen Kontext, die allerdings mit dem selben *Coordinator* verknüpft sind. Ein weiterer Ansatz wäre, für jeden *Thread* einen eigenen Kontext und einen eigenen *Coordinator* zu erstellen. Dieser Ansatz sorgt für eine größere Nebenläufigkeit,

erhöht aber auch die Komplexität, sowie die Speichernutzung, insbesondere dann, wenn Änderungen zwischen verschiedenen Kontexten kommuniziert werden müssen [16].

2.7 Web-Services und PHP

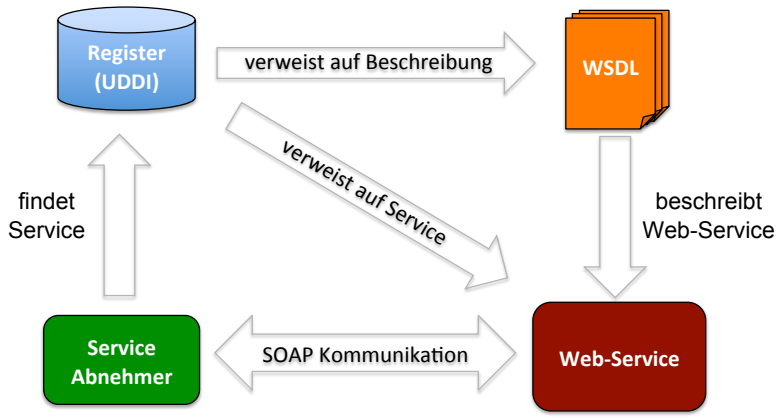

Abbildung 2.11: Web-Service Ablauf im Überblick

Der Erfolg des Webs liegt im Wesentlichen an dem ihm zugrunde liegenden offenen, skalierbaren Konzept und den zugrundeliegenden Technologien. Ein genauerer Blick auf das aktuelle Web zeigt allerdings, dass es mehr ein statisches, als ein dynamisches System ist. Statisch in dem Sinne, dass die verfügbaren Anwendungem im Web nicht ohne weiteres in anderen Kontexten und Szenarien wiederverwendet werden können. In diesem Zusammenhang wurde die Idee des Web-Services als Technologie eingeführt, dass die Integration und Interpolarität von geschäftlichen Funktionalitäten erlaubt.

Der Begriff *Service* ist in seiner Bedeutung sehr überladen. Dieter Fensel und seine Kollegen unterscheiden daher zwischen den beiden Begriffen *Service* und Web-Service [18]: Während *Service* als eine reine Rückstellung eines Wertes einer Domäne verstanden wird, ist ein Web-Service in diesem Sinne als eine rechnerische Einheit definiert, die eine Dienstleistung unter Verwendung eines vereinbarten Standards über das Internet anfordert.

Web-Services verbinden demnach Computer und andere Geräte miteinander über das Internet unter Verwendung von Standard-Protokollen zum Austausch von Daten. Es sind modular aufgebaute, selbsterklärende und in sich geschlossene Anwendungen, die über das Unternet zugänglich sind.

Eine Reihe von Web-Service Technologien (oft von diversen Normungsgremien wie *OASIS*[23] oder *W3C*[24] standartisiert) bilden die Grundlage für die Kommunikation und Integration Web-Service-basierter Anwendungen. Die Abbildung 2.11 zeigt ein Schema eines Web-Services im Zusammenhang mit drei verbreiteten Standardtechnologien wie *WSDL*[25] – eine XML Beschreibungssprache für Web-Services –, *SOAP*[26] – ein XML-basiertes Nachrichtenformat, um beliebige XML-Daten zwischen Services und Clients auszutauschen –, *UDDI*[27] – ein Datenmodell und API für Web-Service Register.

An dieser Stelle soll der Vollständigkeit halber erwähnt werden, dass es noch weitere Alternative für die Realisierung von Web-Services existieren. Besonders verbreitet ist die *Respresentational State Transfer Architecture* oder kurz *REST* dar. Sie basiert auf Prinzipen, die bereits im *World Wide Web* eingesetzt werden. Für die Umsetzung des REST-Architekturstils werden meist Internet-Technologien verwendet, als Anwendungsschicht-Protokoll dient hauptsächlich HTTP.

PHP Für die Umsetzung von Web-Services und der oben genannten Standarts kann unter anderem die Server-seitige Skriptsprache PHP verwendet werden. Sie zeichnet sich besonders durch eine breite Datenbankunterstützung und Internet-Protokolleinbindung, sowie die Verfügbarkeit zahlreicher Funktionsbibliotheken aus. Der Programmcode wird nicht wie bei anderen Programmiersprachen kompiliert, sondern mithile eines Interpreters interpretiert. Mittlerweile kann PHP auch als objektorientierte Programmiersprache durchgehen, da dieses Konzept seit PHP 4 zwar schon vorhanden, aber erst mit PHP 5 ausgebaut wurde.

```
1 $pdo = new PDO('mysql:host=host_name;dbname=db_name',$user_name,
      $password);
2
3 $id=100;
4 $stmt = $pdo->prepare('SELECT * FROM User WHERE id=?');
5 $stmt->execute(array($id));
6 $user = $stmt->fetch(PDO::FETCH_OBJ);
```

Listing 2.7: Einfache Datenbankanbindung über PDO. Ein User mit der ID 100 wird abgefragt und als Objekt mit den Spaltennamen als Attribute gespeichert.

[23] Organization for the Advancement of Structured Information Standards
[24] World Wide Web Consortium: `www.w3.org`
[25] **W**eb Services **D**escription **L**anguage
[26] **S**imple **O**bject **A**ccess **P**rotocol
[27] **U**niversal **D**escription, **D**iscovery and **I**ntegration

Die *PHP Data Objects*-Erweiterung (PDO) nutzt diese neue Eigenschaft und stellt eine konsistente Datenbankschnittstelle bereit. Sie bietet einen transpratenten Datenzugriff, was bedeutet, dass welche Datenbank letztendlich benutzt wird, dieselben Funktionen für die Abfragen verwendet werden können. Das Beispiel in Listing 2.7 soll verdeutlichen, wie mithilfe von PDO eine Anfrage an die Datenbank gestellt werden kann.

3 Konzeption und Umsetzung

Ausgehend von den in Kapitel 2 dargelegten Grundlagen und Technologien, widmet sich dieses Kapitel der konkreten Umsetzung der in der Einleitung beschriebenen iPhone App *iLocator*.

Abbildung 3.1 zeigt die Komponenten, die für die Umsetzung dieser App verwendet wurden. Entwickelt wurde diese auf einem von der Universität zur Verfügung gestellten Mac Mini mit Mac OS X 10.7.5 als Betriebssystem und *XCode* 4.3.2 als Entwicklungsumgebung. Ein ebenfalls von der Universität bereitgestelltes iPhone 4 mit iOS 5.1 als Testgerät, sowie der Uni-eigene Server, standen für die Implementierung der Applikation zur Verfügung.

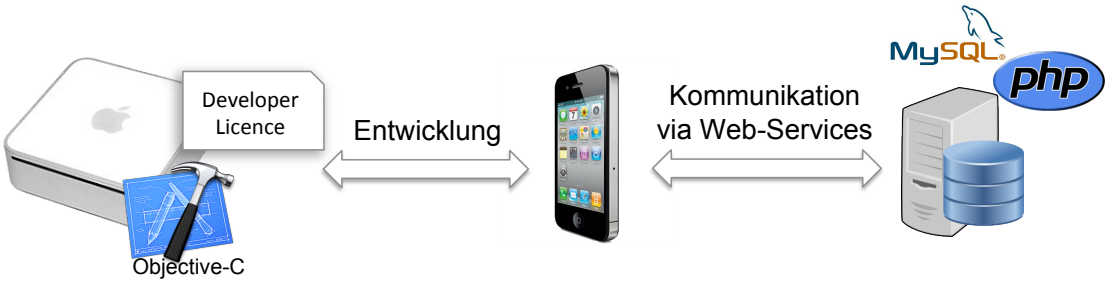

Abbildung 3.1: Bei der Entwicklung verwendete Komponenten und ihre Zusammengehörigkeit.

Zunächst wird in Abschnitt 3.1 und Abschnitt 3.2 erläutert, welche Komponenten und Technologien für die Kommunikation und Datenhaltung zwischen den Clients einerseits auf dem Server und andererseits in der Applikation selber verwendet wurden. Anschließend wird in Abschnitt 3.3 beschrieben, wie die Datenhaltung mit dem *Core Data* Framework umgesetzt worden ist. Abschließend werden die unterschiedlichen *Features* der Applikation im Abschnitt 3.4 vorgestellt.

3.1 Entwurf der Datenbank

Der Grund, dass für die Verwaltung der Daten MySQL als Datenbank verwendet wird, liegt nicht nur darin, dass diese bereits auf dem Server der Universität installiert ist, sondern auch an der hohen Geschwindigkeit, des geringen Speicherverbrauchs, sowie der hohen Popularität. Außerdem ist die Nutzung kostenlos, welches die Argumentation nochmals bekräftigt. MySQL lag während der Implementierungsphase in der Version

5.0.8 vor, bei dem Server handelt es sich um einen *Apache* 2.2.16 mit diversen Erweiterungen wie dem PHP 5.3.14 Paket.

Die Hauptarbeit bei dem Entwurf der Datenbank liegt in der Konzeption des Schemas. Ineffiziente Tabellenstrukturen können bei komplexeren Datenbankanfragen und größeren Datenmengen zu langen Rechenzeiten führen. Für die Implementierung dieses Projekts sind im Wesentlichen fünf Entitäten von Bedeutung, die jeweils eine eigene Tabelle repräsentieren (vgl. Abbildung 3.2). Dazu gehört zum einen die Entität *User*, die alle Daten bezogen auf den Anwender der Applikation speichert, zum anderen die Entität *Nodes* zum Speichern einfacher Geokoordinaten, sowie *POI* für die geteilten *Points Of Interests* bzw. Treffpunkte, *Message* zur Kommunikation der Anwender über den *Chat* untereinander und schließlich die Entität *Authentication*, die für den Registrierungsprozess verwendet wird.

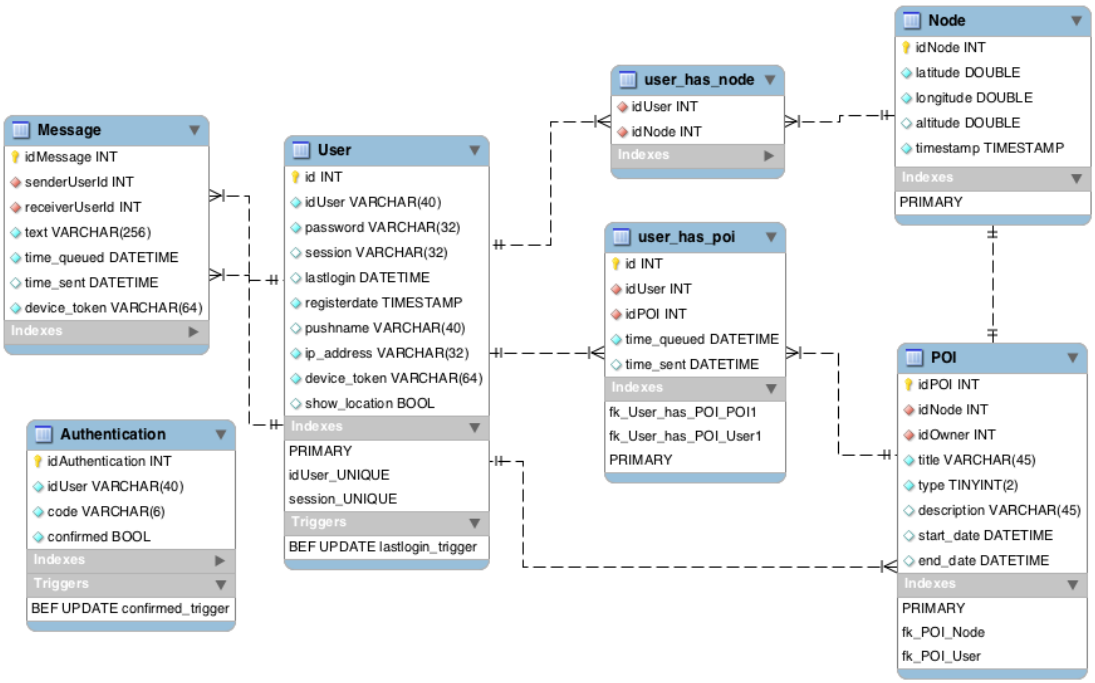

Abbildung 3.2: Aus *MySQLWorkbench* erstelltes ER-Diagramm der Datenbank

Im Folgenden werden diese genannten Entitäten und dessen Attribute, sowie Beziehungen untereinander näher dargelegt:

User Diese Entität stellt die zentrale Einheit der Datenbank dar. Alle anderen Entitäten sind gleichwie über den Primärschlüssel `id` an diese Tabelle geknüpft. Sollte ein Eintrag dieser Tabelle entfernt werden, werden automatisch auch alle über die Fremdschlüssel verknüpften Daten gelöscht. Jeder registrierte Nutzer erhält einen eigenen Eintrag, der ihn über die Emailaddresse (`idUser`) eindeutig identifiziert. Weitere Attribute werden zum Beispiel für die Authentifizierung des Benutzes beim Login (`password`) oder jeder Web-Service Anfrage (`session`) benötigt. Ein weiteres nennenswertes Attribut (`show_location`) gibt an, ob die Position des Users für andere sichtbar sein soll oder nicht. Die Tabelle enthält außerdem noch einen *Trigger*, der, sobald sich ein Nutzer einloggt, also eine *Session* vergeben wird, das Attribut `lastlogin` mit dem aktuellen Zeitstempel aktualisiert.

Node Auch die `Nodes` bilden einen wichtigen Bestandteil. Die Tabelle speichert geographische Koordinaten wie `latitude`, `longitude` und `altitude`, wie auch immer diese mit den anderen Entitäten verknüpft sind. Identifiziert wird ein solcher Eintrag eindeutig durch eine fortlaufende Nummer des Primärschlüssels `idNode`. Für eventuell spätere Verwendung existiert ein weiteres Attribut (`timestamp`), welches den Zeitpunkt des Koordinateneintrags angibt.

```
1  SELECT
2    lastEntries.idUser, n.latitude, n.longitude, n.altitude
3  FROM
4    Node n,
5    (SELECT
6       u.idUser AS idUser, MAX(n.idNode) AS lastNode
7    FROM
8       user_has_node uhn, User u, Node n
9    WHERE
10      u.id = uhn.idUser AND n.idNode = uhn.idNode AND
11      u.show_location <> 0 AND
12      uhn.idUser IN(
13           SELECT u.id
14           FROM   User u
15           WHERE  idUser IN ("buddy1@uos.de", "buddy2@uos.de") )
16   GROUP BY u.idUser ) AS lastEntries
17 WHERE
18   n.idNode = lastEntries.lastNode
```

Listing 3.1: SQL-Query, die für zwei `User` jeweils den letzten Standort liefert, sofern diese freigegeben werden.

Da ein Nutzer mehrere Standorte (`Nodes`) haben kann, aber jedem `Node` jeweils nur ein `User` zugeordnet ist, reicht im Prinzip eine 1:n Beziehung aus. Da es allerdings in Zukunft voraussichtlich möglich sein soll, etwa die letzten n Positionen des Nutzers darzustellen, um zum Beispiel die gelaufene Route in einem gewissen Zeitraum zu *tracken*, erschien die Implementierung einer n:m Beziehung über eine weitere Tabelle (`user_has_node`) als am sinnvollsten. Diese Tabelle enthält zwei Spalten. In der einen werden die ID's der `User` und in der anderen die ID's der `Nodes` vermerkt. So können über eine *SQL-Query* zum Beispiel alle oder nur der letzte Standort eines oder mehrerer `User` abgefragt werden. Die in Listing 3.1 dargestellte etwas komplexere *SQL-Query* liefert exemplarisch die letzten bekannten Koordinaten zweier User, sofern sie diese auch angeben.

POI Die *Points of Interests* stellen zwar innerhalb des Datenbankschemas keine besonders zentrale Entität dar, sind aber für das hier beschriebene Projekt von großer Bedeutung. Der Kern dieser Anwendung besteht, wie bereits in der Einleitung beschrieben, nicht nur in der Möglichkeit, sich eigene POIs zu setzen, sondern diese aber auch direkt oder in abgewandelter Form als Treffpunkt mit seinen Freunden zu teilen. Sobald der Anwender einen derartigen POI mit einem oder mehreren seiner Freunde teilen möchte, werden die Daten über die Applikation in dieser Tabelle gespeichert.

```
1 SELECT
2   uhp.id AS idUHP, p.*
3 FROM
4   user_has_poi uhp, POI p
5 WHERE
6   uhp.time_sent IS NULL AND
7   uhp.idUser = ? AND
8   p.idPOI = uhp.idPOI
```

Listing 3.2: SQL-Query, mit der alle noch nicht gesendeten POIs für einen Benutzer ausgegeben werden.

POIs sind in dieser Applikation über ihren Standort (`idNode`), einen Titel (`title`), eine eventuelle Beschreibung (`description`) und einen Typ (`type`) definiert. Sie werden auch hier eindeutig über den Primärschlüssel `idPOI` identifiziert. Die geographischen Koordinaten für den Standort werden jeweils seperat in der Tabelle `Node` abgelegt und sind über den Verweis der ID in `idNode` lokalisierbar. Über das Attribut `idOwner` wird zudem die ID des Users vermerkt, der diesen POI geteilt hat, damit die anderen Anwender diesen in der App sehen können.

Aufgrund der Tatsache, dass ein Benutzer mehr als nur einen POI teilen kann, sowie diverse mit ihm geteilt werden können, besteht zwischen den beiden Entitäten `User` und `POI` ebenfalls eine n:m Beziehung. Diese wird auch hier durch eine Tabelle `user_has_poi` beschrieben, die neben den jeweiligen IDs zwei weitere Attribute `time_queued` und `time_sent` enthält. Über diese beiden Attribute kann festgestellt werden, ob ein POI bereits von den jeweiligen Benutzern empfangen worden ist.

Das Beispiel in Listing 3.2 zeigt, wie alle „neuen“ POIs, deren Attribut `time_sent` also noch nicht gesetzt ist, für einen bestimmten Nutzer abgerufen werden können.

Message Eine weitere Tabelle ist der Entität `Message` gewidmet. Diese wird benötig, um die Kommunikation zwischen den Clients via Chat umzusetzen. Eine `Message` wird im Wesentlichen durch einen Sender (`senderUserId`), einen Empfänger (`receiverUserId`) und den eigentlichen Text (`text`) beschrieben. Wie schon bei den POIs gibt es zudem auch hier zwei weitere Attribute (`time_queued`) und (`time_sent`), in denen der Sende- und Empfangszeitpunkt vermerkt wird.

In diesem Fall ist eine 1:n Relation zwischen den Entitäten `Message` und `User` vorgesehen, sodass die Beziehung untereinander nicht über eine weitere Tabelle beschrieben werden muss. Der Grund dafür ist, dass derzeit keine Gruppenchatfunktion geplant ist, also ein Nutzer nur mit einem weiteren gleichzeitig über den Chat kommunizieren kann. So ist jeder `Message` im Prinzip nur ein `User` zugeordnet, wobei jedem `User` mehrere `Messages` zugeordnet werden können. „Im Prinzip“ deswegen, weil jede `Message` zwei Verweise auf die Tabelle `User` enthält, einen für den Sender und einen für den Empfänger.

Das nachfolgende Beispiel zeigt, wie mit einer simplen *SQL-Query* alle von einem `User` noch nicht empfangenen `Messages` geliefert werden.

```
1 SELECT
2   m.idMessage, m.text, u.idUser
3 FROM
4   Message m, User u
5 WHERE
6   m.receiverUserId = ? AND
7   m.time_sent IS NULL AND
8   u.id=m.senderUserId
```

Listing 3.3: SQL-Query, die alle neuen Messages für einen User liefert.

Authentication Die Tabelle `Authentication` verwaltet alle Einträge, die Bestandteil des Registrierungsprozesses sind. Da sich die Nutzer über ihre E-Mail-Adresse registrieren, wird diese zum einen in der Spalte `idUser` vermerkt. Zum anderen enthält die Spalte einen Registrierungscode, bestehend aus einem sechs-stelligen *String*. Solange dieser nicht von dem Anwender bestätigt wird, ist das boolesche Attribut `confirmed` auf `false` gesetzt. Erst nach Bestätigung des *Codes* erhält der Anwender einen Eintrag in der Tabelle `User` und kann sein Passwort vergeben.

Nachfolgendes Beispiel zeigt eine *SQL-Query*, die während des Registrierungsprozesses für eine E-Mail-Adresse einen Registrierungscode einfügt. Der Nutzer erhält diesen Code per E-Mail und muss ihn über die App bestätigen, damit die E-Mail-Adresse verifiziert und in die Tabelle `User` eingefügt werden kann.

```
1 INSERT INTO
2   Authentication (idUser, code)
3 VALUES
4   ("buddy1@uos.de", "Qb1a42")
```

Listing 3.4: Beispiel einer SQL-Query, die für eine Email-Adresse einen Code in die Tabelle `Authentication` einfügt.

3.2 Kommunikation zwischen Server und iPhone

Eine direkte Verbindung zwischen zwei oder mehreren iPhones ist nicht möglich. Damit die Geräte dennoch miteinander kommunizieren können, muss ein Server als Bindeglied implementiert werden. Nicht nur zur Kommunikation, sondern auch für die Verwaltung der im vorherigen Abschnitt dargelegten Benutzerdaten wie Login Name, Passwort, Standort oder geteilte POIs ist die Verwendung eines Servers mit Datenbank als Backend unvermeidbar.

Die erste Intention, einen eigenen kleinen *multithreaded* Server in Java zu schreiben, erwies sich jedoch schnell als zu aufwendig und zeitintensiv. Alleine das Testen und *Debuggen* der Routinen hat mehr Zeit in Anspruch genommen, als die Entwicklung der eigentlichen Applikation. Die eingängigste Lösung war, den Uni-eigenen Server zu verwenden und eine Schnittstellte – mithilfe von Web-Services – zwischen den *Clients* und einer Datenbank zu implementieren. Der Vorteil lag darin, dass kein extra Code für den Server geschrieben und getestet werden muss und dass sich der Server selbst um *multithreading* Verhalten bei parallelen Anfragen kümmert.

Die folgenden beiden Abschnitte gehen auf den implementierten Web-Service in PHP als

Schnittstelle zwischen der Datenbank und den *Clients* ein und legen dar, wie das iPhone diese Dienste letztendlich nutzt.

3.2.1 Implementierung des PHP Web-Services

In Abschnitt 2.7 wurden bereits ausführlich die Vorzüge von Web-Services erörtert. Es wurden einige standardisierte Technologien, wie die *Web Services Description Language* zur Beschreibung von Web-Services und das *Simple Object Access Protocol* für den Nachrichtenaustausch, vorgestellt. Da einerseits die implementierten Dienste nur den Anforderungen dieses Projekts dienen und keinem anderen zur Verfügung stehen sollen, andererseits JSON als Ausgabeformat dienen soll, ist hier ein eigener Ansatz angestrebt, der in diesem Abschnitt näher erläutert wird.

Abbildung 3.3: Kommunikation zwischen iPhone, Web-Service und der Datenbank.

Aus den in Abschnitt 2.7 genannten Eigenschaften von PHP wird der Web-Service in dieser Sprache implementiert. Hierzu ist eine `api.php` auf dem Server[28] angelegt worden, die sämtliche Anfragen der Clients verarbeitet. Der Ablauf einer Anfrage ist im folgenden Code-Ausschnitt dargestellt.

```
1  $config     = $config[APPLICATION_ENV];
2  $response   = new Response();
3  $api        = new API($config, $response);
4
5  try {
6  $api->handleCommand();          // handle command
7  exitWithResponse($response);    // send response
8  } catch (Exception $e) {
9    // [...] some exception handling
10 }
```

Listing 3.5: Einstiegspunkt einer Anfrage der `api.php`

[28] `http://dbs.informatik.uni-osnabrueck.de/pbmaster/api2/api.php`

Zunächst wird die aktuelle Konfiguration in Abhängigkeit der Entwicklungsumgebung geladen, woraufhin ein `Response`-Objekt einerseits und das eigentliche `API`-Objekt mit der aktuellen Konfiguration und dem `Response`-Objekt andererseits erstellt wird. Die eigentliche Verarbeitung der Anfrage geschieht in Zeile 6 mit dem Aufruf der Methode `handleCommand()` über das `API`-Objekt. Wie zu erkennen ist, wird dieser Aufruf von einem `try-catch`-Block umgeben. Sollte bei der Verarbeitung der Anfrage eine *Exception* geworfen werden, kann diese innerhalb des `catch`-Bereichs abgefangen werden.

Der Aufbau der zentralen `API`-Klasse ist im nachfolgenden Code-Ausschnitt dargestellt.

```
1  class API {
2
3    public $pdo;
4    public $response;
5
6    function __construct($config, $response) {
7      $this->response = $response;
8
9      // Create a connection to the database.
10     $this->pdo = new PDO(
11       'mysql:host='.$config['db']['host'].';dbname='.
12               $config['db']['dbname'],
13       $config['db']['username'],
14       $config['db']['password'],
15       array());
16
17     // on error, pdo should throw an exception
18     $this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
19
20     // We want the database to handle all strings as UTF-8.
21     $this->pdo->query('SET NAMES utf8');
22   }
23
24   function handleCommand() { ... }
25
26   // [...] handling functions
27
28 }
```

Listing 3.6: Aufbau der Klasse `API` aus `api.php`

Die Klasse besteht im Wesentlichen aus einer Reihe von Methoden zur Verarbeitung der Anfragen, sowie aus zwei Attributen: `pdo`,das Datenbankobjekt, und `response`, der die Daten für die Antwort einer Anfrage hält. Bei der Instanziierung der Klasse werden

dem Konstruktor zwei Parameter übergeben. Der erste Parameter `config` enthält die Konfigurationsdaten für die Datenbank; `response` verweist lediglich auf das bereits im vorherigen Code erzeugte `Response`-Objekt. Als Datenbankschnittstelle wird hier die *PHP Data Objects* Erweiterung (ebenfalls beschrieben in Abschnitt 2.7) verwendet.

Außerdem enthält die Klasse alle nötigen Methoden zur Verarbeitung der Anfragen, angedeutet durch den Kommentar in Zeile 26. Wie bereits in Listing 3.5 erwähnt, gibt es die zentrale Methode `handleCommand()`, die je nach Anfrage die dazugehörige Methode aufruft oder gegebenfalls eine Error-Nachricht zurückliefert. Alle Daten werden bei einer Anfrage per *POST* an die `api.php` übermittelt.

```
1 function handleCommand() {
2   if (isset($_POST['cmd'])) {
3     switch (trim($_POST['cmd'])) {
4       case 'logout' : $this->handleLogout(); return;
5       // [...] handle other commands
6     }
7   }
8   exitWithHttpError(400, 'Unknown command'.$_POST['cmd']);
9 }
```

Listing 3.7: Die Methode `handleCommand()`

Dieser Ausschnitt zeigt diese Methode in verkürzter Form. Zunächst wird die Art der Anfrage aus den POST-Daten über den Key `cmd` ausgelesen. Sollte dieser nicht existieren oder im weiteren Verlauf nicht abgearbeitet werden, so endet die Anfrage und es wird eine Fehlermeldung gesendet. In Zeile 4 wird zum Beispiel das Kommando `logout` abgefangen und die dazugehörige Methode `handleLogout()` aufgerufen, die im nachfolgenden Code-Ausschnitt exemplarisch dargestellt ist.

```
1  function handleLogout() {
2   $session = $this->getSession();
3
4   $sql_logout =  "UPDATE User SET session=NULL WHERE session=?";
5   $stmt = $this->pdo->prepare($sql_logout);
6   $stmt->execute(array($session));
7
8   // Logout successful
9   $this->response->setData(true);
10 }
```

Listing 3.8: Die Methode `handleLogout()`

Bei anderen Anfragen, die erfordern, dass der Nutzer angemeldet ist, wird immer die aktuelle *Session* von der App mitgeschickt. In diesem Beispiel wird zunächst geprüft, ob die *Session* gültig ist. Ist dies nicht der Fall, so wirft die Methode `getSession()` eine *Exception*, die wiederum abgefangen und verarbeitet wird. Bei einem Logout wird lediglich die *Session* des Nutzers auf `NULL` gesetzt. Daher folgt im nächsten Schritt eine SQL Anweisung, die eben dieses Attribut in der Tabelle `User` aktualisiert. War der Logout erfolgreich, wird das Datenobjekt der `Response` auf `true` gesetzt, welches der Applikation dieses auch signalisiert.

Wie bereits erwähnt, wird als Ausgabeformat die *JavaScript Object Notation* (JSON) verwendet. Dieses Format hat gegenüber XML den Vorteil, dass es sehr kompakt und leserlich ist. Außerdem unterstützt Objective-C dieses Format, sodass die JSON-Struktur direkt in ein `NSDictionary` konvertiert werden kann. Da die Ausgabe der Logout-Anfrage nur aus einem booleschen Wert besteht, wird an dieser Stelle eine etwas komplexere Ausgabe dargestellt.

```
1  {
2    data = ({alt = 0;
3             iduser = "buddy1@uos.de";
4             lat = "52.285592";
5             lng = "8.027401";
6            },
7            {alt = 0;
8             iduser = "buddy2@uos.de";
9             lat = "52.286616";
10            lng = "8.024354";
11           });
12     error = 0;
13 }
```

Listing 3.9: Beispiel einer Antwort im JSON-Format.

So kann eine Antwort aussehen, wenn die Applikation eine Anfrage sendet, welche die aktuelle Position aller Freunde liefert. Wie schön zu sehen ist, enthält das `data`-Feld ein *Array* aus Dictionaries, in denen schließlich die geographischen Koordinaten, sowie die ID des jeweiligen Freundes vermerkt sind.

3.2.2 Zugriff auf die Web-Services über das iPhone

Die Dienste der im vorangehenden Abschnitt beschriebenen API können nun von den Clients genutzt werden. Hierzu muss die Applikation lediglich das jeweilige Komman-

do und die dazugehörigen Daten an `api.php` unter der oben genannten URL senden. Das externe Framework `ASIHTTPRequest`[29] stellt zu diesem Zweck geeignete Klassen für die Kommunikation via HTTP zur Verfügung. Das folgende Beispiel zeigt, wie mit der Unterklasse `ASIFormDataRequest` schnell und einfach Daten gesendet werden können.

```
__weak ASIFormDataRequest *request;
request = [ASIFormDataRequest requestWithURL:self.serverAPIURL];

[request setPostValue:@"my_location"  forKey:@"cmd"];
[request setPostValue:session         forKey:@"session"];

[request setCompletionBlock:^{ /* ... Code ...*/ }];
[request setFailedBlock:^{ /* ... Code ... */ }];

[request startAsynchronous];
```

Listing 3.10: Senden von POST-Daten via `ASIFormDataRequest`

Nachdem eine Instanz dieser Klasse mit der entsprechenden URL angelegt wurde, können dem Objekt die nötigen POST-Daten, wie zum Beispiel das Kommando und die Session, hinzugefügt werden. Sollte die (asynchrone) Anfrage, die mit `startAsynchronous` gestartet wird, fehlschlagen oder erfolgreich sein, wird der jeweils zugewiesene Block ausgeführt.

Für die Kommunikation zwischen iPhone und Server ist hierfür eine zentrale Klasse `ServerCommunicator` implementiert worden, die nun etwas genauer besprochen wird.

ServerCommunicator Diese Klasse besteht, wie im Ausschnitt von Listing 3.11 zu sehen ist, im Grunde aus drei Properties und den Methoden zur Ansteuerung der Web-Services auf dem Server.

```
@interface ServerCommunicator : NSObject

@property (nonatomic, copy)     SCBasicBlock completionBlock;
@property (nonatomic, copy)     SCBasicBlock failedBlock;;
@property (nonatomic, strong)   NSURL *serverAPIURL;

- (void)postMyLocation:(CLLocation*)location;
@end
```

Listing 3.11: Verkürzter Auschnitt aus `Servercommunicator.h`

[29] urlhttp://allseeing-i.com/ASIHTTPRequest/

Wie auch schon bei dem `ASIFormDataRequest` besteht auch hier die Möglichkeit, je einen Block zu setzen, der ausgeführt wird, wenn die Anfrage scheitert oder erfolgreich ist. Über das Attribut `serverAPIURL` kann die entsprechende URL angegeben werden, die in diesem Fall aber immer auf `api.php` verweist. Der folgende Ausschnitt zeigt anhand der implementierten Methode `postMyLocation:`, wie sich die Applikation mit dem Web-Service verbindet.

```
1  - (void)postMyLocation:(CLLocation*)loc
2  {
3      // Create the HTTP request object for our URL
4    __weak ASIFormDataRequest *request;
5    request = [ASIFormDataRequest requestWithURL:self.serverAPIURL];
6
7    // [...] code that prepares the data
8
9    // set POST values
10   [request setPostValue:@"my_location"  forKey:@"cmd"];
11   [request setPostValue:session         forKey:@"session"];
12   [request setPostValue:lat             forKey:@"lat"];
13   [request setPostValue:lng             forKey:@"lng"];
14   [request setPostValue:alt             forKey:@"alt"];
15
16     // This code will be executed when the HTTP request is successful
17   [request setCompletionBlock:^{
18     // Parse HTTP-Response
19     Response *response = [Response responseWithRequest:request];
20
21     if (response.hasError)
22       _failedBlock(response);
23     else
24       _completionBlock(response);
25
26      }];
27
28   // This code is executed when the HTTP request fails
29   [request setFailedBlock:^{  _failedBlock(nil); }];
30
31   [request startAsynchronous];
32 }
```

Listing 3.12: Die Methode `postMyLocation:`

Diese Methode aktualisiert den derzeitigen Standort des iPhones auf dem Server. Dazu wird zunächst eine Instanz mit der URL zur PHP-API erzeugt, woraufhin die POST-

Daten gesetzt werden. Diese enthalten zum einen das Kommando `my_location`, die aktuelle Session und die geographischen Koordinaten. Anschließend werden die jeweiligen Blöcke gesetzt und die Anfrage schließlich asynchron gestartet.

Folgendes Beispiel zeigt, wie diese Klasse letztendlich von der Applikation genutzt werden kann.

```
1 ServerCommunicator *scom = [ServerCommunicator new];
2
3 scom.completionBlock = ^(Response *response){ /* completion code */};
4 scom.failedBlock = ^(Response *response){ /* failed code */ };
5
6 [scom postMyLocation:currentLocation];
```

Listing 3.13: Anwendungsbeispiel des `Servercommunicator`'s

3.3 Integration von Core Data für die Datenhaltung auf dem iPhone

Persistente Datenhaltung ist für diese Applikation kein triviales Thema. Sowohl die eigens gesetzten POIs, wie auch die Chatnachrichten und Freunde müssen in irgendeiner Art und Weise auf dem iPhone gespeichert werden, sodass diese Daten auch bei jedem Neustart wieder geladen werden können. Zudem müssen Probleme bezüglich *Multithreading* behandelt werden, da viele Prozesse im Hintergrund laufen und dies zur Inkonsistenz der Daten führen kann, wenn diese Prozesse gleichzeitig auf die Daten zugreifen.

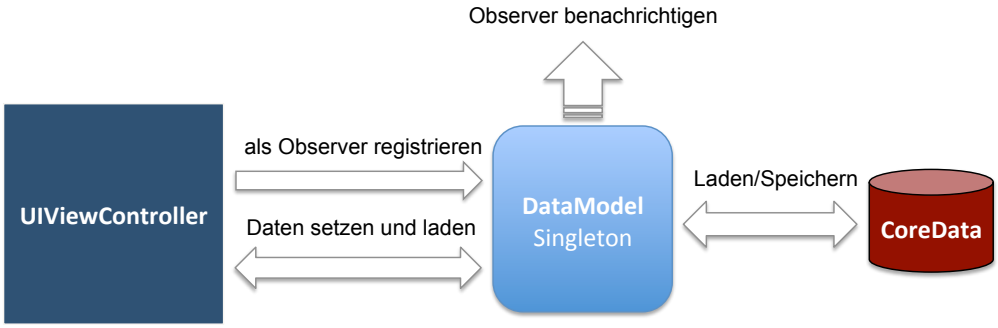

Abbildung 3.4: Das Data-Model zur Verwaltung der Daten mit *Core Data* als *Backend*.

Heutzutage richten sich fast alle objektorientierten Frameworks an das bereits mehrfach erwähnte MVC Paradigma. Auch die Designer von *Cocoa Touch* verfahren mit diesem Konzept der logischen Trennung zwischen der graphischen Benutzeroberfläche und

den Daten, die diese anzeigt. Wie bereits in Abschnitt 2.6 beschrieben, existieren viele Möglichkeiten persistente Datenhaltung umzusetzen. Aufgrund der dort beschriebenen Vorzüge wird an dieser Stelle das leistungsstarke *Core Data* Framework verwendet.

Hierzu ist die zentrale Klasse `DataModel` implementiert, die sich um die Verwaltung der Daten über Core Data kümmert und im Folgenden näher erläutert wird.

Das Data-Model

Die Klasse `DataModel` stellt, wie der Name schon andeutet, das *Model*-Objekt im MVC-Konzept dar. Diese Klasse verwaltet alle Daten, die von der Applikation dauerhaft gespeichert oder auch wieder geladen werden. Sie ist als *Singleton* entworfen. Eine Referenz auf die Instanz kann daher über die Klassenmethode `sharedInstance:` erhalten werden.

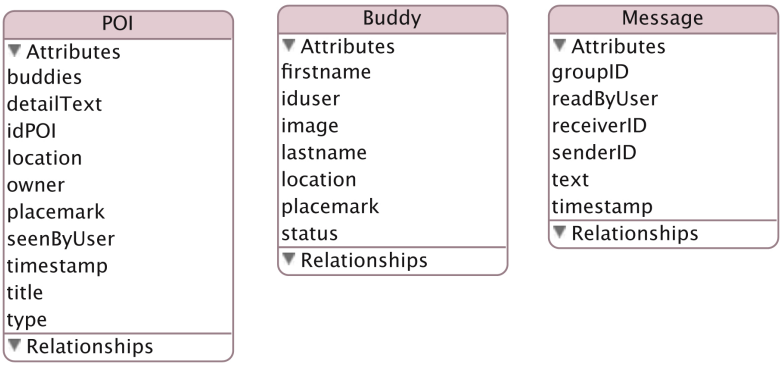

Abbildung 3.5: Die drei Entitäten `POI`, `Buddy` und `Message` als Datenobjekte für Core-Data.

Das Entwerfen der Datenstruktur mit *Core Data* kann gut mit dem Entwurf eines Datenbankschemas verglichen werden. Allerdings erhält jede Entität in diesem Sinne keine eigene Tabelle. In Abschnitt 2.6 wurde beschrieben, dass Datenobjekte in *Core Data* über Instanzen von `NSManagedObject` verwaltet werden. Daher wird für jede Entität ein eigene Klasse bereitgestellt, die von `NSManagedObject` erbt. Derzeit gibt es drei solcher Klassen: eine für die `POI`s bzw. für die Treffpunkte, eine für die Freunde (`Buddy`) und eine für die Chatnachrichten (`Message`). Eine Übersicht dieser Entitäten ist in Abbildung 3.5 dargestellt.

Die Header Datei, sowie auch die Implementierung einer solchen Entität ist in der Regel sehr übersichtlich strukturiert und kann über das Modellentwurfswerkzeug von XCode automatisch generiert werden. Der folgende Code zeigt als Beispiel die Header Datei für

die Chatnachrichten.

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface Message : NSManagedObject

@property (nonatomic, retain) NSString * senderID;
@property (nonatomic, retain) NSString * text;
@property (nonatomic, retain) NSDate * timestamp;
@property (nonatomic, retain) NSString * receiverID;
@property (nonatomic, retain) NSNumber * readByUser;
@property (nonatomic, retain) NSNumber * groupID;

// [...]

@end
```

Listing 3.14: Header Klasse der Entität `Message`

Diese besteht lediglich aus einer Reihe von *Properties* der Attribute. Die konkrete Implementierung dieser Entität ist im nächsten Ausschnitt dargestellt.

```
#import "Message.h"

@implementation Message

@dynamic senderID;
@dynamic text;
@dynamic timestamp;
@dynamic receiverID;
@dynamic readByUser;
@dynamic groupID;

// [...]

@end
```

Listing 3.15: Implementierung der Entität `Message`

Während dem *Compiler* bisher mit dem Schlüsselwort `@sythesize` signalisiert wurde, für die *Properties* die *Getter* und *Setter* Methoden zu generieren, wird an dieser Stelle `@dynamic` verwendet. Dies signalisiert dem *Compiler*, dass diese Methoden zwar nicht von der Klasse selbst, jedoch irgendwo anders (in diesem Fall von `NSManagedObject`) implementiert werden.

Die bereits erwähnte Klasse `DataModel` verwendet diese Objekte. Im Grunde stellt die Klasse sämtliche Methoden zur Modifizierung jeder einzelnen Entität zur Verfügung. Da diese allerdings sehr komplex ist, wird an dieser Stelle nur ein Teil der Funktionalität vorgestellt.

```
1  #define kKVOKeyPathMessage @"messages"
2
3  @interface DataModel : NSObject
4
5  @property (nonatomic, readonly) NSArray *messages;
6
7  - (NSArray*)messages;
8  - (void)commitMessageChangesNotifyObserver:(BOOL)notify;
9
10 @end
```

Listing 3.16: Ausschnitt der Klasse `DataModel.h`

In diesem Beispiel ist die Funktionalität auf die Chatnachrichten beschränkt, da der Code sonst den Rahmen dieser Arbeit überschreiten würde. Die Klasse definiert zunächst eine Reihe von *Strings*, die für das *Key-Value-Observing* (KVO) verwendet werden. KVO is ein Mechanismus, der erlaubt, Objekte zu benachrichtigen, wenn die spezifischen *Properties* modifiziert worden sind. In der Regel fügt sich ein `UIViewController` als *Observer* in das Datenmodell hinzu, sodass dieser seine View aktualisieren kann, wenn sich ein bestimmter Wert in diesem Model geändert hat. Dies ist exemplarisch in folgendem Code dargestellt.

```
1  DataModel *dataModel = [DataModel sharedInstance];
2  [dataModel addObserver:self forKeyPath:kKVOKeyPathMessages
3           options:NSKeyValueObservingOptionNew
4           context:nil];
5
6  // somewhere else in code
7  [dataModel removeObserver:self forKeyPath:kKVOKeyPathMessages];
```

Listing 3.17: Beispiel für eine Key-Value-Observing Anmeldung

Im Beispiel von Listing 3.17 „will“ ein `UIViewController` benachrichtigt werden, sobald sich etwas in den Chatnachrichten verändert. Dabei wird dann jedes Mal die unten stehende Methode aufgerufen, in der dann die *View* entsprechend aktualisiert werden kann. Der übergebene Pfad muss hierbei aber mit dem Namen der *Property* übereinstimmen. Um sich wieder als *Observer* zu entfernen, muss lediglich die Methode

`removeObserver:forKeyPath` aufgerufen werden, wie in Zeile 7 zu sehen ist.

```
1  - (void)observeValueForKeyPath:(NSString*)keyPath
2              ofObject:(id)Object
3              change:(NSDictionary*)change
4               context:(void*)context
5  {
6    if ([keyPath isEqualToString:kKVOKeyPathMessages])
7    {
8      // mark incomming messages as read, because we can read it directly
9          [self updateDataSourceAnimated:YES markMessagesRead:YES];
10   }
11 }
```

Listing 3.18: Diese Methode wird jedes Mal aufgerufen, sobald eine neue Nachricht im *Data Model* hinzugefügt wurde. Die *View* wird anschließend entsprechend aktualisiert.

Des Weiteren sind in dem Auschnitt der Klasse `DataModel` zwei Methoden aufgeführt. Über `messages` werden alle gespeicherten Chatnachrichten aus *Core Data* geladen und als `NSArray` an den Aufrufer übergeben.

```
1  - (NSArray*)messages
2  {
3    NSEntityDescription *entity;
4    entity = [NSEntityDescription entityForName:kEntityNameMessage
5                          nManagedObjectContext:_managedObjectContext];
6
7      NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
8      fetchRequest.entity = entity;
9
10     NSError *error;
11     NSArray *fetchedObjects = [_managedObjectContext executeFetchRequest:
           fetchRequest error:&error];
12
13     return fetchedObjects;
14 }
```

Listing 3.19: Implementierung der Methode `messages:`. Diese lädt über das *Core Data* Framework alle vorhandenen Nachrichten und liefert diese als Array zurück.

Es existieren noch weitere derartiger Methoden, die Nachrichten zum Beispiel sortiert nach Datum zurückliefern. Diese werden hier allerdings nicht näher behandelt. Das Datenmodell ist außerdem so entworfen, dass jede Änderung an den Daten nicht auto-

matisch gespeichert wird. Erst wenn dies explizit gefordert ist, wird dazu die Methode `commitMessageChangesNotifyObserver:` aufgerufen, mit der Option, die angemeldeten *Observer* über diese Änderung zu informieren.

```
- (void)commitMessageChangesNotifyObserver:(BOOL)notify
{
    if (notify)
        [self willChangeValueForKey:kKVOKeyPathMessages];

    NSError *error;
    [_managedObjectContext save:&error]; // commit changes
    if (error)
        ALog(@"Error: %@", [error description]);

    if (notify)
        [self didChangeValueForKey:kKVOKeyPathPOIS];
}
```

Listing 3.20: Diese Methode wird aufgerufen, um Änderungen am Data-Model persistent zu speichern und ggf. die *Observer* zu benachrichtigen.

Multithreading und Core Data

Das Thema *Multithreading* im Zusammenhang mit der Datenhaltung über das *Core Data Framework* soll in diesem Abschnitt diskutiert werden. Bereits in Abschnitt 2.6 wurde mit der Einführung des *Managed Object Contexts* auf die Problematik von threadübergreifenden Zugriffen hingewiesen. Ein Lösungsansatz ist, für jeden *Thread* einen eigenen Kontext bereitzustellen und alle Änderungen abschließend mit dem *Main*-Kontext zu *mergen*, der an dieser Stelle auch verwendet wird. Die Applikation lässt einige Prozesse in regelmäßigen Zeitintervallen im Hintergrund ablaufen, um zum Beispiel nach neuen Nachrichten auf dem Server oder nach Positionsänderungen der Freunde zu schauen. Bei zeitaufwendigeren Prozessen würde die Applikation jedes Mal einfrieren, wenn diese auf dem *Main-Thread* laufen würden.

Über die Klasse `BackgroundProcessController` können diese *Update*-Prozesse mit den jeweiligen Intervallen gestartet und wieder angehalten werden. Für jeden Prozess wird hier ein eigener Kontext bereitgestellt, über den die Änderungen am *Data Model* vorgenommen werden. Dieser Kontext muss dem *Data Model* übergeben werden, da dieser sonst auf den Haupt-Kontext arbeitet. Sobald die Änderungen persistent in dem übergebenen Kontext gespeichert werden, müssen diese anschließend mit dem Haupt-Kontext zusammengefasst werden. Ein Beispiel hierzu zeigt folgender Code-Ausschnitt.

```
1 AppDelegate *appDelegate = [AppDelegate sharedAppDelegate];
2 NSManagedObjectContext *mainContext = [appDelegate managedObjectContext];
3
4 // Merge changes into the main context on the main thread
5 [mainContext performSelectorOnMainThread:@selector(
      mergeChangesFromContextDidSaveNotification:)
6                                withObject:notification
7                             waitUntilDone:YES];
```

Listing 3.21: *Mergen* der Änderungen zweier Kontexte

Sobald also die Änderungen an den Daten des Kontexts eines weiteren *Threads* gespeichert werden, müssen diese mit dem Haupt-Kontext zusammengeführt werden. Hierzu wird als erstes aus dem `AppDelegate` eine Instanz dieses Kontextes erstellt (Zeile 1-2). Dieses Objekt führt schließlich das *Merging* beider Kontexte auf dem *Main Thread* durch (Zeile 5-7).

3.4 Kommponenten der Applikation

Die bisherigen Abschnitte haben zum größten Teil Technologien der Applikation erklärt, die zwar sehr wichtig, aber im Hintergrund ablaufen und für den Anwender nicht sichtbar sind. Dieser Abschnitt geht auf die Steuerung und die grafische Oberfläche der einzelnen Bestandteile der Applikation ein.

3.4.1 Authentifizierung und Registrierung

Beim Starten der Applikation wird der Nutzer zunächst aufgefordert, sich mit seiner E-Mail-Adresse, und dem Password anzumelden (vgl. Abbildung 3.6 a). Sollte noch kein Account vorhanden sein, muss dieser über den in Abbildung 3.6 dargestelten Registrierungsprozess angelegt werden. Hierzu muss der Nutzer im Anmeldebildschirm auf den im unteren Bereich befindlichen Text drücken, sodass eine neue Oberfläche erscheint.

Die Registrierung läuft in drei Schritten. Als erstes muss eine gültige E-Mail-Adresse angegeben werden, an die der Server eine E-Mail mit einem sechsstelligen Code sendet (vgl. Abbildung 3.6 b). Dieser Code ist nötig, um die Gültigkeit dieser E-Mail-Adresse im zweiten Schritt des Prozesses zu verifizieren (vgl. Abbildung 3.6 c). Bei korrekter Eingabe des Registrierungscodes wird der Nutzer im letzten Schritt aufgefordert, ein Passwort für diesen Account zu vergeben (vgl. Abbildung 3.6 d). Erst jetzt wird ein Eintrag in die Tabelle `User` mit den eingegebenen Daten erzeugt und kann in Zukunft verwendet werden. Die Kommunikation zwischen iPhone und Server geschieht, wie auch

bei allen anderen Anfragen, über den `ServerCommunicator`, der die nötigen Anfragen an den Web-Service schickt und verarbeitet.

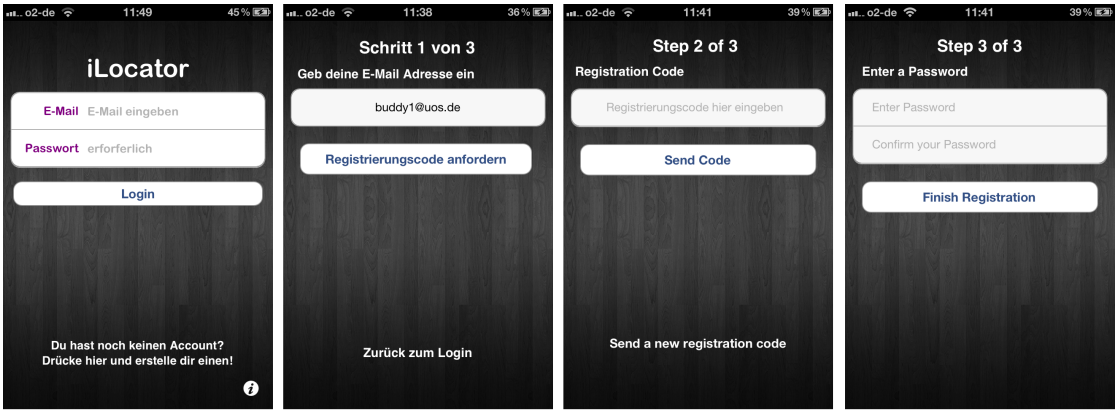

Abbildung 3.6: **Authentifizierung:** **(a)** Ameldebischirm, **(b)** Verfifizierungs-Code für E-Mail Adresse anfordern, **(c)** Code verfifizieren, **(d)** Passwort einrichten

Nach der Registrierung ist der Nutzer automatisch eingeloggt. Beim Login vergibt der Server eine eindeutige Session, die im weiteren Verlauf aller Anfragen mitgesendet werden muss, um die Anfrage zu autorisieren. Ohne die *Session* müsste sonst jedes Mal das Passwort und die E-Mail-Adresse mitgesendet werden. Außerdem ist es so möglich, den Online Status eines Nutzers zu bestimmen. Bis auf das Passwort werden diese Daten ebenfalls auf dem Gerät über das *Data Model* gespeichert, damit diese beim Starten der Applikation direkt für die Anmeldung verwendet werden können. So entfällt die erneute Anmeldung bei jedem Programmstart.

3.4.2 Kartenansicht

Bei erfolgreichem Login wird der Nutzer direkt zur Hauptansicht geführt – der Karte (siehe Abbildung 3.7). Diese bietet dem Anwender eine zweidimensionale Übersicht über die Standorte seiner Freunde, sowie der POIs. Für diesen Zweck wird das Framework `MapKit` eingesetzt, welches eine *Google*-Karte verwednet. Es ist möglich zwischen verschiedenen Kartentypen zu wechseln, wie es auch schon bei anderen Applikationen üblich ist. Hierzu kann entweder über den *Settings Button* in der rechten oberen Ecke des Bildschirms oder bequem mit einer Schüttelgeste zwischen den drei Kartentypen, *Standard*, *Hybrid* und *Satellite* gewechselt werden (vgl. Abbildung 3.7 a-c).

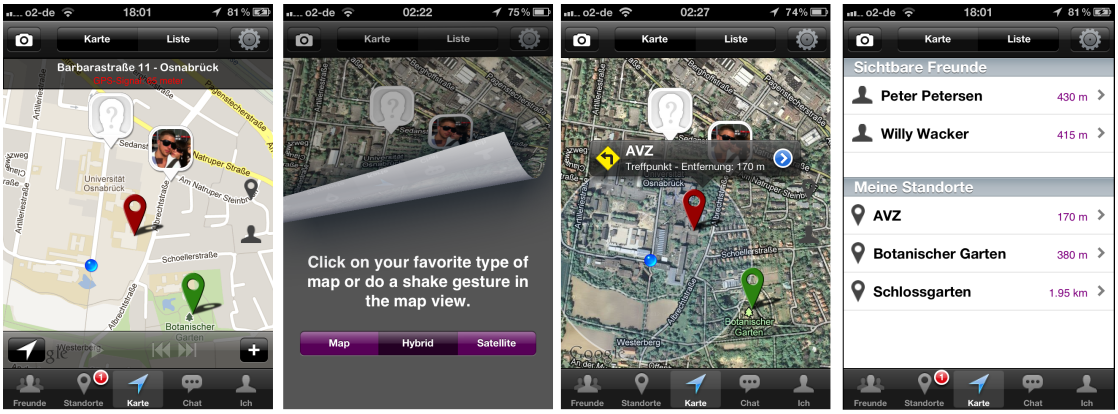

Abbildung 3.7: **Karte**: **(a)** Standard Ansicht, **(b)** Kartentyp ändern, **(c)** Hybrid Ansicht mit ausgewähltem POI, **(d)** Listen Ansicht

Da die Standorte der Freunde oder auch POIs auf der ganzen Welt verteilt sein können, besteht die Möglichkeit, über ein `SegmentedControl` im obreren Bereich zu einer anderen Ansicht *List* zu wechseln (vgl. Abbildung 3.7 d). Hier werden alle Freunde, die ihren Standort nicht verbergen, sowie alle POIs nach Distanz sortiert aufgelistet. Wählt man einen Eintrag dieser Tabelle aus, gelangt man sofort zu diesem Objekt auf der Karte.

Die Applikation empfängt zu diesem Zweck etwa alle zehn Sekunden die neuen Standorte der Freunde vom Server und benachrichtigt anschließend alle angemeldeten *Observer* über den KV-Mechanimus. Auch der `MapViewController` ist als *Observer* registriert und reagiert entsprechend auf die Änderungen. Sollte ein Freund zum Beispiel seinen Standort verbergen, so muss die entsprechende `Annotation` von der Karte entfernt oder ggf. hinzugefügt werden, wenn der Standort wieder freigegeben wird. Falls es sich jedoch um eine Standortänderung handelt, wird die neue Position `Annotation` per Animation aktualisiert. Da POIs in der Regel statisch sind und sich nicht bewegen, muss hier nicht viel beachtet werden.

Die Kartenansicht stellt noch weitere Features bereit. Es zeigt die aktuelle Adresse in Kurzform im oberen Bereich. Detaillierte Informationen über die Adresse erhält man durch Klicken auf dieses Adressfeld. Das Berühren einer `Annotation` auf der Karte öffnet zudem ein kleines Fenster mit den wichtigsten Informationen. Hierüber lässt es sich zum einen zur Detail-Ansicht wechseln (rechter Button); zum anderen kann eine Route zu diesem Ort berechnet werden (linker Button). Das Thema Routing wird in Abschnitt 3.4.5 näher vorgestellt.

Zu guter Letzt gelangt man über den Button in der linken oberen Ecke zu der *Augmented Reality* Ansicht, die in Abschnitt 3.4.8 näher beschrieben ist.

3.4.3 Eigenes Profil

Die Applikation verfügt auch über ein Menü, in der das eigene Profil angezeigt wird (vgl. Abbildung 3.8 a). Derzeit zeigt diese Sicht lediglich die eigene Position auf einer Karte, sowie die aktuelle Adresse an. Besonders wichtig ist hier die Funktion, seinen eigenen Standort zu verbergen oder anzuzeigen. Im Hintergrund läuft permanent ein `LocationController`, der, je nach eingestelltem Distanzfilter, die aktuellen geographischen Koordinaten an den Server schickt, damit die Freunde den momentanen Standort ebenfalls auf dem eigenen iPhone sehen können. So wird etwa nach einer Distanz von jeweils zehn Metern die momentante Position aktualisiert, wenn dies zugelassen wird.

Abbildung 3.8: Profil und Einstellungen

Ebenfalls gelangt man von hier aus zu den Einstellungen der Applikation (vgl. Abbildung 3.8 b). Hier können diverse Einstellungen vorgenommen werden, die momentan eher für die Entwicklung der Applikation von Bedeutung sind. Der Nutzer hat jedoch die Möglichkeit, sich hierüber auszuloggen (vgl. Abbildung 3.8 c).

3.4.4 Freunde verwalten

Ein soziales Netzwerk entsteht erst, wenn der Anwender in irgendeiner Art und Weise mit anderen Nutzern dieses Systems verbunden ist (vgl. Abschnitt 1.2). Diese Nutzer

werden nicht nur in der Kartenansicht, sondern auch im Bereich *Freunde* (oder engl. *Buddies*) mit dem `BuddyViewController` dargestellt. Die Applikation geht dabei die Kontakte auf dem iPhone durch und vergleicht die gespeicherten E-Mail-Adressen mit denen der registrierten Nutzer in der Datenbank. Der dafür vorgesehene Web-Service liefert eine Liste mit allen gefundenen Adressen, sodass anschließend alle weiteren Information (Name, Foto etc.) zu diesen Personen über die Kontakte des iPhones automatisch ausgelesen und über das *Data Model* gespeichert werden können.

Abbildung 3.9: **Freunde**: **(a)** Leere Liste, **(b)** Liste mit zwei Freunden, sortiert nach Vorname, **(c)** Detailansicht, **(d)** Routing Menü

Bei der ersten Anmeldung ist die Liste der Freunde leer (vgl. Abbildung 3.9 a). Über den Refresh-Button in der rechten oberen Ecke wird veranlasst, über die Kontakte des iPhones nach Freunden in der Datenbank zu suchen und diese in der App hinzuzufügen. Diese werden dann in einer Tabelle sortiert nach Vorname, Nachname oder Entfernung aufgelistet (vgl.Abbildung 3.9 b).

Durch Berühren einer dieser Freunde öffnet sich eine neue Ansicht, die mehr Details zu dieser Person anzeigt (vgl.Abbildung 3.9 c). Auch hier besteht die Möglichkeit, sich mit dem rechten Button eine Route zu der Person berechnen zu lassen. Der linke Button öffnet den Chat, sodass hierüber mit dem Freund kommuniziert werden kann.

3.4.5 Routing

Die Applikation bietet die Möglichkeit, Routen zu seinen POIs oder Freunden zu berechnen und sich navigieren zu lassen. Dies ist, wie bereits erwähnt, zum einen direkt in der Kartenansicht möglich, indem auf den *Route-Button* einer *Annotation* gedrückt wird. Zum anderen kann diese Funktion auch in den Detail-Ansichten der Freunde und POIs selber genutzt werden.

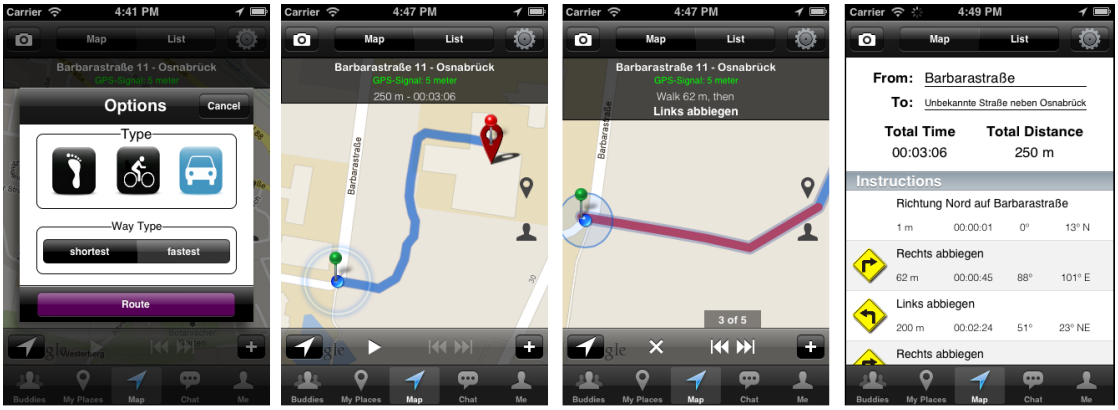

Abbildung 3.10: **Routing: (a)** Einstellungen der Routenberechnung, **(b)** Gesamte Route auf der Karte **(c)** Wegsemgent mit Instruktion, **(d)** Instruktionsliste

Das in Abbildung 3.10 gezeigte Beispiel nutzt die Funktion des *Routing* innerhalb der Kartenansicht. Sobald der *Route-Button* betätigt wird, öffnet sich ein neues Fenster (der Klasse `RoutePopupViewController`), über das diverse Einstellungen zur Berechnung der Route vorgenommen werden können (vgl. Abbildung 3.10 a). Hier kann zum Beispiel die Art der Fortbewegung, sowie die schnellste oder kürzeste Route angegeben werden. Diese Daten werden von der Applikation an den in Abschnitt 2.2 beschriebenen *Routing Service* von *CloudMade* geschickt. Die Empfangenen Daten zur errechneten Route werden anschließend so aufbereitet, dass diese auf der Karte als Polygonzug (`MKPolyline`) angezeigt werden kann (vgl. Abbildung 3.10 a,b).

Ist eine Route über *CloudMade* errechnet worden, wechselt die Kartenansicht in den Navigationsmodus und skaliert die Karte so, dass die komplette Route dargestellt wird. Das Informationslabel im oberen Bereich zeigt nun neben der aktuellen Adresse auch die Länge, sowie die Zeit der gesamten Route (vgl. Abbildung 3.10 b). Die Toolbar im

unteren Bereich verfügt nun über einen *Play-Button*, mit dem die Navigation gestartet, sowie beendet werden kann. Ähnlich wie bei der *GoogleMaps*-App auf dem iPhone kann sich der Anwender anhand der von *CloudMade* mitgelieferten Instruktionen zu den Route-Segmenten zum Ziel navigieren lassen. Dabei wird jedes Wegsegment farbig hervorgehoben und die Instruktion ebenfalls im oberen Informationsbereich angezeigt (vgl. Abbildung 3.10 c). Mit den beiden Buttons auf der rechten Seite der Toolbar kann zwischen diesen Segmenten gewechselt werden. In der weiter oben beschriebenen Listen-Ansicht werden nun nicht mehr die Freunde und POIs angezeigt, sondern detaillierte Informationen zu der Route mit den jeweiligen Instruktionen aufgelistet (vgl. Abbildung 3.10 d).

3.4.6 Kommunikation zwischen den Clients

Eine weitere Eigenschaft der Applikation ist die Kommunikation der *Clients* untereinander. Über den Chat können die Nutzer ihren Freunden Nachrichten schicken und auch welche von ihnen empfangen. Das Senden einer Nachricht geschieht wieder über den `ServerCommunicator` mit der Methode `postMessage:toUserID:`. Der dazugehörige Web-Service benötigt lediglich die zu sendende Nachricht, sowie die E-Mail-Adresse des Empfängers. Anhand der mitgeschickten *Session* kann der Sender über die Datenbank ermittelt und ebenfalls in der Tabelle `Message` vermerkt werden. Jede neu eingefügte Nachricht wird als noch nicht gesendet markiert, indem das Attribut `time_sent` auf Null gesetzt wird.

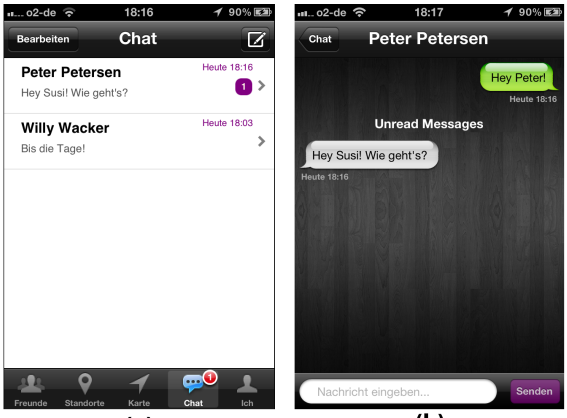

Abbildung 3.11: **Chat: (a)** Liste aller Konversationen, **(b)** Unterhaltung mit einem Freund

Die Applikation schaut mithilfe des bereits erwähnten `BackgroundProcessController`'s in regelmäßigen Zeit-Intervallen, ob neue Nachrichten auf dem Server liegen, lädt die entsprechen herunter und markiert sie als gesendet. So werden diese bei der nächsten Anfrage ignoriert. Anhand der mitgelieferten E-Mail-Adresse des Senders kann somit ermittelt werden, von welchem Freund die Nachricht stammt. Sobald eine oder mehrere Nachrichten empfangen werden, zeigt das Icon dieser Ansicht im unteren Bereich der *Tabbar* die Anzahl der ungelesenen Konversationen in einer roten Blase an.

In der Hauptansicht des Chat werden alle Konversationen nach Datum sortiert aufgelistet. Eine Konversation ist in diesem Sinne die Menge aller gelesenen und ungelesenen Nachrichten zwischen dem Nutzer und einem weiteren Freund. Die Anzahl der ungelesenen Nachrichten einer Konversation wird wiederum in der Tabelle angezeigt.

3.4.7 Verwaltung von Treffpunkten und POI's

Abbildung 3.12: **POI: (a)** Liste aller POIs und Treffpunkte, **(b)** Detailansicht eines POIs, **(c)** Teilen eines POIs mit seinen Freunden, **(d)** Hinzufügen eines neuen POI's auf der Karte

Eine weitere zentrale Komponente der Applikation stellt die Verwaltung der POIs dar. Hier werden alle selbsterstellten, sowie sämtliche von oder mit seinen Freunden geteilten POIs bzw. Treffpunkte aufgelistet. Auch diese können nach Name, Distanz oder Erstellungsdatum sortiert werden (vgl. Abbildung 3.12 a). In der Detailansicht werden die gesamten Informationen zu diesem Standort tabellarisch aufgeführt. Der Anwender kann sich hierüber wieder eine Route zu diesem Standort berechnen und sich anschlie-

ßend dorthin navigieren lassen. Mit dem grünen Button öffnet sich ein kleines Popup, in der ausgewählt werden kann, wo oder mit wem dieser POI geteilt werden soll. Derzeit ist lediglich das Teilen mit den eigenen Freunden implementiert, daher sind die anderen beiden Möglichkeiten nur Platzhalter (vgl. Abbildung 3.12 c).

Soll ein neuer POI bzw. Treffpunkt hinzugefügt werden, geschieht das über ein eigenes Fenster (vgl. Abbildung 3.12 d). Das Fadenkreuz in der Mitte gibt die Position des zu erstellenden Standorts auf der Karte an. Zur Orientierung werden hier ebenfalls alle anderen POIs und Freunde mit angezeigt. Eine kleine Besonderheit stellt die Suchleiste im oberen Bereich dar. Hier kann eine beliebige Adresse eingegeben werden. Aus einer Liste möglicher Treffer zentriert sich die Karte auf das ausgewählte, sodass diese anschließend als neuer Standort abgespeichert und mit in die Liste der POIs aufgenommen werden kann.

3.4.8 Integration von Augmented-Reality

Die Integration der in Abschnitt 1.3 beschriebenen *Augmented Reality* Technologie in diese App war eine der größten Herausforderungen dieses Projekts. Dieser Abschnitt befasst sich daher etwas genauer mit der Implementierung. Es wird zunächst ein Überblick über das eigens implementierte Framework gegeben. Anschließend folgt eine nähere Beschreibung des eigentlichen Algorithmus zur Darstellung der AR, sowie der Transformation der GPS-Koordinaten in ein für diese Zwecke geeigneteres Koordinatensystem.

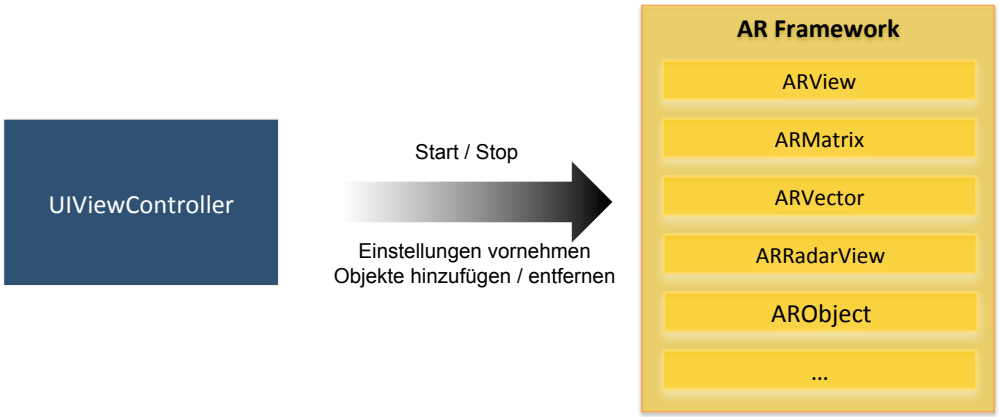

Abbildung 3.13: Die Klassen des AR-Frameworks und die Verwendung der `ARView` in einem `UIViewController`.

Das AR-Framework im Überblick

Leider gab es zum Zeitpunkt der Bearbeitung kein geeignetes, kostenfreies AR-Framework, welches an dieser Stelle hätte eingesetzt werden können. Daher musste ein eigenes kleines Framework implementiert werden, welches in diesem Abschnitt näher erklärt wird.

Das Framework besteht aus diversen Klassen, wie in Abbildung 3.13 dargestellt ist. Dazu gehört vor allem die Hauptklasse `ARView`, welche die gesamte AR-Logik enthält, sowie die Helferklassen `ARMatrix` und `ARVektor` mit den nötigen algebraischen Funktionen. Zudem repräsentiert die Klasse `ARRadarView` ein Radar, welche `ARObject`-konforme Objekte in einer *View* darstellt. Auf die anderen Klassen wird hier nicht näher eingegangen, da diese weniger von Bedeutung sind.

Das Framework ist so entworfen, dass der *View* eines `UIViewControllers` lediglich eine Instanz von `ARView` zugewiesen werden muss. Das folgende Beispiel soll dies verdeutlichen.

```
1  ARView *arview = [[ARView  alloc] initWithFrame:self.view.frame];
2  self.view = arview;
3
4  [arview initilize];
5  arView.motionManager  = [MotionManager sharedInstance];
6  arView.updateInterval = 1.0/5.0;
7  arView.delegate       = self;
8
9  // [...] prepare AR-Objects
10 [arView addARObjects:arObjects];
11
12 [arView start];
```

Listing 3.22: Konfiguration und Start der `ARView`

Nach Zuweisung der `ARView`-Instanz muss die *View* zunächst über `initialize` initialisiert werden. Hier werden unter anderem alle *Subviews* angepasst, sowie die nötigen Projektionsmatrizen und der *Location Manager* initialisiert. In Abschnitt 2.5 wurde erwähnt, dass es empfehlenswert ist, *Core Motion* als *Singleton* zu verwenden. Daher stellt `ARView` keinen eigenen `CMMotionManager` zur Verfügung, sondern lediglich ein Objekt, das darauf verweist. Dieses wird, wie in Zeile 5 zu sehen ist, separat zugewiesen. Das *Update*-Intervall gibt an, wie schnell `ARView` sich aktualisieren soll. Zudem verwendet die Klasse das *DelegatePattern*, sodass bestimmte Methoden an den *View Controller* weitergeleitet werden, sofern dieser `ARViewDelegate` konform ist.

Damit auch Objekte in der *View* angezeigt werden können, müssen schließlich `ARObject`-konforme (vgl. Listing 3.23) Objekte hinzugefügt werden. Das Protokoll `ARObject` erfordert hierbei nur, dass die implementierenden Objekte zwei *Views* bereitstellen, eine für die AR-Ansicht und eine für das Radar. Somit ist das Aussehen vom Framework entkoppelt und kann jederzeit und für jedes Objekt selber gestaltet werden. Außerdem muss das Objekt einen Standort bereitstellen, damit diese auch angezeigt werden können.

```
1  #import <Foundation/Foundation.h>
2
3  @class CLLocation;
4  @protocol ARObject <NSObject>
5
6  @required
7  @property (nonatomic, strong) UIView *arView;
8  @property (nonatomic, strong) UIView *radarView;
9  @property (nonatomic, strong) CLLocation *location;
10
11 @end
```

Listing 3.23: Das `ARObject` Protokoll

Zu guter Letzt startet die `ARView` ihren Algorithmus über den Methodenaufruf `[arView start]`. Dieser kann wiederum durch den Aufruf von `[arView stop]]` angehalten werden. Der folgende Abschnitt geht auf die Arbeitsweise dieses Algorithmus ein.

Der AR-Algorithmus

Über die Methode `start:` wird das `ARView`-Objekt veranlasst, den AR-Algorithmus zu starten. Dieser Algorithmus sorgt dafür, dass die eingefügten `ARObjekte` anhand ihrer geographischen Position korrekt auf das Kamerabild projiziert werden, sodass der Eindruck entsteht, diese Objekte befänden sich in der realen Umgebung. So wird beim Beginn nicht nur ein `CLLocationManager` und der übergebene `CMMotionManager` gestartet, damit die geographische Lage, sowie die Ausrichtung des iPhones ermittelt werden kann, sondern auch ein *Timer*, der in den eingestellten Intervallen die Methode `updateAR:` aufruft.

Wie in Abbildung 3.14 zu sehen ist, besteht die Methode im Wesentlichen aus zwei weiteren Methodenaufrufen. Sie aktualisiert zum einen die Kamera-Matrix, also die Ausrichtung des iPhones über den `CMMotionManager` und passt anschließend die Projektionen der AR-Objekte entsprechend an. Die größte Schwierigkeit hierbei liegt in der korrekten Berechnung der Transformationsmatrix in Abhängigkeit der Ausrichtung des iPhones,

sowie der gewählten Projektion. Für diese Zwecke stellt das *Core Motion* Framework, wie breits in Abschnitt 2.5 diskutiert, seit iOS 4.0 eine Rotationsmatrix zur Verfügung, die die Ausrichtung des iPhones beschreibt. Somit entfällt das Aufstellen einer eigenen Matrix anhand der Rotationswinkel der drei Raumachsen.

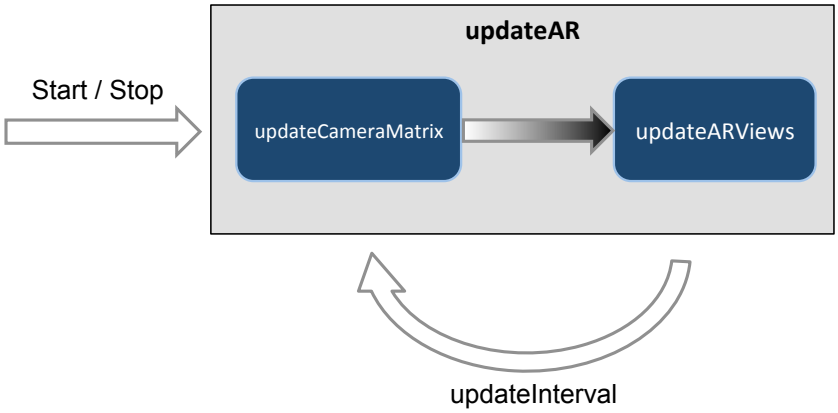

Abbildung 3.14: Ablauf der Methode `updateAR:`. Sie wird in regelmäßigen Zeitintervallen von einem *Timer* aufgerufen, um die AR-Objekte an die Ausrichtung des iPhones anzupassen.

Abbildung 3.16 stellt den schematischen Verlauf dieser Methode dar. Zunächst wird eine Transformationsmatrix (`PCM`) als Produkt der Projektionsmatrix und der aktuellen Kameramatrix aufgestellt. Anhand dieser Matrix wird schließlich für jedes AR-Objekt die Projektion neu berechnet und die Position diesbezüglich angepasst. Zugleich wird über den *Motion Manager* die Neigung des iPhones ermittelt, sodass sich die *Views* der Objekte entsprechend mitdrehen und stets vertikal zum Betrachter ausgerichtet sind.

Das Anpassen der Projektionen anhand der Ausrichtung des iPhones reicht jedoch alleine nicht aus. Da sich ebenfalls die geographische Position der AR-Objekte einerseits ändern kann, wenn es sich zum Beispiel um einen durch die Stadt gehenden Freund handelt, sich aber auch andererseits die eigene Position ändern kann, müssen die Koordinaten der Objekte neu berechnet werden. An dieser Stelle muss erwähnt werden, dass nicht mit den rohen GPS-Koordinaten der Standorte gerechnet wird. Diese werden zunächst in ein geeignetes Koordinatensystem transformiert.

In Abschnitt 2.3.1 wurden drei geographische Koordinatensysteme vorgestellt. Zum einen das WGS84, in dem die GPS-Koordinaten über Längen- und Breitengrad, sowie die Hö-

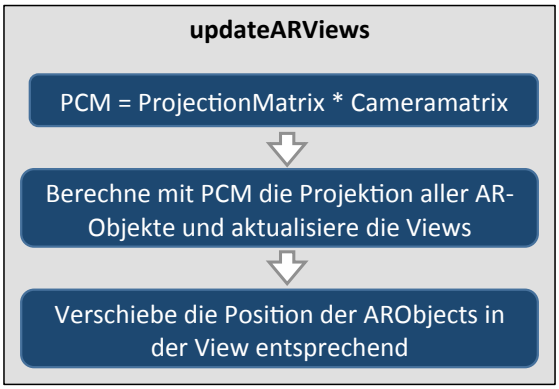

Abbildung 3.15: Algorithmus zum Anpassen der Projektionen an die Ausrichtung des iPhones.

he in Bezug auf das Geoid angegeben werden und zum anderen das ECEF, welches die Position eines Objekts in kartesischen Koordinaten angibt, wobei der Ursprung im Schwerpunkt der Erde liegt. Das dritte vorgestellte geographische Koordinatensystem ENU ist besonders für Tracking-Anwendungen geeignet, da sich der Ursprung an einem lokalen Punkt der Erdoberfläche befindet. Hierbei zeigt die x-Achse nach Osten, die y-Achse nach Norden und die z-Achse nach oben. Daher werden die GPS-Koordinaten in eben dieses Koordinatensystem transformiert. Der Ablauf ist in Abbildung 3.16 dargestellt.

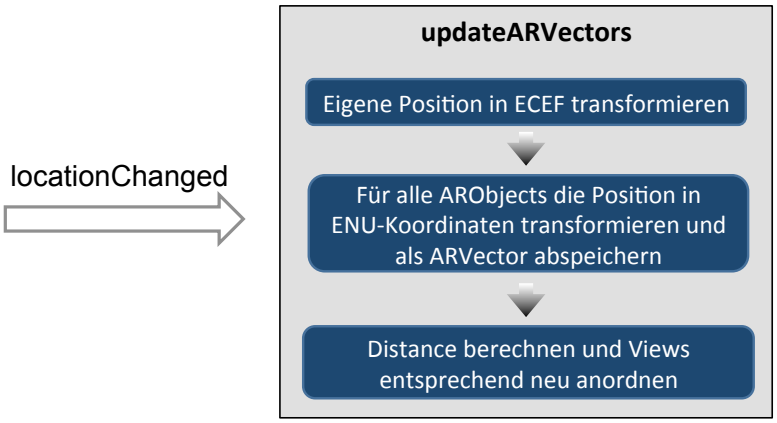

Abbildung 3.16: Schema der Methode `updateARVector`, die jedes Mal aufgerufen wird, wenn sich die eigene Position oder die der AR-Objekte ändert.

Die Transformation geschieht in zwei Schritten. Zunächst muss die geographische Position des iPhones, sowie die der AR-Objekte in ECEF-Koordinaten umgerechnet werden. Erst dann ist es möglich, die Position der AR-Objekte im Bezug auf das iPhone in ENU-Koordinaten zu beschreiben. Diese neuen Koordinaten werden hierbei als Vektoren (`ARVector`) in einem eigenen *Array* gespeichert, mit denen der Algorithmus dann letztendlich rechnet. Schließlich werden auch die *Views* anhand der Distanz zum iPhone selbst neu angeordnet, sodass die weiter weg liegenden Objekte auch in der *View* von den näher gelegenen überdeckt werden. Hier ist es auch möglich, die Views entsprechend der Distanz zu skalieren. Darauf wurde aber bei der Implementierung verzichtet.

Der folgende Abschnitt zeigt, wie die GPS-Koordinaten in das ENU-Koordinatensystem transformiert werden.

Transformation der GPS Koordinaten

Im Rahmen dieser Masterarbeit müssen für die Umsetzung der *Augmented Reality*, wie in Abschnitt 1.3 beschrieben, Koordinatentransformationen vorgenommen werden. Die Transformation der von dem Gerät gewonnenen geodätischen GPS-Koordinaten in das ENU-Koordinatensystem geschieht in zwei Schritten. Zunächst müssen die Koordinaten mit Längen- (λ) und Breitengrad (ϕ) in das ECEF-Koordinatensystem transformiert und von da aus in das ENU-Koordinatensystem überführt werden (vgl. [13]).

1. WGS84 $\rightarrow$ ECEF Wie bereits in Abschnitt 2.3.1 erwähnt, verwendet das WGS84 ein Referenzellipsoid, um die Erdoberfläche Best möglichst mathematisch anzunähern. Dazu muss die große Halbachse a (*Semi-Major Axis*) und die kleine Halbachse b (der Polarradius) oder die große Halbachse und die Abplattung (*Flattening*) $f = (a - b)/a$ bekannt sein. Für das WGS84 sind folgende Werte festgelegt worden:

$$\begin{aligned} \text{Semi-Major Axis a} &= 6378137.0\ m \\ \text{Reciprocal of Flattening 1/f} &= 289.257223563 \end{aligned}$$

Mit diesen Werten und der folgenden Gleichung

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} (h+N)\cos(\lambda)\cos(\phi) \\ (h+N)\cos(\lambda)\sin(\phi) \\ (h+N(1-e^2))\sin(\lambda) \end{pmatrix} \tag{3.1}$$

wobei

$$N(\phi) = \frac{a}{\sqrt{1 - e^2 \sin^2 \phi}}$$

gilt, mit der Höhe (*Altitude*) h, Längengrad (*Longitude*) λ und Breitengrad (*Latitude*) ϕ, können im ersten Schritt die vom iPhone gelieferten GPS-Koordinaten in das ECEF-Koordinatensystem transformiert werden.

2. ECEF → ENU Die neuen Koordinaten liegen nun als $[x, y, z]$ Tupel in kartesischen Koordinaten vor, wobei der Ursprung im Massenmittelpunkt der Erde liegt (vgl. Abschnitt 2.3.1). Da es für die weiteren Rechenoperationen auf dem iPhone wesentlich intuitiver und effektiver ist, wenn sich der Ursprung direkt am iPhone befindet, muss im zweiten Schritt eine weitere Koordinatentransformation vorgenommen werden. Hierzu hat sich das ENU-Koordinatensystem als adäquat erwiesen.

Für die Umrechnung einer Koordinate (p) im ECEF-System wird ein Referenzpunkt (r) benötigt, der den Ursprung des neuen Koordinatensystems darstellt. In dieser Arbeit ist dieser Referenzpunkt also stets die Position des eigenen iPhones. Auch diese Position muss in ECEF-Koordinaten vorliegen. Mit diesen Informationen und der (Gleichung 3.2) kann also schließlich in das gewünschte ENU-Koordinatensystem transformiert werden.

$$\begin{pmatrix} e \\ n \\ u \end{pmatrix} = \begin{pmatrix} -\sin\lambda_r & \cos\lambda_r & 0 \\ -\sin\phi_r \cos\lambda_r & -\sin\phi_r \sin\lambda_r & \cos\phi_r \\ \cos\phi_r \cos\lambda_r & \cos\phi_r \sin\lambda_r & \sin\phi_r \end{pmatrix} \cdot \begin{pmatrix} X_p - X_r \\ Y_p - Y_r \\ Z_p - Z_r \end{pmatrix} \tag{3.2}$$

Die ursprünglichen GPS-Koordinaten sind letztendlich so transformiert worden, dass sie die geographische Lage relativ zum iPhone beschreiben. Das hat viele Vorteile: zum einen ist die Abstandsberechnung zum eigenen Gerät einfacher (Abstand zum Ursprung), da nur noch der Betrag des Vektors ausgerechnet werden muss. Zum anderen ist es so wesentlich effektiver alle virtuellen Objekte in Abhängigkeit der Position und Ausrichtung des iPhones zu transformieren.

Das Ergebnis

Das Ergebnis dieser Berechnung ist in Abbildung 3.17 dargestellt. Die AR-Objekte werden nun korrekt in das Kamerabild eingefügt, sodass der Eindruck entsteht, sie würden tatsächlich in der realen Umgebung existieren.

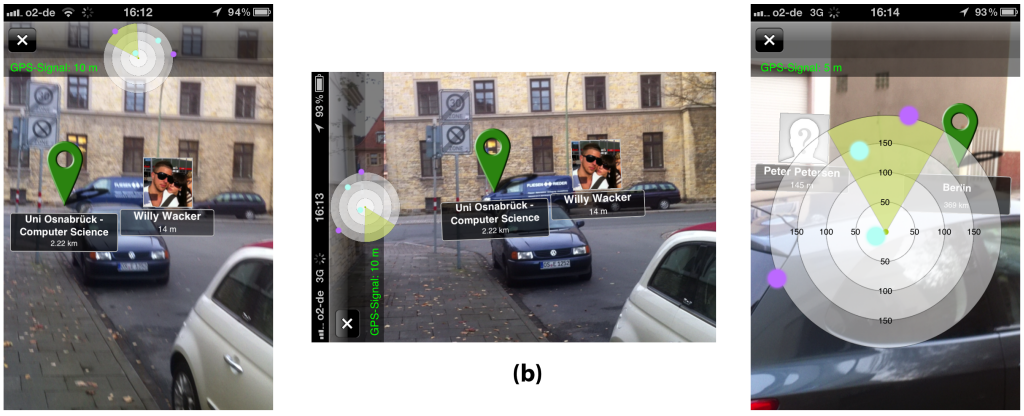

Abbildung 3.17: **Kamera mit AR:** `(a)` Ein POI und ein Freund sind zu sehen, `(b)` Die Objekte drehen sich entsprechend der Ausrichtung des iPhones mit, `(c)` Das kleine Radar wird größer, wenn man es berührt.

Die AR-Ansicht besteht im Grunde aus dem Kamerabild, dem die Projektionen der AR-Objekte überlagert werden. Im oberen Bereich befindet sich eine Leiste mit einem kleinen Radar (`ARRadarView`), der die AR-Objekte in der Umgebung anzeigt (vgl. Abbildung 3.17 a). Außerdem wird die Genauigkeit des GPS-Signals dargestellt. So kann abgewogen werden, wie genau die Objekte relativ zu dem iPhone projiziert werden. Wie bereits erwähnt, drehen sich die Projektionen der AR-Objekte entsprechend der Orientierung des iPhones mit, sodass diese stets vertikal zu dem Betrachter ausgerichtet sind (vgl. Abbildung 3.17 b). Schließlich kann das Radar im oberen Bereich durch eine Berührung vergrößert und durch erneute Berührung wieder verkleinert werden.

4 Fazit und Ausblick

Die im Rahmen dieser Masterarbeit erzielten Erfahrungen und Ergebnise sind sehr facettenreich. Die von den in der Einleitung erörterten Anforderungen des Themas sind im Laufe der Entwicklungsphase höher gewesen, als anfänglich vermutet. Es musste nicht nur eine iPhone Applikation mit den geforderten Eigenschaften implementiert werden, sondern auch ein serverseitiges System mit Datenbank und Web-Services, dass auf den Funktionsumfang dieser App zugeschnitten ist.

Allerdings gibt es noch weitere Überlegungen und Technologien, die in einer eventuell weiteren Entwicklungsphase umgesetzt werden können. Die folgenden beiden Abschnitte erklären kurz, welche konkreten Technologien gemeint sind, bevor schließlich auf ds Fazit eingegangen wird.

Native Apps vs. Web Apps

Bei dem Ergebnis dieser Arbeit handelt es sich um eine native App. Sie ist speziell für das iOS Betriebssystem programmiert und läuft dementsprechend ausschließlich auf iOS Geräten wie dem iPhone, iPad oder dem iPod Touch. Das hat den großen Vorteil, dass sichergestellt ist, dass die Ressourcen des Geräts optimal genutzt werden. Leider ist dieses Konzept nicht mehr zeitgemäß. Derzeit existieren zahlreiche große Plattformen mobiler Geräte, wie *iOS*, *Android*, *Windows Phone OS* und viele mehr. Die Vermarktung nativer Apps verschiedener Plattformen und Geräte ist daher sehr zeitaufwendig und kostenintensiv.

Aus diesem Grund gibt es Diskussionen über die beste Methode zur Entwicklung mobiler Apps. Ein großes Thema stellt die Entwicklung so genannter Web Apps dar. Hier wird eine Applikation über speziell programmierte HTML5 Seiten entwickelt. Das Endgerät interpretiert diesen Code und stellt den Inhalt entsprechend optimiert dar. Das hat den großen Vorteil, dass mit nur einem Code, gleich mehrere Endgeräte und Plattformen in der Lage sind, diese App zu verwenden.

Es ist nicht auszuschließen, dass auch diese App eines Tages als Web App implementiert wird, allerdings gibt es derzeit einige Funktionen, die nur native Apps aufweisen. Dazu gehört zum Beispiel die Erfassung von Bewegung und Neugung durch ein Gyroskop.

Nutzung von Push Notification

Für mobile Endgeräte ist es wegen der begrenzten Ressourcen, wie Akkulaufzeit und Rechenleistung, immer von Vorteil, wenn Prozesse ausgelagert (z.B. auf einen Server) werden können. iOS-Applikationen sind leider nur bedingt in der Lage, im Hintergrund weiterzuarbeiten, nachdem der *Home Button* gedrückt wurde. Daher ist es so nicht möglich von der App benachrichtigt zu werden, wenn zum Beispiel neue Nachrichten empfangen wurden. Apple stellt zu diesem Zweck einen *Service* zur Verfügung, mit der eine Applikation Benachrichtigungen vom Server erhält, auch wenn diese nicht mehr aktiv ist – den *Push Notification Service*.

Der gegenwärtige Stand der App verwendet diesen Service nicht. Neue Chatnachrichten oder geteilte POIs bzw. Treffpunkte können somit nur im laufenden Betrieb erhalten werden. Ist die Anwendung inaktiv, so wird der Anwender über neue Inhalte nicht benachrichtigt. Für die Zukunft könnte man sich allerdings überlegen, diesen Service zu implementieren. Zur Zeit der Bearbeitung und im Zusammenhang mit den Anforderungen dieser Arbeit wurde jedoch vorerst davon abgesehen, da die Implementierung nicht ganz trivial ist.

4.1 Fazit

Zusammenfassend kann jedoch gesagt werden, dass mit *iLocator* eine funktionsfähige iPhone Appliaktion entstanden ist, die allen in der Einleitung dargelegten Anforderungen entspricht. Der Anwender kann mit dieser App zum seine Freunde verwalten, mit ihnen kommunizieren, sowie Treffpunkte oder andere POIs mit ihnen teilen, sodass diese entweder über eine Karte oder *Augmented Reality* Ansicht dargestellt werden. Außerdem ist eine *Routing*-Funktion implementiert, wodurch der Anwender in der Lage ist, sich zu einen Freunden oder anderen Standorten navigieren zu lassen. Des Weiteren ist die App so entworfen worden, dass sie auch in mehreren Sprachen angezeigt werden kann (*Localized*) – derzeit auf englisch und zum Teil bereits auf deutsch.

Literatur- und Webverzeichnis

[1] **S.P. Drake**, *Converting GPS Coordinates ($\phi\lambda h$) to Navigation Coordinates (*ENU*)*, DSTO Electronics and Surveillance Research Laboratory 2002

[2] **Dr. Jens Groß**, *Kartographie* (Vorlesungsskript), Leibniz Universität Hannover 2010

[3] **Prof. Dr. Norbert de Lange**, *Grundlagen der Geoinformatik und GIS* (Vorlesungsskript), Universität Osnabrück 2010

[4] **v.A.**, Geoinformatik-Service, Universität Rostock
`http://www.geoinformatik.uni-rostock.de/lexikon.asp`

[5] **D. Mark, J. Nutting, J. LaMarche**, *Beginning iOS 5 Development - Exploring the iOS SDK*, Apress 2011

[6] **v.A.**, *The View Controller Programming Guide for iOS*
`http://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/AboutViewControllers/AboutViewControllers.html#//apple_ref/doc/uid/TP40007457-CH112-SW10`

[7] **v.A.**, *The Objective-C Programming Guide*
`http://developer.apple.com/library/mac/#documentation/cocoa/conceptual/objectivec/Introduction/introObjectiveC.html`

[8] **D. M. Boyd, N.B. Ellison**, *Social Network Sites: Definition, History, and Scholarship*, Journal of Computer-Mediated Communication Vol. 13 Iss. 1 2007, S. 210-230

[9] **Lee Humphreys**, *Mobile Social Networks and Social Practice: A Case Study of Dodgebal*, Journal of Computer-Mediated Communication Vol. 13 Iss. 1 2007, S. 341-360

[10] **W. Broll et al.**, *Toward Next-Gen Mobile AR Games*, Frauenhofer Institute for Applied Information Technology - IEEE Computer Society 2008

[11] **A.H. Behzadan, V.R. Kamat**, *Visualization of Construction Graphics in outdoor Augmented Reality*, Center for Contruction Engineering and Management 2005

[12] **T. Turunen et al.**, *Mobile AR Requirements for Location Based Social Networks*, International Journal of Virtual Reality 2010

[13] **Z.Y. Zhou et al.**, *Robust Pose Estimation for Outdoor Mixed Reality with Sensor Fusion*, Universal Access in Human-Computer Interaction, Springer 2009

[14] **W. Piekarski, B. Thomas**, *ARQuake: The Ourdoor Augmented Reality Gaming*

System, Communications of the ACM, 2002, Vol. 45 No. 1

[15] **v.A.**, *Core Data Programming Guide*
`http://developer.apple.com/library/ios/#DOCUMENTATION/DataManagement/Conceptual/iPhoneCoreData01/Introduction/Introduction.html`

[16] **v.A.**, *Concurrency with Core Data*
`http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/CoreData/Articles/cdConcurrency.html`

[17] **v.A.**, *Allgemeine Informationen über CloudMade*
`http://cloudmade.com/about`

[18] **Dieter Fensel et al.**, *Semantic Web Services*, Springer-Verlag Berlin Heidelberg 2011, S. 37-65

Aufgrund der Aktualität der in dieser Masterarbeit eingesetzten Techniken sind zum Beleg viele Internetseiten und verlinkte pdf-Dokumente aufgeführt. Da sich die Erreichbarkeit und der Inhalt dieser verlinkten Seiten schnell ändern können, kann nicht sichergestellt werden, dass die unter den Links zu findenden Informationen noch aktuell sind. Zum Zeitpunkt der Fertigstellung dieser Masterarbeit am 01.11.2012 waren alle im Literatur- und Webverzeichnis, sowie in den Fußnoten aufgeführte Quellen zu finden.

Abbildungsverzeichnis

Quellcodeverzeichnis

Danksagung

An dieser Stelle möcht ich allen danken, die mich bei der Erstellung dieser Masterarbeit unterstützt haben.

- Herrn Prof. Dr. Oliver Vornberger danke ich für die Einwilligung des interessanten Themas und die Übernahme des Erstgutachters.
- M.Sc. Nicolas Neubauer danke ich für die Betreuung, sowie Beratung und Hilfestellung bei allen Fragen und Anliegen.
- Dipl. Phys. Friedhelm Hofmeyer danke ich vor allem für die Unterstützung bei der Konfiguration und Einrichtung der Hard- und Software.
- Für die Übernahme des Zweitgutachtens möchte ich mich herzlich bei Frau Jun. Prof. Dipl. Ing. Elke Pulvermüller bedanken.
- Schließlich gilt ein großer Dank meiner Freundin Agnieszka Musiol, die mir während dieser Zeit stets zur Seite stand.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Osnabrück, 01. November 2012 _____

Unterschrift Phillipp Bertram