UNIVERSITÄT OSNABRÜCK

Institut für Informatik

Bachelorarbeit

Indoor Positionierung mittels Bluetooth Low Energy

Kevin Seidel

24.09.2013

Erstgutachter:	Prof. Dr. Oliver Vornberger
Zweitgutachterin:	Prof. Dr. Elke Pulvermüller

Danksagungen

Hiermit möchte ich allen Personen danken, die mich bei der Erstellung der Arbeit unterstützt haben:

- Herrn Prof. Dr. Oliver Vornberger für die Tätigkeit als Erstgutachter und für die Bereitstellung der interessanten Thematik.
- Frau Prof. Dr. Elke Pulvermüller, die sich als Zweitgutachterin zur Verfügung gestellt hat.

Zusammenfassung

Bluetooth

Abstract

Bluetooth

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Die GPS-Navigation ist seit Jahren aus keinem Auto mehr wegzudenken. Wo früher Karten genutzt wurden und nach Straßennamen geschaut wurde, wird heute die Zieladresse in das Navigationssystem eingegeben und das System bestimmt selbstständig die aktuelle Position, die Zielposition und errechnet die bestmögliche Route. Ein Problem der GPS-Navigation ist jedoch, dass diese nur unter freiem Himmel akzeptabel funktioniert. In der Realität verbringen wird jedoch den Großteil unserer Zeit in Gebäuden, wo uns dieser Ansatz wenig weiterhilft.

Daher wäre es sinnvoll, eine Alternative zu GPS zu schaffen, welche diese Funktionien in Innenräume realisiert. Da man jedoch für Innenräume kein eigenes Navigationssystem kaufen möchte, liegt die Idee nah, diese Konzept auf einem Gerät zu realisieren, was sowieso schon viele Leute besitzen und auch schon für die GPS-Navigation nutzen. Das Smartphone. Wie in Abbildung 1 zu sehen, hat die Verbreitung der Smarthphones in den letzten Jahren sehr stark zugenommen, sodass man annehmen kann, das ein Großteil der potentiellen Nutzer der Indoor Positionierung auch ein Smartphone besitzt.

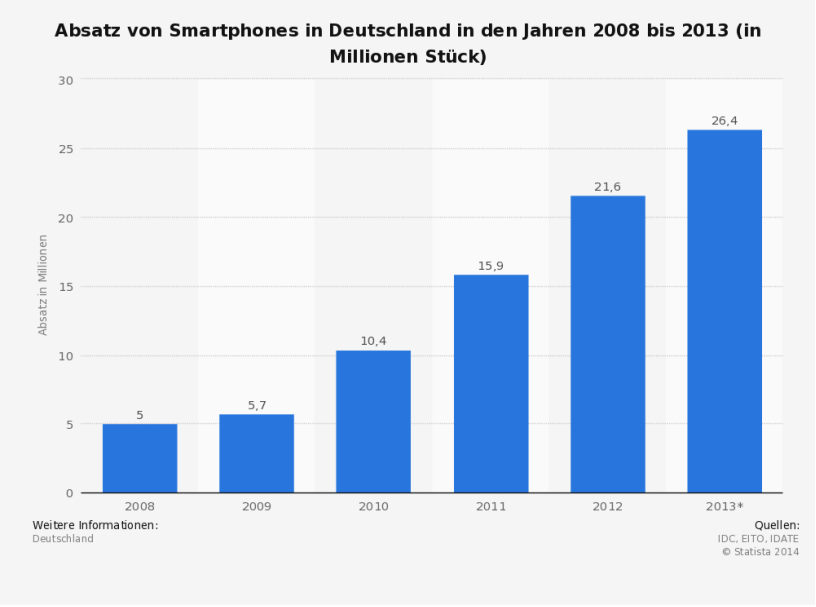

Abbildung 1.1: Smartphoneabsatz in Deutschland

Für die Indoor Positionierung würden verschiedene Technologien in Frage kommen, wie zum Beispiel Wireless LAN, RFID oder Bluetooth. Diese Technologien bieten sich an, da sie schon von Haus aus in vielen Smartphones integriert sind und so kein Bedarf an neuen Geräten oder Erweiterungen besteht.

Schlussendlich viel die Entscheidung der zu verwendenden Technologie auf Bluetooth, da dieses die höchste Verbreitung bietet und auch weitere Vorteile mit sich bringt. Zum einen ermöglicht Bluetooth eine schnelle und einfache Einrichtung und zum anderen benötigen die Bluetooth-Sendestationen nicht zwingend einen Stromanschluss, sondern erlauben auch einen Batteriebetrieb über mehrere Monate bis Jahre.

Die Positionierung in Innenräumen mittels Bluetooth ist ein relativ neuer Ansatz, welcher jedoch seit der Präsentation von Bluetooth Low Energy und der Vorstellung der iBeacons-Technologie von Apple immer mehr an Aufmerksamkeit gewonnen hat.

1.2 Ziele der Bachelorarbeit

Das Hauptziel dieser Arbeit ist es zu untersuchen, in wie weit sich Bluetooth Low Energy beziehungsweise die darauf basierende iBeacons-Technologie für eine akzeptable Indoor Positionierung eignet, um Endgeräte zum Beispiel in Verkaufsräumen zu orten und zu identifizieren.

Dabei soll untersucht werden, welches Verfahren sich dafür am Besten eignet und ob es Unterschiede zwischen verschiedenen Sende- und auch Empfangsgeräten gibt. Für die initialen Tests werden ausschließlich Apple-Geräte genutzt, da hier eine übersichtlichere Auswahl auf dem Markt ist, sodass man sich nicht mit unzähligen verschiedenen Messungen ausseinander setzen muss.

Im Laufe der Bachelorarbeit soll deshalb eine iOS-Applikation entwickelt werden, welche eine Positionierung in einem Innenraum implementiert. Dabei wird die von Apple bereitgestellte CoreLocation-API genutzt, welche die Verarbeitung der iBeacon-Daten übernimmt. Die genutzten iBeacon-Sender kommen von Drittherstellern und sind derzeit noch in einem Vorserienstadium.

Zum Abschluss soll eine grundlegende Positionierung, eine Anzeige der aktuellen Position auf eine Karte und das Auslösen bestimmter Aktionen an festgelegten Orten implementiert sein.

2 Technologien

2.1 Bluetooth 4.0

Die Bluetooth-Version 4.0, oder auch Bluetooth Smart genannt, wurde 2009 final spezifiziert und wird seit Ende 2010 in Endgeräten eingesetzt. Dieser Standard beinhaltet neben dem klassischen Bluetooth, eine neue Version, mit dem Namen Bluetooth Low Energy, welche, wie der Name schon andeutet, einen sehr viel geringeren Stromverbrauch vorweißt. Dabei ist der Stromverbrauch zwischen zwei und 100 mal geringer als beim klassischen Bluetooth.

2.2 Bluetooth Low Energy

Bluetooth Low Energy wurde Anfangs von Nokia unter dem Namen "Wibree" entwickelt. Die Zielsetzung dabei war es eine Technologie zu entwickeln, mit der sich Computer und Mobilgeräte schnell und einfach mit Peripherie-Geräten verbinden lassen. Das Hauptaugenmerk galt dabei dem geringen Stromverbrauch, kompakter Bauweise und den Kosten für die benötigte Hardware. Im Jahr 2007 wurden diese Spezifikationen dann in den, sich in der Entwicklung befindenden, Bluetooth-Standard 4.0 aufgenommen und daraufhin in Bluetooth Low Energy, oder kurz BLE umbenannt.

Bluetooth Low Energy arbeitet wie das klassische Bluetooth im 2,4 GHz Band, bringt aber in der Funktionsweise einige Unterschiede mit sich.

So wurde, im Vergleich zum klassischem Bluetooth, die Datenrate von bis zu 3 Mbit/s auf maximal 1 Mbit/s reduziert. Dies führt dazu, dass BLE zum Beispiel nicht für Headsets genutzt werden kann, da die zur Verffügung stehende Übertragungsrate nicht für die Audioübertragung ausreicht.

Die Vorteile die BLE mit sich bringt, liegen vor allem in der niedrigen Latenz, welche von 100ms auf bis zu unter 3ms reduziert wurde, und, wie bereits erwähnt, der Energieverbrauch drastisch gesenkt wurde.

Bluetooth Low Energy bietet darüber hinaus eine Vielzahl sogennanter GATT-Profile (Generic Attribute Profile). Die bereitgestellten GATT-Profile sind Richtlinien für die Bluetooth-Funktionialität, sprich, welche Daten übertragen werden und in welcher Form. Dies erlaubt eine einfache und schnell Interoperabilität zwischen verschiedenen Geräten. Ein Beispiel für ein GATT-Profil wäre zum Beispiel das "Heart Rate Profile", welches besipielsweise die Verbindung und Kommunikation eines Pulsmessgurtes mit einem Endgerät beschreibt. So wird sichergestellt, dass dieser Gurt mit jedem Endgerät auf die selbe Weise funktioniert.

2.2.1 iBeacons

Die iBeacons-Technologie wurde am 10.Juni 2014 von Apple auf der Worldwide Developers Conference vorgestellt. Diese basiert auf Bluetooth Low Energy und arbeitet mit einem von Apple entwickelten GATT-Profil.

Beacon bedeutet übersetzt "Leuchtfeuer" und die Funktionsweise der Beacons ist dem sehr ähnlich. Einmal in Betrieb genommen, sendet das Beacon kontinuierlich ein Signal, in welchem sich Daten zur Identifizierung des Beacons befinden.

Neben den Identifikationsdaten kann das Emfangsgerät noch weitere Größen bestimmen. Es ist so zum Beispiel möglich die ungefähre Entfernung einzuschätzen. In der iBeacons-API sind dafür vier verschiedene Zustände definiert: *Far*, *Near*, *Immediate* und *Unknown*. Diese Werte erlauben eine grobe Entfernungseinschätzung und für eine genauere Bestimmung lässt sich noch eine weitere Kenngröße bestimmen, der *Accuracy*-Wert. Dabei handelt es sich um eine ungefähre Entfernungsangabe in Metern, welche jedoch ausdrücklich nur zur Differenzierung zwischen zwei Beacons genutzt werden soll und keinesfalls eine genaue Entfernung angibt.

Daten	Format	Beschreibung	Beispiel
UUID	16-stellige Hexadezimalzahl	Identifizierung	3F4
Major	Integerzahl	Identifizierung eine Region	12
Minor	Integerzahl	Identifizierung eines einzelnen Beacons	132
Proximity	Drei Entfernungsstufen	Ungefähre Entfernung	Far, Near, Immediate und Unknown
Accuracy	Wert in Meter	Bestimmung der ungefähren Entfernung	1.243 m
RSSI	Signalstärke in dBm	Signalstärke des emfangenen Signals	-42 dBm

Die von dem Beacon gesendeten Daten lassen sich mit jedem BLE-kompatiblem Gerät empfangen.

Die großen Vorteile der iBeacons sind zum einen ihr kleiner Formfaktor, welcher es erlaubt die Beacons an fast jedem beliebigem Ort anzubringen, als auch ihr geringer Stromverbrauch, der es möglich macht, die Beacons bis zu mehreren Jahren mit einer Knopfzellenbatterie zu betreiben. Der Aufbau eines solchen Beacons lässt sich in Abbildung 2.3 sehr gut erkennen. Den Großteil des Beacons nimmt dabei die Batterie ein.

Unter genauerer Betrachtung des Chipsatzes in Abbildung 2.4, erkennt man, dass er im Grunde aus zwei Teilen besteht. Dem Bluetooth-Chipsatz, welcher an sich ist nur wenige Zentimeter groß und der Antenne, welche im vorderen Bereich der Platine eingearbeitet ist und die über welche letztendlich die Daten gesendet werden.

Abbildung 2.1: Außenhülle

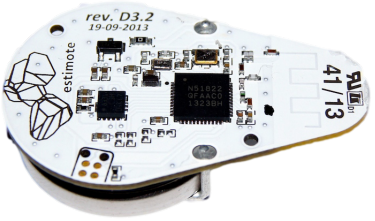

Abbildung 2.2: Chipsatz mit Bluetooth-Modul

Abbildung 2.3: Ein iBeacon der Firma "estimote"

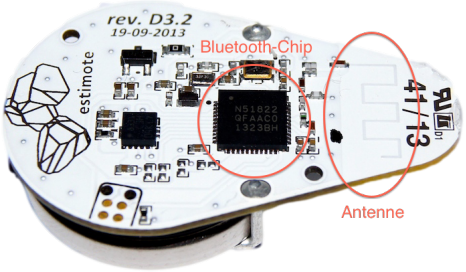

Abbildung 2.4: Aufbau des estimote-Beacons

2.3 iOS, OS X und Xcode

Für die Entwicklung der Applikation zur Indoor Positionierung war eine der Vorgaben, dass diese für iOS programmiert werden soll. Daher waren drei Dinge zwingend notwendig: ein Mac, Xcode und ein iOS-Gerät.

Für die Entwicklung setzte ich deshalb auf ein Macbook Pro mit installiertem Xcode und als iOS-Gerät setzte ich ein iPhone 5 ein. Als minimale iOS-Version musste iOS 7 verwendet werden, da die iBeacon-Features des CoreLocation-Frameworks (mehr dazu im Kapitel 2.4) erst ab dieser Version zur Verfügung stehen.

2.4 CoreLocation-Framework

Das Core Location-Framework erlaubt es aktuelle Positions- und Richtungsinformationen eines Gerätes zu bestimmen. Die Positionsbestimmung lässt sich dabei über verschiedene Werte und Sensoren bestimmen und auch der Grad der Genauigkeit ist variabel. Auch die Aktualiserungsrate der Position lässt sich festlegen, wobei eine höhere Aktualisierungsrate auch gleichbedeutend mit einem höherem Akkuverbrauch ist.

Bei der Genauerigkeit gibt es dabei verschiedene Konstanten, die die gewollte Genauigkeit bestimmen.

Diese Genauigkeiten beziehen sich hauptsächlich auf die Positionierung mittels GPS und sind daher für die Indoor Positionierung nur bedingt geeignet.

Konstante	Erwartete Genauigkeit
kCLLocationAccuracyThreeKilometers	Genauigkeit auf 3 Kilometer
kCLLocationAccuracyKilometer	Genauigkeit auf 1 Kilometer
kCLLocationAccuracyHundredMeters	Genauigkeit auf 100 Meter
kCLLocationAccuracyNearestTenMeters	Genauigkeit auf 10 Meter
kCLLocationAccuracyBest	Höchstmögliche Genauigkeit
kCLLocationAccuracyBestForNavigation	Höchstmögliche Genauigkeit und weitere Sensordaten für die Navigation

Table 2.1: Mögliche Optionen der Positionsgenauigkeit

Eine weitere Funktion des Core Location-Frameworks ist die Bestimmung der Himmelsrichtungen. Durch den eingebauten Kompass in den neueren iOS-Geräten ist es möglich, die aktuelle Ausrichtung des Gerätes zu bestimmen. Dies ist im Bezug auf die Indoor Navigation hilfreich, da diese Informationen in die Positionsbestimmung einbezogen werden können. Des Weiteren erlaubt diese Funktion eine dynamische Ausrichtung der Karte, abhängig davon in welche Richtung man momentan schaut.

Die für uns zentrale Funktion dieses Frameworks ist die Erkennung von iBeacons und die Funktionen zur Verarbeitung der gesendeten Daten. Damit können Beacons anhand ihres UUID erkannt werden und einer Region zugeordnet werden. Die genaue Funktionsweise wird in Kapitel 2.4.1.

Die Funktionen zum Positionsupdate und zur Erkennung der Beacons werden dabei im *LocationManager* verwaltet. In der *LocationManagerDelegate* lassen sich dabei die Aktionen bestimmen, welche bei verschiedenen Events ausgeführt werden.

In Listing 1 wird die Initialisierung eines LocationManager gezeigt, welcher eine Genauigkeit von einem Kilometer haben soll und bei Positionsänderungen von mehr als 500 Metern aktualisiert wird.

2.4.1 iBeacons-API

Seit der iOS Version 7 wurde das Core Location Framework um die Beacon-Funktionen erweitert. Dazu wurden zwei neue Klassen geschaffen. Einmal die *CLBeacon*-Klasse, welche ein iBeacon repräsentiert und alle zur verfügungsteheneden Informationen enthält und zum anderen die *CLBeaconRegion*-Klasse, welche eine Region mit mehreren Beacons, abhängig von ihrem UUID, beschreibt.

Die *CLBeacon*-Klasse besteht dabei lediglich aus Propertys mit den gegenbenden Beacon-Informatinen, wie *UUID*, *major*, *minor*, *accuracy*, *proximity* und *rssi*.

Die *CLBeaconRegion*-Klasse ist etwas umfangreicher und bestimmt letztendlich, nach welchen Beacons gesucht werden soll. Dabei ist es möglich die Region in verschiedene Genaugikeitsstufen einzuteilen.

initWithProximityUUID:identifier:

```
1  - (void)startStandardUpdates
2  {
3      // Create the location manager if this object does not
4      // already have one.
5      if (nil == locationManager)
6          locationManager = [[CLLocationManager alloc] init];
7
8      locationManager.delegate = self;
9      locationManager.desiredAccuracy = kCLLocationAccuracyKilometer;
10
11     // Set a movement threshold for new events.
12     locationManager.distanceFilter = 500; // meters
13
14     [locationManager startUpdatingLocation];
15 }
```

Listing 1: Beispielinitialisierung für einen LocationManager.

Die Region ist nur abhängig von dem UUID und dem Identifier der Beacons, das heißt es werden alle Beacons mit dem gegebenen UUID gesucht.

initWithProximityUUID:major:identifier:

Die Region ist abhängig von dem UUID, dem Identifier und dem Major-Wert der Beacons. Es werden nur Beacons eines bestimmten Major-Wertes gesucht.

initWithProximityUUID:major:minor:identifier:

Die Region ist abhängig von dem UUID, dem Identifier, dem Major-Wert und dem Minor-Wert der Beacons. Es werden nur Beacons mit passendem Major und Minor-Wert gesucht. In diesem Fall ist bei mehreren erkannten Beacons keine Unterscheidung mehr möglich.

Mittels des Location Manager lässt sich dann gezielt nach bestimmten Regionen suchen.

2.5 MapBox

MapBox ist ein Online-Karten Anbieter, welcher es erlaubt eigene Karten zu erstellen und über ihren Service bereitzustellen. Die Karten werden dabei vom OpenStreetMap Projekt bereitgestellt und MapBox erlaubt es diese Karten grafisch zu überarbeiten, um zum Beispiel das Farbschema zu ändern, eigene Markierungen zu setzen oder auch eigene Layer über die Karte zu legen. Ausserdem stellt MapBox ein SDK für iOS bereit, welche es erlaubt diese individuell angepassten Karten in iOS anzuzeigen und gleichzeitig die Funktionen des native MapKit-Framework mit sich bringt. Ausserdem besitzt MapBox

eine größere Flexibilität im Bezug auf die individuelle Anpassung und den Offline-Betrieb, das die Karten direkt auf dem Gerät gespeichert werden können.

Bisher ist die Unterstützung von Indoor-Karten jedoch noch nicht gegeben, sodass man hierbei nicht auf schon vorhandenes Kartenmaterial zurückgreifen kann.

Google hat mit Google Maps Indoor bereits einen Dienst gestartet, welcher Gebäudepläne in Google Maps integriert, bisher handelt es sind jedoch dabei hauptsächlich um öffentliche Gebäude in US-amerikanischen Städten. Das hinzufügen von neuen Gebäudeplänen ist nur bei öffentlichen Gebäuden möglich und nicht für den privaten Gebrauch vorgesehen, daher konnte ich nicht darauf zurückgreifen.

Die Indoor-Karten mussten daher selbst erstellt und in ein, von Mapbox verständliches Format umgewandelt werden. Die Ausgangsdatei ist dabei eine Bilddatei mit der Karte des Innenraumes, welche selbst angefertigt wurde. Dieses Bild der Karte muss nun in ein passendes Geo-Format überführt werden. Dazu wurde ein von "Tom MacWright" (MacWright (2014)) bereitgestellte Python-Script verwendet, welches JPEG Dateien in GeoTIFF Dateien umwandelt. Die GeoTiff Datei speichert neben den eigentlichen Bilddaten noch Koordinaten für die Georeferenzierung. Ritter and Ruth (2014)

Mit diesem GeoTIFF ist es nun möglich eine eigene Karte zu erstellen, welche letztendlich auf dem iOS-Gerät ausgegeben wird. Dafür stellt MapBox das Programm *TileMill* zur Verfügung. Dieses erlaubt es eigene Karten zu erstellen und zu bearbeitet. Die erstellte Karte kann anschließend in verschiedenen Formaten exportiert werden. TileMill bietet einen Import von GeoTIFF-Dateien an, sodass unsere Karte direkt eingefügt werden kann.

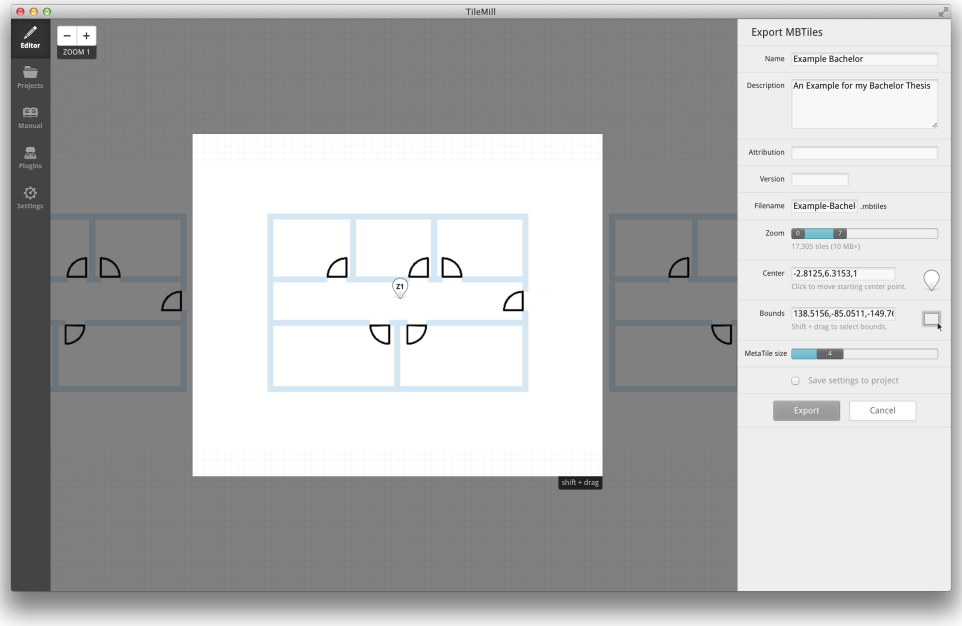

Abbildung 2.5: Karte in TileMill

TileMill erlaubt es nun die eingefügte Karte weiter zu bearbeiten, was in unserem Fall jedoch nicht nötig ist. Der nächste Schritt ist es die Karte in ein für iOS beziehungsweise

das Mapbox SDK, verständliches Format zu überführen. Dazu wird die Karte als *mbtiles* exportiert. Dies ist ein von Mapbox entwickeltes Dateiformat, welches die Karte in Kacheln speichert um das Laden der einzelnen Kartenabschnitte bei größeren Karten zu beschleunigen, sodass nicht die komplette Karte geladen werden muss, sondern nur die benötigten Kacheln.

Die erzeugte *.mbtiles*-Datei lässt sich nun in die iOS Appikation einbinden und über das SDK auf dem iOS-Gerät ausgeben. In Abbildung 2.6 sieht man die Ausgabe einer Karte auf dem iPhone 5.

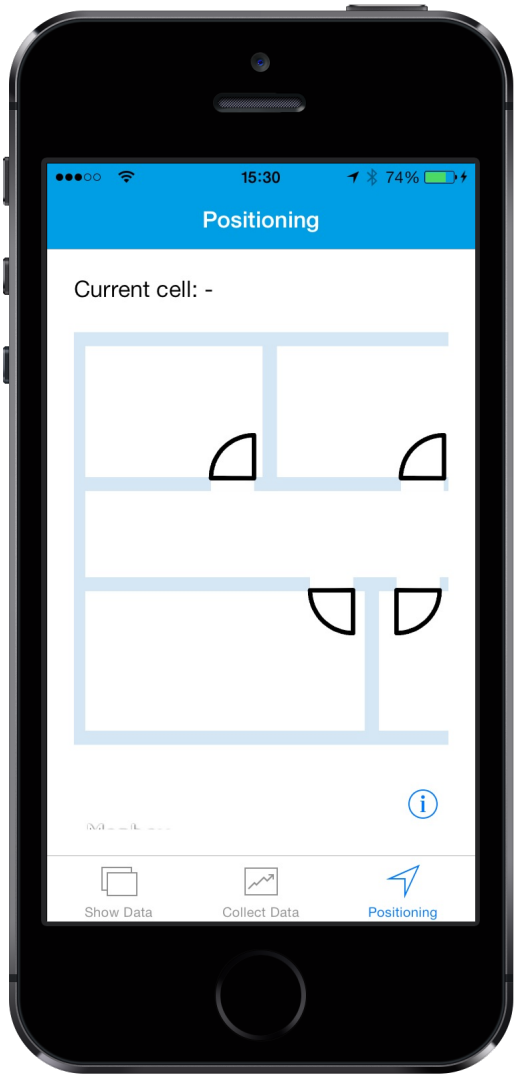

Abbildung 2.6: Kartenausgabe mittels Mapbox SDK auf dem iPhone

Diese Karte wird offline auf dem Gerät gespeichert, es ist also keine Internetverbindung nötig und diese anzuzeigen.

Für die Anzeige auf dem Gerät ist es zunächst nötig einen *RMMapView* anzulegen, welcher für die Ausgabe der Karte verantwortlich ist und gleichzeitig die gewohnten MapView-Features wie zum Beispiel *Pinch-to-Zoom* oder die automatische Ausrichtung auf den Mittelpunkt mit sich bringt. Da wir unser eigenes Kartenmaterial verwendet

ist es zudem nötig die Quelle für Kartendaten des MapViews zu ändern, da ansonsten die Daten des OpenStreetMap-Projekts genutzt werden. Dazu wird eine eigene *RMTileSource* angelegt, welche die zuvor generierten *mbtiles* läd und dem MapView zur Verfügung stellt. In Listing 2 wird diese Inintialisierung gezeigt.

```
1  RMMBTilesSource *customTileSource =
2          [[RMMBTilesSource alloc] initWithTileSetURL:
3                  [NSURL fileURLWithPath:[
4                          [NSBundle mainBundle]
5                          pathForResource:@"Example-Bachelor"
6                          ofType:@"mbtiles"]]];
7
8  RMMapView *mapView =
9          [[RMMapView alloc] initWithFrame:self.outerMapView.bounds
10                 andTilesource:customTileSource];
```

Listing 2: Initialisierung des MapView mit eigenem Kartenmaterial

2.5.1 Weitere API's

2.6 CoreData-Framework

Core Data ist ein Framework für die Organisation und Speicherung von Daten in einem Entity-Realationship-Modell. Core Data vereinfacht die Speicherung und den Zugriff auf die Daten, da es für jede Entity ein eigenes Objekt erstellt. Die eigentliche Datenspeicherung sieht dabei drei Verschiedene speichermöglichkeiten vor, entweder als Binärdatei, als XML-Datei oder in einer SQLite-Datenbank. Nachdem eine Core Data-Datenbank erstellt wurde, benötigt man zunächst ein *NSManagedObjectContext*, welcher Lese- und Schreibzugriffe auf die Datenbank steuert und verwaltet. Objekte aus der Datenbank werden durch *NSManagedObject* repräsentiert. Nach dem anlegen der Datenbank ist es jedoch auch möglich sich direkt eigene Klasse für die einzelnen Datenbank-Objekte erzeugen zu lassen, welche alle Attribute und Funktionen der einzelnen Objekte und Verbindungen beinhalten und somit den Zugriff enorm erleichtern.

Um auf die Daten zuzugreifen und diese zu verändern ist es zunächst nötig sie aus der Datenbank zu extrahieren. Dazu verwendet man einen *NSFetchRequest*, welcher Objekte nach bestimmten Kriterien aus der Datenbank holt. Dabei ist es möglich den *NSFetchRequest* zu spezialisieren und so nur Objekte mit bestimmten Eigenschaften anzuzeigen. Dafür verwendet man ein *NSPredicate*, welches umfangreiche Vergleiche von Attributen und logische Operationen erlaubt.

```
1  NSManagedObjectContext *moc = [self managedObjectContext];
2  NSEntityDescription *entityDescription = [NSEntityDescription
3      entityForName:@"Employee" inManagedObjectContext:moc];
4  NSFetchRequest *request = [[NSFetchRequest alloc] init];
5  [request setEntity:entityDescription];
6
7  NSNumber *minimumSalary = 3000;
8  NSPredicate *predicate = [NSPredicate predicateWithFormat:
9      @"(lastName LIKE[c] 'mann') AND (salary > %@)", minimumSalary];
10 [request setPredicate:predicate];
11
12 NSArray *array = [moc executeFetchRequest:request error:&error];
```

Listing 3: Fetch Request für alle Objekte die im Namen "mannënthalten und mehr als 3000 Euro im Monat verdienen

3 Werkzeuge

3.1 Xcode

Xcode ist eine integrierte Entwicklungumgebung von Apple, welche es ermöglicht iOS und OS X Applikationen zu programmieren, zu testen und zu debuggen. Standardmäßig werden dabei die Programmiersprachen *Objective C*, *C* und *C++* unterstützt.

Xcode stellt viele Features für die Programmierung bereit, wie zum Beispiel *code completion*, vorgefertigte *Templates*, einen umfangreichen *Debugger* und eine *iOS-Simulator* für das Testen der Applikationen.

Bei Erstellung einer neuen Applikation kann man unter mehreren Templates wählen, welche jeweils verschiedene Funktionen mit sich bringen. In Abbildung 3.1 lassen sich die verschiedenen Auswahlmöglichkeiten erkennen.

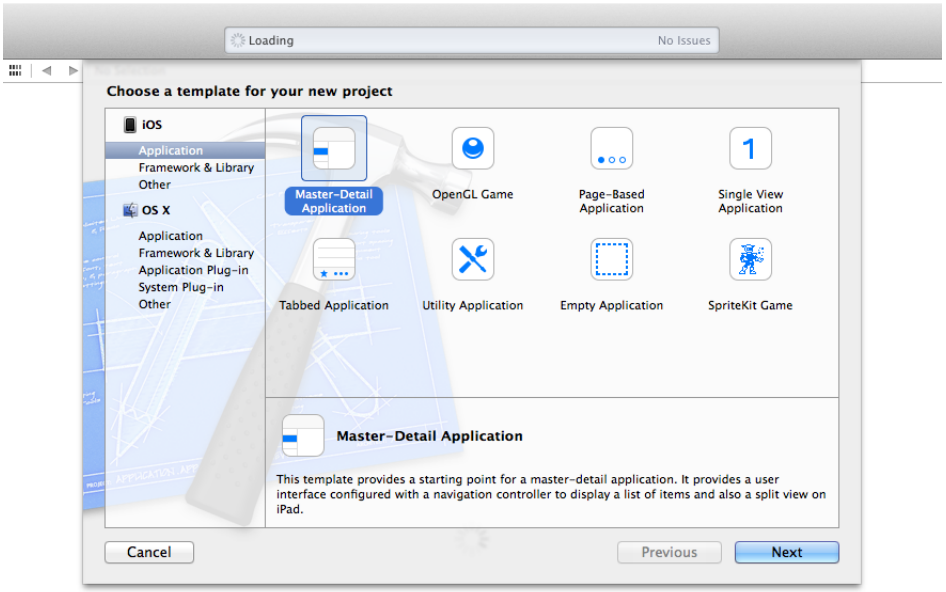

Abbildung 3.1: Auswahlbildschirm der verschiedenen Templates

Nach dem man das passende Template gewählt hat, werden die benötigten Dateien angelegt. Dazu gehören beispielsweise die *AppDelegate* und das *Storyboard*.

Die AppDelegate-Klasse steuert applikationsweite Ereignisse, wie etwa das Aufrufen und Schließen der Applikation. Außerdem wird durch die AppDelegate der aktuelle Zustand der Applikation gespeichert und wiederhergestellt.

Das Storyboard ist eine grafische Oberfläche für die Erstellung des User Interfaces. Es ermöglicht verschiedene Elemente wie zum Beispiel Views, Textfelder, Buttons oder Tabellen einzufügen und diese zu verbinden. Wie in Abbildung 3.2 zu erkennen, besteht das Storyboard aus meheren View Controllern, die jeweils eine gezeigte Szene auf dem Gerät repräsentieren. Die einzelnen View Controller sind mit sogennanten *Segue's* verbunden, welche sich durch bestimmte Aktionen, wie zum Beispiel den Druck auf einen Button, auslösen lassen.

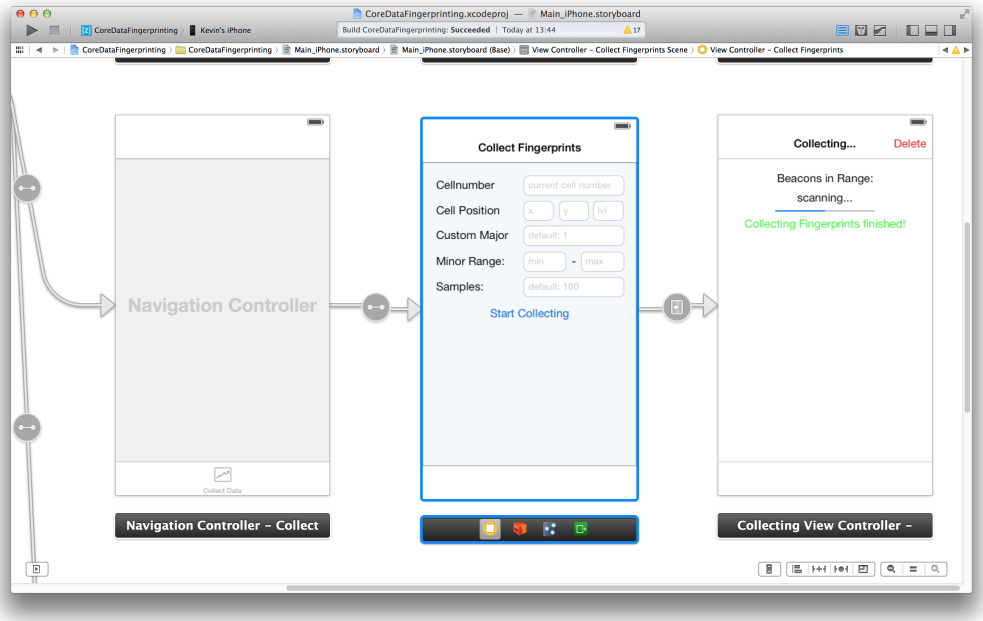

Abbildung 3.2: Beispiel eines Storyboards für iPhones

Das Storyboard bietet außerdem noch die Funktion des *Auto Layout*. Dabei werden sogenannte *Constraints* genutzt, welche die Positionsbeziehungen zwischen den einzelnen Elementen festlegen. Diese Contstraints erzeugen so ein dynamisches Interface, welches sowohl im Hochformat, als auch im Querformat ein sauberes und geordnetes Format hat. In Abbildung 3.3 lassen sich die Abstands und Ausrichtungsbeziehungen zwischen den einzelnen Objekten gut erkennen.

ufgabe
r Cons-
aints be-
hreiben

3.2 Objective-C

Objective-C ist eine Programmiersprache, welche in den 80er Jahren entwickelt worden ist. Sie ist eine strikte Obermenge von C und erweitert diese um objektorientierte Konzepte. Objective-C ist die Hauptsprache für die Programmierung von Cocoa-Applikationen, wie sie unter iOS und OS X genutzt werden.

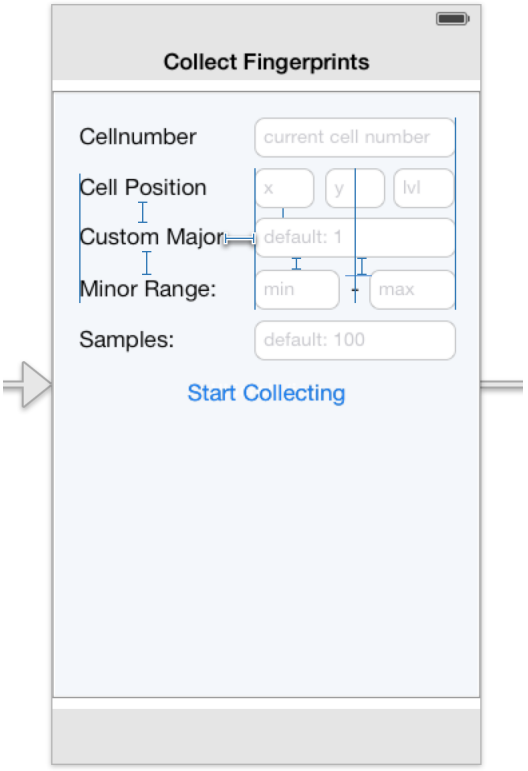

Abbildung 3.3: View Controller mit Constraints

3.3 Versionsverwaltung mit Git

3.4 iOS Developer Program

3.5 iPhone

4 Daten und Messungen

4.1 Mobile iBeacons

4.2 Stationäre iBeacons

4.3 Außenmessungen

4.4 Innenraummessungen

5 Umsetzung und Implementation

5.1 Ansatz zur Positionsbestimmung

5.2 Trilateration

5.3 Fingerprinting

6 Fingerprinting

6.1 Positionsbestimmung

6.1.1 Nearest-Neighbor-Verfahren

6.1.2 Prohabilistisches-Verfahren

7 Fazit und Ausblick

Literaturverzeichnis

T. MacWright. Images as maps. 2014. URL http://www.macwright.org/2012/08/13/images-as-maps.html.

N. Ritter and M. Ruth. Geotiff format specification. 2014. URL http://www.remotesensing.org/geotiff/spec/geotiffhome.html.

Erklärung

Ich versichere, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, den 24.09.2013

(Kevin Seidel)