



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. Da wir in der Realität jedoch den Großteil unserer Zeit in Gebäuden aufhalten, ist der GPS-Ansatz dort wenig hilfreich.

Daher ist es sinnvoll, eine Alternative zu GPS zu schaffen, welche diese Funktionen in Innenräumen realisiert. Da man jedoch für Innenräume kein eigenes Navigationssystem kaufen möchte, liegt die Idee nah, diese Konzept auf einem Gerät zu realisieren, welches viele Menschen schon besitzen und auch für die GPS-Navigation nutzen. Das Smartphone.

In Abbildung 1 ist zu erkennen, wie die Verbreitung der Smartphones in den letzten Jahren sehr stark zugenommen hat. Dadurch kann man annehmen, dass ein Großteil der potentiellen Nutzer der Indoor Positionierung auch ein Smartphone besitzen.

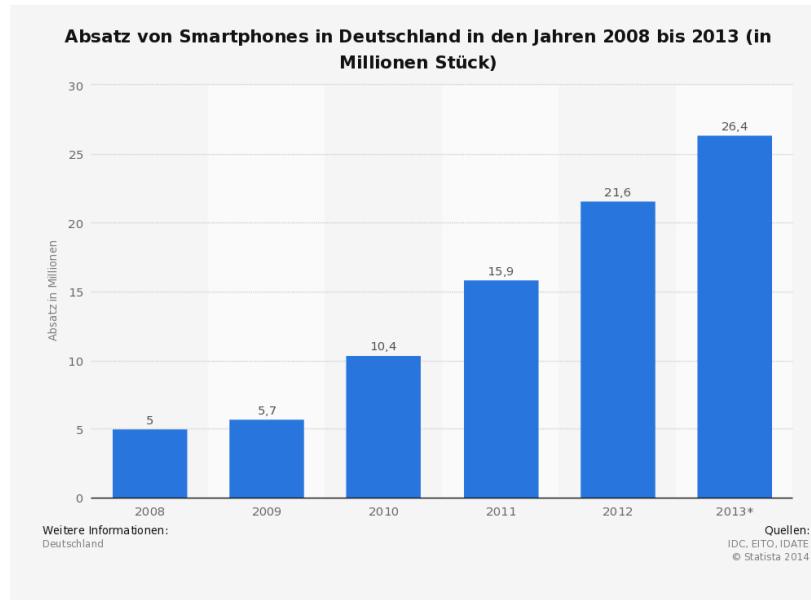


Abbildung 1.1: Smartphoneabsatz in Deutschland

Für die Realisierung der Indoor Positionierung kommen verschiedene Technologien in Frage. Darunter zum Beispiel Wireless LAN, RFID oder Bluetooth. Diese Technologien

bieten sich an, da sie standardmäßig in vielen Smartphones integriert sind und so nicht der Zwang besteht ein neues Gerät oder eine Erweiterung zu kaufen.

Schlussendlich viel die Entscheidung der zu verwendenden Technologie auf Bluetooth, da dieses eine sehr hohe Verbreitung bietet und auch viele Vorteile mit sich bringt. Zum einen ermöglicht Bluetooth eine schnelle und einfache Einrichtung und zum anderen benötigen die Bluetooth-Sendestationen wenig Energie, sodass nicht zwingend einen Stromanschluss vorhanden sein muss, sondern auch einen Batteriebetrieb über mehrere Monate bis Jahre hinweg möglich ist.

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. So setzt zum Beispiel Apple selbst die iBeacons-Technologie in ihren Geschäften ein, um dem Kunden gezielte Werbung zu den in der Nähe befindlichen Produkten zu bieten.

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 eignen, um Endgeräte zum Beispiel in Verkaufsräumen zu orten und zu identifizieren.

Dabei soll bestimmt werden, welches Verfahren sich am Besten für die Positionierung in Innenräumen eignet und ob es Unterschiede zwischen verschiedenen Sende- und auch Empfangsgeräten gibt.

Für diese Tests werden ausschließlich Apple-Geräte genutzt, da hier eine übersichtlichere Auswahl auf dem Markt herrscht, sodass die Basis der zu nutzenden Geräte überschaubar bleibt und nur wenige Geräte getestet werden müssen. Ein weiterer Grund ist, dass iOS, das Betriebssystem des Apple iPhone und iPads, bisher das einzige mobile Betriebssystem ist, welches die iBeacons-Technologie offiziell und nativ unterstützt.

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.

1.3 Überblick

2 Technologien und Werkzeuge

2.1 Bluetooth 4.0

Der Bluetooth-Standard 4.0, auch Bluetooth Smart genannt, wurde am 30.Juni 2010 verabschiedet. Darin enthalten sind alle Protokolle der vorherigen Version 3.0, sowie Fehlerkorrekturen und Erweiterungen und ein neues Protokoll, Bluetooth Low Energy.

Das erste unterstützte Mobilfunkgerät war das iPhone 4s, welches am 4. Oktober 2011 vorgestellt wurde. Im Jahr 2012 integrierten auch andere Smartphone-Hersteller Bluetooth 4.0 in ihre Geräte, sodass alle neueren Geräte diesen Standard beherrschen.

2.1.1 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 sollten. Das Hauptaugenmerk galt dabei dem geringen Stromverbrauch, einer kompakten Bauweise und den geringen Kosten der benötigten Hardware. Im Jahr 2007 wurden diese Spezifikationen in den, sich in der Entwicklung befindlichen, Bluetooth-Standard 4.0 aufgenommen. Wibree wurde 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 beispielsweise nicht für Headsets genutzt werden kann, da die zur Verfügung stehende Übertragungsrate nicht für eine 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 den drastisch gesenkten Energieverbrauch im Vergleich zu den Vorgänger-Versionen.

Des Weiteren wird eine 24-Bit-Fehlerkorrektur eingesetzt, welche die Verbindung unempfindlicher für Störungen und Übertragungsfehler machen soll und unnötige Neubertragungen verhindert.

Auch die Verschlüsselung des zu übertragenden Signals wurde verbessert. Dabei kommt der Advance Encryption Standard (AES) mit einer Schlüssellänge von 128 Bit zum Einsatz.

Central und Peripheral Bei einer Bluetooth Low Energy-Verbindung unterscheidet man zwischen *Central* und *Peripheral*. Das Gerät in der Rolle des *Central* horcht dabei auf gesendete Pakete in der Umgebung. Das Gerät in der Rolle des *Peripheral* hingegen sendet Pakete an das verbundene oder an alle in der Umgebung befindlichen Geräte.

Link Layer

ATT Das *Attribute Protocol*, oder kurz *ATT*, wird bei allen Datenübertragungen mittels Bluetooth Low Energy genutzt. Es wurde speziell für Bluetooth LE entworfen, um eine schnelle und sparsame Übertragung zu ermöglichen. Es basiert dabei auf einer Client-Server-Architektur. Der Server verwaltet die Daten und der Client fordert Daten vom Server an. Nach einer erfolgreichen Anfrage sendet der Server diese zum Client.

GATT Das *Generic Attribute Profile (GATT)* ist verpflichtend für alle Bluetooth LE-Profile. Es basiert dabei auf ATT und erlaubt ein einfaches Auffinden der Attribute und einen schnellen Zugriff. Ein Beispiel für ein GATT-Profil ist das *Heart Rate Profile*, welches für Herzfrequenzmessungen designet wurde. Jedes Profil besteht dabei aus verschiedenen Attributen, welche hierarchisch aufgebaut sind.

An oberster Stelle steht dabei das *Service-Attribut*. Ein Service besteht aus einer Sammlung von *Characteristics*. Eine Beispiel für einen GATT-Service ist der *Heart Rate Service*, welcher aus mehreren *Characteristics* besteht, wie zum Beispiel *Heart Rate Measurement Characteristic*, *Body Sensor Location Characteristic*, usw. Das *Characteristic-Attribut* besteht aus einem Messwert und einem *Descriptor*. Der aktuelle Wert ist abhängig von der *Characteristic*, so gibt die *Heart Rate Measurement Characteristic* beispielsweise die aktuelle Herzfrequenz als Wert zurück. Der *Descriptor* liefert zusätzliche Information über die aktuelle *Characteristic*, wie zum Beispiel den erlaubten Wertebereich oder die Einheit in welcher der Messwert vorliegt.

2.1.2 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, proprietären GATT-Profil.

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

Für die Identifizierung sendet das Beacon drei Werte, den *Universally Unique Identifier (UUID)*, den *Major-Wert* und den *Minor-Wert*. Der *UUID* ist ein Identifier, welcher Beacons einem bestimmten Typ oder einem Unternehmen zuordnen. Dieser UUID lässt sich mittels diversen Programmen generieren.

Der *Major-Wert* dient zur Unterscheidung von Beacons mit dem selben UUID und wird dazu eingesetzt, verschiedene Standorte beziehungsweise Regionen zu unterscheiden. Ein Beispiel dafür wäre ein Unternehmen mit mehreren Standorten, sodass bei gleichem UUID eine eindeutige Bestimmung des Standortes möglich ist.

Der *Minor*-Wert dient zur weiteren Unterscheidung der Beacons mit gleichem UUID und Major-Wert. Vorgesehen ist der Minor-Wert zur Bestimmung eines einzelnen Beacons in einer bestimmten Region, es ist jedoch nicht verboten mehreren Beacons die gleichen UUID, Major und Minor-Werte zu zuweisen, wodurch jedoch keine eindeutige Identifizierung mehr möglich ist.

Neben den Identifikationsdaten kann das Empfangsgerät noch weitere Größen bestimmen. Es ist so zum Beispiel möglich die ungefähre Entfernung zu erhalten. Für diesen *Proximity*-Wert sind dabei vier verschiedene Entfernungs-Zustände definiert: *Far*(mehr als 10m), *Near*(wenige Meter), *Immediate* (wenige Zenitmeter) und *Unknown*(Entfernung konnte nicht bestimmt werden). Diese Werte erlauben eine sehr grobe Entfernungseinschätzung zum Beacon. Für eine differenziertere Entfernungsbestimmung lässt sich eine weitere Kenngröße auslesen, der *Accuracy*-Wert. Dabei handelt es sich um eine ungefähre Entfernungsangabe in Metern, welche jedoch ausdrücklich nur zur Differenzierung der Entfernung zweier Beacons genutzt werden soll und keinesfalls einen genauen Abstand zum Beacon angibt. Der Accuracy-Wert soll dabei erlauben, bei gleichem Proximity-Wert, dass nächstgelegene Beacon zu bestimmen.

	Format	Beschreibung	Beispiel
UUID	16-stellige Hexadezimalzahl	Identifizierung der Beacons	9E711191-7DE1-4CA8-850C-7368BD1DD449
Major	Integer	Identifizierung der Region	42
Minor	Integer	Identifizierung des einzelnen Beacons	1337
Proximity	3 Entfernungsstufen	Große Entfernung	Far, Near, Immediate oder Unknown
Accuracy	Entfernung in Meter	Ungewährte Entfernung	1,3
RSSI	Signalstärke in dBm	Signalstärke des empfangenen Signals	-47

Abbildung 2.1: Daten in der iBeacon-Übertragung

Die von dem Beacon gesendeten Daten lassen sich mit jedem BLE-kompatiblem Gerät empfangen, bisher bietet jedoch nur iOS eine entsprechende, native Unterstützung für das iBeacon-Profil.

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 mit einer Knopfzellen Batterie zu betreiben und das, laut Herstellerangaben, für bis zu zwei Jahre. Der Aufbau eines solchen Beacons lässt sich in Abbildung 2.4 erkennen. Den Großteil des Beacons nimmt dabei die Batterie ein.

Unter genauerer Betrachtung der Platine in Abbildung 2.5, erkennt man, dass diese 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 über die letztendlich die Daten gesendet werden.

2.2 Xcode

Xcode ist eine integrierte Entwicklungsumgebung, welche von Apple entwickelt wurde. Xcode ermöglicht es *iOS* und *OS X* Applikationen zu erstellen, zu testen und zu debug-



Abbildung 2.2: Außenhülle



Abbildung 2.3: Chipsatz mit Bluetooth-Modul

Abbildung 2.4: Ein iBeacon der Firma "estimote"

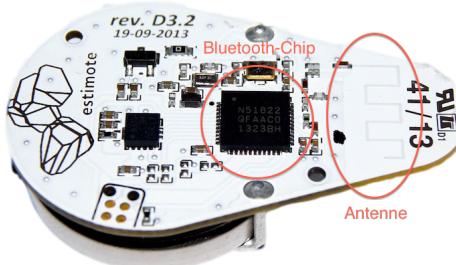


Abbildung 2.5: Aufbau des estimote-Beacons

gen. 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, ohne diese auf ein reales Gerät zu übertragen.

Bei Erstellung einer neuen Applikation kann man unter mehreren Templates wählen, welche jeweils verschiedene Funktionen mit sich bringen. In Abbildung 2.6 lassen sich die verschiedenen Auswahlmöglichkeiten erkennen.

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 2.7 zu erkennen, besteht das *Storyboard* aus mehreren View Controllern, die jeweils eine gezeigte Szene auf dem Gerät repräsentieren. Die einzelnen View Controller sind mit so genannten *Segue's* verbunden, welche sich durch bestimmte Aktionen, wie zum Beispiel den Druck auf einen Button, auslösen lassen. Diese *Segue's* definieren, wie die Übergänge zwischen den ViewControllern ablaufen.

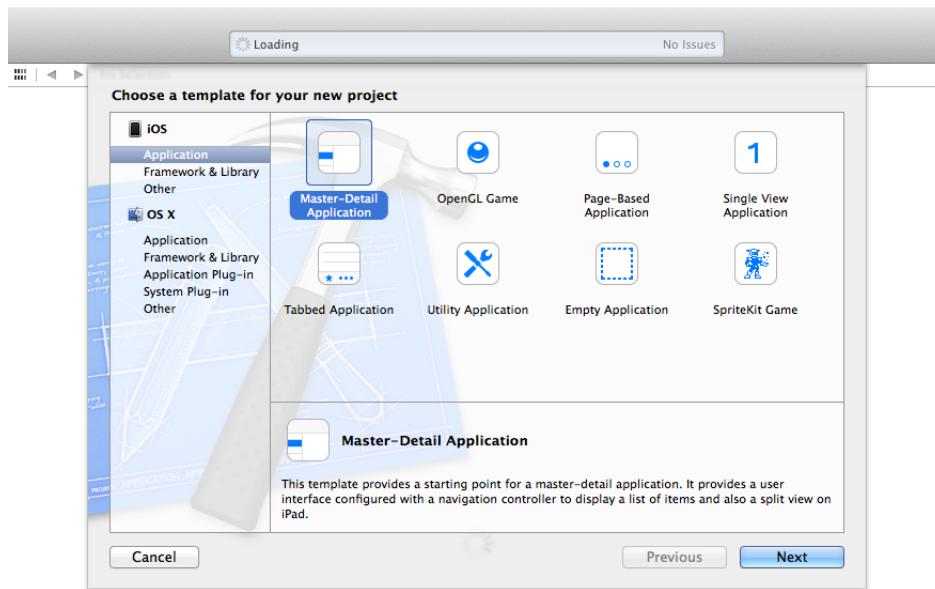


Abbildung 2.6: Auswahlbildschirm der verschiedenen Templates

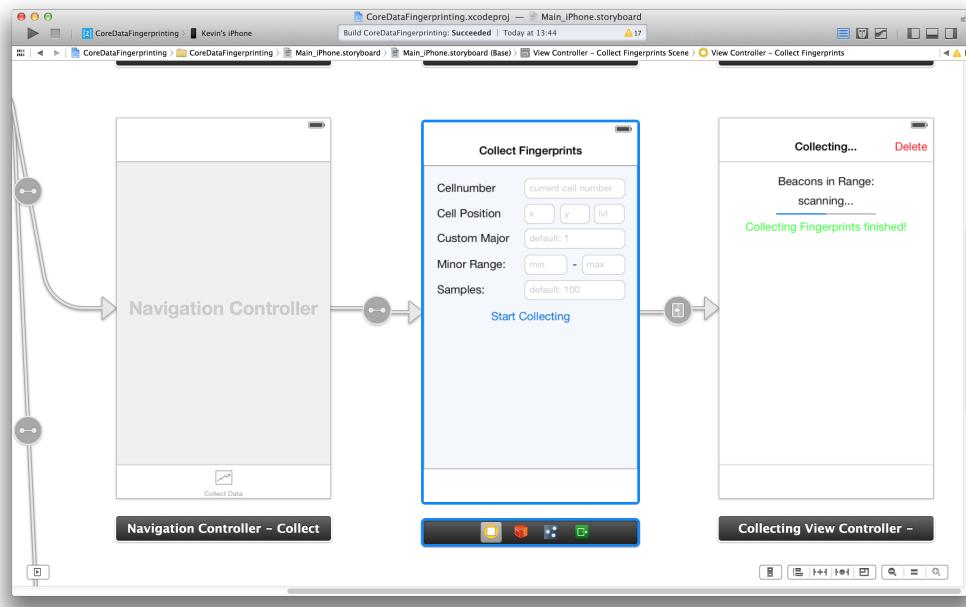


Abbildung 2.7: Beispiel eines Storyboards für iPhones

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 sich an das verwendete Gerät anpasst und so ein passendes User Interface, unabhängig der Bildschirmgröße oder der aktuellen Orientierung des Bildschirms, darstellt. In Abbildung 2.8 lassen sich die Constraints, also die Abstands- und Ausrichtungsbeziehungen zwischen den einzelnen Objekten, gut erkennen.

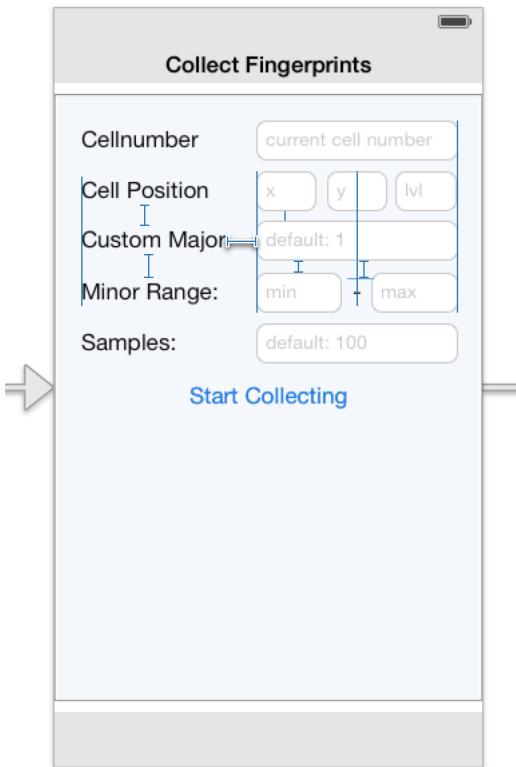


Abbildung 2.8: View Controller mit Constraints

Diese Constraints beschreiben dabei die Abstände und Ausrichtungen der einzelnen Elemente zu dem umschließenden ViewController oder den anderen Elementen.

2.3 iOS

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 5.1 und als iOS-Gerät setzte ich ein iPhone 5 mit iOS 7.1 und ein iPhone 4s mit iOS 7.0.6 ein. Als minimale iOS-Version musste iOS 7 verwendet werden, da die iBeacon-API des CoreLocation-Frameworks (mehr dazu im Kapitel 2.4) erst ab dieser Version zur Verfügung stehen.

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 App Store 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. Dabei

beschränkt sich der Funktionsumfang jedoch auf das Testen der Applikation auf dem Gerät, da die Veröffentlichung im AppStore und der iOS Beta-Programm nicht Teil des Pakets für Universitäten ist.

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.

Das Ausführen der geplanten Applikation auf einem realen Endgerät ist dabei unverzichtbar, da der iOS-Simulator nicht die Möglichkeit besitzt Bluetooth-Signale zu empfangen.

Das iPhone 4s wurde dazu genutzt, zu überprüfen in wie weit die Ergebnisse der Messungen übertragbar sind, beziehungsweise wie sie sich zwischen den einzelnen Modellen unterscheiden, da diese verschiedene Hardware einsetzen. So setzt das iPhone 4s auf den Broadcom BCM4330-Chipsatz, welches ein Wireless LAN-Chip mit integriertem Bluetooth 4.0 ist. Das iPhone 5 dagegen setzt auf den BCM4334, ebenfalls von Broadcom. Aber auch der Aufbau der Antennen und das Material der iPhones unterscheidet sich zwischen diesen beiden Generationen deutlich. So ist das iPhone 4s mit einer Glas-Rückseite ausgestattet, wohingegen das iPhone 5 einen Rückseite aus Aluminium besitzt.

Daher ist es wichtig zu vergleichen in wie weit die Änderungen die Empfangsqualität beeinflussen und zu bestimmen, ob eine Übertragung der Ergebnisse möglich ist oder ob jedes Gerät individuell behandelt werden muss.



Abbildung 2.9: Die für die Messungen und Tests genutzten iPhones

Objective-C Als Programmiersprache kam dabei *Objective-C* zum Einsatz, welches die primäre Sprache zur Programmierung von iOS-Anwendungen ist. Diese bietet viele nützliche Eigenschaften, wie etwa dynamisches Binden, eine dynamische Typisierung oder *Fast Enumeration*. Im Vergleich zu Java arbeitet *Objective-C* nicht mit Methodenaufrufen, sondern mit Nachrichten. Diese werden vom Sender zum Empfänger gesendet, wobei der Empfänger daraufhin entscheidet, welche Methode ausgeführt wird. Ein weitere Besonderheit von Objective-C sind die Properties (Eigenschaften), welche den Instanzvariablen entsprechen, jedoch noch weitere Konfigurationsmöglichkeiten hinsichtlich Schreibschutz und Atomarität bieten. Eine weiterer Unterschied ist die Syntax von Objective-C.

Der Hauptunterschied liegt in der Methodendeklaration. Statt des *static* Schlüsselwortes wird bei Objective-C eine Klassenmethode mit einem "+" gekennzeichnet, wobei eine

```

1  /** Objective-C **/
2  + (void) helloWorldWithName: (NSString*) name
3                                andSurname: (NSString*) surname {
4      NSLog(@"Hello %@ %@", name, surname);
5  }
6
7  + (void) main: (NSArray*) args {
8      [self helloWorldWithName: @"Max" andSurname: @"Mustermann"];
9  }
10
11 /**
12 public static void helloWorldWithNameAndSurname(String name, String surname)
13     System.out.print("Hello " + name + " " + surname);
14 }
15
16 public static void main(String[] args) {
17     helloWorldWithNameAndSurname("Max", "Mustermann");
18 }

```

Listing 1: Hello World-Beispiel in Objective-C und Java

Instanzmethode mit einem "-" deklariert wird. Der nächste Unterschied wird bei den Übergabewerten deutlich. Diese werden bei Objective-C mit einem ":" abgetrennt. Ein weiterer Unterschied wird beim Aufruf der Methoden deutlich. Da Objective-C nicht explizit die Methode aufruft, sondern eine Nachricht an die Klasse sendet, unterscheidet sich die Syntax deutlich. So werden hier geschweifte Klammern genutzt, wobei der Empfänger der Nachricht zuerst aufgeführt wird und danach die zu sendende Nachricht eingefügt wird.

2.4 CoreLocation-Framework

Das CoreLocation-Framework ist ein iOS-Framework, welches es erlaubt die aktuellen Positions- und Richtungsinformationen eines Gerätes zu bestimmen und auszugeben. Die Positionsbestimmung lässt sich dabei über verschiedene Sensoren und Werte bestimmen, wobei der Grad der Genauigkeit variabel ist. Für die Positionsbestimmung lässt sich dabei zum Beispiel das integrierte GPS-Modul verwenden. Auch die Aktualisierungsrate der Position lässt sich festlegen, wobei eine höhere Aktualisierungsrate und eine höhere Genauigkeit auch gleichbedeutend mit einem höherem Akkuverbrauch sind.

Bei der Genauigkeit gibt es dabei verschiedene Konstanten, welche bestimmen, mit welcher Genauigkeit die Position bestimmt werden soll.

Diese Genauigkeiten beziehen sich hauptsächlich auf die Positionierung mittels GPS und sind daher für die Indoor Positionierung nur bedingt geeignet. Es wäre jedoch denkbar,

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

Table 2.1: Mögliche Optionen der Positionsgenauigkeit

die Positionierung mittels GPS und die Positionierung mittels iBeacons zu verbinden und nahtlos in einander übergehen zu lassen.

Eine weitere Funktion des CoreLocation-Frameworks ist der Kompass, also die Bestimmung der Himmelsrichtungen. Durch den eingebauten Kompass in den neueren iOS-Geräten ist es möglich, die aktuelle Ausrichtung des Gerätes sehr genau zu bestimmen. Dies ist im Bezug auf die Indoor Navigation hilfreich, da diese Informationen in die Positionsbestimmung einbezogen werden können. Da der menschliche Körper die Signale der Beacons beeinflusst, ist es daher von Vorteil die aktuelle Ausrichtung zu kennen und so auch die Position des Körpers zu berücksichtigen.

Des Weiteren erlaubt diese Funktion eine dynamische Ausrichtung der Karte, abhängig davon wie das Gerät aktuell ausgerichtet ist.

Die für uns zentrale Funktion dieses Frameworks ist die Erkennung von iBeacons und die Funktionen zur Verarbeitung der von den Beacons gesendeten Daten. Mittels des Frameworks können Beacons anhand ihres UUID erkannt und einer Region zugeordnet werden. Die genaue Funktionsweise wird dabei im folgenden Kapitel 2.4.1 behandelt.

2.4.1 iBeacons-API

Seit der iOS Version 7 wurde das CoreLocation Framework um die Beacon-Funktionalitäten erweitert. Dazu wurden zwei neue Klassen hinzugefügt und das bestehende Framework dementsprechend angepasst. Hinzugefügt wurde zum Einen die `CLBeacon`-Klasse, welche ein Beacon repräsentiert und alle zur Verfügung stehenden Informationen enthält und zum Anderen die `CLBeaconRegion`-Klasse, welche eine Region mit mehreren Beacons, abhängig von ihrem UUID und weiteren Werten, beschreibt.

Die `CLBeacon`-Klasse besteht dabei lediglich aus Properties mit den gegebenen Beacon-Informationen, 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 Genauigkeits-Stufen zu initialisieren:

`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.

Die Beacon-Region bestimmt also letztlich nach welchen Beacons gesucht wird, beziehungsweise welche Beacons gefunden werden.

2.5 MapBox

MapBox ist ein Online-Landkarten Anbieter, welcher es erlaubt eigene Karten zu erstellen und über ihren Service online bereitzustellen. Die Grundkarten werden dabei aus dem OpenStreetMap-Projekt entnommen und MapBox erlaubt es diese Karten grafisch zu überarbeiten, um so zum Beispiel das Farbschema zu ändern, eigene Markierungen hinzuzufügen 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, wie zum Beispiel Pinch-to-Zoom, automatische Kompassausrichtung oder Annotationen, mit sich bringt. Ausserdem besitzt MapBox eine größere Flexibilität im Bezug auf die individuelle Anpassung der Karten und den Offline-Betrieb als das von Apple für iOS bereitgestellte MapKit-Framework. Das MapKit-SDK erlaubt die Karten direkt auf dem Gerät zu speichern.

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

Google hat mit *Google Maps Indoor* bereits einen Dienst gestartet, welcher Gebäudepläne in Google Maps integriert. Dabei handelt es sich bisher jedoch hauptsächlich um öffentliche Gebäude in US-amerikanischen Städten. In Deutschland ist der Dienst ebenfalls schon gestartet, beinhaltet jedoch nur wenige Gebäude. Das hinzufügen von neuen Gebäudeplänen ist nur bei öffentlichen Gebäuden möglich und nicht für den privaten Gebrauch vorgesehen, daher konnte nicht auf diesen Dienst zurückgegriffen werden.

Die Indoor-Karten mussten daher individuell für den Einsatzort erstellt und in ein, von Mapbox verständliches Format, umgewandelt werden. Die Ausgangsdatei ist dabei eine Bilddatei in JPEG-Format, welches die Karte des Innenraumes zeigt. Dieses Datei

muss zur weiteren Verwendung in ein von MapBox verständliches Format umgewandelt werden. Dazu wurde ein von "Tom MacWright" (?) bereitgestellte Python-Script verwendet, welches JPEG Dateien in GeoTIFF Dateien umwandelt. Die GeoTiff Datei speichert neben den eigentlichen Bildinformationen zusätzlich Koordinaten für die Georeferenzierung. ?

Mit dieser GeoTIFF-Datei 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.

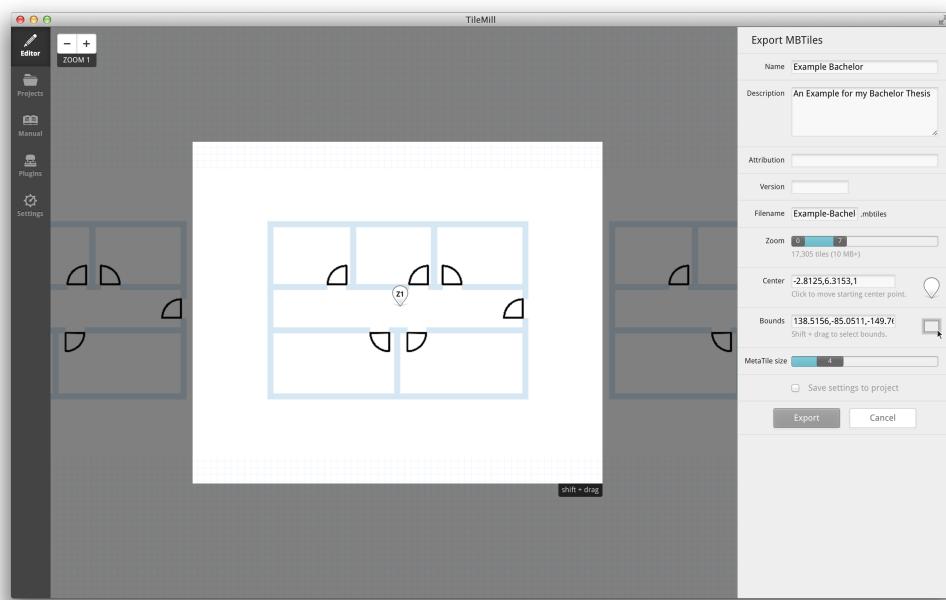


Abbildung 2.10: Karte in TileMill

TileMill erlaubt es nun die eingefügte Karte weiter zu bearbeiten oder weitere Informationen hinzuzufügen. 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 einzelne Kacheln überführt und speichert. Dadurch wird das Laden der einzelnen Kartenschnitte bei größeren Karten beschleunigt, da nicht die komplette Karte geladen werden muss, sondern nur die benötigten Kacheln.

Die erzeugte *.mbtiles*-Datei lässt sich nun in die iOS Applikation einbinden und über das SDK auf dem iOS-Gerät ausgeben. In Abbildung 2.11 sieht man die Ausgabe einer Karte auf dem iPhone 5.

Diese Karte wird offline auf dem Gerät gespeichert, sodass keine Internetverbindung für die Anzeige nötig ist.

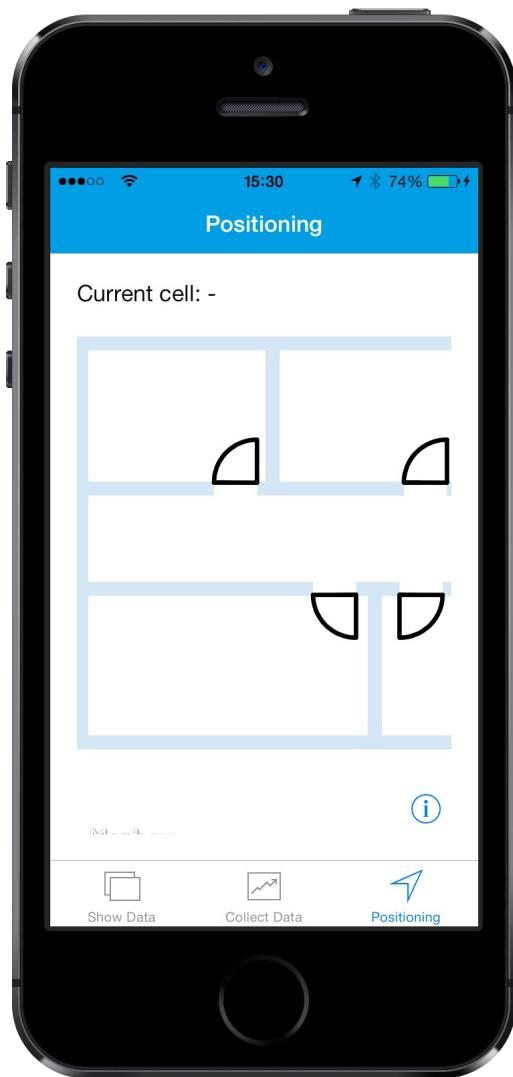


Abbildung 2.11: Kartenausgabe mittels Mapbox SDK auf dem iPhone

2.6 CoreData-Framework

Das CoreData-Framework erlaubt die Modellierung von Objekten und deren Speicherung auf dem Gerät. 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.

Für die Erstellung eines CoreData-Modells bietet Xcode einen eigenen Editor an, welcher es erlaubt, Entitäten zum Modell hinzuzufügen und deren Attribute anzupassen. Die Beziehungen der Entitäten lassen sich dort ebenfalls erstellen und bearbeiten. Das Modell lässt sich dabei sowohl grafisch als auch in Tabellenform anzeigen.

Nachdem ein CoreData-Modell erstellt wurde, benötigt man für den Zugriff auf das Modell ein *NSManagedObjectContext*-Objekt, welcher Lese- und Schreibzugriffe auf das

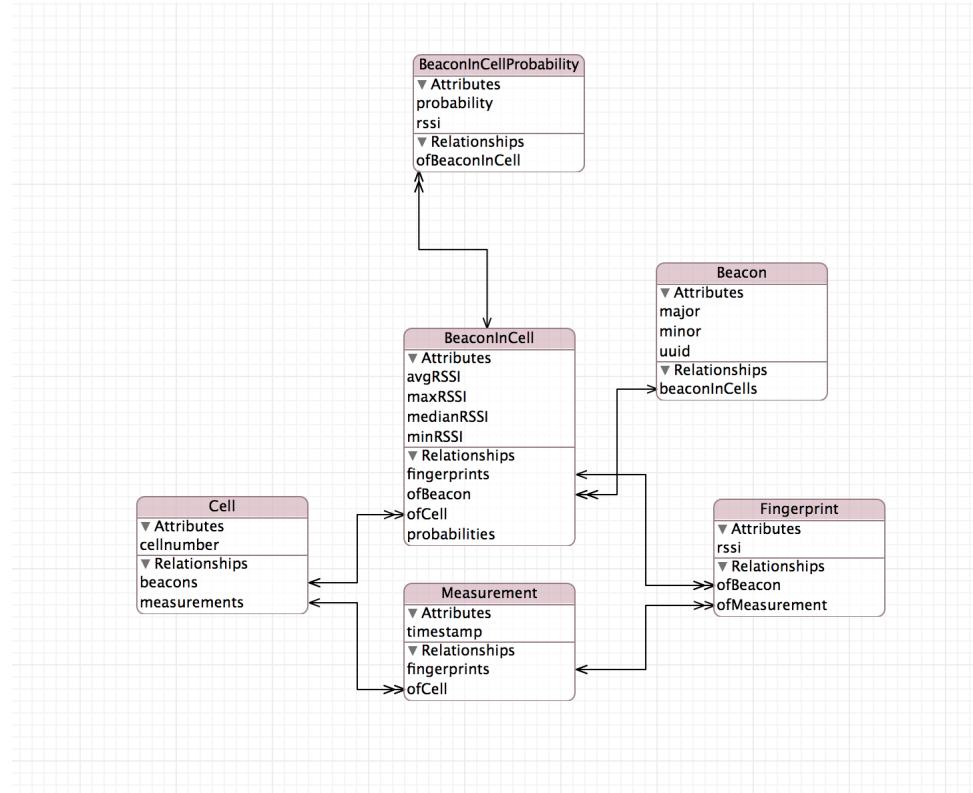


Abbildung 2.12: CoreData-Modell in der grafischen Darstellung.

Datenmodell steuert und verwaltet. Die einzelnen Objekte der Modells sind dabei *NSManagedObject*-Objekte. Nach dem Anlegen des Modells ist es jedoch auch möglich, automatisiert eigene Klassen für die einzelnen Datenobjekte erzeugen zu lassen, welche alle Attribute und Funktionen der einzelnen Objekte und Verbindungen beinhalten und somit den Zugriff und das Auslesen 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 Datenmodell ausliest. Dabei ist es möglich den *NSFetchRequest* genauer zu spezifizieren und so nur Objekte mit bestimmten Eigenschaften auszulesen. Dafür verwendet man ein *NSPredicate*, welches umfangreiche Tests auf bestimmte Attribute und logische Operationen erlaubt.

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

Die Attribute dieser Objekte können nun ausgelesen und verändert werden. Um Veränderungen auch im Datenmodell zu übernehmen, muss lediglich der *save*-Befehl des *NSManagedObjectContext* ausgeführt werden.

```
1 NSManagedObjectContext *moc = [self managedObjectContext];
2 NSEntityDescription *entityDescription = [NSEntityDescription
3     entityForName:@"Employee" inManagedObjectContext:moc];
4 NSFetchedResultsController *fetchedResultsController =
5     [[NSFetchedResultsController alloc] initWithFetchRequest:
6         [NSFetchRequest fetchRequest] managedObjectContext:
7         moc sectionNameKeyPath:nil cacheName:nil];
8 [fetchedResultsController performFetch:&error];
9 NSLog(@"Fetched %d Employee", [fetchedResultsController.fetchedObjects count]);
10 NSArray *array = [moc executeFetchRequest:fetchedResultsController.fetchRequest
11     error:&error];
```

Listing 2: Fetch Request für alle Objekte die mit Nachnamen "muller" heißen und mehr als 3000 Euro im Monat verdienen

2.7 Versionsverwaltung mit Git

Für die Verwaltung und Versionierung des Projektes wurde Git verwendet. Als Hosting-Plattform wurde dabei GitHub genutzt.

Git wurde gewählt, da es schnell und vergleichsweise einfach zu bedienen ist. Des Weiteren bietet Xcode bereits standardmäßig Git-Unterstützung mit einem grafischen Interface, welches alle nötigen Befehle wie Commit, Push, Pull oder die Erstellung eines neuen Branches auf Knopfdruck beherrscht.

Auch ein Diff-Editor ist integriert, welcher es erlaubt die Unterschiede im Quelltext, zwischen zwei Versionen zu begutachten.

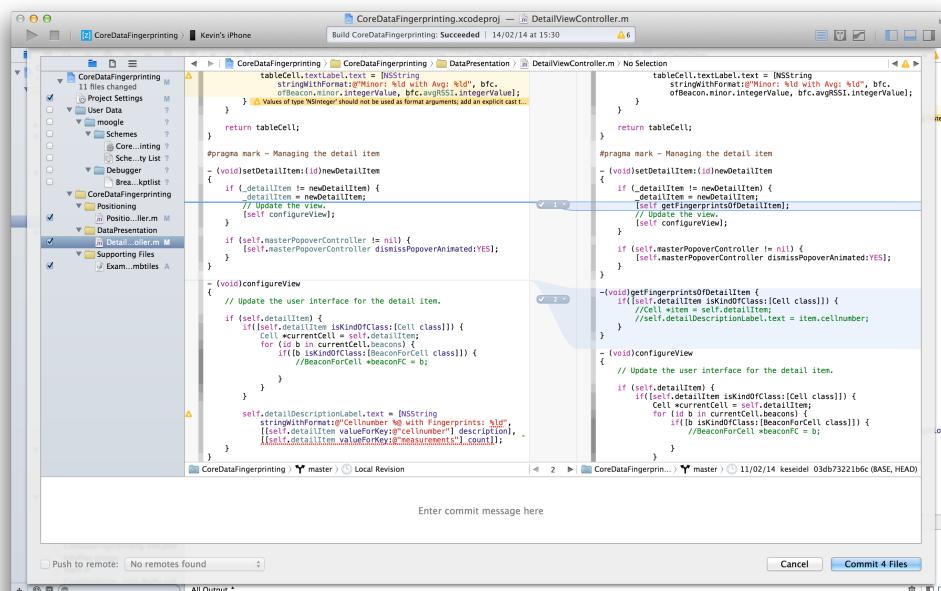


Abbildung 2.13: Xcode Versionsverwaltung mit Diff-Anzeige bei einem Commit

3 Daten und Messungen

3.1 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 mit der gewählten Signalstärke und dem Sendeintervall zusammen, welche die Laufzeit sehr stark beeinflussen können.

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

3.1.1 *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 ein beliebiges 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 Sendeintervall und die Major-Minor-Informationen zu verändern oder die Firmware der Beacons zu aktualisieren.



Abbildung 3.1: 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.

3.1.2 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 *Development Kit*, welches zehn Beacons enthält. Die Beacons sind relativ schlicht gehalten und das Innere ist sehr einfach zugänglich, sodass ein Batteriewechsel ohne Umstände möglich ist.



Abbildung 3.2: Kontakt.io Beacon

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

Auch *kontakt.io* bietet ein eigenes SDK an, welches im Gegensatz zu dem SDK von *estimote* nicht nativ für die einzelnen Plattformen 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. Auch auf die Verwendung dieses SDK wurde verzichtet.

3.2 Stationäre iBeacons

Neben den mobilen iBeacons, welche mittels Batterien funktionieren, gibt es auch stationäre iBeacons, welche auf eine stetige Anbindung an das Stromnetz angewiesen sind. Dabei gibt es verschiedene Ansätze. Zum einen bieten zum Beispiel *PayPal* und *Radius Networks* einen Ansatz, bei dem die komplette Technik in einen USB-Stick integriert wird und über ein USB-Netzteil an jeder Steckdose betrieben werden kann. Die verwendete Technik dieser Beacons entspricht denen der batteriebetriebenen Beacons.

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.

3.2.1 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.

Um den Raspberry Pi zu einem Beacon umzufunktionieren wurde eine Linux-Distribution auf dem Gerät installiert und ein Bluetooth-Dongle über USB angeschlossen. Dabei kam ein Modul von *Plugable Technologies* zum Einsatz, welches spezielle Bluetooth 4.0 und Linux-Unterstützung bietet. Für die Umfunktionierung zum iBeacon wurde die Bluetooth-Implementierung *blueZ* eingesetzt, welche es erlaubt das Bluetooth-Modul anzusprechen und spezifische Nachrichten über Bluetooth zu versenden. Diese Möglichkeit den Raspberry Pi als iBeacon zu nutzen, wurde von der Firma Radius Network vorgestellt. Diese bieten ein ausführliches Tutorial für die Nutzung des Raspberry Pi als iBeacon auf ihrer Webseite an (?), welches von mir genutzt wurde, um den Raspberry Pi zu konfigurieren.

3.3 Außenmessungen

3.4 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.

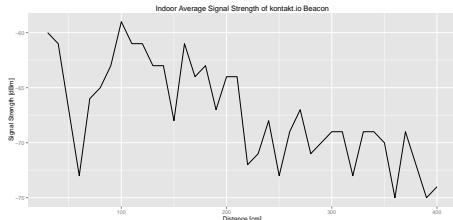


Abbildung 3.3: Messung des iPhone 5

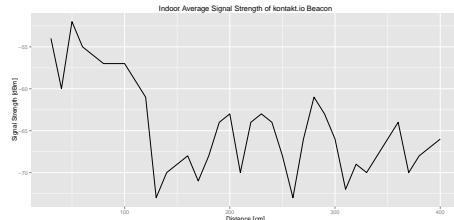


Abbildung 3.4: Messung des iPhone 4s

Abbildung 3.5: Durchschnittliche Signalstärke eines kontakt.io Beacons

In Abbildung 3.3 und Abbildung 3.4 lässt sich dabei sehr gut erkennen, dass die Signallösung, 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 Bluetooth-Signal 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.

Ein weiterer wichtiger Punkt ist die Untersuchung der Stabilität des Signals. Dabei wurden die gleichen Daten wie zuvor verwendet, jedoch um die minimalen und maximalen Werte ergänzt. In Abbildung 3.6 lässt sich dabei gut erkennen, dass die Ergebnisse eine ähnliche Tendenz haben, aber sich dennoch über einen sehr großen Bereich der Signallösung verteilen.

Zusätzlich wurde untersucht in wie weit sich die Sendeleistung und Signalqualität der batteriebetriebenen Beacon von stationären Beacons unterscheidet. Dazu wurde der gleiche Messaufbau wie zuvor genutzt, jedoch das *kontakt.io* Beacon durch den Raspberry Pi ausgetauscht. Danach wurden die gleichen Messungen erneut durchgeführt.

Wie aus Abbildung 3.7 zu entnehmen, sind die Ergebnisse im Vergleich zu der Messung der *kontakt.io* Beacons näher am erwarteten Ergebnis, welches eine konstante Abnahme der Signallösung sein sollte. Es sind jedoch immernoch einige Ausreißer zu erkennen. Bei der Betrachtung der minimalen und maximalen Signallösungen fällt auch auf, dass die ähnlich stark schwanken, wie schon zuvor bei den *kontakt.io* Beacons. Die Stabilität des Signals des stationären Beacons ist also genauso schwach beziehungsweise noch schwächer als die des batteriebetriebenen Beacons. Dies wird in Abbildung 3.8 noch einmal deutlich.

Für die weiteren Test und Messungen wurden daher die *kontakt.io* Beacons verwendet, da die beiden zur Verfügung stehenden Beacons sich in Signallösung und Stabilität nicht

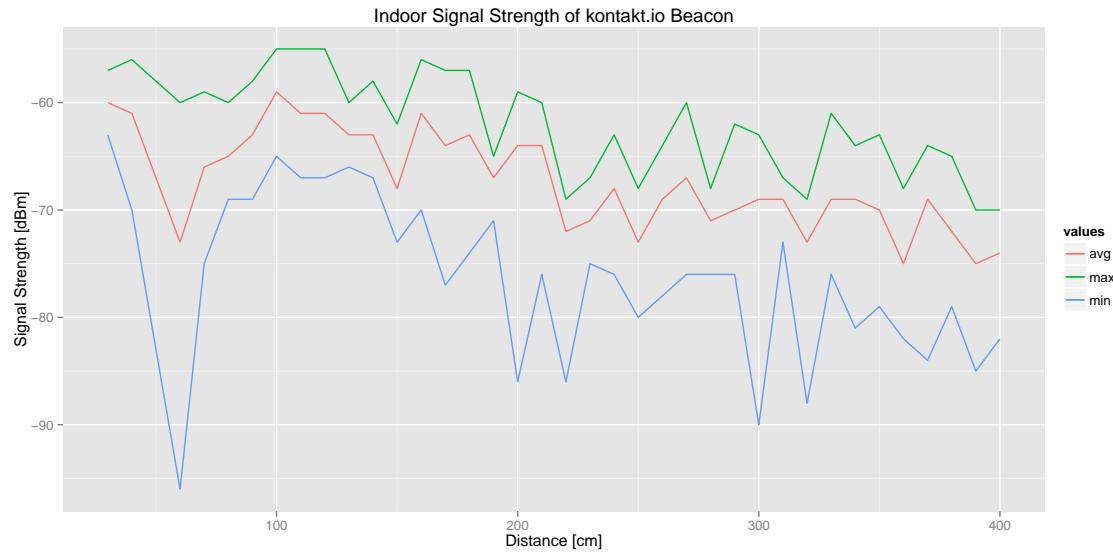


Abbildung 3.6: Minimale, maximale und durchschnittliche Signalstärke des Beacons gemessen vom iPhone 5

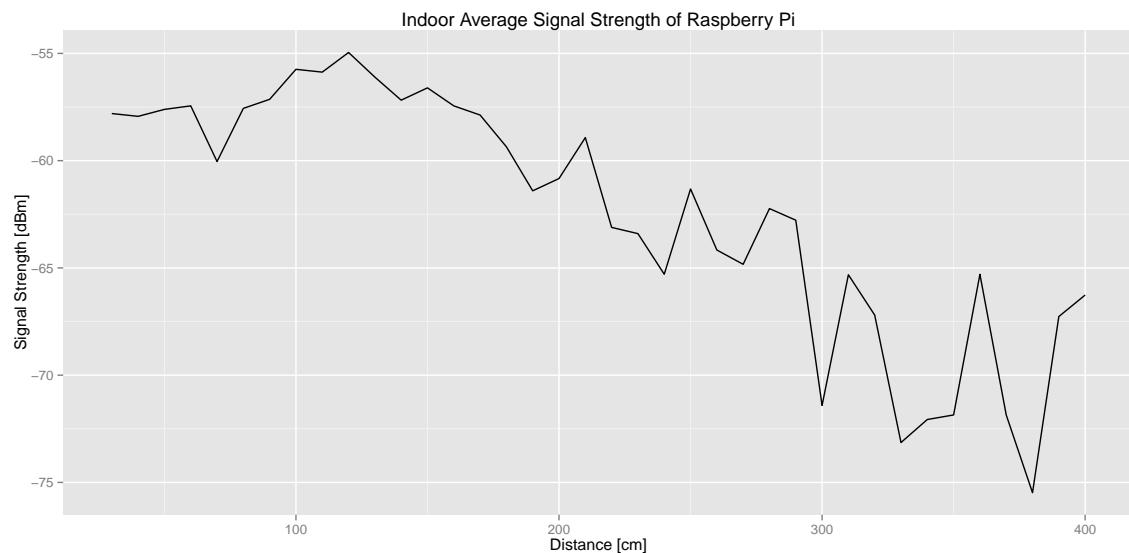


Abbildung 3.7: Durchschnittliche Signalstärke eines Raspberry Pi Beacons

zu stark unterscheiden, von den *kontakt.io* Beacons jedoch deutlich mehr Exemplare verfügbar sind und diese deutlich variabler im Bezug auf die Positionierung der Beacons sind.

3.5 Mögliche Störfaktoren

Wie die obigen Messungen zeigen, weichen die realen Ergebnisse stark von den, durch die physikalischen Ausbreitungseigenschaften der elektromagnetischen Wellen, angenommenen Ergebnissen ab. Dies hängt vor allem damit zusammen, dass die Antenne der Be-

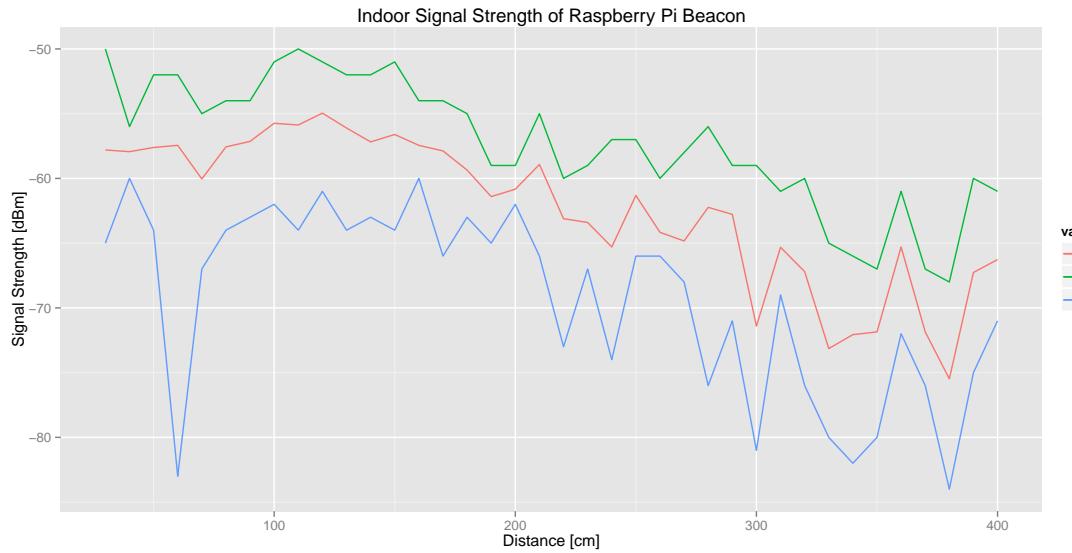


Abbildung 3.8: Minimale, maximale und durchschnittliche Signalstärke des Raspberry Pi gemessen vom iPhone 5

acons nicht gerichtet ist, sondern in alle Richtungen sendet. Dies führt dazu, dass das Signal der Beacons von diversen Flächen im Raum reflektiert und so nicht auf direktem Weg zum Endgerät gelangt.

Ein weiterer wichtiger Störfaktor ist der Benutzer selbst, da der menschliche Körper größtenteils aus Wasser besteht, welche elektromagnetische Wellen abschirmt. Daher ist zu beobachten, dass die Ausrichtung des Nutzers einen deutlichen Einfluss auf die Signalstärke nimmt. In Abbildung 3.9 ist diese Auswirkung des Körpers deutlich zu erkennen. Hierbei wurden jeweils aus zwei Metern Entfernung 100 Stichproben der Signalstärke genommen. Einmal mit freier Sicht zum Beacon und einmal mit dem Körper zwischen Beacon und iPhone.

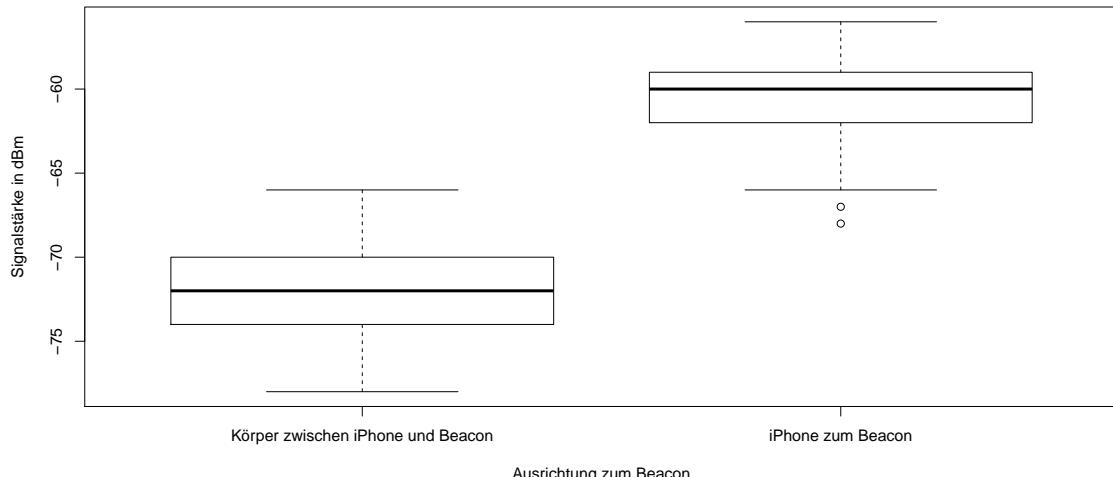


Abbildung 3.9: Signalstärke bei 2m Entfernung zum Beacon

Auf der Abbildung ist deutlich zu erkennen wie der Körper die Signalstärke verringert. Dieser Faktor muss also in die Positionsbestimmung einbezogen werden.

Zusätzlich zum Körper kann es an realen Einsatzorten weitere Gegenstände geben, welche das Signal abschirmen beziehungsweise abschwächen, wie zum Beispiel Wände oder Möbelstücke.

4 Umsetzung und Implementation

4.1 Initialisierung und Beacon-Daten

Für die Implementierung unter iOS müssen zunächst einige grundlegende Programmbestandteile erzeugt und eingerichtet werden. Wie bereits in Kaptiel 2.2 gezeigt, gibt es bei der Erstellung eines neuen Projektes in Xcode verschiedene Vorlage, aus welchen gewählt werden kann. Für diese Applikation wurde die Vorlage der *Master-Detail Application* gewählt, da diese eine CoreData-Unterstützung mit sich bringt. Ausserdem lassen sich über die Master-Detail Applikation die bereits gemessenen Werte in einem Tabellen View anzeigen und zusätzlich, bei Klick auf die Tabellenzelle, weitere Informationen zu der Daten ausgeben.

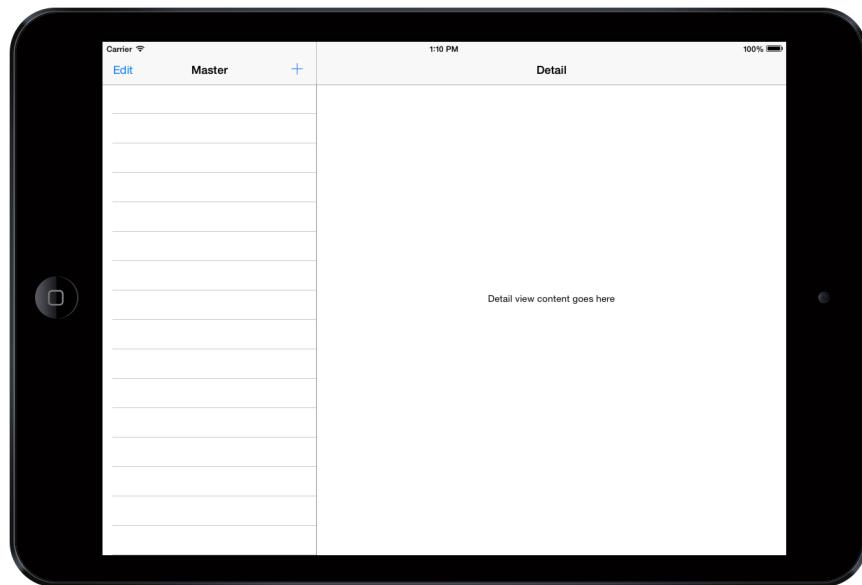


Abbildung 4.1: Beispiel einer Master-Detail Applikation auf dem iPad

Durch die Verwendung dieses Templates werden die benötigten Klassen, das Datenmodel und die Storyboards direkt generiert. Der *NSManagedObjectContext* des Datenmodels wird in der *AppDelegate* erzeugt und dort an die jeweiligen ViewController weitergegeben, sodass der Zugriff auf das Datenmodell in der gesamten Applikation möglich ist. Alle selbstständig generierten Klassen und Dateien lassen sich selbstverständlich, den eigenen Bedürfnissen nach, verändern und anpassen.

Die geplante Applikation soll mehrere Funktionen abdecken, im Genaueren soll sie es ermöglichen Fingerprints zu sammeln, die gesammelten Fingerprints beziehungsweise Informationen über die Fingerprints anzeigen und eine Positionierung des Geräte im aktuellen Raum ermöglichen.

Dahingehend muss das Storyboard, welches das User Interface repräsentiert dementsprechend angepasst werden. Dazu kommt ein Tab Bar Controller zum Einsatz, welcher es ermöglicht, mittels einer Tab Bar im unteren Bereich des Bildschirms, zwischen verschiedenen View Controllern auszuwählen. Dies ermöglicht einen schnellen Wechsel zwischen den ViewControllern für das Sammeln der Fingerprints, für die Ausgabe der Informationen über die Fingerprints und für das Anzeigen der aktuellen Position.

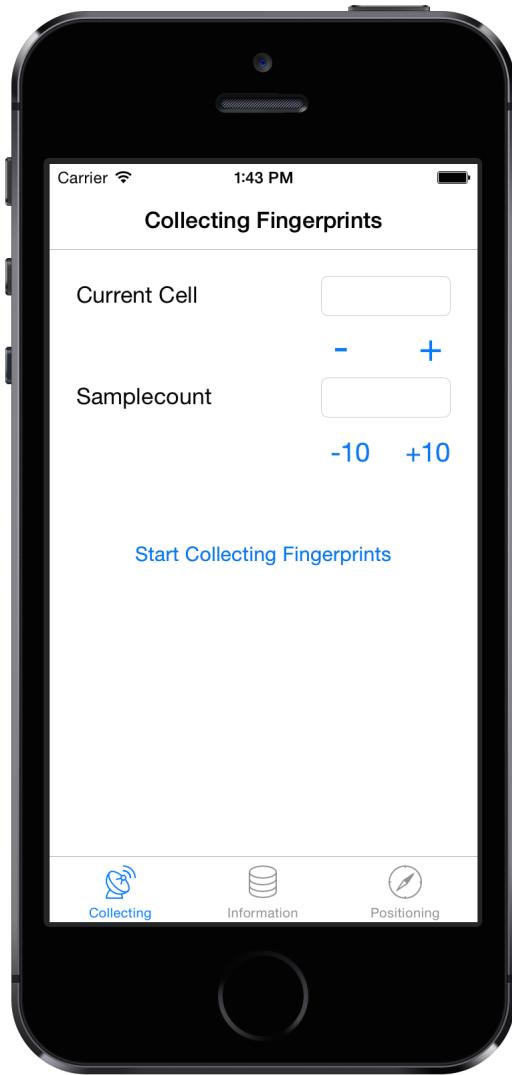


Abbildung 4.2: Beispiel eines TabView Controller auf dem iPhone 5

Um ein Fingerprinting zu ermöglichen, ist es zunächst wichtig eine geeignete Datenstruktur zur Speicherung der Fingerprints zu haben. Hierfür wird das von Apple bereitgestellte CoreData-Framework genutzt, welches einen einfachen Zugriff und Speicherung der Daten ermöglicht. Dazu muss zunächst ein Datenmodell angelegt werden, welches die Entitäten, ihre Attribute und die Beziehungen zwischen den Entitäten beschreibt. Durch die Nutzung des Master-Detail-Templates wurde schon ein Modell erstellt, welches jedoch nur aus einer Entität besteht. Das Modell muss daher für das Figerprinting erweitert werden.

Zunächst werden daher die wichtigsten Eigenschaften eines Fingerprints bestimmt. Diese bestehen aus der aktuellen Position, der empfangenden Signalstärke und dem sendenden Beacon. Daraus lassen sich die grundlegenden Entitäten des Datenmodells bestimmen: **Beacon**, **Zelle** und **Fingerprint**. Des Weiteren ist der Zeitpunkt der Messung ebenfalls relevant und sollte gespeichert werden. Da eine Messung zu einem Zeitpunkt stattfindet und mehrere Fingerprints enthält bietet es sich an, für eine Messung eine eigene Entität zu erstellen, welche einen Zeitstempel enthält und eine Beziehung zu den Fingerprints der Messung besitzt. Dieses Modell lässt sich durch den in Xcode integrierten CoreData-Modell-Editor einfach erstellen und das fertige Modell ist in Abbildung 4.3 zu sehen.

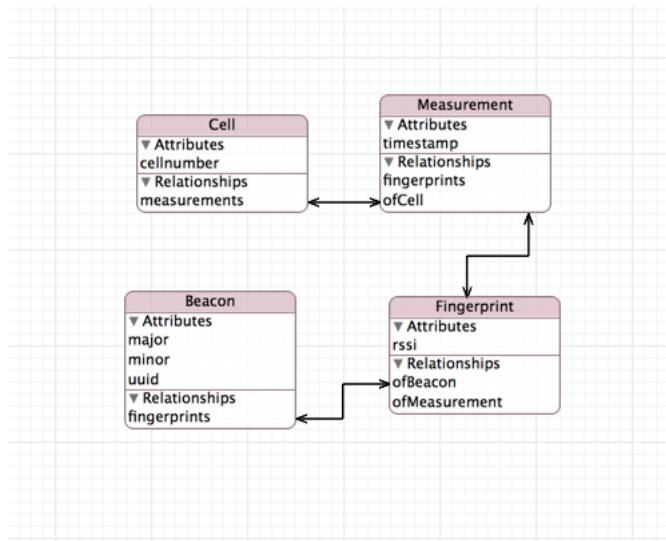


Abbildung 4.3: CoreData-Modell im grafischen Editor

Die Beziehungen zwischen den einzelnen Entitäten lassen sich hier ebenfalls erstellen und bearbeiten.

Nachdem dieses Modell erstellt wurde, ist es zusätzlich möglich, für jede Entität eine eigene Klasse zu generieren. Dieses vereinfacht die Handhabung und den Zugriff auf die Attribute, da nicht mit einem generischen `NSManagedObject` gearbeitet werden muss. Die generierten Klassen enthalten alle Attribute der Entitäten in Form von Properties. Bei *To-Many* Beziehungen zu anderen Entitäten werden zusätzlich Methoden zum Hinzufügen und Entfernen dieser Entitäten erstellt.

Sammeln der Fingerprints Die erste Aufgabe der Applikation ist das Sammeln der aktuellen Fingerprints an einem festgelegten Ort. Dieser Zweig der Applikation setzt sich aus zwei ViewControllern zusammen. Zunächst wird ein ViewController für die Konfiguration der aktuellen Zelle benötigt. In diesem lassen sich etwa der Ort oder die gewünschte Anzahl an Fingerprints bestimmen. Außerdem ist es möglich weitere Einstellungsmöglichkeiten bereitzustellen, wie etwa die manuelle Festlegung des zu suchenden UUID oder Major-Wertes.

Nachdem alle benötigten Daten für die Konfiguration eingegeben wurden, sollen nun die Fingerprints gesammelt werden.

```

1  @interface Cell : NSManagedObject
2
3  @property (nonatomic, retain) NSNumber * cellnumber;
4  @property (nonatomic, retain) NSSet *measurements;
5  @end
6
7  @interface Cell (CoreDataGeneratedAccessors)
8
9  - (void)addMeasurementsObject:(Measurement *)value;
10 - (void)removeMeasurementsObject:(Measurement *)value;
11 - (void)addMeasurements:(NSSet *)values;
12 - (void)removeMeasurements:(NSSet *)values;
13
14 @end

```

Listing 3: Generierte Klasse für die Zelle im CoreData-Modells

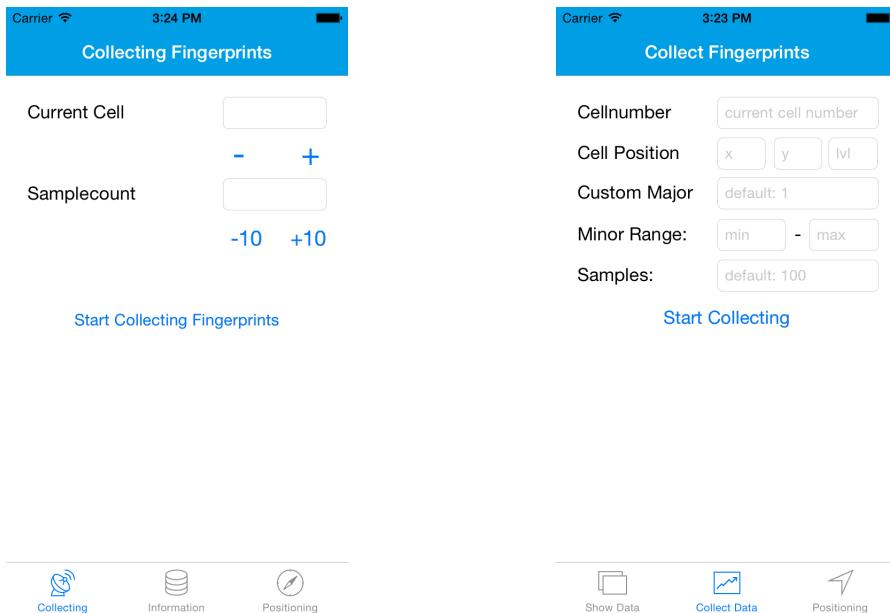


Abbildung 4.4: ViewController für einfache Konfiguration.

Abbildung 4.5: ViewController mit mehreren Konfigurationsmöglichkeiten.

Abbildung 4.6: Mögliche ViewController für die Konfiguration des Fingerprinting

Dazu muss ein weiterer ViewController angelegt werden. In diesem wird ein *CLLocationManager*-Objekt angelegt, welcher für den Empfang der Beacon-Daten zuständig ist. Nachdem dieser erstellt und initialisiert wurde, wird der ViewController als *delegate* der CLLocationManagers gesetzt, da dies nötig ist, um die eigene Methoden bei der Erkennung von Beacons auszuführen. Ausserdem muss die zu suchende Region spezifiziert werden. Dazu wird ein *CLBeaconRegion* Objekt erstellt, welcher die Informationen einer Region enthält. Dazu zählen zum Beispiel der UUID und der Identifier der Region. Dies sind die beiden zwingend notwendigen Angaben. Darüberhinaus lässt sich zudem

der Major-Wert festlegen, nach dem gesucht werden soll. Eine weitere Möglichkeit ist es nur nach Beacons mit bestimmtem UUID, Major und Minor-Wert zu suchen. Hier wird die CLBeaconRegion nur mittels UUID und Identifier initialisiert.

Da der ViewController nun die *delegate* ist, lässt sich die Methode, welche nach Beacons sucht, im ViewController implementieren. Die *didRangeBeacons:inRegion* Methode des CLLocationManager, welche für die Suche nach Beacons zuständig ist, bekommt dabei ein *NSArray* mit allen gefundenen Beacons zurückgegeben.

Die Suche nach den Beacons wird dabei mit der *startRangingBeaconsInRegion:* Methode des CLLocationManagers gestartet. Nach dem Start, erhält die *didRangeBeacons:inRegion* Methode periodisch ein Array der gefundenen Beacons auf welche die Spezifikationen der CLBeaconRegion zutreffen. Das Array beinhaltet dabei *CLBeacon* Objekte, welche ein gefundenes Beacon repräsentieren. Für das Fingerprinting müssen nun die gefundenen Beacons in das Datenmodell gespeichert werden. Dies geschieht über eine selbstgeschriebene Methode, welche über alle gefundenen Beacons iteriert und diese, unter Berücksichtigung der aktuellen Zelle, zum Datenmodell hinzufügt. Während der Aufzeichnung der Fingerprints wird eine Fortschrittsanzeige und die aktuelle Anzahl der in Reichweite liegenden Beacons angezeigt, wie auf Abbildung 4.7 zu erkennen.

Ausgabe der Fingerprints Der vom Template generierte Master-Detail-ViewController wird verwendet um die gesammelten Fingerprints zu verwalten. Dabei wird der Tabellen-View (Master) genutzt um die bisher aufgezeichneten Zellen anzuzeigen. Bei einem Klick auf die jeweilige Tabellenspalte der Zelle lassen sich zusätzliche Informationen ausgeben. Außerdem wurde eine Übertragung der gesammelten Daten an einen Server implementiert. Dazu werden die Daten in das json-Format überführt und dann an einen lokalen Server geschickt, welcher mittels *Javascript* programmiert wurde. Dieser Server empfängt die Datei und speichert sie anschließend auf die Festplatte. Dies ermöglicht eine einfache Auswertung der gesammelten Daten.

Positionsbestimmung Für die Bestimmung der aktuellen Position ist die aktuelle Zelle der wichtigste Wert. Die Ausgabe einer Karte mit der aktuellen Position ist zwar hilfreich für eine praktische Anwendung, für das Testen jedoch nicht notwendig. Für die Positionsbestimmung werden, wie schon beim Sammeln der Fingerprints, zunächst aktuelle Werte der Beacons benötigt. Die Konfiguration und Initialisierung des *CLLocationManager* und der *CLBeaconRegion* ist dabei identisch zu der Vorgehensweise des Fingerprint ViewControllers.

Statt die erhaltenen Beacon-Daten jedoch in die Datenbank zu übernehmen, werden diese direkt weiterverarbeitet. Dazu werden verschiedene Positionierungsalgorithmen angewandt, auf welche im Kapitel 4.4.2 genauer eingegangen wird. Diese Algorithmen liefern nun die, nach den Berechnungen, am nächsten liegende Zelle zurück. Der ViewController kümmert sich dann um die Ausgabe der Zellenummer auf dem Bildschirm.

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.

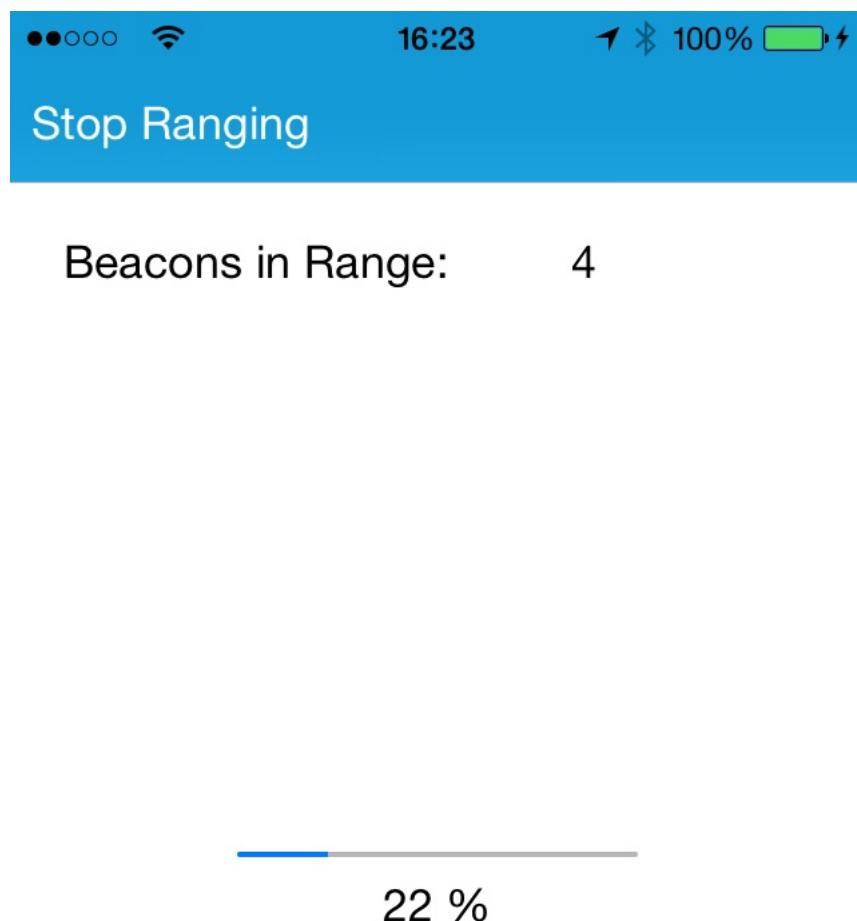


Abbildung 4.7: Anzeige während des Sammelns von Fingerprints

In Listing 4 wird die Initialisierung eines LocationManager gezeigt, welcher eine Genauigkeit von einem Kilometer haben soll und bei Positionsänderungen von mehr als 500 Metern aktualisiert wird.

```

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 4: Beispielinitialisierung für einen LocationManager.

4.2 Ansatz zur Positionsbestimmung

Bei der Positionsbestimmung geht es um die Bestimmung des aktuellen Ortes in Echtzeit und das auf bis zu 10cm genau. Bei der Positionsangabe handelt es sich hier um eine zweidimensionale Position, da dies für unsere Zwecke ausreicht.

Bei der Positionsbestimmung wurden zwei verschiedene Ansätze untersucht. Zum einen die Trilateration, welche eine Positionierung mittels Entferungen zu verschiedenen Fixpunkten ermöglicht und zum Anderen die Positionierung mittels Fingerprinting, welches eine Datenbank mit sogenannten Fingerprints, also vorher aufgezeichneten Messwerten und damit verbundenen Positionsdaten, voraussetzt und darüber die aktuelle Position bestimmt.

Die Positionsbestimmung soll dabei in einem 2D-Raum erfolgen, da die Höhe vernachlässigt werden kann. In der realen Welt kann die Höhe ebenfalls vernachlässigt werden, da dort Stockwerke meist einen deutlichen Höhenunterschied aufweisen, sodass dieser über andere Faktoren eindeutig bestimmt werden kann.

4.3 Trilateration

Die Trilateration ist eine Methode zur Bestimmung der aktuellen Position. Im Gegensatz zur Triangulation, welche die Position anhand der Winkelgrößen zwischen verschiedenen Fixpunkten bestimmt, wird bei der Trilateration die Position durch die Abstände zu den Fixpunkten bestimmt.

Bei der Trilateration wird der Abstand zu einem Fixpunkt genutzt, um die Position des Objektes zu bestimmen. Das Objekt muss dabei auf einer Kreisbahn um den Fixpunkt liegen, welche den Radius des zuvor bestimmten Abstand besitzt. Um nun einen

genauen Standpunkt zu bestimmen, sind mindestens drei Fixpunkte und die dazugehörige Abstände nötig, da so im zweidimensionalem Raum ein eindeutiger Schnittpunkt entsteht.

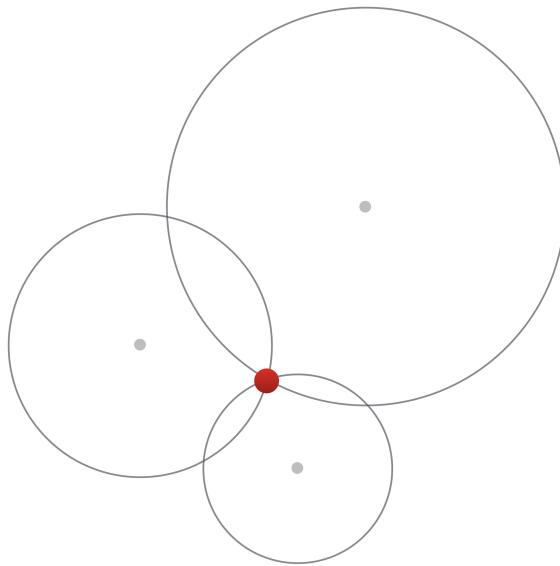


Abbildung 4.8: Funktionsprinzip der Trilateration

In Abbildung ?? sieht man dabei die Funktionsweise der Trilateration bei genauer Abstandsbestimmung. In realen Messungen und Positionsbestimmungen ist es jedoch nicht möglich genaue Abstände zu bestimmen, da es immer zu Messungenauigkeiten kommen kann. Bei solchen ungenauen Messungen ist es nun nicht mehr möglich einen genauen Schnittpunkt zu finden.

Um diese Ungenauigkeit auszugleichen wird das Verfahren entsprechend angepasst. Dabei werden Geraden durch die Schnittpunkte der einzelnen Umkreise gelegt. Dadurch entsteht zwischen den Geraden ein neuer Schnittpunkt, welcher die aktuelle Position repräsentiert. Dieses Verfahren wird in Abbildung 4.9 dargestellt.

Damit ist es möglich auftretende Ungenauigkeiten zu kompensieren und trotzdem eine genaue Positionsbestimmung durchzuführen.

Bei der genutzten iBeacons beziehungsweise Bluetooth-Technologie ist eine genaue Entfernungsgabe jedoch nicht vorgesehen, wodurch das Verfahren der Trilateration nicht direkt angewandt werden kann. Dafür muss zunächst ein Ersatzindikator für die Entfernungsmessung bestimmt werden. Bei der Bluetooth-Technologie bietet sich dafür die Signalstärke an. Dabei wird die Tatsache genutzt, dass die Signalstärke mit zunehmendem Abstand sinkt und man somit aus der aktuellen Signalstärke auch die aktuelle Entfernung bestimmen kann. Das Verhältnis zwischen Entfernung und Signalstärke bei elektromagnetischen Wellen wird durch das Abstandsgesetz beschrieben, welches besagt, dass die Signalstärke quadratisch zum Abstand abnimmt.

$$\text{Signalstärke} = \text{Ausgangssignalstärke} / \text{Entfernung}^2 \quad (4.1)$$

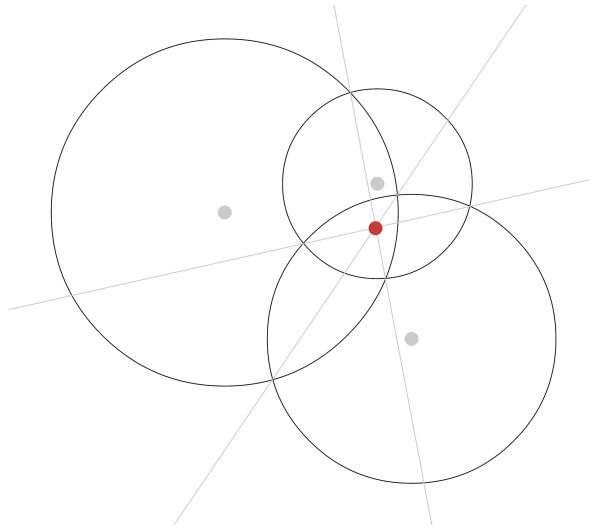


Abbildung 4.9: Trilateration bei ungenauen Abständen zu den Fixpunkten

Diese Annahme mag bei freien Flächen korrekt sein, in Innenräumen kommen jedoch weitere Faktoren hinzu. Durch Wände und Hindernisse im Raum, wie zum Beispiel Schränke, Regale, usw., kommt es dort zu einer Dämpfung des Signals, wodurch die Signalstärke beeinflusst wird. Des Weiteren kann es in Innenräumen auch zu Streuung und Reflexionen kommen, welche das Signal zusätzlich verfälschen.

Diese Annahme bestätigt sich auch bei den Messungen. Diese zeigen, dass die gemessene Signalstärke nicht, wie angenommen, mit der Entfernung stetig abnimmt, sondern sehr stark schwankt, wodurch keine genaue Entfernungsbestimmung durchgeführt werden kann.

Die Methode der Trilateration wurde auf Grund der fehlenden Genauigkeit verworfen.

4.4 Fingerprinting

Das Fingerprinting ist ein Verfahren der Positionsbestimmung auf der Grundlage von zuvor erhobenen Messwerten, den sogenannten Fingerprints. Die Funktionsweise des Fingerprintings unterscheidet sich grundlegend von der Methode der Trilateration, da hierbei keine direkte Berechnung der Position über Entfernungswerte geschieht, sondern die Positionierung über, in einer Datenbank abgelegten, Erfahrungswerten geschieht.

Um dieses Verfahren umzusetzen muss der Messraum, in welchem die Positionierung statt finden soll, zunächst in ein Gitternetz eingeteilt werden, wobei jede Zelle des Gitters eine mögliche Position im Raum repräsentiert. Die Größe dieser Zellen ist prinzipiell frei wählbar, wird jedoch im wesentlichen durch zwei Faktoren bestimmt. Zum einen die gewünschte Genauigkeit. Da jede Zelle eine mögliche Position repräsentiert, wird durch die Größe der Zellen auch die Genauigkeit der Position bestimmt. Daraus ergibt sich, dass die Genauigkeit zunimmt, wenn die Zellengröße verkleinert wird. Der zweite Faktor bei der Wahl der Zellengröße, ist die eindeutige Bestimmung der Zelle. Dies ist darauf

zurückzuführen, dass bei kleineren Zellen die Differenzen zwischen den einzelnen Zellen ebenfalls abnehmen. Um nun eine genaue Bestimmung der Zelle zu ermöglichen, sollte jede Zelle so groß gewählt werden, dass dies noch möglich ist.

Auf Grund dieser zwei Kriterien sollte die Zellengröße so gewählt werden, dass eine gute Unterscheidbarkeit zwischen den einzelnen Zellen gewährleistet ist, hinzu jedoch eine möglichst genaue Positionsbestimmung erzielt werden kann.

Das Fingerprintingverfahren besteht dabei im Wesentlichen aus zwei Phase.

Die erste Phase ist die sogenannte *Trainingsphase* (auch Offline-Phase). Dabei werden die *Fingerprints* gesammelt, welche letztlich zur Positionsbestimmung genutzt werden. In der Trainingsphase werden daher für jede Zelle unseres Messraumes eine Reihe von Fingerprints gesammelt. Die Anzahl der Fingerprints sollte dabei relativ groß sein, sodass Messfehler kompensiert werden können. Ein Fingerprint kann sich dabei aus verschiedenen Daten zusammensetzen. In diesem Fall besteht ein Fingerprint aus der aktuellen Zellennummer beziehungsweise der Zellenkoordinate, einem Zeitstempel mit aktuellem Datum und Uhrzeit und den Signalstärken zu den verschiedenen, in Reichweite befindlichen Sendestationen, welches in diesem Fall die Beacons sind.

Die Sammlung der Fingerprints muss für jede Zelle geschehen und macht die Trainingsphase daher sehr zeitaufwendig.

Die der zweiten Phase, auch *Onlinephase* genannt, werden die zuvor gesammelten Informationen verwendet um die aktuelle Position zu bestimmen. Dafür werden die gesammelten Fingerprints mit den aktuell gemessenen Signalstärken verglichen. Wenn eine Übereinstimmung gefunden wird, wird die Position des passenden Fingerprints als aktuelle Position angenommen.

nfügen
n Zeich-
ng wel-
e Zellen
nd Fin-
erprints
rdeutlicht

4.4.1 Sammlung und Speicherung von Fingerprints

Um eine Positionierung mittels Fingerprinting zu ermöglichen, ist es zunächst nötig einen grundlegenden Datensatz mit Fingerprints zu sammeln, welcher später für die Positionsbestimmung genutzt werden kann. Dazu ist es wichtig zu bestimmen, welche Informationen benötigt werden.

Für jeden Fingerprint muss dabei die aktuelle Position der Messung, das zugehörige Beacon und die aktuelle Signalstärke zwingend vorhanden sein. Zusätzlich ist es sinnvoll den aktuellen Zeitpunkt der Messung zu speichern, um so, wenn nötig, veraltete Fingerprints zu entfernen.

Für das iOS-Programm lag es daher nahe, ein CoreData-Datenmodell anzulegen und die Speicherung der Daten darüber abzuwickeln. Dafür wurden diverse Entitäten angelegt:

Beacon:

Repräsentiert ein Beacon und beinhaltet UUID, Major und Minor-Wert

Cell:

Repräsentiert eine Zelle und beinhaltet die Zellenummer

Fingerprint:

Repräsentiert einen Fingerprint eines Beacons und beinhaltet die gemessene Signalstärke

Measurement:

Repräsentiert eine Messung von Fingerprints und beinhaltet den Zeitstempel des Zeitpunktes der Messung

Untereinander verfügen die Entitäten über diverse Beziehungen, sodass jeder Fingerprint eindeutig einer Zelle und einem Beacon zuzuordnen ist.

4.4.2 Positionsbestimmung

Bei der Positionsbestimmung mittels Fingerprinting gibt es verschiedene Ansätze. Das erste Verfahren vergleicht alle Fingerprints in der Datenbank mit den aktuellen Messwerten und bestimmt damit die aktuelle Position. Eine weitere Möglichkeit besteht darin, den Durchschnittswert der Fingerprints zu bilden um diesen dann mit den aktuellen Werten zu vergleichen. Die letzte untersuchte Möglichkeit ist die der Wahrscheinlichkeitsverteilung der Werte. Hier wird über die Wahrscheinlichkeitswerte der einzelnen Messwerte die aktuelle Position bestimmt.

4.4.3 Einfache Positionsbestimmung mittels Nearest-Neighbor-Verfahren

Algorithmus

Bei der einfachen und naiven Bestimmung der aktuellen Position, werden alle zuvor gesammelten Fingerprints mit den aktuell gemessenen Signalstärken verglichen. Dies führt dazu, dass bei größeren Fingerprint-Datenbanken auch die Rechenzeit und der Energieverbrauch steigt.

Bei dem Vergleich der Messwerte mit den gespeicherten Fingerprints wird das Nearest-Neighbor-Verfahren verwendet. Dabei werden sowohl die aktuellen Messwerte, als auch die Fingerprints als Vektoren aus den Signalstärken zusammengefasst und aus diesen Vektoren wird die jeweilige Entfernung der beiden Werte berechnet. Die einzelnen Signalstärken-Werte sind die Werte von allen in Reichweite befindlichen Beacons.

Bei der Berechnung wird dabei für jeden Fingerprint ein Vektor erzeugt, welcher die Signalstärken zu den in Reichweite befindlichen Beacons beinhaltet. Die Signalstärke für die Beacons wird hier als *FSig* bezeichnet, wobei ein Zusatz angibt auf welches Beacon sich der Wert bezieht, zum Beispiel *FSigB1* für die Signalstärke des Beacons