



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
1.3 Überblick	3
2 Technologien und Werkzeuge	5
2.1 Bluetooth 4.0	5
2.1.1 Bluetooth Low Energy	5
2.1.2 iBeacons	7
2.2 Xcode	9
2.3 iOS	12
2.4 CoreLocation-Framework	14
2.4.1 iBeacons-API	14
2.5 MapBox	15
2.6 CoreData-Framework	18
2.7 Versionsverwaltung mit Git	19
3 Daten und Messungen	21
3.1 Mobile iBeacons	21
3.1.1 estimote Beacon	21
3.1.2 kontakt.io Beacon	22
3.2 Stationäre iBeacons	23
3.2.1 Raspberry Pi als iBeacon	23
3.3 Physikalische Grundlagen	24
3.4 Innenraummessungen	25
3.5 Mögliche Störfaktoren	27
4 Umsetzung und Implementation	29
4.1 Ansatz zur Positionsbestimmung	29
4.1.1 Trilateration	29
4.1.2 Fingerprinting	31
4.2 Erstellung der iOS-Applikation	32
4.2.1 Initialisierung des Projektes	33
4.2.2 Erstellung der Oberfläche	33
4.2.3 Erstellung des CoreData-Modells	36
4.2.4 Implementierung der Oberfläche	38
4.2.5 Implementierung des FingerprintingViewControllers	40
4.2.6 Implementierung des PositioningViewController	41
5 Versuchsergebnisse	51
5.1 Positionierung eines beweglichen Objektes	52

5.2 Statische Positionierung	52
6 Fazit und Ausblick	55
Bibliography	56

Abbildungsverzeichnis

1.1	Smartphoneabsatz in Deutschland	1
2.1	Verbindug zwischen Client und Server bei GATT-Übertragung	7
2.2	Beacon-Daten beim Empfangsgerät	8
2.3	Außenhülle	8
2.4	Chipsatz mit Bluetooth-Modul	8
2.5	Ein iBeacon der Firma "estimote"	8
2.6	Aufbau des estimote-Beacons	9
2.7	Auswahlbildschirm der verschiedenen Templates	10
2.8	Interface Builder für eine iPhone-Applikation	11
2.9	View Controller mit Constraints	11
2.10	Die für die Messungen und Tests genutzten iPhones	13
2.11	Karte in TileMill	16
2.12	Kartenausgabe mittels Mapbox SDK auf dem iPhone	17
2.13	CoreData-Modell in der grafischen Darstellung.	18
2.14	Xcode Versionsverwaltung mit Diff-Anzeige bei einem Commit	20
3.1	Das Developer-Kit von estimote	22
3.2	Kontakt.io Beacon	22
3.3	Messung des iPhone 5	25
3.4	Messung des iPhone 4s	25
3.5	Durchschnittliche Signalstärke eines kontakt.io Beacons	25
3.6	Minimale, maximale und durchschnittliche Signalstärke des Beacons ge- messen vom iPhone 5	26
3.7	Durchschnittliche Signalsteäines Raspberry Pi Beacons	26
3.8	Minimale, maximale und durchschnittliche Signalstrke des Raspberry Pi gemessen vom iPhone 5	27
3.9	Signalstärke bei 2m Entfernung zum Beacon	28
4.1	Funktionsprinzip der Trilateration	30
4.2	Trilateration bei ungenauen Abständen zu den Fixpunkten	30
4.3	Beispiel einer Master-Detail Applikation auf dem iPad	33
4.4	Beispiel eines TabView Controller auf dem iPhone 5	34
4.5	CoreData-Modell im grafischen Editor	37
4.6	Erstellung des FingerprintingSetupViewController	38
4.7	Vergleich zwischen dem arithmetischen Mittel und dem Median	45
4.8	Wahrscheinlichkeitsverteilung von Signalstärke bei einem Beacon	47
5.1	Abbildung der Messraumes	51
5.2	Genauigkeitswerte der statischen Lokalisierung	53

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 Navigationsfunktionen 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 bereits 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.

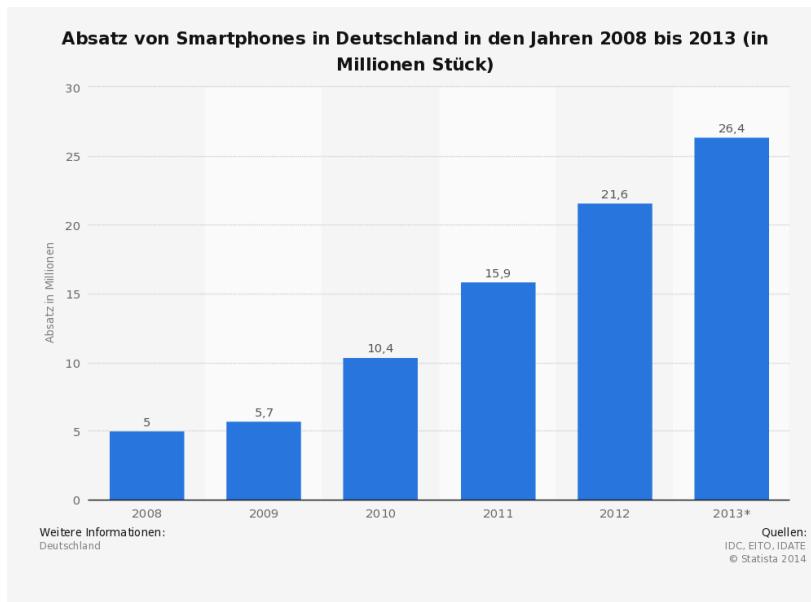


Abbildung 1.1: Smartphoneabsatz in Deutschland

Für die Realisierung der Indoor Positionierung kommen verschiedene Technologien in Frage. Darunter zum Beispiel Wireless LAN, RFID oder Bluetooth (vgl. Mautz (2012)).

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 für eine Existierendes zu kaufen.

Es gibt bereits existierende Indoor Navigations-Lösungen auf dem Markt, zum Beispiel von *AlterGeo* oder *Skyhook Wireless*, welche jedoch auf Wireless LAN basieren. Dies bringt jedoch einige Nachteile mit sich, wie zum Beispiel die Erfordernis eines festen Stromanschlusses. Außerdem sind die bisherigen Systeme nicht sehr genau, so gibt *AlterGeo* eine Genauigkeit von 20-30 Meter an, was den Ansprüchen der Indoor Positionierung nicht gerecht wird.

Daher fiel 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. Außerdem bietet Bluetooth eine gute Signalreichweite, sodass eine großflächige Abdeckung mittels weniger Sendestationen 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. Auch andere Unternehmen haben das Potenzial von iBeacons und Bluetooth erkannt und arbeiten an verschiedenen Einsatzmöglichkeiten für diese Technologie.

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 bestimmt werden, welches Verfahren sich am Besten für die Positionierung in Innenräumen eignet und ob mit der zur Verfügung stehenden Hardware eine ausreichend genaue Positionsbestimmung realisiert werden kann. Außerdem soll die Vergleichbarkeit der Messungen zwischen verschiedenen Endgeräten untersucht werden, also ob diese Werte ohne weiteres übertragbar sind.

Für diese Tests werden eingangs 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 iPad, 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 befinden sich derzeit noch in einem Vorserienstadium.

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

1.3 Überblick

Zu Beginn der Arbeit wird ein Überblick über die verwendeten Hardware- und Softwartechnologien, welche zur Umsetzung der Indoor Positionierung genutzt werden, gegeben. Dabei werden besonders die zugrundeliegende Bluetooth-Technologie als auch das CoreLocation-Framework, welches für die Verarbeitung der Beacon-Daten zuständig ist, genauer betrachtet. Im zweiten Abschnitt werden diverse Messdaten erhoben, welche die Signalausbreitung der Bluetooth-Beacons verdeutlichen. Dabei werden unter anderem die batteriebetriebenen Beacons mit stationären Beacons verglichen. Außerdem wird untersucht in wie weit sich die Empfangsstärken zwischen verschiedenen Endgeräten unterscheiden. Im darauf folgenden Kapitel werden auf die verwendeten Techniken zur Positionierung erörtert. Dabei wird zuerst eine Positionierung mittels Trilateration beschrieben, welche die Position anhand von Entfernungswerten zu den Basisstationen bestimmt. Danach wird die Methode des Fingerprinting näher erläutert, welche die Positionierung mittels zuvor gesammelter Positions- und Signalstärkedaten realisiert. Dabei werden verschiedene Algorithmen zur Bestimmung der Ähnlichkeit zwischen den gesammelten Fingerprintdaten in der Datenbank und den aktuell empfangenen Signalstärkedaten beschrieben. Im letzten Teil wird die Leistungsfähigkeit der verschiedenen Algorithmen untersucht und verglichen. Dabei werden unter anderem die Genauigkeit, als auch die Robustheit der Positionsbestimmung untersucht. Abschließend folgt ein Fazit und ein Ausblick für die Zukunft der iBeacon-Technologie und der Indoor Positionierung.

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 (vgl. SIG (2010)). Außerdem wurde ein neues Protokoll hinzugefügt, Bluetooth Low Energy.

Das erste unterstützte Mobilfunkgerät war das iPhone 4s (vgl. Inc. (zuletzt besucht 20.März 2014i)), 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 (Wiki (zuletzt besucht 20.März 2014)).

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 dabei mit sich bringt, liegen vor allem in der niedrigen Latenz, welche von 100ms auf bis zu weniger als 3ms reduziert wurde, und dem drastisch gesenkten Energieverbrauch im Vergleich zu den Vorgänger-Versionen. Der gesenkter Energieverbrauch hängt dabei mit verschiedenen Faktoren zusammen. Dabei wurde unter anderem die maximale Paketgröße auf 20 Byte festgelegt. Außerdem wurden die Empfangs- und Sendefenster verkleinert. Darüberhinaus arbeitet BLE nach dem Race-to-idle Prinzip. Dabei werden anstehende Daten schnellstmöglich gesendet und das Gerät geht, sobald die Daten verschickt worden sind, direkt in den Schlafmodus bis neue Daten vorliegen.

Des Weiteren wird eine 24-Bit-Fehlerkorrektur eingesetzt, welche die Verbindung unempfindlicher für Störungen und Übertragungsfehler machen soll und unnötige Neuübertragungen 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 (vgl. Decuir (2010)).

Das Bluetooth Low Energy Architektur besteht dabei aus verschiedenen Schichten und Komponenten (Bui (April 2013)).

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 von verbundenen oder allen in der Umgebung befindlichen Geräten. Das Gerät in der Rolle des *Peripheral* hingegen sendet Pakete an die verbundenen oder an alle Geräte.

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. Ein Attribut-Paket des ATT besteht dabei aus drei Bestandteilen. Einem 16 Bit *Handle*, einem 128 Bit *Universally Unique Identifier (UUID)* und einem *Value* mit variabler Länge. Der *Handle* ist eine Nummer, welche das Attribut eindeutig identifiziert, da mehrere Attribute mit gleichem UUID möglich sind. Der *UUID* beschreibt den Typ des gesendeten Attributes, wobei die verschiedenen UUID's nicht im ATT festgelegt werden, sondern in darüberliegenden Profilen wie etwa GATT. Die Länge des *Value's* hängt dabei von dem Typ des gesendeten Attributes ab, lässt sich also in höheren Protokollen aus dem UUID bestimmen.

Das ATT basiert dabei auf einer Client-Server-Architektur, wobei nur der Server Attribute speichert und der Client diese ausliest oder schreibt.

GATT Das *Generic Attribute Profile (GATT)* ist verpflichtend für alle Bluetooth LE-Profile. Es basiert dabei auf ATT und erlaubt eine einfache und geregelte Übertragung der Attribute. 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. Die einzelnen Attribut-Typen unterscheiden sich dabei durch ihre UUID, welche in den GATT-Profilen eindeutig festgelegt sind.

An oberster Stelle steht dabei das *Service*-Attribut. Ein Service besteht aus einer Menge von *Characteristics*. Eine Beispiel für einen GATT-Service ist der *Heart Rate Service*, welcher aus mehreren *Characteristics* besteht, wie zum Beispiel der *Heart Rate Measurement Characteristic*, welche die aktuelle Herzfrequenz beschreibt oder der *Body Sensor Location Characteristic*, welche die Lage des Herzfrequenzmessers beschreibt. 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.

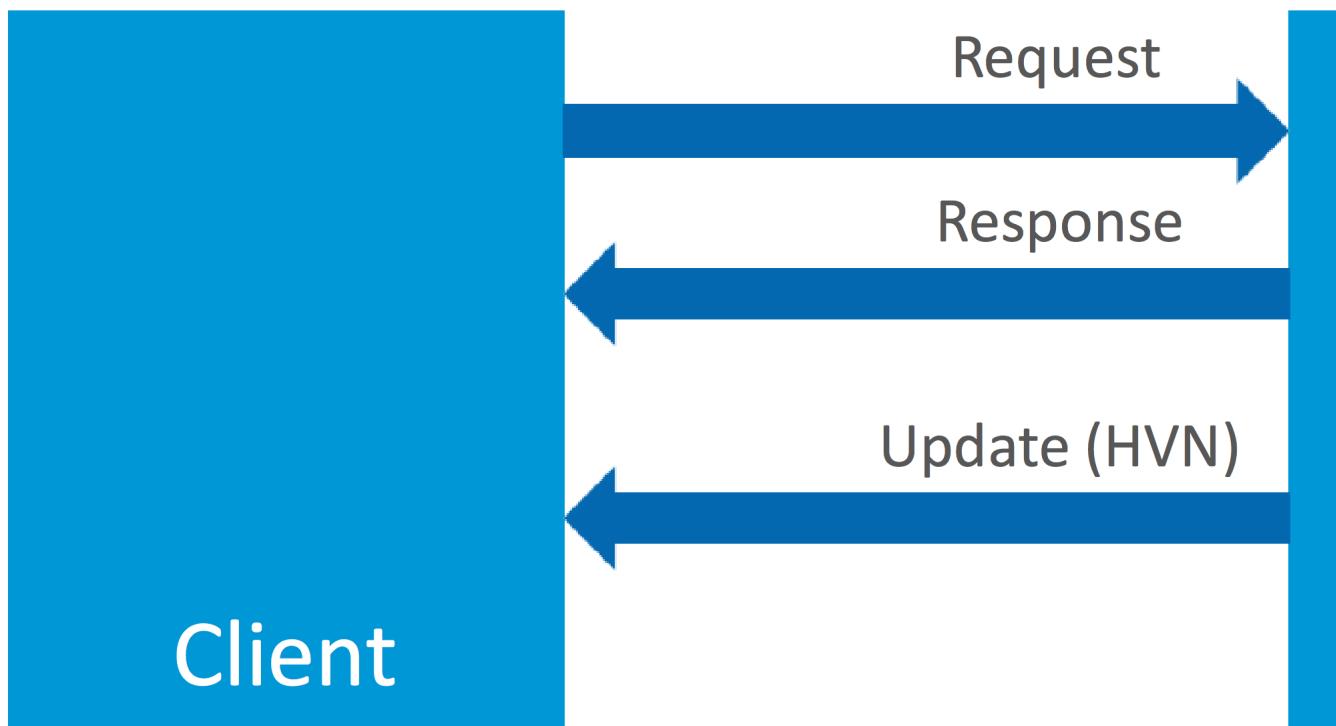


Abbildung 2.1: Verbindung zwischen Client und Server bei GATT-Übertragung

2.1.2 iBeacons

Die iBeacons-Technologie wurde am 10.Juni 2014 von Apple auf der Worldwide Developers Conference vorgestellt (Dilger (Juni 2013)). 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 von Beacon gesendeten Identifikationsdaten kann das Empfangsgerät selbst noch weitere Größen bestimmen. Es ist so zum Beispiel möglich die ungefähre Entfernung zu erhalten. Dafür wird der *Proximity*-Wert genutzt. Dieser definiert vier verschiedene Entfernungs-Zustände : *Far*(mehr als 10m), *Near*(wenige Meter), *Immediate* (wenige Zenitmeter) und *Unknown*(Entfernung konnte nicht bestimmt werden). Diese Werte erlauben eine sehr grobe Entfernungsabschätzung zum Beacon. Für eine differenziertere Entfernungsbestimmung lässt sich eine weitere Kennzahl bestimmen, der *Accuracy*-Wert. Dabei handelt es sich um eine ungefähre Entfernung 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 (Inc. (zuletzt besucht 20.März 2014a)).

	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	Ungefähre Entfernung	1,3
RSSI	Signalstärke in dBm	Signalstärke des empfangenen Signals	-47

Abbildung 2.2: Beacon-Daten beim Empfangsgerät

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 2.5 erkennen. Den Großteil des Beacons nimmt dabei die Batterie ein.



Abbildung 2.3: Außenhülle



Abbildung 2.4: Chipsatz mit Bluetooth-Modul

Abbildung 2.5: Ein iBeacon der Firma "estimote"

Unter genauerer Betrachtung der Platine in Abbildung 2.6, erkennt man, dass diese 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 über die letztendlich die Daten gesendet werden.

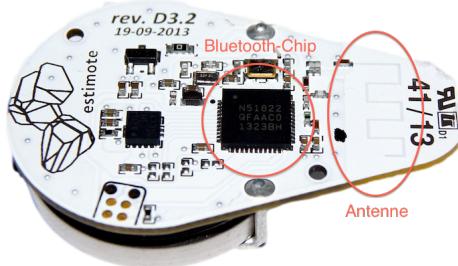


Abbildung 2.6: Aufbau des estimote-Beacons

2.2 Xcode

Xcode ist eine integrierte Entwicklungsumgebung, welche von Apple entwickelt wurde. Xcode ermöglicht es native *iOS* und *OS X* Applikationen zu erstellen, 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, ohne diese auf ein reales Gerät zu übertragen (Inc. zuletzt besuch 20.März 2014)).

Templates

Xcode stellt verschiedene, vorgefertigte iOS-Templates zur Verfügung. Dabei bringt jedes Template eigene Funktionen mit sich. In Abbildung 2.7 lassen sich die verschiedenen Auswahlmöglichkeiten erkennen.

Nach der Auswahl des Templates, werden die dem Template entsprechenden Dateien im Dateiexplorer angelegt.

Dazu gehören beispielsweise die *AppDelegate* und die *Storyboards*.

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.

Interface Builder

Der Interface Builder ist ein in Xcode integrierter, grafischer Editor, welcher es erlaubt die Benutzeroberflächen einer iOS-Applikation zu erstellen. Dabei werden Storyboards genutzt. Ein *Storyboard* ist die Repräsentation der grafischen Oberfläche einer Applikation und besteht dabei aus einem oder mehreren Views.

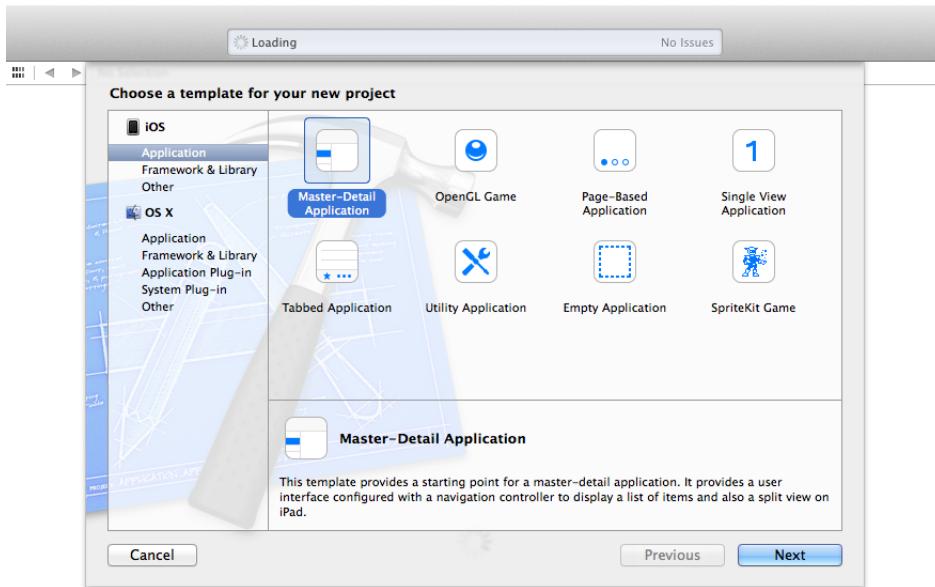


Abbildung 2.7: Auswahlbildschirm der verschiedenen Templates

Die Views lassen sich nach dem *Drag and Drop*-Prinzip zusammenstellen. Dabei stehen weitere Interface-Elemente, wie zum Beispiel Textfelder, Buttons oder Schalter, zur Verfügung.

Wie in Abbildung ?? zu erkennen, besteht das Storyboard aus mehreren Views, die jeweils eine gezeigte Szene auf dem Gerät repräsentieren. Die einzelnen Views sind mit *Segue's* verbunden. Ein Segue, auf deutsch Übergang, steuert dabei den Wechsel zwischen den Views. Dabei wird zum Beispiel festgelegt durch welche Aktion dieser ausgelöst wird und wie dieser abläuft.

Der Interface Builder bietet außerdem noch die Funktion des *Auto Layout*. Dabei werden sogenannte *Constraints* genutzt, welche die Positionsbeziehungen zwischen den einzelnen Elementen des Views festlegen. Diese Constraints erzeugen so ein dynamisches Interface, welches sich an das verwendete Gerät anpasst und so ein User Interface, unabhängig der Bildschirmgröße oder der aktuellen Orientierung des Bildschirms, darstellt. (Inc. (zuletzt besucht 20.März 2014k))

Diese Constraints beschreiben dabei die Abstände und Ausrichtungen der einzelnen Elemente zu dem umschließenden View oder den Elementen innerhalb des Views.

Eine iOS-Applikation kann über mehrere Storyboards verfügen, wodurch es möglich ist, die gleiche Applikation sowohl für das iPhone als auch das iPad anzupassen.

iOS Simulator

Für das Testen und Debuggen der Applikation bringt Xcode einen iOS-Simulator mit sich. Dieser erlaubt das Ausführen einer Applikation auch ohne ein iPhone oder iPad. Der Simulator bietet dabei verschiedene Konfigurationen, sowohl für iPhone (3,5 Zoll und 4 Zoll) als auch für das iPad (retina und non-retina). Im Bezug auf Bluetooth bietet

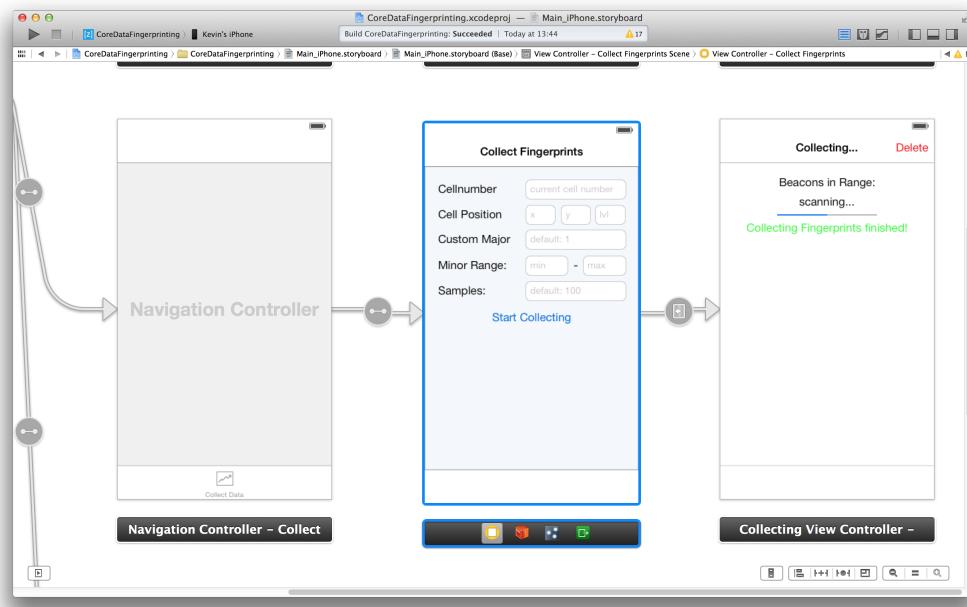


Abbildung 2.8: Interface Builder für eine iPhone-Applikation

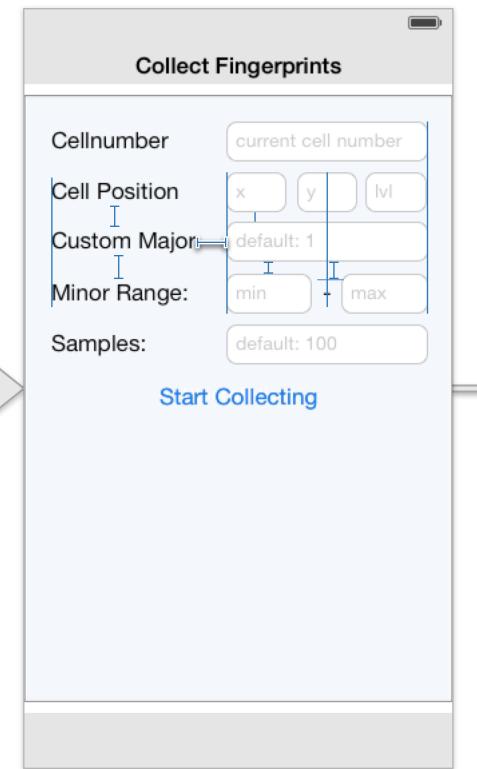


Abbildung 2.9: View Controller mit Constraints

der Simulator jedoch keine entsprechende Unterstützung (Inc. (zuletzt besucht 20.März 2014h)).

2.3 iOS

Für die Entwicklung der Applikation zur Indoor Positionierung war eine der Vorgaben, dass diese für iOS programmiert werden soll. Apple hat dabei für die iOS Programmierung verschiedene Voraussetzungen. Zum einen benötigt man einen Mac, da nur hier die benötigte Software installiert werden kann. Dazu zählt in erster Linie Xcode, welches als Entwicklungsumgebung für iOS-Applikationen genutzt wird (Inc. (zuletzt besucht 20.März 2014g)). Außerdem wird ein iOS-Gerät vorrausgesetzt, da, wie bereits erwähnt, der iOS-Simulator kein Unterstützung für Bluetooth mit sich bringt. Die eingesetzten iOS-Gerät sind ein iPhone 5 und ein iPhone 4s, welche beide mit iOS 7.1 laufen.

Die vorrausgesetzte, minimale iOS-Version ist dabei iOS 7, 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 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 (Inc. (zuletzt besucht 20.März 2014f)). Im Rahmen meiner Bachelorarbeit wurde 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 Messwerte zu vergleichen oder 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 Messwert zwischen einzelnen Geräten übertragbar sind, beziehungsweise wie sich diese zwischen einzelnen Modellgenerationen 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 (iFixit (2014a)). Das iPhone 5 dagegen setzt auf den BCM4334 (iFixit (2014b)), ebenfalls von Broadcom. 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 diese Unterschiede Einfluss auf die Empfangsqualität haben und zu bestimmen, ob eine Übertragung der Messwerte zwischen den Geräten möglich ist oder ob jedes Gerät individuell behandelt werden muss.



Abbildung 2.10: 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, was geschieht beziehungsweise 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.

```

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 /** Java */
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

Der Hauptunterschied liegt in der Methodendeklaration. Statt des *static* Schlüsselwortes wird bei Objective-C eine Klassenmethode mit einem "+" gekennzeichnet, wobei eine 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 eckige Klammern genutzt, wobei der Empfänger der Nachricht zuerst aufgeführt wird und danach die zu sendende Nachricht eingefügt wird (Inc. (zuletzt besucht 20.März 2014j)).

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.

Die Genauigkeit lässt sich dabei nur bei der Positionierung mittels GPS einstellen und ist daher für die Indoor Positionierung nur bedingt geeignet. Es wäre jedoch denkbar, 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. (Inc. (zuletzt besucht 20.März 2014e))

2.4.1 iBeacons-API

Mit 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 (Inc. (zuletzt besucht 20.März 2014a)), welche ein iBeacon repräsentiert und alle zur Verfügung stehenden Informationen enthält und zum Anderen die *CLBeaconRegion*-Klasse

(Inc. (zuletzt besucht 20.März 2014b)), 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 Genaugikeits-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 (Mapbox (zuletzt besucht 20.März 2014b)). 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 (Mapbox (zuletzt besucht 20.März 2014a)) 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 auf der Karte, 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 MapBox-SDK erlaubt es zum Beispiel 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 (siehe Inc. (zuletzt besucht 20.März 2014l)). 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 eine Karte des Innenraumes zeigt. Dieses Datei muss zur weiteren Verwendung in ein von MapBox verständliches Format umgewandelt werden. Dazu wurde ein von "Tom MacWright" (MacWright (2014)) bereitgestellte Python-Script verwendet, welches JPEG Dateien in GeoTIFF Dateien umwandelt. Die GeoTiff-Datei (Ritter and Ruth (2014)) 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 (Mapbox (zuletzt besucht 20.März 2014c)). 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.

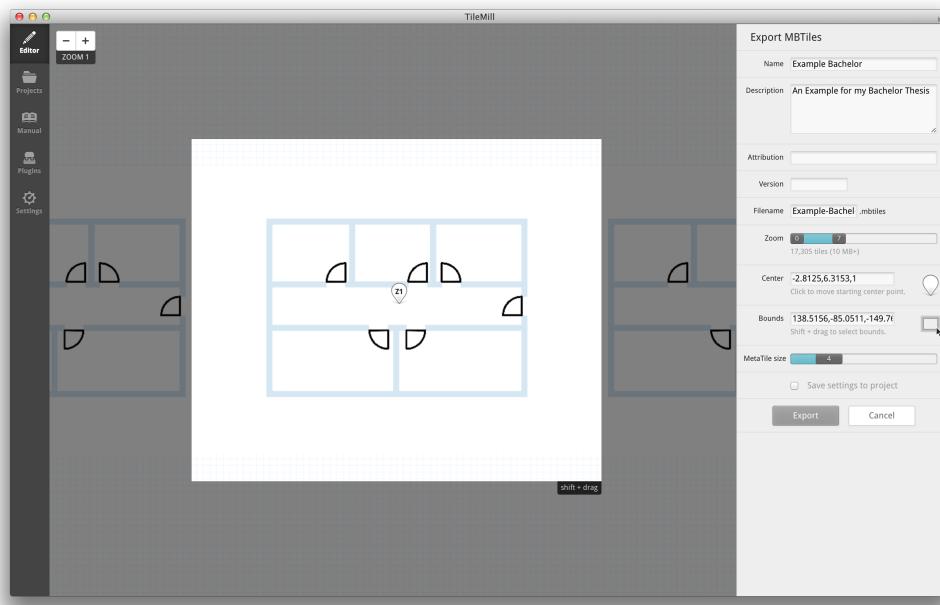


Abbildung 2.11: Karte in TileMill

TileMill erlaubt es nun die eingefügte Karte weiter zu bearbeiten oder 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 gesamte Karte geladen werden muss, sondern nur die aktuell 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 2.12 sieht man die Ausgabe einer Karte auf dem iPhone 5.

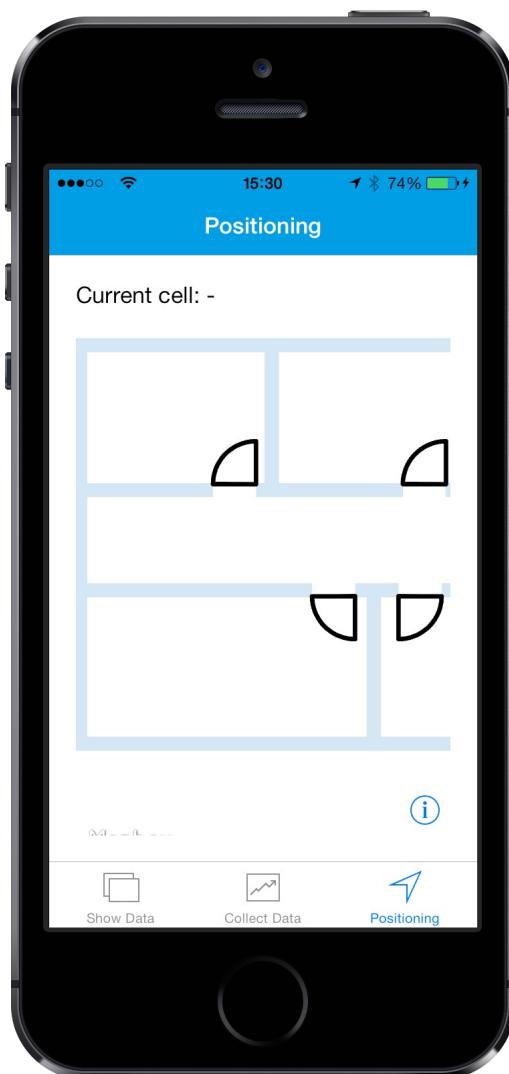


Abbildung 2.12: Kartenausgabe mittels Mapbox SDK auf dem iPhone

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

2.6 CoreData-Framework

Das CoreData-Framework erlaubt die Verwaltung von Datenobjekten und deren Speicherung auf dem Gerät. Dabei kommt ein relationales Datenmodell zum Einsatz, welches sich mittels grafischer Oberfläche in Xcode erstellen lässt. In dem Modell lassen sich sowohl die Entitäten und ihre Attribute einstellen, als auch die Beziehungen zwischen den Entitäten definieren. 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.

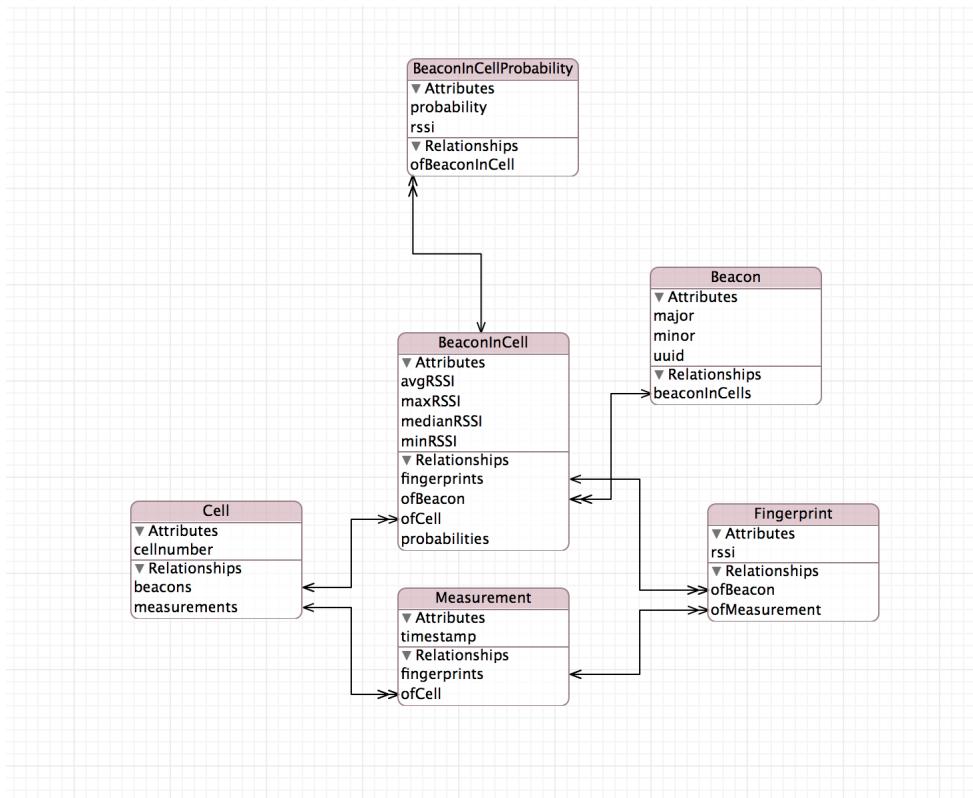


Abbildung 2.13: CoreData-Modell in der grafischen Darstellung.

Nachdem ein CoreData-Modell konfiguriert wurde, benötigt man für den Zugriff auf das Modell ein `NSManagedObjectContext`-Objekt, ein `NSPersistentStoreCoordinator`-Objekt und ein `NSManagedObjectModel`-Objekt. Das `NSManagedObjectContext`-Objekt, verwaltet dabei alle Datenobjekte des Modells. Die Objekte sind dabei vom generischen Typ `NSManagedObject`. Über den `NSManagedObjectContext` werden neue Objekte zum Datenmodell hinzugefügt oder vorhandene Objekte ausgelesen.

Nach dem Anlegen des Datenmodells ist es auch möglich, automatisiert eigene Klassen für die einzelnen Entitäten erzeugen zu lassen. Dadurch werden in den einzelnen Klassen

alle Attribute der Entität entsprechend ihres Typs generiert. Dies erleichtert den Zugriff auf die einzelnen Attribute der Datenobjekte.

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.

```

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] 'meier') AND (salary > %@", minimumSalary];
10 [request setPredicate:predicate];
11
12 NSArray *array = [moc executeFetchRequest:request error:&error];

```

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

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. (Inc. (zuletzt besucht 20.März 2014d))

2.7 Versionsverwaltung mit Git

Für die Verwaltung und Versionierung des Projektes wurde Git verwendet. Als Hosting-Plattform wurde dabei GitHub (GitHub (zuletzt besucht 20.März 2014)) 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 verschiedenen Versionen zu begutachten.

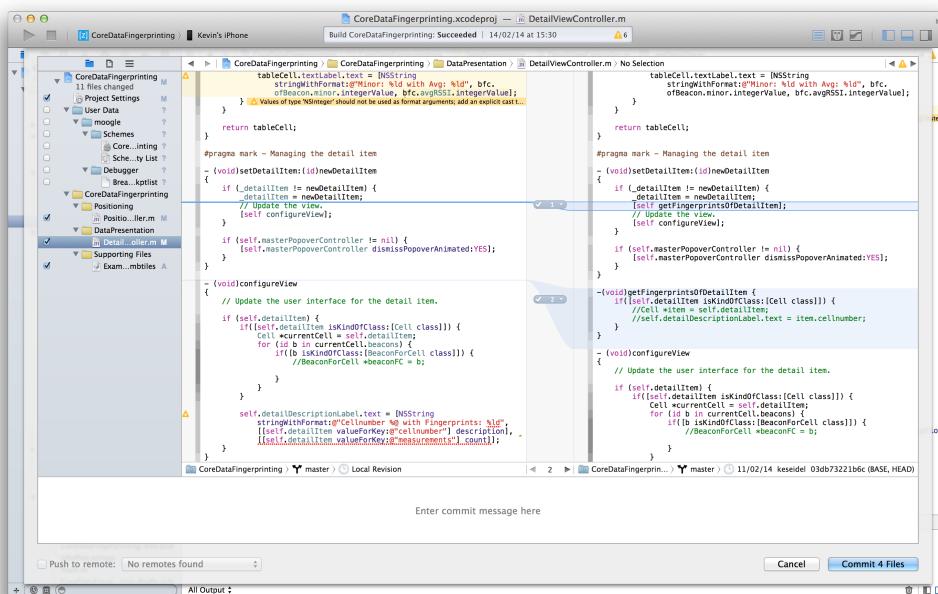


Abbildung 2.14: Xcode Versionsverwaltung mit Diff-Anzeige bei einem Commit

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

Bisher gibt es nur wenige Hersteller von iBeacons, wobei sich der Großteil der Produkte momentan noch in der Entwicklungsphase befindet. Die genutzten iBeacons von *estimote* und *kontakt.io* sind ebenfalls noch in der Entwicklungsphase und wurden hauptsächlich als Testgeräte für Entwickler ausgegeben. 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* (*estimote* (zuletzt besucht 20.März 2014a)) mit Sitz in Polen, war eine der ersten, welche ein funktionstüchtiges iBeacon vorgestellt hat 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 der nRF51822 Bluetooth-Chipsatz von Nordic Semiconductor (Nordic Semiconductor (zuletzt besucht 20.März 2014)), 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. (Allan and Mistry (zuletzt besucht 20.März 2014)) In den *estimote* Beacons wurde zusätzlich noch ein Temperatursensor und ein Accelerometer integriert, welche sich jedoch softwareseitig noch nicht ansprechen lassen.

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 (*estimote* (zuletzt besucht 20.März 2014b)). Dabei bietet es neben den Funktionen der iBeacon-API noch die Funktionalität, sich mit den *estimote* Beacons zu verbinden und deren Konfiguration anzupassen. So erlaubt es zum Beispiel

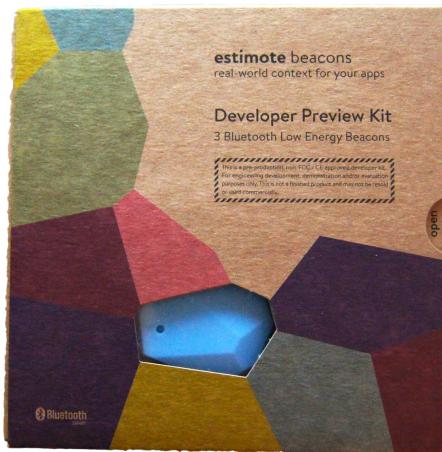


Abbildung 3.1: Das Developer-Kit von estimote

die Signalstärke, das Sendeintervall und die Major-Minor-Informationen zu verändern oder die Firmware der Beacons zu aktualisieren.

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*. (bluegiga (zuletzt besucht 20.März 2014)) Die Beacons verfügen dabei einen ARM Cortex M0 Prozessor mit 256 KB Flashspeicher. Die Ausgangssignalstärke des Chipsatz lässt sich zwischen +4dBm und -20dBm variabel einstellen.

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 eine Funktion zuweisen kann. Auch auf die Verwendung dieses SDK wurde verzichtet, da es nicht dem geplanten Anwendungszweck gerecht wird.

3.2 Stationäre iBeacons

Neben den mobilen iBeacons, welche mittels Batterien betrieben werden, 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. Diese Beacon basieren dabei auf den gleichen Chipsätzen wie auch die 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 dem BCM2835 ARM-Prozessor von Broadcom basiert und als günstiger Computer für Programmieranfänger konzipiert wurde (siehe raspberrypi.org (zuletzt besucht 20. März 2014)). 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 (Nebeker and Young (2014)), welches von mir genutzt wurde, um den Raspberry Pi zu konfigurieren.

Radius Network hat dabei den Inhalt der, von den iBeacons gesendeten, Nachrichten untersucht und die gesendeten Pakete aufgeschlüsselt. Dadurch wurde der Aufbau der Nachrichten, die das iBeacon aussendet, ausgelesen.

3.3 Physikalische Grundlagen

Im Vergleich zu Kabelverbindungen, wo die Ausbreitung des Signals entlang eines bestimmten Leiters geschieht, ist die Ausbreitung eines drahtlosen Signals von deutlich mehr Faktoren abhängig (Heine and Nitschke (zuletzt besucht 20.März 2014)).

Dämpfung

Dämpfung ist die Schwächung der Energie, mit welcher das Signal übertragen wird. Diese tritt auch bei einer freien Sichtverbindung zwischen Sender und Empfänger auf. Die Verringerung der Leistung ist dabei abhängig von der Ausgangssendeleistung P_0 und der Entfernung r zwischen Sender und Empfänger. Die letztlich am Empfänger ankommende Signalstärke P_r berechnet sich dabei wie folgt:

$$P_r = \frac{P_0}{r^2} \quad (3.1)$$

Die Abschwächung des Signals resultiert aus der Art der Aussendung. Da die hier verwendeten Beacons einem Punktstrahler entsprechen, wird die anliegende Signalstärke auf eine Kugeloberfläche verteilt. Die Kugeloberfläche O lässt sich dabei abhängig vom Radius r mittels der Formel $O = 4\pi r^2$ berechnen. Die Oberfläche vergrößert sich damit quadratisch zum Radius.

Abschattung

Die Abschattung ist eine stärkere Form der Dämpfung, welche durch Objekte zwischen der Signalquelle und dem Empfänger verursacht wird. Die Abschattung ist dabei abhängig von der Frequenz des Signals und der Geschaffenheit des Objektes. Bei höherer Frequenz ist eine stärkere Abschattung des Signals gegeben.

Reflexion

An größeren Oberflächen kann es zu Reflexionen des Signals kommen. Dabei trifft es auf die Oberfläche und wird dann in abgeschwächter Form reflektiert.

Streuung

Ähnlich der Reflexion trifft das Signal auf eine Oberfläche, wird dabei jedoch aufgespalten und in verschiedene Richtung abgelenkt.

Mehrwegeausbreitung

Durch die obige genannten Effekte ist es möglich, dass ein Signal einen Empfänger auf mehreren Wegen erreicht. Dabei ist die Signalstärke stark abhängig von dem zurückgelegtem Weg. Außerdem unterscheiden sich die Laufzeiten der Signale, da diese ebenfalls vom Weg abhängig sind. Dadurch kann es auch zu Überlagerungen zwischen Signalen kommen, welche zur gleichen Zeit gesendet wurden, jedoch zu unterschiedlichen Zeiten beim Empfänger eintreffen. Durch diese Phasenverschiebung kann das Signal zusätzlich abgeschwächt werden.

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.

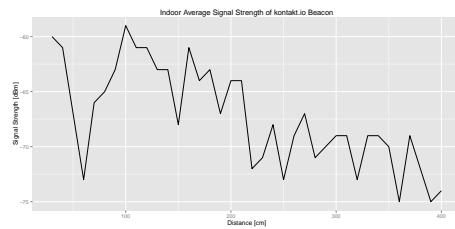


Abbildung 3.3: Messung des iPhone 5

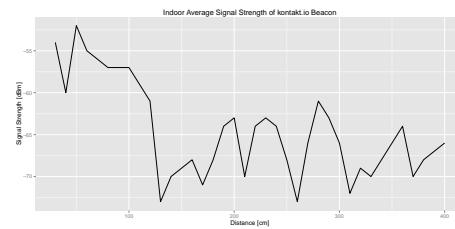


Abbildung 3.4: Messung des iPhone 4s

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 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 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 Signal. 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 Signalstärke 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

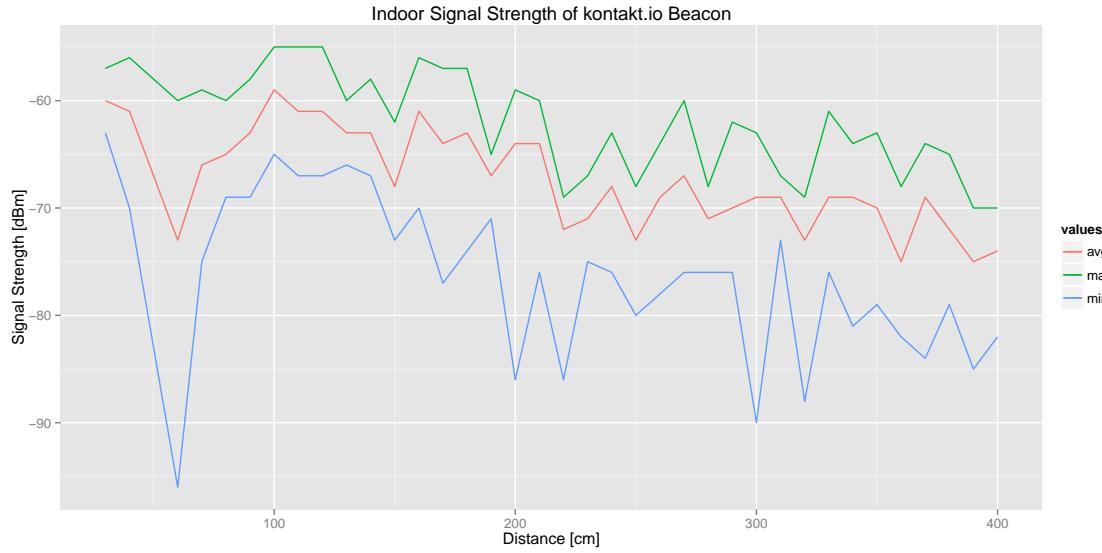


Abbildung 3.6: Minimale, maximale und durchschnittliche Signalstärke des Beacons gemessen vom iPhone 5

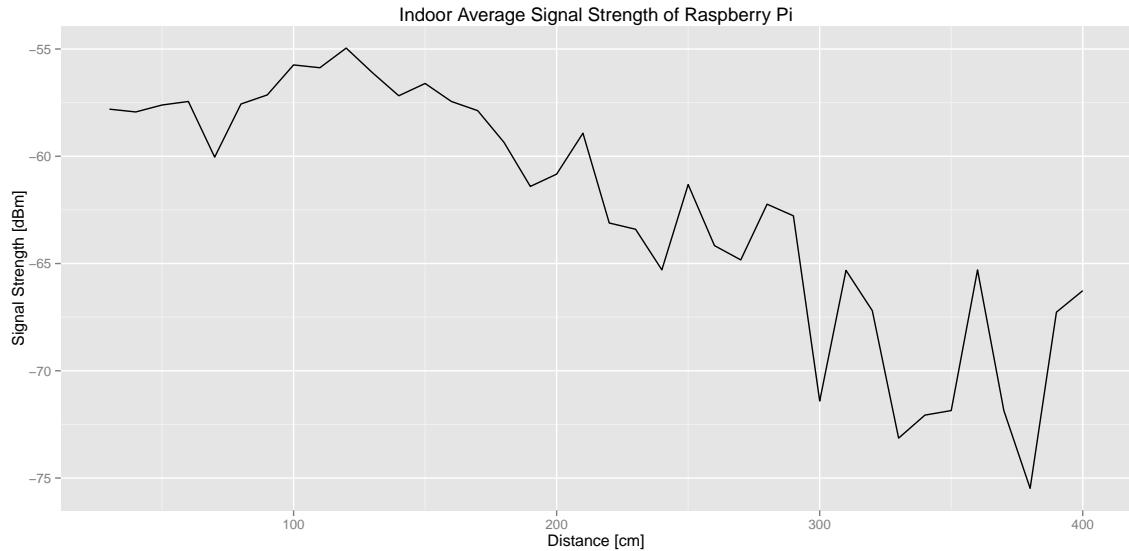


Abbildung 3.7: Durchschnittliche Signalstärke eines Raspberry Pi Beacons

der Signalstärke sein sollte. Es sind jedoch immernoch einige Ausreißer zu erkennen. Bei der Betrachtung der minimalen und maximalen Signalstärken fällt auch auf, dass die ähnlich stark schwanken, wie schon zuvor bei den kontakt.io Beacons. Die Stabilität des Signal des stationären Beacons ist also genauso schwach beziehungsweise noch schwächer als die des batteriebetriebenen Beacons. Dies wird in Abbildung 3.8 nocheinmal deutlich.

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

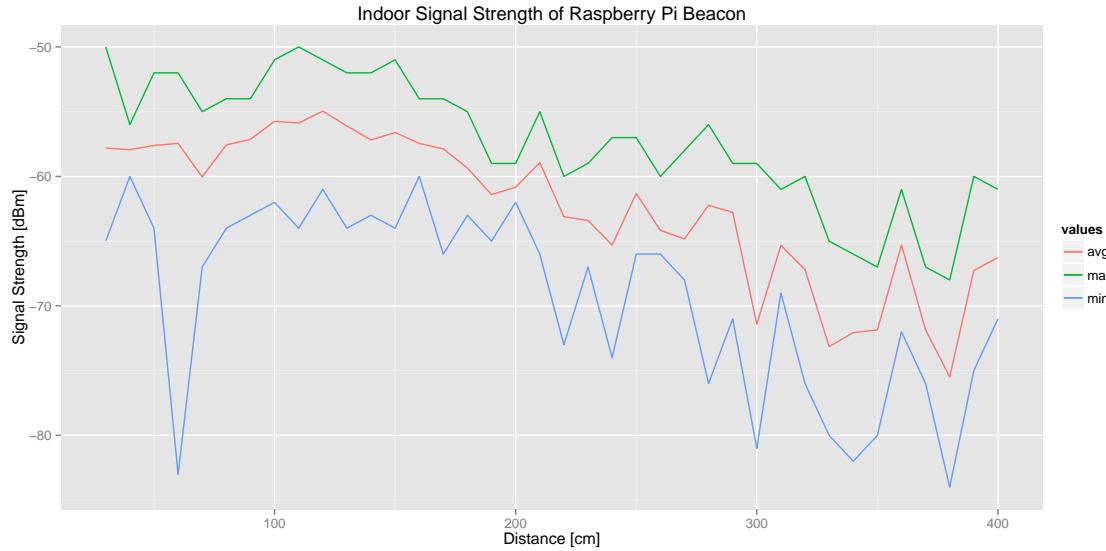


Abbildung 3.8: Minimale, maximale und durchschnittliche Signalstärke des Raspberry Pi gemessen vom iPhone 5

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 Beacons 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.

Auf der Abbildung ist deutlich zu erkennen, wie der Körper die Signalstärke verringert. Dieser Faktor muss also bei der Positionsbestimmung berücksichtigt werden.

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

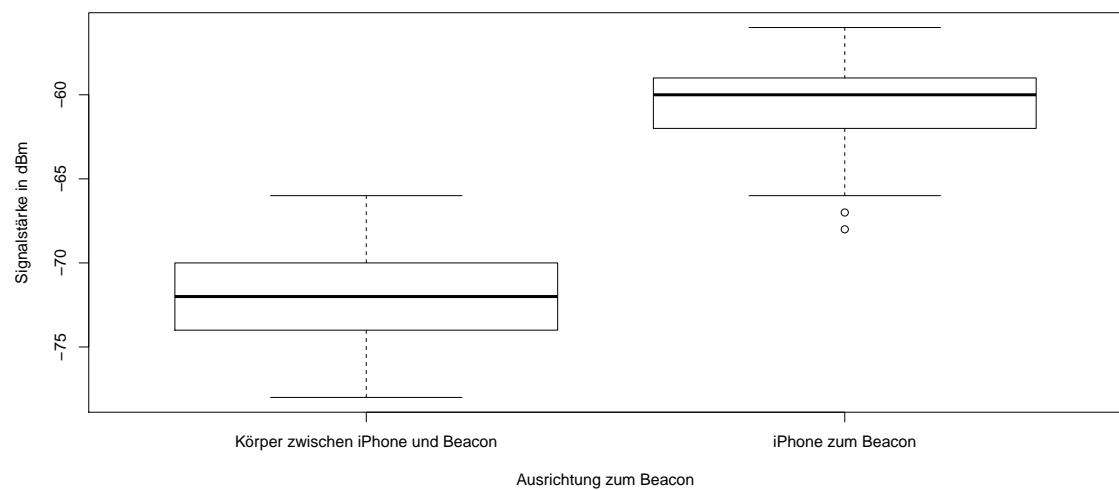


Abbildung 3.9: Signalstärke bei 2m Entfernung zum Beacon

4 Umsetzung und Implementation

4.1 Ansatz zur Positionsbestimmung

Bei der Positionsbestimmung geht es um die Bestimmung des aktuellen Ortes in Echtzeit und das auf bis zu wenige Meter genau. Bei der Positionsangabe handelt es sich hier um eine zweidimensionale Position, da dies für die Indoor-Positionierung ausreichend ist.

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 über diese Daten 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 Höhenunterschied über andere Faktoren eindeutig bestimmt werden kann.

4.1.1 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.

Die Trilateration macht sich dabei zunutze, dass sich ein Objekt, bei gegebenem Abstand zum Fixpunkt, auf einer Kreisbahn um diesen befinden muss. Der Radius des Kreises entspricht dabei dem Abstand. Für eine genaue Positionierung sind dabei mindestens drei Fixpunkte und deren Abstände nötig, da nur so ein eindeutiger Schnittpunkt bestimmen werden kann. (Kushki et al. (2012)).

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

Um diese Ungenauigkeit auszugleichen wird das Verfahren entsprechend angepasst. Dabei werden Geraden durch die Schnittpunkte der einzelnen Umkreise gelegt. Dadurch

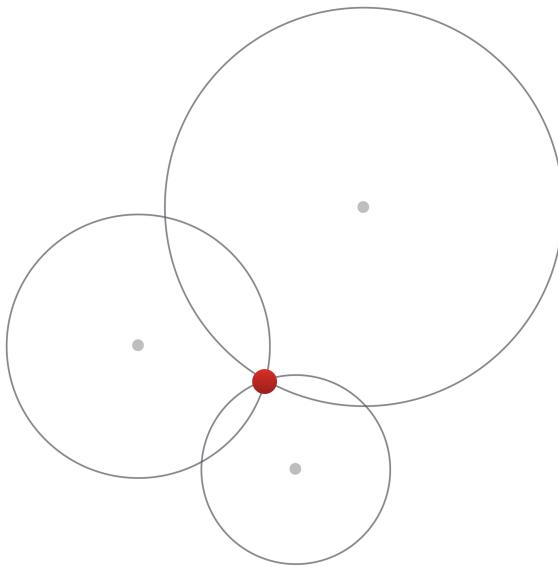


Abbildung 4.1: Funktionsprinzip der Trilateration

entsteht zwischen den Geraden ein neuer Schnittpunkt, welcher die aktuelle Position repräsentiert. Dieses Verfahren wird in Abbildung 4.2 dargestellt.

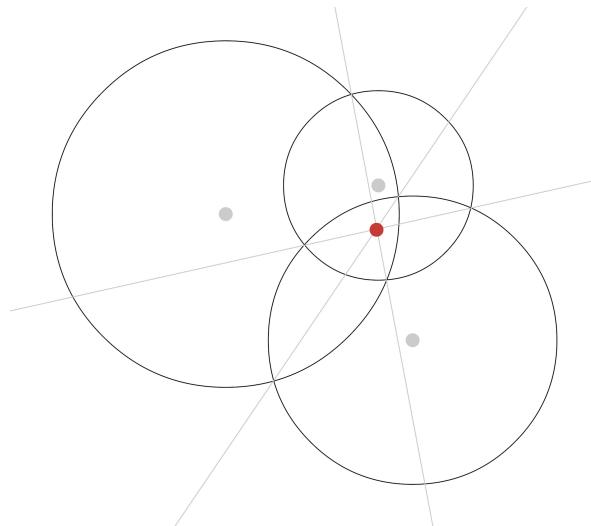


Abbildung 4.2: Trilateration bei ungenauen Abständen zu den Fixpunkten

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.

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

Diese Annahme mag bei freien Flächen korrekt sein, in Innenräumen kommen jedoch weitere Faktoren hinzu, wie in Kapitel 3.5 beschrieben. 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 in Kapitel 3. 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.

4.1.2 Fingerprinting

Das Fingerprinting ist ein Verfahren zur Positionierung auf der Grundlage von gesammelten Signalstärke-Werten.

Für das Fingerprinting werden dabei für verschiedene Positionen im untersuchten Messraum die Signalstärken der vorhandenen Bluetooth-Signale aufgezeichnet. Dadurch entsteht ein charakteristischer Fingerabdruck der Signalstärken an der aktuellen Position (vgl. Navarro et al.).

Bei der Positionierung wird daraufhin der aktuelle Fingerabdruck mit den zuvor aufgezeichneten Abdrücken verglichen, wodurch die aktuelle Position bestimmt werden kann.

Dieses Verfahren ist vor allem für die Positionierung mittels Signalstärken gedacht, da die möglichen Störungen des Signals bereits bei der Aufzeichnung der Fingerprints berücksichtigt werden.

Das Fingerprinting-Verfahren besteht aus zwei Phasen.

Vorbereitungen

Um eine Positionierung mittels Fingerprinting zu erlauben, muss der Messraum, in welchem die Positionierung stattfindet, zunächst in ein Gitternetz unterteilt werden. Dabei repräsentiert jede Zelle dieses Gitternetzes eine mögliche Position im Raum. Die Größe dieser Zellen ist prinzipiell frei wählbar, wird jedoch im Wesentlichen durch zwei Faktoren bestimmt.

Zum einen hängt die Zellengröße von der gewünschten Genauigkeit ab, da jede Zelle eine mögliche Position repräsentiert. Folglich erhöht eine kleinere Zellengröße die Genauigkeit der Positionierung.

Der zweite Faktor bei der Wahl der Zellengröße, ist die Eindeutigkeit einer Zelle. Dies ist darauf zurückzuführen, dass bei kleineren Zellen die Differenzen der Messwerte zwischen den einzelnen Zellen abnehmen. Es ist daher nötig die Zelle ausreichend groß zu wählen, sodass eine gute Unterscheidung zwischen einzelnen Zellen möglich ist.

Aufgrund dieser zwei Kriterien muss die Zellengröße so gewählt werden, dass eine akzeptable Genauigkeit gegeben ist und eine eindeutige Unterscheidung der Zellen möglich ist.

Trainingsphase

Die erste Phase ist die sogenannte *Trainingsphase* (auch Offlinephase). Dabei werden die Fingerprints gesammelt, welche letztlich zur Positionsbestimmung genutzt werden.

Ein Fingerprint besteht dabei aus einem Zeitstempel, der aktuellen Position und den Signalstärken der Sendestationen, in diesem Fall der Beacons, in Reichweite.

In der Trainingsphase werden dabei für jede Zelle eine Reihe von Fingerprints gesammelt. Die Anzahl der zu sammelnden Fingerprints sollte dabei ausreichend groß sein, sodass Messfehler kompensiert werden können.

Da in der Trainingsphase eine Reihe von Fingerprints für jede vorhandene Zelle erhoben werden müssen, ist die Trainingsphase relativ aufwendig.

Onlinephase

In 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 abgeglichen. Für die Positionierung kann dabei auf verschiedene Algorithmen zurückgegriffen werden.

Es wurden hier drei verschiedene Algorithmen untersucht. Die ersten beiden Algorithmen basieren auf dem Nearest-Neighbor-Verfahren. Dabei werden im ersten Algorithmus alle gesammelten Fingerprints mit dem aktuell gemessenen Fingerprint verglichen. Der zweite Algorithmus arbeitet ähnlich, nutzt jedoch die Mittelwerte der jeweiligen Signalstärken in einer Zelle. Der letzte Algorithmus arbeitet über eine Wahrscheinlichkeitsverteilung der Signalstärken.

Durch die Vorteile des Fingerprintings bei der Positionierung mittels Signalstärken und der Robustheit des Verfahrens gegenüber Signalverfälschungen durch statische Objekte, wurde dieses Verfahren für die Indoor Positionierung ausgewählt.

4.2 Erstellung der iOS-Applikation

Die geplante iOS-Applikation soll alle Bereiche des Fingerprintings abdecken. Das heißt sie soll sowohl die Sammlung von Fingerprints an bestimmten Orten ermöglichen, als auch die Lokalisierung des Gerätes innerhalb des Messraumes ermöglichen. Außerdem soll eine Ausgabe der gesammelten Daten ermöglicht werden, sodass man in der App zum Beispiel die Anzahl der gesammelten Fingerprints für jede Zelle auslesen kann. Um

dies zu erreichen muss die iOS-Applikation über mehrere Views verfügen, welche jeweils eine dieser Aufgaben erfüllen.

Für die Navigation zwischen den einzelnen Views kommt dafür ein TabBarController zum Einsatz. Dieser ermöglicht es, mittels einer Leiste am unteren Bildschirmrand, zwischen verschiedenen Views zu wechseln.

4.2.1 Initialisierung des Projektes

Für die Implementierung der iOS-Applikation muss zunächst ein neues Projekt in Xcode erstellt werden. Wie bereits in Kapitel 2.2 erwähnt, gibt es dabei verschiedene Vorlagen, aus welchen gewählt werden kann. Für diese Applikation wurde dies Vorlage der *Master-Detail Application* mit CoreData-Unterstützung gewählt. Dies erzeugt automatisch ein CoreData-Modell und alle erforderlichen Objekte um dieses zu nutzen. Außerdem werden noch weitere Dateien generiert, wie etwa das Storyboard und die AppDelegate. Das Storyboard beinhaltet bereits einen Master-Detail-View, lässt sich jedoch beliebig erweitern und bearbeiten.

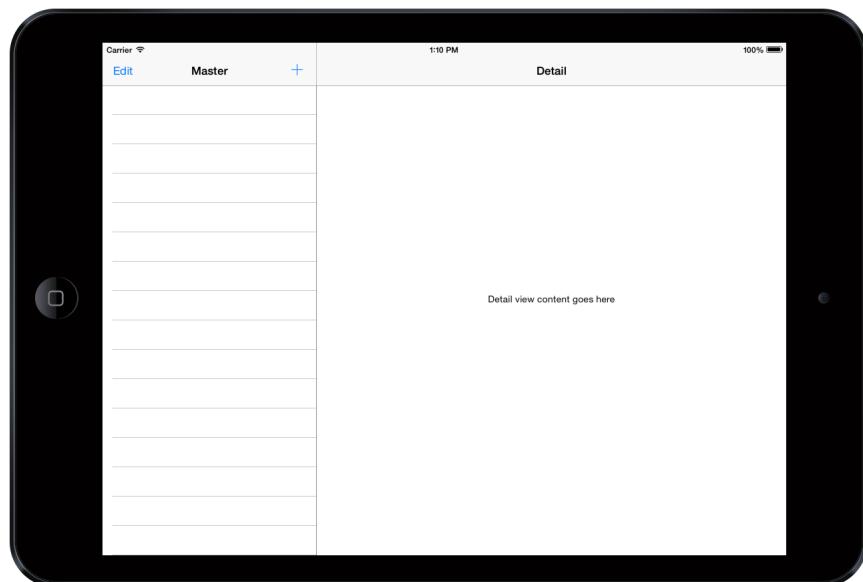


Abbildung 4.3: Beispiel einer Master-Detail Applikation auf dem iPad

Die AppDelegate ist die Kerndatei der Applikation und steuert dabei das Verhalten bei Start und bei Beendigung der Applikation. So werden dort bei Start der Applikation die nötigen Objekte für den Zugriff auf die CoreData-Funktionalitäten erzeugt und an die jeweiligen ViewController der Applikation weitergegeben.

4.2.2 Erstellung der Oberfläche

Mittels des Interface Builders wird die Oberfläche der Applikation angepasst. Dazu wird die Storyboard-Datei für das iPhone editiert. Durch die Nutzung der Master-Detail-Application Vorlage beinhaltet das Storyboard bereits zwei Views, welche in

einen Navigation Controller eingebettet sind. Zum einen den Master View, welcher aus einer Tabelle besteht und dem Detail View, welcher bei Klick auf eine Tabellenzeile weitere Informationen über diese ausgibt. Der Navigation Controller steuert dabei die Übergänge zwischen den beiden Views. Diese Views werden in dieser Applikation behalten und später für die Anzeige der bereits gesammelten Fingerprint-Informationen genutzt.

Als erstes wird nun ein TabBarController zum Storyboard hinzugefügt. Dieser besitzt dabei bei der Erzeugung bereits zwei Tabs mit jeweils einem leeren View. Diese beiden Views werden später für die Sammlung der Fingerprints und die Positionierung verwendet.

Der bereits vorhandene Master-View wird in den TabBarController integriert, sodass dieser nun aus drei Tabs besteht. Außerdem werden die beiden leeren Views in einen NavController eingebettet.

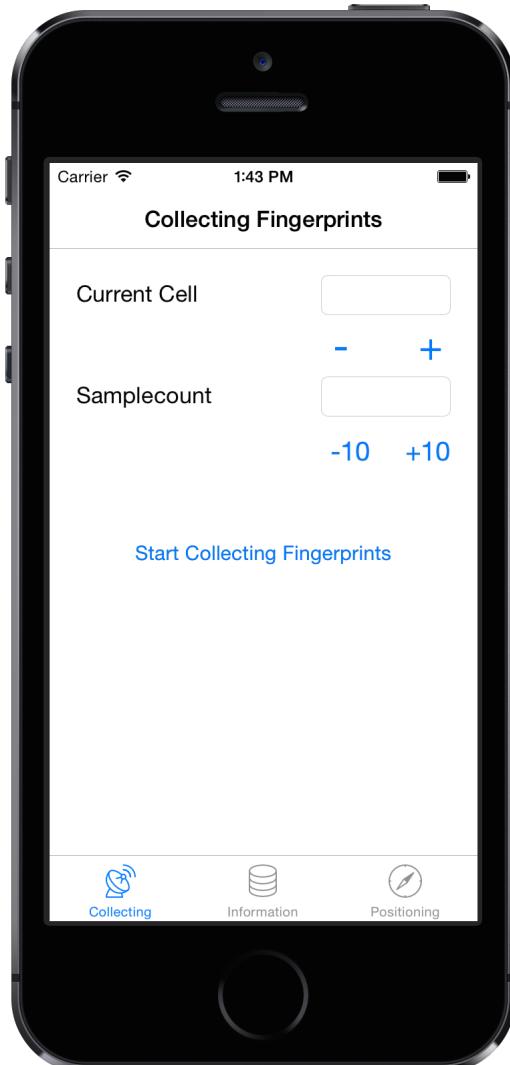


Abbildung 4.4: Beispiel eines TabView Controller auf dem iPhone 5

Nachdem das grundlegende Layout erstellt wurde, werden die einzelnen Views konfiguriert.

FingerprintingSetupView

Der *FingerprintingSetupView* wird genutzt, um die aktuelle Zelle für die Sammlung der Fingerprints zu konfigurieren und um die gewünschte Anzahl von Fingerprints einzustellen. Da die Applikation nur zu Testzwecken eingesetzt werden soll, wird die Erscheinung des View sehr simpel gehalten. Der View besteht dabei lediglich aus zwei Textfeldern und mehreren Button. Ein Textfeld erlaubt die Eingabe der aktuellen Zelle und das andere die Anzahl der zu sammelnden Fingerprints. Außerdem befinden sich unter jedem Feld zwei Buttons, welche es erlauben die Werte innerhalb der Felder zu inkrementieren oder zu dekrementieren.

Außerdem befindet sich in dem View ein weiterer Button, welcher es erlaubt die Sammlung der Fingerprints zu starten. Nach dem Start wird dabei der *FingerprintingView* aufgerufen.

FingerprintingView

Der *FingerprintingView* ist ebenfalls sehr simpel aufgebaut. Er besteht aus einem Label, welches die Anzahl der aktuell gefundenen Beacons anzeigt. Außerdem zeigt er den aktuellen Vorschritt des Sammelsorganges an. Dafür dient zum Einen ein Progress View, welcher mittels einer Statuslinie den aktuellen Vortschritt anzeigt. Zum Anderen liegt darunter ein Label, welches den aktuellen Fortschritt in Prozent anzeigt.

In der NavigationBar des NavController existiert außerdem ein Button, welcher es erlaubt das Sammeln der Fingerprints direkt abzubrechen und damit zum vorherigen View zurückzukehren. Nach Abschluss des Sammelsorgangs soll die Applikation automatisch zum vorherigen View zurückkehren.

PositioningView

Der *PositioningViewController* besteht nur aus Labels, welche die aktuelle Zelle anzeigen. Dabei wird die aktuelle Position für jeden implementierten Algorithmus ausgegeben, sodass sich diese Ergebnisse direkt vergleichen lassen.

InformationView Der InformationView basiert auf dem MasterView, welcher durch die Nutzung des Templates automatisch generiert wurde. Daher besteht dieser bereits aus einem TableView. Dieser soll dabei die Zellen anzeigen, für welche bereits Fingerprints gesammelt wurden. Bei einem Klick auf ein Zelle wird dabei der InformationDetailView aufgerufen.

InformationDetailView Der InformationDetailView zeigt detaillierte Informationen der Zelle an, wie etwa die Anzahl der gesammelten Fingerprints oder die durchschnittliche Signalstärke der Beacons. Dies wird ebenfalls mit Labels gelöst.

4.2.3 Erstellung des CoreData-Modells

Für die Speicherung der Fingerprints wird das CoreData-Framework verwendet. Dafür wurde bei der Erstellung des Projektes ein Modelldatei erzeugt. In dieser lässt sich der Aufbau des Projektes erstellen und bearbeiten. Zuerst müssen verschiedene Entitäten erstellt werden, welche die Daten im Speicher repräsentieren. Für die grundlegende Sammlung von Fingerprints benötigt man vier Entitäten: Cell, Beacon, Fingerprint und Measurement.

Für eine schnellere Durchführung des zweiten Algorithmus, welcher mit den Mittelwerten der Beacons in einer Zelle arbeitet, wird eine weitere Entität angelegt. Die *BeaconInCell*-Entität repräsentiert dabei ein Beacon in einer Zelle. Um den dritten Algorithmus zu ermöglichen, ist noch eine zusätzliche Entität nötig. Die *BeaconInCellProbability*-Entität verwaltet dabei die Wahrscheinlichkeitsverteilung eines Beacons in einer Zelle.

Im Folgenden werden die einzelnen Entitäten genauer erklärt.

Cell

Repräsentiert eine Zelle des Messraumes. Besitzt als Attribut die Zellennummer. Diese besitzt eine *One-to-many*-Beziehung zu BeaconInCell und Fingerprints.

BeaconInCell

Repräsentiert ein Beacon in einer bestimmten Zelle. Besitzt als Attribute sowohl die durchschnittlichen, maximale und minimale Signalstärke als auch deren Median. Außerdem existiert eine *One-to-many*-Beziehung zu Measurements und BeaconInCellProbability.

BeaconInCellProbability

Repräsentiert eine Wahrscheinlichkeitswert für eine Signalstärke eines Beacons in einer Zelle. Daher besitzt es als Attribute die Warhscheinlichkeit und die Signalstärke.

Beacon

Repräsentiert ein Beacon und beinhaltet als Attribute die UUID, den Major-Wert und den Minor-Wert des Beacons. Außerdem besitzt die Entität eine *One-to-many*-Beziehung zu BeaconInCell.

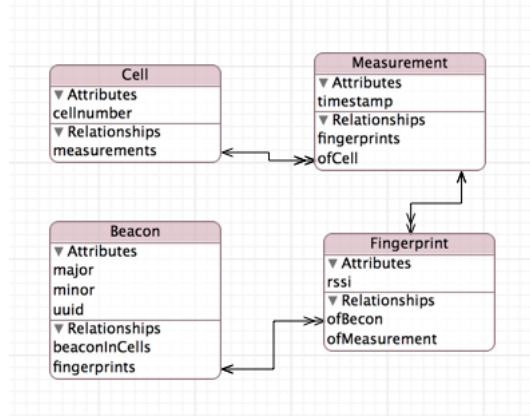
Fingerprint

Repräsentiert einen Fingerprint und beinhaltet als Attribut einen Zeitstempel. Die Entität verfügt darüberhinaus über eine *One-to-many*-Beziehung zu Measurements.

Measurements

Repräsentieren einen einzelnen Signalstärke-Wert und beinhalten dafür ein Attribut Signalstärke.

Aus diesen Entitäten lässt sich im CoreData-Editor von Xcode, dieses Modell erstellen (Abbildung 4.5).

**Abbildung 4.5:** CoreData-Modell im grafischen Editor

Nach der Erstellung des Modells ist es darüberhinaus noch möglich, für die einzelnen Entitäten eigene Klassen zu erzeugen. Dies hat den Vorteil, dass beim Zugriff auf diese Entitäten nicht mit einem generischen *NSManagedObject* gearbeitet werden muss. Stattdessen können Objekte genutzt werden, welche die jeweilige Entität repräsentieren. Diese verfügen dabei über alle benötigten Attribute der Entität in Form von Properties.

```

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: Header der Cell-Entität

Wie in Listing 3 zu sehen, werden für alle Attribute der Entität entsprechende Properties erzeugt. Außerdem werden den Beziehungen entsprechende Properties angelegt, welche, je nach Art der Beziehung, entweder ein Objekt der entsprechenden Entität repräsentieren oder ein Set mit Objekten der Entität.

Bei entsprechenden *One-to-many*-Beziehungen werden zusätzlich Methoden hinzugefügt, welche es erlauben Objekte der Beziehung hinzuzufügen oder zu entfernen.

4.2.4 Implementierung der Oberfläche

Nachdem die Oberfläche erstellt wurde, müssen nun die nötigen Funktionen implementiert werden. Dazu muss für jeden View ein passender Controller implementiert werden, welcher die Funktionalität bereit stellt.

FingerprintingSetupViewController Für die Implementierung des *FingerprintingSetup-ViewController* muss zunächst einen neuen Klasse mit gleichem Namen im Projekt angelegt werden. Da es sich um einen ViewController handelt, muss dieser von der Klasse UIViewController erben.

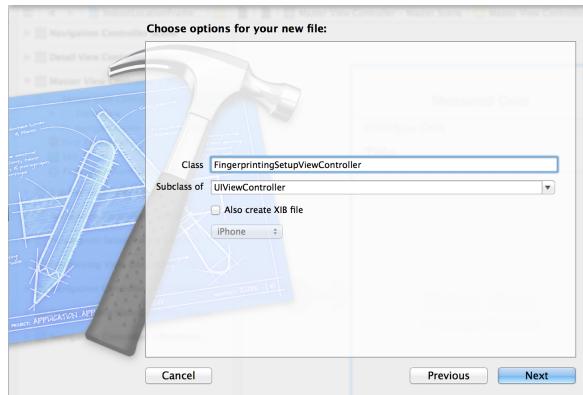


Abbildung 4.6: Erstellung des FingerprintingSetupViewController

Die so erzeugte Klasse beinhaltet bereits drei Standardmethoden eines ViewController. Die einzige wichtige Methode dabei ist die *viewDidLoad*-Methode, welche aufgerufen wird, sobald der View geladen wurde. In dieser lassen sich Properties initialisieren, da dies nur einmal, beim Laden des Views, nötig ist.

Nach der Erzeugung der Klasse muss diese zunächst dem View bekannt gemacht werden. Dazu ist es möglich dem View im Interface Builder eine eigene Klasse zuzuweisen. Statt dem standardmäßig ausgewählten UIViewController muss dort der FingerprintingSetupViewController ausgewählt werden.

Nachdem dies geschehen ist, müssen die zuvor erstellten Textfelder auch dem ViewController bekannt gemacht werden. Dazu bietet Xcode den *Assistend Editor* an. Dieser erlaubt es in zwei Spalten nebeneinander den Quellcode des ViewController und auf der anderen Seite den Interface Builder anzuzeigen. Für die Verbindung zwischen Storyboard und Quellcode wird dabei die sogenannte *Control-Drag*-Geste genutzt. Dabei ist es möglich bei gedrückter *Strg*-Taste das Textfeld in den Quellcode zu ziehen. (Inc. (zuletzt besucht 20.März 2014c)) Xcode erstellt so eigenständig ein Property, welches dem Textfeld entspricht.

So kann man im Quellcode einfach auf das Textfeld zugreifen und dieses auslesen. Das gleiche Verfahren lässt sich auch mit allen anderen Objekten des Views durchführen, wobei es zum Beispiel bei Buttons auch die Option gibt, eine entsprechende Methode zu erstellen, welche bei Druck des Buttons aufgerufen wird.

Mittels dieses Verfahrens werden im FingerprintingSetupViewController Properties für die Textfelder angelegt. Außerdem lassen sich so ebenfalls Methoden für die Buttons,

welche die Textfeldinhalte inkrementieren und dekrementieren. Diese werden bei Druck des Buttons ausgeführt. Außerdem wird der Button zum Starten des Fingerprint-Sammelns mittels der Control-Drag-Geste auf den FingerprintingViewController gezogen. Dies erzeugt einen Segue zwischen diesen beiden Views, welcher bei einem Klick auf den Button ausgelöst wird. Dies bedeutet, dass bei einem Klick auf den Button der Fingerprinting-View geöffnet wird.

Dem FingerprintingViewController müssen jedoch die Werte aus dem FingerprintingSetup-ViewController übergeben werden. Um dies zu ermöglichen müssen die Methoden *prepareForSegue:sender:* und *shouldPerformSegueWithIdentifier:sender:* implementiert werden, welche vor Ausführung des Segues aufgerufen werden.

Die Methode *shouldPerformSegueWithIdentifier:sender:* gibt dabei ein boolschen Wert zurück, welcher bestimmt, ob der Segue ausgeführt wird. Dies ist in diesem Fall wichtig, da die Sammlung der Fingerprints nur gestartet werden kann, falls eine valide Eingabe für die Zellenummer und die Anzahl der zu sammelnden Fingerprints getätigt wurde. Diese beiden Werte werden daher in der Methode überprüft und abhängig von deren Korrektheit wird der Wechsel zwischen den Views erlaubt.

Die Methode *prepareForSegue:sender:* erlaubt es Variablen zwischen den beiden am Segue beteiligten ViewControllern zu übergeben. Dabei besitzt das übergebene *UIStoryboardSegue*-Objekt ein Property *destinationViewController*, wodurch es möglich wird Werte an diesen zu übergeben.

FingerprintingViewController

Der *FingerprintingViewController* verwaltet die Sammlung der Fingerprints. Für den Empfang von Beacon-Daten ist es dabei zunächst notwendig ein *CLLocationManager*-Objekt zu erstellen. Der CLLocationManager ist dabei für das Auffinden der Beacons verantwortlich. Des Weiteren muss der ViewController selbst als *delegate* für den CLLocationManager gesetzt werden. Das Prinzip hinter einem *delegate* ist es, einem Objekt zu erlauben Methoden in der aktuellen Klasse aufzurufen. Dies erlaubt dem CLLocationManager in diesem Fall, zu signalisieren, dass Beacons gefunden wurden und deren Informationen an die Klasse zu übergeben.

Um diese Funktionalität der *delegate* zu nutzen, müssen die entsprechenden Klassen implementiert werden. Für die Rückgabe der Beaconinformationen handelt es sich dabei um die *locationManager:didRangeBeacons:inRegion:-Methode*, welche die aktuell empfangenden Beacons und die aktuelle Region überliefert. Diese Methode wird periodisch vom CLLocationManager aufgerufen.

Um die Suche nach den Beacons zu starten, muss die *startRangingBeaconsInRegion:-Methode* aufgerufen werden, welche ein CLBeaconRegion-Objekt benötigt. Das CLBeaconRegion-Objekt beinhaltet dabei die benötigten Beacon-Information, wie zum Beispiel die UUID.

Im FingerprintingViewController wird in der *locationManager:didRangeBeacons:inRegion:-Methode* die Speicherung der aktuell empfangenden Beacons in die Datenbank durchgeführt.

Für die Speicherung werden verschiedene Methode aus ZDatabaseConnector Klasse verwendet.

InformationViewController

Der *InformationViewController* gibt die Zellen aus, welche bereits über Fingerprint-Daten verfügen. Dabei wurde die Struktur des MasterViewController, welcher durch die Wahl des Templates bereits erstellt wurde, größtenteils beibehalten. Es musste nur die CoreData-Abfrage angepasst werden, sodass alle Zellen des Modells zurückgegeben werden.

InformationDetailViewController

Wie auch schon beim *InformationViewController* kann die grobe Struktur der Klasse beibehalten werden und nur die Datenabfrage muss angepasst werden, um das eigene CoreData-Modell zu unterstützen.

PositioningViewController

Wie bereits im FingerprintingViewController muss zunächst ein CLLocationManager-Objekt und ein CLBeaconRegion-Objekt angelegt werden, um die Beacon-Informationen zu empfangen. Statt die Beacondaten zu speichern, wird in der *locationManager:didRangeBeacons:inRegion:* Methode jedoch der Positionierungsalgorithmus aufgerufen, welchem die aktuellen Informationen übergeben werden. Die verschiedenen Positionierungsalgorithmen wurden in der Klasse ZPositionLocator implementiert.

4.2.5 Implementierung des FingerprintingViewControllers

Der FingerprintingViewController ist für die Sammlung der Fingerprints zuständig. Für die Suche und Erkennung der Beacon wird hier ein CLLocationManager-Objekt genutzt.

Die CLLocationManager bietet in iOS die Möglichkeit die aktuelle Position und Ausrichtung der Gerätes auszulesen. Bei der Bestimmung der Position gibt es dabei verschiedene Möglichkeiten. Neben GPS-Positionierung bietet der CLLocationManager seit iOS 7 die Möglichkeit nach Beacons in der Umgebung zu suchen.

In iOS ist es dabei nicht möglich nach beliebigen Beacons zu suchen. Es muss eine CLBeaconRegion angegeben werden, welche die zu suchenden Beacons genauer spezifiziert.

Das CLBeaconRegion-Objekt muss dabei mindestens mit der zu suchenden UUID und dem Identifier initialisiert werden. Zusätzlich ist es noch möglich sich auf einen Major-Wert oder einen Major-Wert und Minor-Wert festzulegen.

Da die Lokalisierung in diesem Fall nur an einem Ort stattfinden soll, ist eine tiefere Unterscheidung nicht nötig, sodass die CLBeaconRegion nur mittels UUID und Identifier initialisiert wird.

Um bei erfolgreicher Auffindung von Beacons in der Umgebung benachrichtigt zu werden, muss zunächst der FingerprintingViewController als *delegate* des CLLocationManager-Objektes gesetzt werden. Dies ermöglicht das Senden von Nachrichten vom CLLocationManager-Objekt an den ViewController. Dabei sind verschiedene Methoden zu implementieren, welche vom CLLocationManager-Objekt aufgerufen werden, sobald ein Ereignis eintritt. Dabei gibt es verschiedene Ereignisse, die Methoden aufrufen, wie etwa das Finden von Beacons, das Betreten einer BeaconRegion oder das Verlassen einer BeaconRegion.

Um bei diesen Ereignissen verschiedene Aktionen auszuführen, müssen zunächst die Delegate-Methoden implementiert werden. Die Methode, welche beim Finden von Beacons aufgerufen wird lautet dabei *locationManager:didRangeBeacons:inRegion:* und übergibt dabei ein Array mit den aktuell gefundenen Beacon und die CLBeaconRegion, zu der die Beacons gehören.

Da jeder Aufruf dieser Methode einem neuen Fingerprint entspricht, muss dieser gespeichert werden. Dies übernimmt die Methode *updateRangingDataForCurrentCellWithBeacons:*, welche ein Array von Beacons erwartet. In der Datenbank wird für diese Daten ein neuer Fingerprint erstellt, welcher bereits alle nötigen Beziehungen zu den anderen Entitäten besitzt.

Dies geschieht dabei so lange, bis die Suche nach Beacons manuell abgebrochen wird oder das im FingerprintingSetupViewController angegebene Limit erreicht wurde.

Bevor der FingerprintingView geschlossen wird und die Applikation zum FingerprintingSetupViewController zurückkehrt, werden noch einige benötigte Werte errechnet. Zum einen werden die minimale, maximale und durchschnittliche Signalstärke sowie der Median der Signalstärke für jedes Beacon in der aktuellen Zelle bestimmt. Darüberhinaus wird auch die Wahrscheinlichkeitsverteilung für die Beacons der aktuellen Zelle erstellt beziehungsweise aktualisiert.

Dies muss nach jedem Hinzufügen von Fingerprints geschehen.

4.2.6 Implementierung des PositioningViewController

Für den PositioningViewController müssen ebenfalls ein CLLocationManager-Objekt und eine CLBeaconRegion entsprechend denen aus dem FingerprintingViewController angelegt werden.

Im Unterschied zum FingerprintingViewController muss der PositioningViewController in der *locationManager:didRangeBeacons:inRegion:-Methode* jedoch keine Speicherung der Daten durchführen, sondern die aktuelle Position bestimmen. Dafür wurden verschiedene Algorithmen implementiert.

Einfaches 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 der in der Umgebung befindlichen Beacons, als auch die zuvor erhobenen Fingerprints der Beacons 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 der Fingerprints der Beacons n ist dabei F_{b_n} . Die Signalstärke der aktuellen Messung des Beacons n wird als M_{b_n} bezeichnet.

$$\begin{pmatrix} F_{b_1} \\ F_{b_2} \\ \dots \\ F_{b_n} \end{pmatrix} - \begin{pmatrix} M_{b_1} \\ M_{b_2} \\ \dots \\ M_{b_n} \end{pmatrix} = \begin{pmatrix} F_{b_1} - M_{b_1} \\ F_{b_2} - M_{b_n} \\ \dots \\ F_{b_n} - M_{b_n} \end{pmatrix} \quad (4.2)$$

$$\begin{pmatrix} F_{b_1} - M_{b_1} \\ F_{b_2} - M_{b_2} \\ \dots \\ F_{b_n} - M_{b_n} \end{pmatrix} = \begin{pmatrix} Diff_1 \\ Diff_2 \\ \dots \\ Diff_n \end{pmatrix} \hat{=} \sqrt{Diff_1^2 + Diff_2^2 + \dots + Diff_n^2} \quad (4.3)$$

Aus den Differenzen beziehungsweise die Abstände zwischen den einzelnen Signalstärke-Vektoren lässt sich nun der Nearest-Neighbor bestimmen und damit die wahrscheinlichste Position im Raum.

Implementierung Für die Implementierung werden dabei die einzelnen Fingerprints mit den aktuellen Messwerten verglichen. Die aktuellen Werte stehen dabei in Form eines Arrays zur Verfügung, welches CLBeacon-Objekte beinhaltet. Diese bieten neben der Identifizierung der Beacon-Information, die aktuelle Signalstärke des Beacons. Bei diesem Algorithmus wird die Ähnlichkeit aller gesammelten Fingerprints gegenüber den aktuellen Messwerten bestimmt und der Fingerprint mit der größten Ähnlichkeit als aktuelle Position angenommen.

Für die Positionierung wird dabei die Klasse ZPositionLocator genutzt, in welcher der Algorithmus zur Bestimmung der akutellen Position implementiert ist.

ZPositionLocator

Für die Positionierung mittels Nearest-Neighbor-Verfahren über alle Fingerprints wird die Methode *nearestCellWithFingerprintMethodWithBeacons:andCellsInRegion* genutzt. Dieser Methode werden die aktuell gemessenen Beacons und die bereits eingemessenen Zellen übergeben. In der Methode wird daraufhin über die Zellen und anschließend über

deren Fingerprints iteriert. Dabei wird für jeden Fingerprint die euklidische Distanz zu den aktuell gemessenen Werten berechnet. Dazu werden die einzelnen Messwerte der Beacons in einen Vektor übertragen, welcher im Programm als Array gespeichert wird. Diese Form ist für die Berechnung der euklidischen Distanz nötig. Die Zeilen des Vektors entsprechen dabei jeweils den gleichen Beacons. Die euklidische Distanz wird dabei über eine eigene Methode (Listing 4) berechnet. Während der Ausführung der Methode werden dabei die aktuell kleinste Distanz und die aktuell nächste Zelle gespeichert. Während jedes Iterationsschrittes wird dabei die aktuelle euklidische Distanz mit der bisher kleinsten Distanz verglichen werden. Falls diese kleiner ist, wird die aktuelle Distanz als kleinste Distanz gesetzt und die Zelle wird gespeichert.

```

1  +(float)euclideanDistanceBetweenVector:(NSArray*)vector1 andVector:(NSArray*)
2
3      /** check if both vectors are the same size */
4      if([vector1 count] != [vector2 count]) {
5          return -1;
6      }
7
8      /** first calculate the sum under the square root */
9      float euclideanDistanceSum = 0;
10     for (int i = 0; i < [vector1 count]; i++) {
11         /** check if both array only contain NSNumber objects */
12         if([vector1[i] isKindOfClass:[NSNumber class]] && [vector2[i] isKindOfClass:[NSNumber class]])
13             /** calculate sum of square of v1 - v2 */
14             euclideanDistanceSum += ([vector1[i] floatValue] - [vector2[i] floatValue]) * ([vector1[i] floatValue] - [vector2[i] floatValue]);
15     }
16     else {
17         /** error if wrong input file format */
18         NSLog(@"Wrong file format, no NSNumber. (ZLocationPositioner:euclideanDistanceBetweenVector)");
19         return -1;
20     }
21 }
22
23     /** calculate final euclidean distance */
24     float euclideanDistance = sqrt(euclideanDistanceSum);
25
26     return euclideanDistance;
27
28 }
```

Listing 4: Bestimmung der euklidischen Distanz zweier Vektoren

Nachdem die Iteration abgeschlossen wurde, liegt der Fingerprint mit der geringsten euklidischen Distanz vor.

Die für die Positionierung relevante Zelle lässt sich dabei aus dem Fingerprint auslesen und die nächstgelegene Zelle wird von der Methode zurückgegeben. Der Positioning-

Viewcontroller verarbeitet dieses Objekt und gibt die aktuelle Zelle auf dem Bildschirm aus.

Probleme Bei dem Standard Nearest-Neighbor-Verfahren kommt es jedoch zu einigen Problemen. Die Masse der zu überprüfenden Daten kann, je nach der Größe der Fingerprint-Datenbank, sehr groß werden. Bei sehr großen Datenmengen kann es zu einer längeren Laufzeit bei der Überprüfung der Fingerprints kommen und außerdem wird mehr Systemspeicher belegt. Eine Möglichkeit dies zu beheben, wäre die Verlagerung der Berechnungen auf einen Server, welche als Ergebnis die aktuelle Position liefert. Die zu übertragenden Daten dabei sind sehr gering, da es sich nur um die aktuellen Beacon-Signalstärken handelt beziehungsweise die aktuelle Position, welche vom Server zurückgesendet wird. Außerdem würde die Rechenlast komplett von iPhone genommen, was sich positiv auf die Batterielaufzeit und Performance auswirkt.

Ein weiteres Problem sind Messfehler beziehungsweise Messausreißer, welche das Ergebnis verfälschen können. So werden auch Ausreißer in das Nearest-Neighbor-Verfahren mit einbezogen, wodurch die Berechnung der aktuellen Position verfälscht werden kann.

Daher wurde überlegt, wie dieses Problem behoben werden könnte. Dies wurde so gelöst, dass statt aller Fingerprint-Werte nur der Mittelwert der Signalstärke eines Beacons genutzt wird.

Nearest-Neighbor-Verfahren mit Mittelwerten

Algorithmus Der zweite Ansatz arbeitet ähnlich wie das zuvor erklärte Verfahren, nutzt jedoch nicht die komplette Datenbank der Fingerprints. Stattdessen wird für jede Zelle ein Mittelwert über alle Fingerprints der vorhandenen Beacons berechnet und dieser für die Bestimmung der Position verwendet. Dieser Mittelwert muss dabei nur bei einer Änderung der grundlegenden Fingerprint-Datenbank angepasst werden, wodurch der Rechenaufwand niedriger gehalten wird, da bei jeder Positionsbestimmung nur auf den Mittelwert zugegriffen werden muss. Außerdem wird der Einfluss von Störungen und Messungenauigkeiten der Fingerprints verringert.

Bei der Implementierung wurden zwei Mittelwerte getestet. Zum einen der Median, welcher den mittleren Wert einer sortierten Reihe aller Werte nutzt und zum anderen das arithmetische Mittel, welcher alle Werte aufaddiert und dann durch die Anzahl der Werte dividiert.

Dabei wird die durchschnittliche Signalstärke eines Beacons $RSSI_{avg}$ über die vorhandenen Werte der Fingerprints zu diesem Beacon in einer bestimmten Zelle errechnet.

$$RSSI_{avg} = \frac{RSSI_1 + RSSI_2 + \dots + RSSI_n}{n} \quad (4.4)$$

Der arithmetische Mittelwert lässt sich sehr leicht berechnen und schafft es kleinere Messungenauigkeiten zu beseitigen. Falls jedoch sehr starke Messfehler einfließen, können diese das arithmetische Mittel deutlich verfälschen.

Im Gegensatz dazu ist der Median deutlich robuster gegenüber Messfehlern als das arithmetische Mittel.

Der Median errechnet sich dabei wie folgt:

Für $RSSI$ gilt: $RSSI_1 \leq RSSI_2 \leq \dots \leq RSSI_n$

$$RSSI_{median} = \begin{cases} RSSI_{\frac{n+1}{2}} & \text{für } n \text{ ungerade} \\ \frac{RSSI_{\frac{n}{2}} + RSSI_{\frac{n}{2}+1}}{2} & \text{für } n \text{ gerade} \end{cases} \quad (4.5)$$

Bei der Berechnung wird klar, warum Messfehler keinerlei Auswirkungen haben. Da beim Median der mittlere Wert der sortierten Reihe genutzt wird, spielen Messfehler, welche sich am Anfang oder Ende der Reihe befinden, keine Rolle.

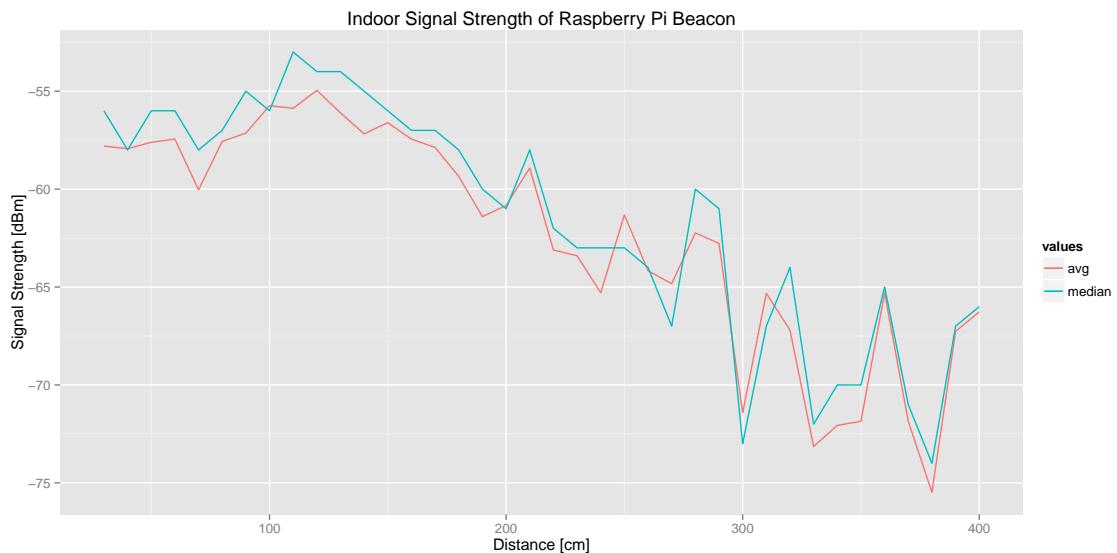


Abbildung 4.7: Vergleich zwischen dem arithmetischen Mittel und dem Median

Wie in Abbildung 4.7 zu erkennen, ist der Unterschied zwischen Median und arithmetischem Mittel jedoch zu vernachlässigen.

Implementierung Der Nearest-Neighbor-Algorithmus über die Mittelwerte wurde ähnlich des vorherigen Nearest-Neighbor über die gesamten Fingerprints implementiert. Dabei wurde eine neue Methode in der ZPositionLocator erstellt.

ZPositionLocator Diese Methode des Fingerprints ist ähnlich der vorherigen Methode implementiert. Dazu wurde die Methode `nearestCellWithAverageSignalStrengthMethod-WithBeacons:andCellsInRegion:useMedian` genutzt. Diese Methode bekommt dabei, wie auch in der Methode des vorherigen Algorithmus, die aktuell gemessenen Beacons und die bereits gemessenen Zelle übergeben. Zusätzlich wird ein boolescher Wert übergeben, welcher entscheidet, ob der Median genutzt wird oder der Durchschnitt der Fingerprints. Die Vorgehensweise der Methode für die Bestimmung der nächsten Zelle läuft

dabei identisch zum ersten Verfahren ab. Es wird wieder über alle Fingerprints iteriert und jeweils die nächste Zelle und deren euklidische Distanz zum aktuellen Messwert gespeichert. Der einzige Unterschied ist, dass nicht über alle Fingerprints iteriert wird, sondern nur über die Durchschnittswerte beziehungsweise die Medianwerte. Dies beschleunigt die Berechnung der aktuell am nächsten gelegenen Zelle im Vergleich zum vorherigen Algorithmus. Die Mittelwerte werden dabei aus dem CoreData-Modell ausgelesen, da diese schon während der Sammlung der Fingerprints generiert wurden und in der BeaconInCell-Entität gespeichert wurden. Für die Berechnung der euklidischen Distanz wird die gleiche Methode genutzt wie in der vorherigen Implementierung.

Für die Implementierung der Positionsbestimmung mit Mittelwerten werden die, während der Sammlung der Fingerprints generierten, Mittelwerte genutzt, welche in der BeaconInCell-Entität gespeichert wurden. Durch die Speicherung der Werte in einer eigenen Entität ist es nicht nötig diese Werte bei jeder Positionierung neu zu errechnen, da sich diese nur nach einer Änderung an der Datenbasis verändern. Dadurch wird die Bearbeitungszeit der Positionierung reduziert.

Auch hier wird die aktuelle Zelle von der Methode zurückgegeben und der Positioning-Viewcontroller verarbeitet diese weiter und gibt die aktuelle Position an den Benutzer aus.

Probabilistisches Verfahren

Algorithmus Der letzte untersuchte Ansatz war ein probabilistisches Verfahren, welches die Wahrscheinlichkeiten einer bestimmten Signalstärke eines Beacons als Referenz-Wert für die Positionbestimmung nutzt. Dabei wurde für die Berechnung der Wahrscheinlichkeiten und Bestimmung der aktuellen Position das Verfahren von Le Dortz et al. (2013) verwendet, welche das Fingerprinting mit Hilfe von Wireless LAN-Routern verwenden.

Da das Fingerprinting mittels Wireless LAN ebenfalls auf der Positionierung mittels Signalstärken basiert, lässt sich das Verfahren auch auf andere Technologien, welche die Signalstärke zur Positionsbestimmung nutzen, übertragen.

Für die Nutzung mit Bluetooth LE mussten einige Anpassungen vorgenommen werden, der verwendete Algorithmus bleibt jedoch sehr ähnlich.

Wie in den vorher vorgestellten Fingerprinting-Verfahren ist es zunächst nötig, während der Offline-Phase, die benötigten Fingerprints zu sammeln. Zusätzlich wird, nachdem die Fingerprints für eine Zelle gesammelt sind, eine Wahrscheinlichkeitsverteilung der Signalstärken für jedes Beacon in der aktuellen Zelle erstellt. Diese wird später als Vergleichswert genutzt, um die Ähnlichkeit der Signalstärken zu bestimmen.

Diese Wahrscheinlichkeitsverteilungen werden dabei für jedes Beacon in jeder aufgezeichneten Zelle errechnet.

Danach folgt die Online-Phase in der die aktuelle Position bestimmt werden soll. Dafür werden die aktuellen Signalstärken der Beacons gemessen und gespeichert, um von diesen Werten ebenfalls eine Wahrscheinlichkeitsverteilung zu berechnen. Die Anzahl der

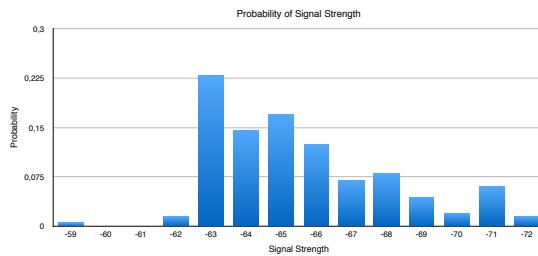


Abbildung 4.8: Wahrscheinlichkeitsverteilung von Signalstärke bei einem Beacon

Werte, welche in die Wahrscheinlichkeitsverteilung einfließen ist dabei entscheidend. Dabei muss man zwischen der statischen Positionierung und der Echtzeit-Positionierung unterschieden werden. Für eine statische Positionsbestimmung sollte die Anzahl der letzten gespeicherten Werte groß gewählt werden, da hier keine Echtzeit-Änderung der Position geschieht und eine große Datenbasis hilfreich ist, um eine umfassende Wahrscheinlichkeitsverteilung zu bestimmen. Für unsere Anwendung der Echtzeit-Positionsbestimmung ist ein relativ schnelles aktualisieren der Position jedoch essenziell. Daher muss hier ein Kompromiss aus akkuratester Positionsbestimmung und Echtzeit-Fähigkeit gefunden werden. Dabei muss die Zahl der Signalstärke-Werte, welche in die Wahrscheinlichkeitsverteilung einfließen, so gewählt werden, das die Position in aufreichenden Abständen aktualisiert wird.

Die Wahrscheinlichkeitsverteilung der aktuell gemessenen Werte muss daraufhin mit den Wahrscheinlichkeitsverteilungen aller Zellen verglichen werden und deren Ähnlichkeit muss bestimmt werden. Dafür wird die Bhattacharyya-Distanz genutzt, welche die Ähnlichkeit zweier Wahrscheinlichkeitsverteilungen beschreibt. Diese Distanz muss für jede Zelle errechnet werden. Dafür muss zunächst der Bhattacharyya-Koeffizient $B_{b,c}$ für jedes Beacon b einer Zelle c berechnet werden.

$$B_{b,c} = \sum_{s \in [s_{min}, s_{max}]} \sqrt{P_b^c(s) \cdot Q_b(s)} \quad (4.6)$$

Um daraus die Bhattacharyya-Distanz für die aktuelle Zelle c zu berechnen, wird zunächst das arithmetische Mittel der Bhattacharyya-Koeffizienten der q stärksten Beacons der aktuellen Zelle O_c^q gebildet. Die Bhattacharyya-Distanz d_c ist dabei der negative Logarithmus über diesen Mittelwert.

$$d_c = \begin{cases} -\ln(\frac{1}{q} \sum_{i \in O_c^q} B_{b,c}) & \text{wenn } \sum_{i \in O_c^q} B_{b,c} > 0 \\ -\infty & \text{sonst} \end{cases} \quad (4.7)$$

Diese Distanz gibt nun die Ähnlichkeit der aktuellen Wahrscheinlichkeitsverteilung mit den Wahrscheinlichkeitsverteilungen der jeweiligen Zelle an. Daraus ergibt sich, dass die Zelle mit der kleinsten Distanz die wahrscheinlichste Zelle für die aktuellen Messwerte ist.

Für unseren Zweck reicht diese Angabe aus. Es lässt sich jedoch, wie im Paper von Le Dortz et al. (2013) weiter ausgeführt, auch noch eine interpolierte Position aus den k wahrscheinlichsten Positionen bilden, welche, gewichtet nach ihrer Distanz, in die finale Position einfließen. Da wir jedoch mit Zellen arbeiten und nicht mit Koordinaten wurde darauf verzichtet.

Implementierung BeaconSampler Für die Implementierung muss zunächst eine Wahrscheinlichkeitsverteilung über die aktuell gemessenen Beaconsignalstärken erstellt werden. Dazu wurde eine neue Klasse namens *BeaconSampler* erstellt. Ein BeaconSampler-Objekt verwaltet dabei die aktuelle Wahrscheinlichkeitsverteilung. Dazu besitzt es eine Metode zum Hinzufügen von Beaconinformationen, eine Methode zum Auslesen der aktuellen Wahrscheinlichkeitsverteilung für ein Beacon und eine Methode, welche die aktuell stärksten Beacons zurück gibt. Zudem wird eine Methode bereitgestellt, welche sowohl CLBeacon-Objekte als Beacon-Objekte in Beaconidentifizierungsstrings umwandelt. Dieser String bestehen aus der UUID, dem Major und dem Minor-Wert.

Die Methode *addBeacons* verlangt dabei ein NSArray der aktuell empfangenen Beacons. Die Signalstärkewerte der Beacons werden daraufhin zu einem, vom BeaconSampler verwaltetem, NSDictionary hinzugefügt. Dabei werden, um dem Echtzeitanspruch gerecht zu werden, nur die letzten fünf gemessenen Signalstärkewerte für jedes Beacon gespeichert. Dadurch wird die Wahrscheinlichkeitsverteilung ausreichend schnell an die aktuelle Position und deren Signalstärken angepasst.

Außerdem verfügt der BeaconSampler über die Methode *getProbabilityDictForBeaconWithIdentifier*, welche ein Beaconidentifizierungs-String erhält. Die Methode errechnet dabei die Wahrscheinlichkeiten der Signalstärken des Beacon, welches zu dem übergebenen Beaconidentifizierungs-String passt. Diese werden dabei in einem NSDictionary gespeichert, wobei der Signalstärke-Wert als Key des Dictionary verwendet und die Wahrscheinlichkeit des Wertes als Value im Dictionary. Dieses NSDictionary wird von der Methode zurückgegeben.

Eine weitere Methode ist die *getKeysOfStrongestBeacons*-Methode, welche ein Array mit Beacons zurückgibt, absteigend sortiert nach der Signalstärke. Dies wird dazu verwendet, um bei der Positionierung Zellen auszuschließen, da nur Zellen berücksichtigt werden, welche auch über die aktuell stärksten Beacons verfügen. Dies erlaubt eine schnelleren Ausführung, da nicht die Wahrscheinlichkeitsverteilungen jeder Zelle berücksichtigt werden müssen.

ZPositionLocator In der Klasse *ZPositionLocator* wurden die Methoden zur Positionierung implementiert. Die Methode *nearestCellsForBeaconSampler:withContext* wird dabei zur Bestimmung der nächsten Zellen genutzt. Die Methode erwartet dabei das aktuelle BeaconSampler-Objekt, welches die Wahrscheinlichkeitsverteilung enthält. Zunächst werden die für die Positionierung in Frage kommenden Zellen bestimmt. Dabei

werden nur Zellen berücksichtigt, deren fünf stärksten Beacons aktuell empfangen werden. Dadurch müssen nicht die Wahrscheinlichkeitsverteilungen aller Zellen berücksichtigt werden, wodurch die Geschwindigkeit der Positionierung erhöht werden soll.

Für die in Frage kommenden Zellen werden nun die Bhattacharyya-Distanzen berechnet. Dazu wird zunächst über die Zellen iteriert und dabei für die fünf stärksten Beacons der Zelle der Bhattacharyya-Koeffizient berechnet. Die Summe der Koeffizienten und die Anzahl der hinzugefügten Koeffizienten werden dabei in separaten Variablen gespeichert. Dadurch lässt sich, nachdem die Bhattacharyya-Koeffizienten für alle Beacons errechnet wurden, der durchschnittliche Bhattacharyya-Koeffizient errechnen. Dieser wird letztlich verwendet, um die Bhattacharyya-Distanz des Beachons zu errechnen. Die Distanzen werden alle in ein NSDictionary gespeichert, welches als Key die Zellenummer verwendet und als Value die Bhattacharyya-Distanz speichert.

Die *nearestCellsForBeaconSampler:withContext* gibt dabei ein, nach den Bhattacharyya-Distanzen, sortiertes NSArray mit den Zellenummern zurück.

5 Versuchsergebnisse

Im Anschluss an die Implementierung der verschiedenen Algorithmen, sollten diese unter realen Bedingungen getestet werden. Dazu wurde ein Versuchsaufbau innerhalb eines Gebäudes aufgebaut, welches aus mehreren Räumen besteht. Dafür werden 9 Beacons innerhalb des Gebäudes verteilt. In der Offline-Phase werden dabei für jede Position 40 Fingerprints gesammelt.

Untersucht werden soll dabei zum einen die Genauigkeit der Positionierung eines sich bewegenden Objektes. Hierfür wird der Pfad aufgezeichnet und mit dem realen Pfad verglichen. Zum Anderen soll die Positionierung an einer statischen Position getestet werden. Dazu wird an verschiedenen Testpunkten geprüft, ob die ausgegebene Position mit der realen Position überein stimmt.

Der Testraum besteht dabei aus drei benachbarten Zimmern, wobei zwei dieser Zimmer durch einen offenen Durchgang verbunden sind. Das andere Zimmer ist über eine Tür zu erreichen. In Abbildung 5.1 lässt sich der Aufbau des Messraumes erkennen, wobei die Dreiecke die eingemessenen Positionen darstellen und die Beacons als blaue Antennen gezeigt werden.

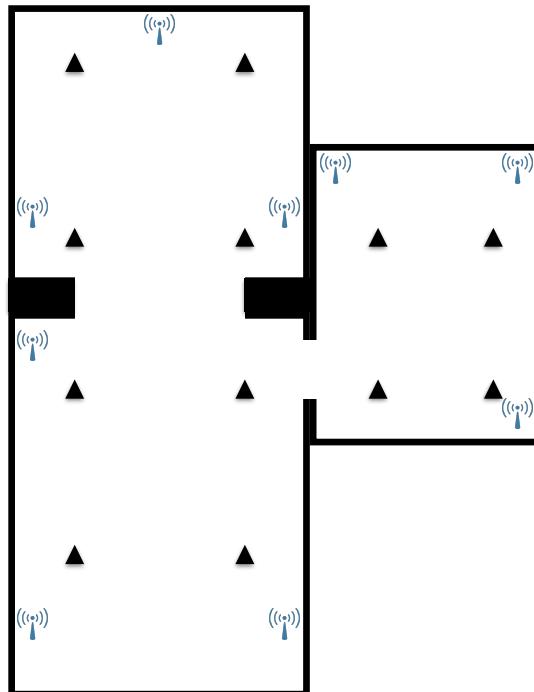


Abbildung 5.1: Abbildung der Messräume

Die Abstände zwischen den einzelnen Messpositionen betragen dabei zwei Meter, woraus sich auch die mögliche Genauigkeit der Messung ergibt.

5.1 Positionierung eines beweglichen Objektes

Bei den Tests der Positionierung bei einem bewegenden Objekt wird zuerst ein abzulaufender Pfad festgelegt, welcher jeweils über die Mittelpunkte der zuvor eingemessenen Zellen führt. Dieser Pfad wird dabei langsam abgelaufen und die Positionierungsangabe für jeden Algorithmus werden aufgezeichnet und nach Beendigung der Messungen wird der gemessene Pfad mit dem realen Pfad verglichen. Dabei wird das Hauptaugenmerk auf die Korrektheit der Position, die Größe der Abweichung und die Echtzeitfähigkeit des verwendeten Algorithmus gelegt.

Bei der Durchführung wurde zwei vorgegebene Wege abgelaufen. Bei der Analyse wird dabei der real abgeschrittene Weg mit dem, durch die Algorithmen errechneten, Weg verglichen.

Alle getesteten Algorithmen unterlagen dabei relativ starken Schwankungen zwischen einzelnen Zelle. Das Verfahren mittels Wahrscheinlichkeitsverteilungen erreichte bei diesem Test die höchste Genauigkeit, wobei hier die errechnete Position immer wieder zwischen der Zelle der aktuellen Position und den benachbarten Zellen wechselte. Durch diese Ungenauigkeit war es daher nicht möglich den genauen Laufweg zu reproduzieren. Eine Aussage darüber, in welchem Raum sich das Gerät aktuell befindet, konnte jedoch mit sehr großer Sicherheit durchgeführt werden.

Auch das Nearest-Neighbor-Verfahren über die gesamte Fingerprintdatenbank erlaubte ähnliche Aussagen zu treffen, obwohl es etwas instabiler als das probabilistische Verfahren ist. Der Vorteil dieses Systems ist die schnellere Aktualisierungsrate, da es nur auf den letzten gemessenen Wert zurückgreift. Das probabilistische Verfahren hingegen nutzt die letzten fünf gemessenen Werte, wodurch sich die Aktualisierung der Position um einige Sekunden verzögert.

Das Nearest-Neighbor-Verfahren über die Mittelwert schnitt bei dieser Untersuchung schlecht ab. Eine Positionsangabe war kaum möglich, da die Zellenangaben stark schwanken und im Großteil der Fälle ein falsche Position angezeigt wurde.

Bei der Positionierung von beweglichen Objekten ist der probabilistische Algorithmus am Besten geeignet, da dieser einen guten Kompromiss zwischen Genauigkeit und Aktualisierungsrate bietet. Falls mehr Wert auf eine sehr schnell Aktualisierung gelegt wird, sollte jedoch der Nearest-Neighbor-Algorithmus über alle Fingerprints genutzt werden, da dieser eine sofortige Aktualisierung der Position mit sich bringt.

5.2 Statische Positionierung

Bei der statischen Positionierung wird an verschiedenen festen Positionen eine Positionsangabe durchgeführt. Dabei werden die Ergebnisse der verschiedenen Algorithmen verglichen. Dabei wird neben der korrekten Positionsangabe besonders auf die Konstanz dieser Positionsangabe geachtet. Diese sollte nicht zu stark schwanken und sich nach kurzer Zeit auf die korrekte Position einpendeln. Bei der Untersuchung der statischen Positionierung wurde an vier verschiedenen Punkten im Messraum für 20 Sekunden Stichproben genommen. Dabei wurden alle implementierten Algorithmen untersucht

und direkt verglichen. Dabei wird eine grobe Unterteilung genutzt, sodass eine Positionierung entweder die richtige Zelle bestimmt, eine benachbarte Zelle oder eine weiter entfernte Zelle als aktuelle Position angibt. Wie bereits erwartet, schafft der Algorithmus mit der Wahrscheinlichkeitsverteilung am Besten ab. Dieser schaffte es in knapp zwei Dritteln der Fälle die genaue Position zu bestimmen. In den restlichen Fällen bestimmte der Algorithmus eine benachbarte Position als aktuelle Position. Dabei wurde kein Mal eine nicht in unmittelbarer Nachbarschaft befindliche Position als die aktuelle angenommen. Überraschender Weise liegt der Nearest-Neighbor-Algorithmus, welcher über alle Fingerprints iteriert, von der Genauigkeit an der zweiten Stelle. Die Quote der richtigen Positionsangaben lag dabei bei etwa 50 Prozent. In etwa 40 Prozent der Fälle wurde eine benachbarte Position als die aktuelle Position bestimmt und nur in einem Zehntel der Fälle wurde eine Position, welche sich nicht in der direkten Nachbarschaft befindet, bestimmt. Der Nearest-Neighbor-Algorithmus mit dem Median und der durchschnittlichen Signalstärke lagen bei der Messung gleich auf. Dabei ergab sich bei dieser Methode nur eine Treffrate von 36 Prozent. Als benachbarte Zelle werden nur etwa 5 Prozent der erkannten Positionen bestimmt. Die Fehlerquote dieser Methode liegt dabei bei über der Hälfte aller bestimmten Positionen und ist so nicht ausreichend genau.

	Nearest-Neighbor (alle Fingerprints)	Nearest-Neighbor (Median/Durchschnitt)	Wahrscheinlichkeitsverteilung
genaue Zelle	51,3%	36,8%	68,4%
benachbarte Zelle	39,5%	5,3%	31,6%
Entfernte Zelle	9,2%	57,9%	0%

Abbildung 5.2: Genauigkeitswerte der statischen Lokalisierung

Für eine statische Positionierung ist das Verfahren zur Positionierung mittels Wahrscheinlichkeitsverteilungen sehr gut geeignet. Die Genauigkeit ist dabei ausreichend, da der größte gemessene Fehler eine Lokalisierung in eine benachbarten Zelle ist. Außerdem ist dieses Verfahren relativ robust und es sind nur geringe Schwankungen in der Positionierung zu beobachten. Auch das Nearest-Neighbor-Verfahren über die gesammte Datenbank der Fingerprints liefert akzeptable Ergebnisse und liegt nur in unter 10 Prozent der Fälle außerhalb der eigentlichen oder benachbarten Zelle. Hierbei sind jedoch schon größere Schwankungen zu erkennen, sodass die Positionierung häufiger zwischen zwei oder mehreren Zelle springt. Die Nearest-Neighbor-Methode mit den Mittelwerte ist weniger geeignet für die Indoor Positionierung, da hier eine sehr hohe Fehlerrate vorliegt und die Ergebnisse zudem sehr stark schwanken.

6 Fazit und Ausblick

Das Ziel dieser Arbeit war es eine iOS-Applikation zu entwickeln, welche eine Lokalisierung der aktuellen Position mittels iBeacons ermöglicht. Dabei sollte außerdem untersucht werden, in wie weit sich die Bluetooth LE-Technologie, auf welcher die iBeacons basieren, für die Indoor Positionierung eignen und welche Vorgehensweise bei der Lokalisierung die geeignetste ist. Das Ziel, eine iOS-Applikation zu entwickeln, wurde dabei erfüllt.

Bei der Untersuchung der Nutzbarkeit von Bluetooth bei der Indoor Lokalisierung ist das Ergebnis nicht ganz zufriedenstellend. Die Funkttechnologie ist dabei sehr störanfällig und stark vom verwendeten Chipsatz und dem Design der Antenne abhängig. So waren zwischen den hier getesteten Beacons teilweise deutliche Unterschiede festzustellen. Auch die Störanfälligkeit durch sich bewegenden Menschen könnte eine Beeinflussung der Genauigkeit des Ergebnisses nach sich ziehen, da der Körper die Signalstärke deutlich abschwächt.

Auch das verwendete Gerät spielt bei der Positionierung eine große Rolle, da auch hier das Antennendesign und der verwendete Chipsatz einen großen Unterschied bei der empfangenen Signalstärke spielen.

Dadurch dass diese ganzen Faktoren in die Positionierung mit einbezogen werden müssen, ist es schwierig ein einheitliches Verfahren für verschiedene Smartphonemodelle zu erstellen. Theoretisch müsste für jedes Gerät eine eigene Einmessung vorgenommen werden, wobei dies sowohl für die Smartphones als auch für die verwendeten Beacons gilt. Eine Alternative wäre es die Abweichungen zwischen den einzelnen Geräten zu bestimmen. Aufgrund fehlender Testgeräte konnte in der Arbeit nicht bestimmt werden, in wie fern sich gleiche Modelle mit identischen Chipsätzen in der Signalqualität unterscheiden oder ob diese Werte vergleichbar sind.

Für die Zukunft ist Bluetooth für eine grobe Indoor Navigation und für die sogenannten Location-based Services eine vielversprechende Technologie. Die Bluetoothsender sind sehr günstig, einfach zu konfigurieren und in ihrer Anbringung äußerst flexibel. Durch Verbesserungen der Signalqualität der Bluetoothsender und der Antennen in den mobilen Geräten könnte die Positionierung zudem an Genauigkeit gewinnen.

Die Integrierung der iBeacons-Technologie in das Betriebssystem iOS wird die Bluetooth-Technologie in der nächsten Zeit fördern und wahrscheinlich viele neue Einsatzmöglichkeiten für diese bieten. So wird die Nutzung von Location-based Services in der nächsten Zeit zunehmen, um zum Beispiel Gutscheine in Geschäften anzubieten oder automatisch Karten vom aktuellen Ort auf dem Gerät anzuzeigen.

Literaturverzeichnis

A. Allan and S. Mistry. Reverse engineering the estimote. zuletzt besucht 20.März 2014.
URL <http://makezine.com/2014/01/03/reverse-engineering-the-estimote/>.

bluegiga. Ble113 bluetooth smart module. zuletzt besucht 20.März 2014.
URL <https://www.bluegiga.com/en-US/products/bluetooth-4.0-modules/ble113-bluetooth--smart-module/>.

H. Bui. Nordic tech tour: Introduction to bluetooth low energy. April 2013. URL http://www.eabeurs.nl/files/7013/7085/2988/3_Introduction_to_Bluetooth_low_energy.pdf.

J. Decuir. Bluetooth 4.0: Low energy. 2010. URL <http://chapters.comsoc.org/vancouver/BTLER3.pdf>.

D. E. Dilger. Inside ios 7: ibeacons enhance apps' location awareness via bluetooth le. *AppleInsider.com*, Juni 2013. URL <http://appleinsider.com/articles/13/12/06/first-look-using-ibeacon-location-awareness-at-an-apple-store>.

estimote. estimote website. zuletzt besucht 20.März 2014a. URL <http://estimote.com/>.

estimote. estimote api. zuletzt besucht 20.März 2014b. URL <http://estimote.com/api/index.html>.

GitHub. Github website. zuletzt besucht 20.März 2014. URL <https://github.com/>.

A. Heine and R. Nitschke. Wlan-antennenbau. zuletzt besucht 20.März 2014.
URL http://www.ronaldnitschke.de/index.php?main=antenna/007&rechts=antenna/antenna_r.

iFixit. iphone 4s teardown. 2014a. URL <http://www.ifixit.com/Teardown/iPhone+4S+Teardown/6610>.

iFixit. iphone 5 teardown. 2014b. URL <http://www.ifixit.com/Teardown/iPhone+5+Teardown/10525>.

A. Inc. What's new in xcode 5. *Apple Developer*, zuletzt besuch 20.März 2014. URL <https://developer.apple.com/xcode/>.

A. Inc. Clbeacon class reference. *iOS Developer Library*, zuletzt besucht 20.März 2014a. URL https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CLBeacon_class/Reference/Reference.html#/apple_ref/doc/uid/TP40013053.

- A. Inc. Clbeaconregion class reference. *iOS Developer Library*, zuletzt besucht 20.März 2014b. URL https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CLBeaconRegion_class/Reference/Reference.html#/apple_ref/doc/uid/TP40013054.
- A. Inc. Establishing a cocoa binding by control-dragging. zuletzt besucht 20.März 2014c. URL https://developer.apple.com/library/ios/recipes/xcode_help-interface_builder/articles-connections_bindings/EstablishingBindingsDragging.html.
- A. Inc. Introduction to core data programming guide. *iOS Developer Library*, zuletzt besucht 20.März 2014d. URL https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html#/apple_ref/doc/uid/TP30001200-SW1.
- A. Inc. About location services and maps. *iOS Developer Library*, zuletzt besucht 20.März 2014e. URL https://developer.apple.com/library/ios/documentation/userexperience/conceptual/LocationAwarenessPG/Introduction/Introduction.html#/apple_ref/doc/uid/TP40009497-CH1-SW1.
- A. Inc. ios developer program. zuletzt besucht 20.März 2014f. URL <https://developer.apple.com/programs/ios/>.
- A. Inc. Start developing ios apps today. zuletzt besucht 20.März 2014g. URL https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/index.html#/apple_ref/doc/uid/TP40011343.
- A. Inc. ios simulator user guide. zuletzt besucht 20.März 2014h. URL https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/Introduction/Introduction.html.
- A. Inc. Technische daten des iphone 4s. zuletzt besucht 20.März 2014i. URL <http://www.apple.com/de/iphone-4s/specs/>.
- A. Inc. Programming with objective-c. zuletzt besucht 20.März 2014j. URL https://developer.apple.com/library/mac/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html#/apple_ref/doc/uid/TP40011210-CH1-SW1.
- A. Inc. Auto layout guide. zuletzt besucht 20.März 2014k. URL https://developer.apple.com/library/ios/documentation/userexperience/conceptual/AutolayoutPG/Introduction/Introduction.html#/apple_ref/doc/uid/TP40010853-CH13-SW1.
- G. Inc. Google maps indoor-karten. zuletzt besucht 20.März 2014l. URL <http://www.google.com/maps/about/partners/indoormaps/>.
- A. Kushki, K. N. Plataniotis, and A. N. Venetsanopoulos. *WLAN Positioning Systems: Principles and Applications in Location-Based Services*. Cambridge University Press, 2012.

- N. Le Dortz, F. Gain, and P. Zetterberg. Wifi fingerprint indoor positioning system using probability distribution comparison. *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2012) : Kyoto, Japan, 25 - 30 March 2012 ; [proceedings]*, pages 2301–2304, 2013. URL <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6288374>.
- T. MacWright. Images as maps. 2014. URL <http://www.macwright.org/2012/08/13/images-as-maps.html>.
- Mapbox. Mapbox ios sdk. zuletzt besucht 20.März 2014a. URL <https://www.mapbox.com/mapbox-ios-sdk/>.
- Mapbox. Mapbox website. zuletzt besucht 20.März 2014b. URL <https://www.mapbox.com/>.
- Mapbox. Tilemill. zuletzt besucht 20.März 2014c. URL <https://www.mapbox.com/tilemill/>.
- D. R. Mautz. Indoor positioning technologies. 2012. URL <http://e-collection.library.ethz.ch/eserv/eth:5659/eth-5659-01.pdf>.
- E. Navarro, B. Peuker, and M. Quan. Wi-fi localization using rssi fingerprinting. URL <http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1007&context=cpesp>.
- 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>.
- raspberrypi.org. Raspberry pi website. zuletzt besucht 20.März 2014. URL <http://www.raspberrypi.org/faqs>.
- N. Ritter and M. Ruth. Geotiff format specification. 2014. URL <http://www.remotesensing.org/geotiff/spec/geotiffhome.html>.
- N. Semiconductor. nrf51822. zuletzt besucht 20.März 2014. URL <http://www.nordicsemi.com/eng/Products/Bluetooth-R-low-energy/nRF51822>.
- B. SIG. Bluetooth specification version 4.0[vol 0]. 2010. URL https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737.
- N. D. Wiki. Wibree. zuletzt besucht 20.März 2014. URL <http://developer.nokia.com/community/wiki/Wibree>.

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)