



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

1 Einleitung	1
1.1 Motivation	1
1.2 Ziele der Bachelorarbeit	2
2 Technologien	3
2.1 Bluetooth 4.0	3
2.2 Bluetooth Low Energy	3
2.2.1 iBeacons	4
2.3 iOS, OS X und Xcode	5
2.4 CoreLocation-Framework	5
2.4.1 iBeacons-API	6
2.5 MapBox	8
2.5.1 Weitere API's	10
2.6 CoreData-Framework	10
3 Werkzeuge	13
3.1 Xcode	13
3.2 Objective-C	14
3.3 Versionsverwaltung mit Git	15
3.4 iOS Developer Program	15
3.5 iPhone	16
4 Daten und Messungen	17
4.1 Mobile iBeacons	17
4.1.1 estimote Beacon	17
4.1.2 kontakt.io Beacon	18
4.2 Stationäre iBeacons	19
4.2.1 Raspberry Pi als iBeacon	19
4.3 Außenmessungen	19
4.4 Innenraummessungen	19
4.5 Mögliche Störfaktoren	20
5 Umsetzung und Implementation	21
5.1 Ansatz zur Positionsbestimmung	21
5.2 Trilateration	21
5.3 Fingerprinting	21
6 Fingerprinting	23
6.1 Positionsbestimmung	23
6.1.1 Nearest-Neighbor-Verfahren	23
6.1.2 Probabilistisches-Verfahren	23

7 Fazit und Ausblick	25
Bibliography	26

Abbildungsverzeichnis

1.1	Smartphoneabsatz in Deutschland	1
2.1	Außenhülle	5
2.2	Chipsatz mit Bluetooth-Modul	5
2.3	Ein iBeacon der Firma "estimote"	5
2.4	Aufbau des estimote-Beacons	5
2.5	Karte in TileMill	9
2.6	Kartenausgabe mittels Mapbox SDK auf dem iPhone	10
3.1	Auswahlbildschirm der verschiedenen Templates	13
3.2	Beispiel eines Storyboards für iPhones	14
3.3	View Controller mit Constraints	15
3.4	Xcode Versionsverwaltung mit Diff-Anzeige bei einem Commit	16
4.1	Das Developer-Kit von estimote	18
4.2	Kontakt.io Beacon	18
4.3	Messung des iPhone 5	20
4.4	Messung des iPhone 4s	20
4.5	Durchschnittliche Signalstärke eines kontakt.io Beacons	20
4.6	Minimale, maximale und durchschnittliche Signalstärke des Beacons gemessen vom iPhone 5	20

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 Funktionen 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 Smartphones in den letzten Jahren sehr stark zugenommen, sodass man annehmen kann, dass ein Großteil der potentiellen Nutzer der Indoor Positionierung auch ein Smartphone besitzt.

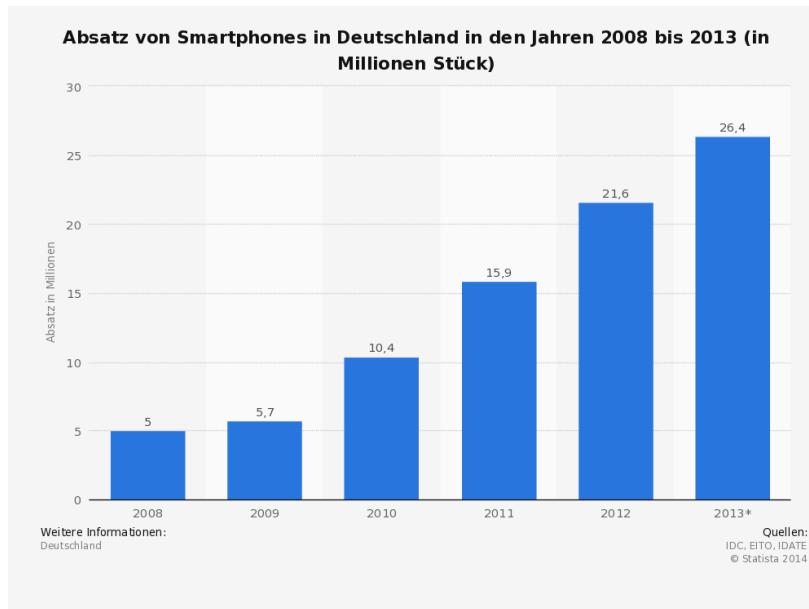


Abbildung 1.1: Smartphoneabsatz in Deutschland

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 ausseinenander 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 vorweist. 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 klassischen 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 Verfügung 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.

Des Weiteren wird eine 24-Bit-Fehlerkorrektur eingesetzt, was die Verbindung unempfindlicher für Störungen machen soll.

Für die Verschlüsselung des Signals wird AES-128 eingesetzt, was eine sichere Kommunikation gewährleisten soll.

Bluetooth Low Energy bietet darüber hinaus eine Vielzahl sogennanter GATT-Profile (Generic Attribute Profile). Die bereitgestellten GATT-Profile sind Richtlinien für die Bluetooth-Funktionalität, 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 beispielsweise 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 Empfangsgerät 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.

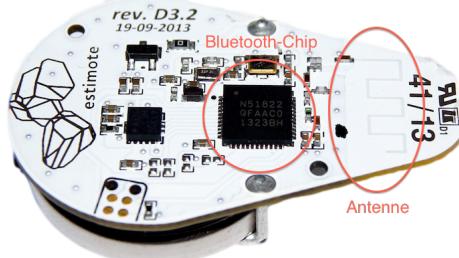
Daten	Format	Beschreibung	Beispiel
UUID	16-stellige Hexadezimalzahl	Identifizierung	3F4
Major	Integerzahl	Identifizierung einer Region	12
Minor	Integerzahl	Identifizierung eines einzelnen Beacons	132
Proximity	Drei Entfernungsstufen	Ungefähre Entfernung	Far, Near, Immediate und Unknown
Accuracy	Wert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

**Abbildung 2.1:** Außenhülle**Abbildung 2.2:** Chipsatz mit Bluetooth-Modul**Abbildung 2.3:** Ein iBeacon der Firma "estimote"

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 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 2.4:** 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 5.0.2 und als iOS-Gerät setzte ich ein iPhone 5 mit iOS 7.0.4 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 Aktualisierungsrate der Position lässt sich festlegen, wobei eine höhere Aktualisierungsrate auch gleichbedeutend mit einem höherem Akkuverbrauch ist.

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

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

Table 2.1: Mögliche Optionen der Positionsgenauigkeit

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

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 Verfügung stehenden Informationen enthält und zum anderen die *CLBeaconRegion*-Klasse, welche eine Region mit mehreren Beacons, abhängig von ihrem UUID, beschreibt.

```

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 *CLBeacon*-Klasse besteht dabei lediglich aus Property's 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 Genaugkeitsstufen einzuteilen.

initWithProximityUUID:identifier:

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 sich 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 bearbeiten. 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.

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.

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

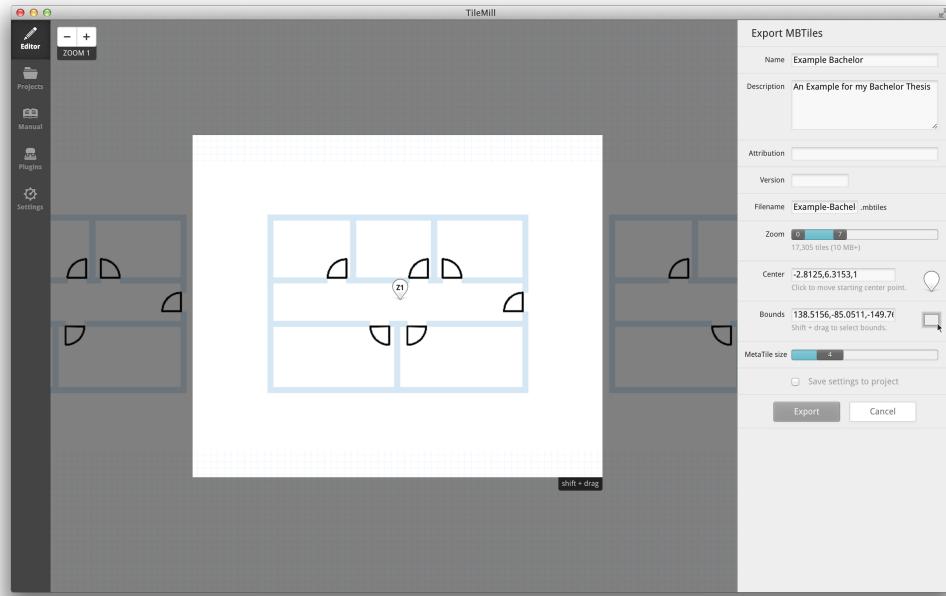


Abbildung 2.5: Karte in TileMill

Für die Anzeige auf dem Gerät ist es zunächst nötig einen *RMMMapView* 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* lädt und dem MapView zur Verfügung stellt. In Listing 2 wird diese Initialisierung gezeigt.

```

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

```

Listing 2: Initialisierung des MapView mit eigenem Kartenmaterial

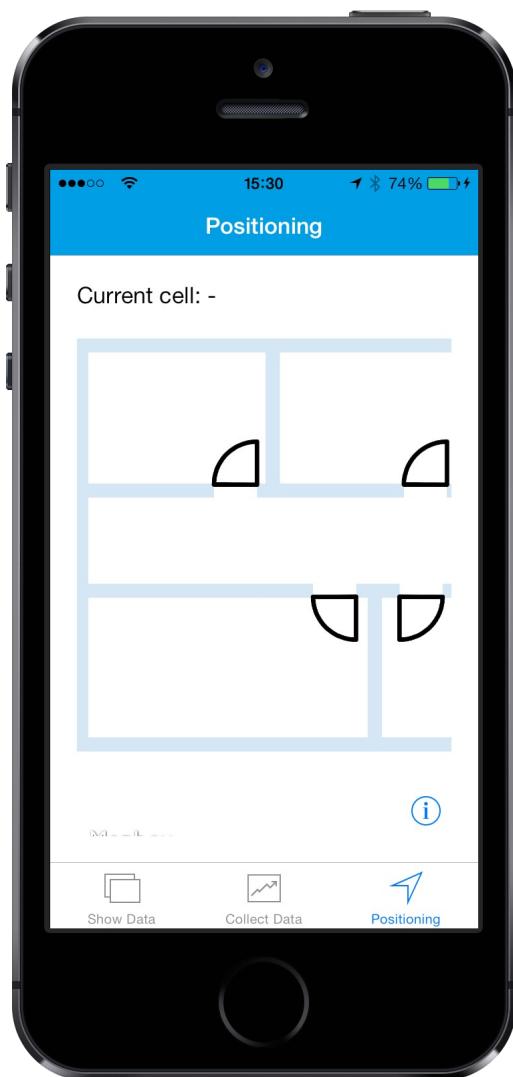


Abbildung 2.6: Kartenausgabe mittels Mapbox SDK auf dem iPhone

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-Relationship-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 NSFetchedResultsController *request = [[NSFetchedResultsController alloc] init];
5 [request setEntity:entityDescription];
6
7 NSNumber *minimumSalary = 3000;
8 NSPredicate *predicate = [NSPredicate predicateWithFormat:
9     @"(lastName LIKE[c] 'muller') AND (salary > %@", minimumSalary];
10 [request setPredicate:predicate];
11
12 NSArray *array = [moc executeFetchRequest:request error:&error];
```

Listing 3: Fetch Request für alle Objekte die mit Nachnamen "muller" heißen und mehr als 3000 Euro im Monat verdienen

Als Rückgabewert erhält man ein Array mit allen Objekten auf die die gegebenen Kriterien zutreffen.

3 Werkzeuge

3.1 Xcode

Xcode ist eine integrierte Entwicklungsumgebung 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.

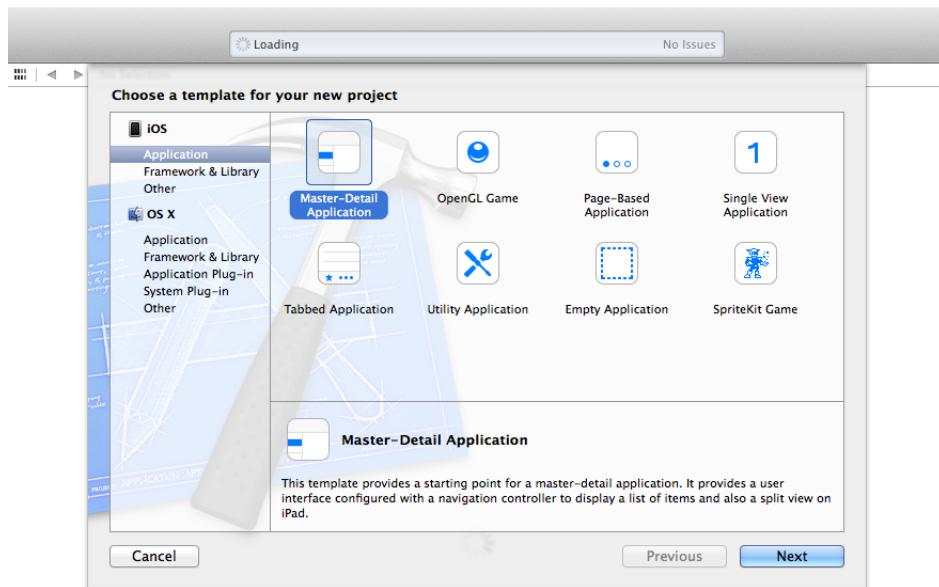


Abbildung 3.1: Auswahlbildschirm der verschiedenen Templates

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 mehreren View Controllern, die jeweils eine gezeigte Szene auf dem Gerät repräsentieren. Die einzelnen View Controller sind mit sogenannten *Segue's* verbunden, welche sich durch bestimmte Aktionen, wie zum Beispiel den Druck auf einen Button, auslösen lassen.

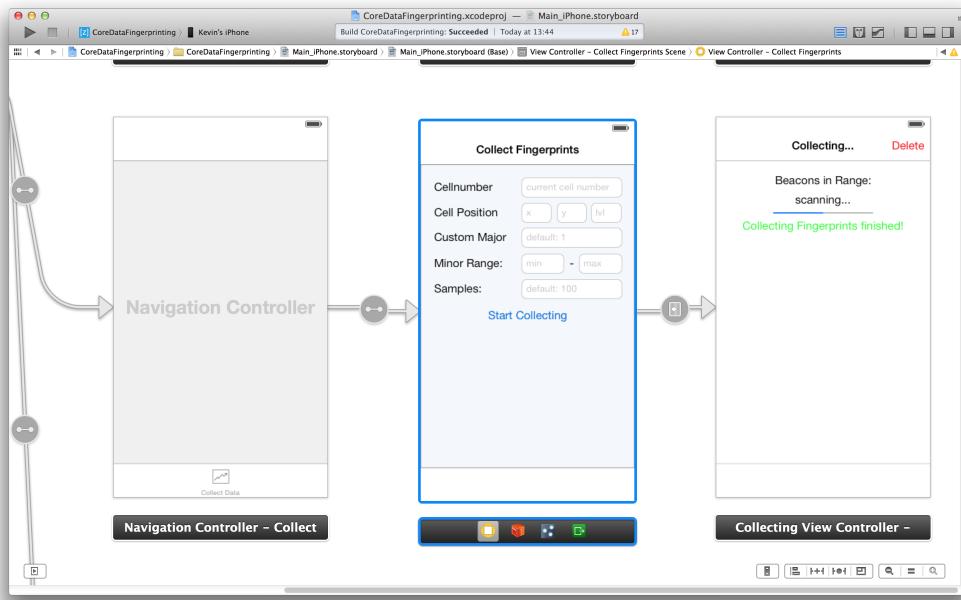


Abbildung 3.2: Beispiel eines Storyboards für iPhones

Das Storyboard bietet außerdem noch die Funktion des *Auto Layout*. Dabei werden sogenannte *Constraints* genutzt, welche die Positionsbeziehungen zwischen den einzelnen Elementen festlegen. Diese Constraints 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.

Aufgabe
r Constraints be-
schreiben

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.

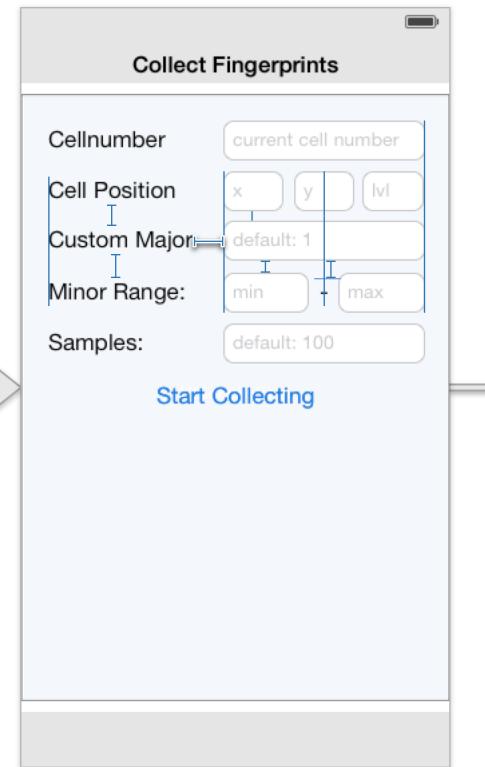


Abbildung 3.3: View Controller mit Constraints

3.3 Versionsverwaltung mit Git

Xcode bietet für die Versionsverwaltung eine integrierte Git-Unterstützung, welche einfach und schnell zu bedienen ist. Dabei werden die Differenzen innerhalb des Verzeichnisses bei einem Commit grafisch dargestellt und auch die Unterschiede innerhalb der Datei werden angezeigt.

Des Weiteren lassen sich die Änderungen auf Knopfdruck auf einen Server *pushen* und vom Server *pullen*.

Ausserdem lassen sich sehr einfach neue Branches erstellen und ein Mergen der Branches ist auch auf Knopfdruck möglich.

3.4 iOS Developer Program

Um eine programmierte Anwendung letztendlich auf einem iOS-Gerät auszuführen, ist die Mitgliedschaft im iOS Developer Program notwendig. Diese erlaubt das Testen der Anwendung auf dem Gerät, die Veröffentlichung im AppStore und gewährt Zugriff auf das iOS Beta-Programm, um Anwendungen für neue Versionen des Betriebssystems zu optimieren. Die Mitgliedschaft in diesem *iOS Developer Program* kostet jährlich 99 Dollar. Im Rahmen meiner Bachelorarbeit wurde mir der Zugang zu diesem Programm von der Universität zur Verfügung gestellt.

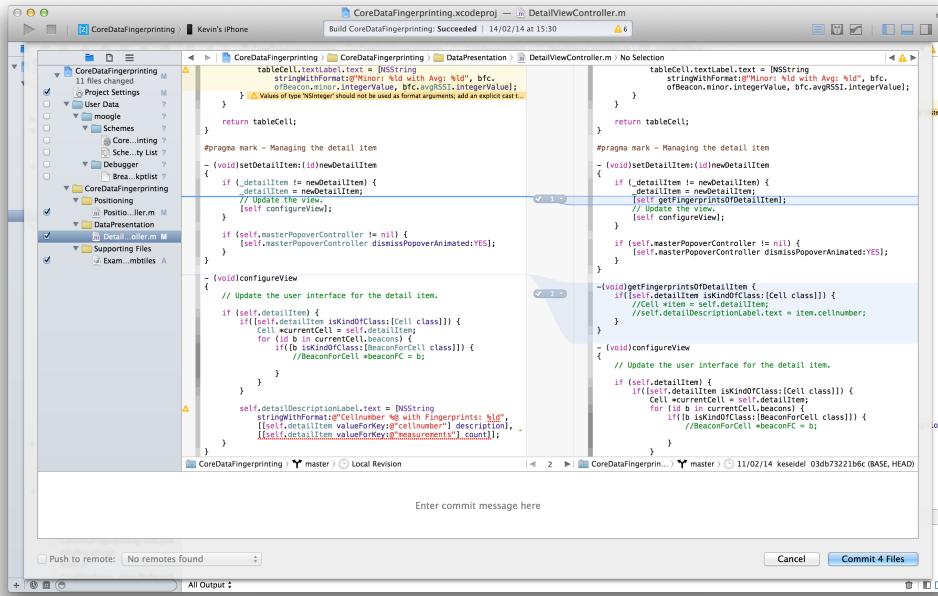


Abbildung 3.4: Xcode Versionsverwaltung mit Diff-Anzeige bei einem Commit

3.5 iPhone

Für die Entwicklung und das Testen der Applikation wurde ein iPhone 5 und ein iPhone 4s verwendet. Hauptsächlich wurde das iPhone 5 genutzt, wobei das iPhone 4s eher als Vergleichsgerät diente, um zum Beispiel Messungen zu überprüfen.

4 Daten und Messungen

4.1 Mobile iBeacons

Die mobilen iBeacon verzichten, wie der Name schon andeutet, auf eine feste Stromquelle und werden ausschließlich mit Batterien betrieben. Zum Einsatz kommen dabei die sogenannten Knopfzellen, welche mit einer Spannung von 3,0 Volt operieren. Da Bluetooth Low Energy extrem energiesparend arbeitet, geben die Hersteller der Beacons, die Akkulaufzeit mit bis zu zwei Jahren, ohne einen Batteriewechsel an. Diese Laufzeit hängt jedoch auch stark von der gewählten Signalstärke und dem gewählten Sendeinterval zusammen, welche die Laufzeit sehr stark beeinflussen können.

Bisher gibt es wenige Hersteller dieser iBeacons und der Großteil der Produkte befindet sich momentan noch in der Entwicklungsphase. Die genutzten iBeacons von *estimote* und *kontakt.io* sind ebenfalls noch in der Entwicklungsphase und hauptsächlich als Testgeräte für Entwickler ausgelegt. Dabei bleibt jedoch unklar in wie weit sich das fertige Produkt in den technischen Spezifikationen von den aktuellen Prototypen unterscheiden wird.

4.1.1 *estimote* Beacon

Die Firma *estimote* mit Sitz in Polen, war ein der ersten, die ein funktionstüchtiges iBeacon vorgestellt haben und es in einem *Developer Preview Kit* zum Verkauf anbieten. Dieses Kit beinhaltet drei verschiedenfarbige Beacons, welche mit einer wiederverwendbaren Klebeschicht an der Unterseite ausgestattet sind. Dies erlaubt das beliebige Anbringen und Abziehen der Beacons auf allen glatten Oberflächen.

Im inneren des Beacons befindet sich ein Bluetooth Chipsatz von Nordic Semiconductor, welcher auf einem 32-bit ARM Prozessor beruht und mit einem 2,4Ghz Bluetooth Low Energy Modul arbeitet. Dabei verfügt der über 256 KB Flash-Speicher für Speicherung der Beacon-Konfiguration. Speziell in den *estimote*-Beacon wurde dazu noch ein Temperatursensor eingebaut, welcher allerdings momentan noch nicht angesprochen werden kann.

Des Weiteren stellt *estimote* noch ein SDK für Android und iOS zur Verfügung, welches in Fall des iOS-SDK auf der iBeacons-API basiert, jedoch speziell auf die *estimote*-Beacons abgestimmt ist. Dabei bietet es neben den Funktionen der iBeacon-API noch die Funktionalität, sich mit den *estimote* Beacons zu verbinden und diese zu programmieren. So erlaubt es zum Beispiel die Signalstärke, das Sendeinterval und die Major-Minor-Informationen zu verändern oder die Firmware der Beacons zu aktualisieren.

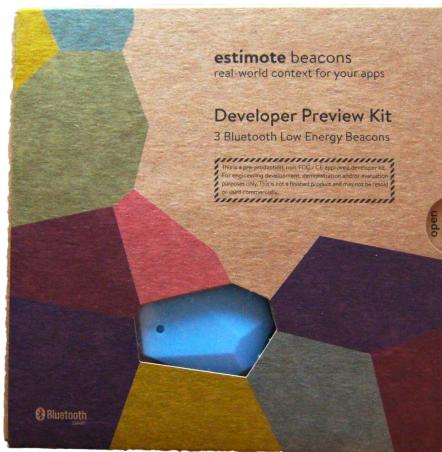


Abbildung 4.1: Das Developer-Kit von estimote

Da das SDK, bis auf die Programmierung der Beacons, keine Vorteile gegenüber dem Core Location-Framework mit der iBeacons-API bietet, wurde jedoch auf die Verwendung verzichtet.

4.1.2 kontakt.io Beacon

Ein weiteres Unternehmen, welches sich eine eigene iBeacons-Lösung anbietet ist *kontakt.io*. Auch hier ist noch kein finales Produkt erhältlich, sondern nur ein *Dev Kit*, welches zehn Beacons enthält. Die Beacons sind relativ einfach gehalten und das Innere ist sehr einfach zugänglich, sodass ein Batteriewechsel sehr einfach ist.



Abbildung 4.2: Kontakt.io Beacon

Die Beacons von *kontakt.io* basieren dabei auf dem BLE113 Chipsatz von *bluegiga*, welcher über 256 KB Flash-Speicher verfügt und über einen 8051 Mikrocontroller von Intel verfügt.

Auch *kontakt.io* bietet ein eigenes SDK an, welches im Gegensatz zu dem SDK von *estimote* nicht nativ für die einzelnen Platformen entworfen wurde, sondern online über eine REST-Schnittstelle arbeitet. Dabei stellt *kontakt.io* ein Webpanel zur Verfügung, in

welchem man die einzelnen Beacons mit ihrem UUID, Major und Minor-Wert registriert und jedem den jeweiligen Ort beziehungsweise die Funktion zuweisen kann.

4.2 Stationäre iBeacons

Neben den mobilen iBeacons, welche mittels Batterien funktionieren, gibt es auch stationäre iBeacons, welche über eine stetige Anbindung an das Stromnetz angewiesen sind. Dabei gibt es verschiedene Ansätze. Zum einen bietet *PayPal* einen Ansatz, bei dem die komplette Technik in einen USB-Stick integriert wird und dann über ein Standard USB-Netzteil an jeder Steckdose betrieben werden kann.

Eine andere Lösung ist die Nutzung eines Bluetooth 4.0-kompatiblen USB-Dongles an einem Computer. Dieser kann mit entsprechender Software zu einem iBeacon umfunktioniert werden.

4.2.1 Raspberry Pi als iBeacon

Der Raspberry Pi ist ein Mini-Computer, welcher auf einem ARM-Prozessor basiert und als günstiger Computer für Programmierersteiger konzipiert wurde. Der kleine Computer ermöglicht aber auch andere Einsatzgebiete, zum Beispiel als Beacon.

Hierbei wurde eine Linux-Distribution auf dem Gerät installiert und als Bluetooth-Dongle kann ein Modul von *Plugable Technologies* zum Einsatz, welches speziell Bluetooth 4.0 Unterstützung bietet. Für die Umfunktionierung zum iBeacon wurde die Bluetooth-Software *blueZ* eingesetzt, welche es erlaubt das Bluetooth-Modul anzusprechen und spezifische Nachrichten über Bluetooth zu schicken. Diese Möglichkeit wurde von der Firma Radius Network vorgestellt, welche auch ein ausführliches Tutorial für die Nutzung des Raspberry Pi als iBeacon auf ihrer Webseite anbieten (Nebeker and Young (2014)).

4.3 Außenmessungen

4.4 Innenraummessungen

Um die Leistungsfähigkeit und das Verhalten der Beacons in Innenräumen zu testen und darzustellen, wurden verschiedene Messungen durchgeführt. Dazu wurden zum einen die mobilen Beacons verwendet und zum anderen der Raspberry Pi, als stationäres Beacon. Die Messungen wurden dabei sowohl mit dem iPhone 5 als auch mit dem iPhone 4s durchgeführt, um auch hier die Unterschiede zwischen den einzelnen Modellen zu erfassen.

Zuerst wurden die Messungen mit den mobilen Beacons, hier die *kontakt.io*-Beacons, durchgeführt. Diese wurden in einem leeren, nur an den Wänden bestellten, Raum durchgeführt, wobei immer freie Sicht zwischen den Beacons und den Empfangsgeräten

bestand. Für jede Entfernung wurden dabei 100 Stichproben genommen, jeweils eine pro Sekunde.

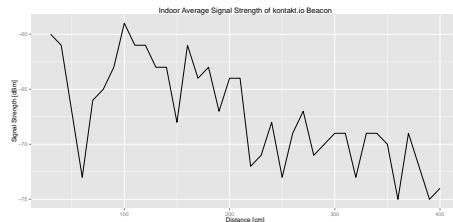


Abbildung 4.3: Messung des iPhone 5

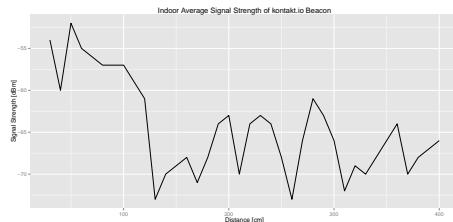


Abbildung 4.4: Messung des iPhone 4s

Abbildung 4.5: Durchschnittliche Signalstärke eines kontakt.io Beacons

In Abbildung 4.3 und Abbildung 4.4 lässt sich dabei sehr gut erkennen, dass die Signalstärke, nicht wie eigentlich erwartet stetig abnimmt, sondern relativ stark schwankt. Dies ist darauf zurückzuführen, dass in Innenräumen sowohl Wände, als auch Gegenstände im Raum, das Bluetoothsignal reflektieren oder blockieren und so die Ergebnisse verfälschen. Des Weiteren ist zu erkennen, dass die Ergebnisse zwischen den verschiedenen iPhone-Modellen deutlich voneinander abweichen. Das lässt darauf schließen, dass der verbaute Chipsatz beziehungsweise die verbaute Antenne innerhalb der Gerät die Ergebnisse deutlich beeinflusst und die Werte daher nur schwer übertragbar sind.

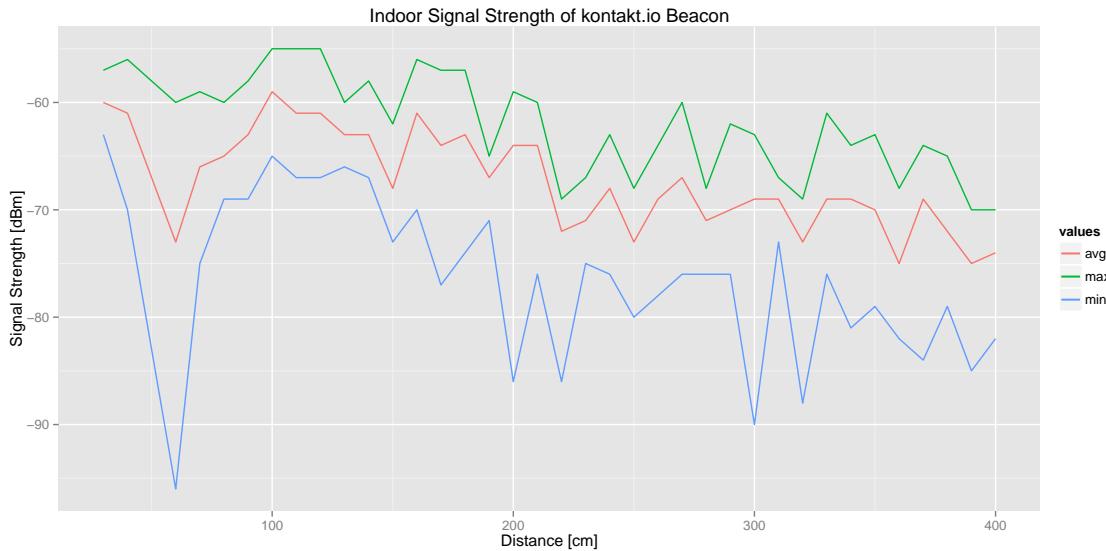


Abbildung 4.6: Minimale, maximale und durchschnittliche Signalstärke des Beacons gemessen vom iPhone 5

4.5 Mögliche Störfaktoren

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>.
- J. Nebeker and D. G. Young. How to make an ibeacon out of a raspberry pi. 2014. URL <http://developer.radiusnetworks.com/2013/10/09/how-to-make-an-ibeacon-out-of-a-raspberry-pi.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)