Bachelorarbeit

Reactive Bluetooth Low Energy Framework for iOS

von

Martin Stöber

Fachhochschule Lübeck
Fachbereich Informatik und Elektrotechnik

ma design GmbH & Co. KG
Abteilung Software Engineering

Betreuender Prüfer: Prof. Dr. Nane Kratzke

Zweitprüfer: Prof. Dr. Monique Janneck

Betreuer ma design: Dipl.-Inf. Eckhard Anders

Aufgabenbeschreibung

Bluetooth Low Energy[1] wurde bereits vor mehreren Jahren vorgestellt und standardisiert, findet aber erst allmählich eine breitere Aufmerksamkeit. Apples iBeacons[2] Technologie trägt dazu sicherlich ihren Teil bei. Ebenso wie ein verstärkt festzustellendes Interesse an Themenfeldern wie dem Internet der Dinge[3] oder bspw. Smart Home[4].

Technologien wie iBeacon beschränken sich aber primär noch auf die Indoor Standortbestimmung. Um die Bluetooth Low Energy Technologie darüber hinausgehend sinnvoll innerhalb des iOS Systems ausschöpfen zu können, wäre ein Framework sinnvoll, dass wiederkehrende Bluetooth Problemstellungen (wie bspw. Verbindungsmanagement und Datenübertragung) löst und mittels einer komfortablen API zugreifbar machen würde.

Ziel dieser Arbeit ist ein solches Framework zu entwickeln, dass die beiden Bluetooth LE Betriebsarten (Central und Periphal Modul) unterstützt. Das Framework soll ereignisbasiert[5] sein und wenn möglich dem Funktional-Reaktiven-Programmierparadigma[6] entsprechen. Sinnvolle Ereignisse (bspw. Statusänderungen, Entfernungsänderungen, Vorhandensein von UUIDs[7]) sollen im Rahmen einer Anforderungsanalyse abgeleitet und definiert werden.

Die Arbeit beinhaltet einen Nachweis der Machbarkeit anhand zweier Beispielimplementierungen für den Einsatz als Central und als Peripheral Modul.

[1] https://de.wikipedia.org/wiki/Bluetooth_Low_Energy
[2] https://de.wikipedia.org/wiki/IBeacon
[3] https://de.wikipedia.org/wiki/Internet_der_Dinge
[4] https://de.wikipedia.org/wiki/Smart_Home
[5] https://en.wikipedia.org/wiki/Event-driven_programming
[6] https://en.wikipedia.org/wiki/Functional_reactive_programming
[7] https://de.wikipedia.org/wiki/Universally_Unique_Identifier

Hierzu müssen im Detail folgende Teilaufgaben bearbeitet und dokumentiert (Bachelorarbeit) werden:

- Analyse des Problemraums und durchführen einer Anforderungsanalyse zur Ableitung sinnvoller Ereignisarten, Datenaustauscharten und -formaten, Interaktionsprotokollen für möglichst vielseitige ereignisbasierte Anwendungsfälle
- Entwicklung einer Architektur für ein ereignisbasiertes Framework
- Entwicklung einer geeigneten Evaluations- und Nachweisstrategie (QS)
- Implementierung inkl. Test/Nachweis des Frameworks anhand zweier Anwendungsfälle
  - Einsatz als Central Modul: Erstellung einer Mobile Applikation, die mittels des Frameworks auf Ereignisse (z.B. einem Feueralarm) von einem Bluetooth Sender (z.B. einen Feuermelder) reagiert (z.B. eine Notruf SMS an die Feuerwehr vorschlägt).
  - Einsatz als Peripheral Modul: Erstellung einer Applikation, die mittels des Frameworks Ereignisse (z.B. einen Feueralarm) und ergänzende Daten (bspw. einen Fluchtplan) versenden kann.
- Begleitende Dokumentation der oben angegebenen Schritte, Designentscheidungen und Ergebnisse (Bachelorarbeit)

Erklärung zur Bachelorarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

Lübeck, 10.03.2015

_____

Martin Stöber

Danksagung

Hiermit möchte ich mich bei Prof. Dr. rer. nat. Nane Kratzke für die für die gute Betreuung und hilfreichen Anmerkungen bei der Erstellung dieser Arbeit bedanken.

Des Weiteren möchte ich mich bei Dipl.-Inf. Eckhard Anders und bei allen weiteren Kollegen von ma design bedanken, die mich bei dieser Arbeit unterstützt haben und mir wertvolle Anmerkungen machten.

Zuletzt möchte ich auch den Korrekturlesern und den Personen danken, die bei der Umfrage zur Vorteilsevaluation teilgenommen haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programm-Listings

1 Einleitung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Entwicklung eines Frameworks zur reaktiven Steuerung von Bluetooth Low Energy für das Betriebsystem iOS von Apple. In diesem einleitenden Kapitel wird zunächst meine Motivation zur Erstellung dargelegt, die Aufgabenstellung erläutert und reflektiert, sowie der Aufbau beschrieben, um einen Überblick für diese Arbeit zu erhalten.

1.1. Motivation

Bluetooth ist seit Jahren eine sehr erfolgreiche Technologie die bereits in 2,5 Mrd. Geräten zur Anwendung kommt [BS14]. Besonders im Bereich der Consumer Hardware wird diese intensiv eingesetzt, um kabellose Übertragungen von Daten stattfinden zu lassen. Die Bluetooth Special Interest Group, eine Vereinigung aus mittlerweile über 800 Unternehmen, arbeitet ständig an Verbesserungen und Neuerungen [Wik14b]. Im Jahr 2007 wurde die Version 4 von Bluetooth veröffentlicht. Neben vielen Veränderungen und Verbesserungen wurde auch ein neuer Ansatz eingeführt. Dieser Bereich trägt den Namen Bluetooth Low Energy, im folgenden Bluetooth LE genannt. Ziel ist es, den Ressourcenbedarf auf ein Minimum zu reduzieren und eine effiziente Datenübertragung zu ermöglichen. Damit ergeben sich viele neue Anwendungsbereiche, da nun allein mit einer Knopfzelle Funkübertragungen über Jahre stattfinden können. Öffentlich zugänglich ist die Technologie seit dem Jahr 2009 und hat deshalb bereits jetzt große Verbreitung erlangt, auch wenn diese noch nicht aktiv genutzt wird.

In letzter Zeit zeigt sich ein Trend zur Vernetzung von Elektrogeräten im Haushalt. So präsentierten auf der Internationalen Funkausstellung (IFA) 2014 praktisch alle großen Hersteller von Haushaltsgeräten wie Bosch, Siemens, Samsung und LG Konzepte zur Steuerung über Mobilgeräte. Diese Entwicklung, auch Smart Home genannt, fusioniert sich immer weiter mit dem Konzept der „Internet der Dinge“. Diese Idee

hat das Ziel, selbst kleine elektronischen Geräte vernetzen zu können. Dabei ist das Potential noch lange nicht ausgeschöpft. Viele Einsatzszenarien befinden sich noch in der Entwicklung oder müssen erst gefunden werden.

Bluetooth LE hat, unter anderen, auch in diesem Bereich großes Potential. Zum einen bildet es mit dem geringen Energie- und Platzbedarf ideale Vorraussetzungen für dessen Einsatz und zum anderen könnten Geräte bereits jetzt durch die weite Verbreitung auch gesteuert werden.

Gesteuert werden sollen diese Geräte oft über Handhelds wie Smartphones und Tablets. Beim Blick auf die Verbreitung sind die Betriebsysteme Android, iOS und Windows Phone zu nennen. Und auch wenn Android mit einem Marktanteil von 80% zunächst am wichtigsten erscheint, so gibt es trotzdem gute Gründe sich zunächst auf Apples Betriebssystem iOS zu fokussieren [Sta14b]. Zum einen haben Marktforschungen ergeben, dass iPhone Nutzer eher höherpreisige Produkte erwerben und eher Trends folgen [Hei14]. Zum anderen ist bereits seit dem iPhone 4s, das 2012 erschien, Bluetooth LE eingebaut und lässt sich nutzen. Ein weiteres Argument ergibt sich aus dem unnötig komplexen Implementierungsaufwand von Bluetooth Low Energy unter iOS. Mit einem Framework könnte die Nutzung erleichtert und die zur Verfügung stehenden Möglichkeiten ausgereizt werden.

Eine effiziente und effektive Nutzung von Bluetooth Low Energy ist somit Hauptgegenstand dieser Arbeit.

1.2. Reflexion der Aufgabenstellung

Seit den letzten Monaten kommen vermehrt viele neue Produkte auf den Markt, die Bluetooth Low Energy zur Datenübertragung nutzen. Angefangen bei Fitnesstrackern bis hin zu elektrischen Zahnbürsten, die Produktpalette ist weit aufgestellt. Oft ist dabei das Ziel, dass sich diese Produkte mit dem iPhone oder dem iPad steuern lassen. Und fast alle Produkte haben eins gemeinsam: Die Bluetooth Steuerung muss immer von Grund auf neu entwickelt werden. Dabei ähneln sich, bedingt durch den Aufbau von Bluetooth LE, viele Anwendungsfälle.

Apple bietet durch diverse Restriktionen keinen vollen Zugriff auf die Hardware. Um Bluetooth-Funktionlitäten umzusetzen, muss man zwangsweise auf das zur Verfügung gestellte Framework CoreBluetooth aufsetzen. Dieses bietet aber zum einen

kaum Komfort bei dessen Nutzung und zum anderen wird durch die verwendete Softwarearchitektur der Quellcode schlecht nachvollziehbar und durch Fragmentierung schlecht wartbar. Es gibt zwar Ansätze von verschiedenen Entwicklern, die fehlende Funktionen ergänzen, die zusammenfassend aber nicht zufriedenstellend sind. Oft sind diese Implementierungen unvollständig, es fehlende Funktionen oder das Konzept schließt Anwendungsfälle aus, da nicht alle Betriebsarten vollständig umgesetzt sind.

Das zu entwerfende Framework soll mit bestehenden Einschränkungen umgehen und trotzdem einen Vorteil für den Nutzer schaffen. Es sollen keine von iOS unterstützten Anwendungsfälle ausgeschlossen sein und somit generische Anwendbarkeit besitzen. Durch die reaktive Architektur ergeben sich gute Möglichkeiten zur Steuerung und durch sinnvolle Funktionen soll zudem noch die Nutzung von Bluetooth Low Energy vereinfacht werden.

Besonderes Augenmerk wird auch auf die gute Nutzbarkeit des Frameworks gelegt. So soll mithilfe der Beispielimplementierung der Anwendungsfälle sichergestellt werden, dass das Framework einen echten Mehrwert bei der Implementierung bietet und sich gut in andere Projekte einbinden lässt. Außerdem wird mit verschiedenen Testarten sichergestellt, dass die Implementierung eine gute Testabdeckung besitzt, um die Softwarequalität zu erhöhen und außerdem eine langfristige Stabilität und Wartbarkeit zu ermöglichen.

1.3. Aufbau der Bachelorarbeit

Diese Arbeit ist in 6 Kapitel und einem Anhang gegliedert. Um einen besseren Überblick über die Struktur zu geben, sind zusammenfassend die Inhalte der folgenden Kapitel beschrieben.

Kapitel 2: Grundlagen

Im zweiten Kapitel werden die Grundlagen der eingesetzten Techniken beschrieben und die eingesetzten Programmierparadigmen erläutert. Die Entwicklung von Bluetooth bis hin zum gegenwärtigen Low Energy Konzept wird erläutert. Außerdem wird auch auf die Besonderheiten bei der Nutzung des Betriebsystems iOS eingegangen.

Kapitel 3: Anforderungsanalyse

Im folgenden dritten Kapitel werden die Analyse der Produktfunktionen, der Softwareanforderungen und zudem die generellen Design-Entscheidungen beschrieben. Außerdem werden die Produktfunktionen zu den jeweiligen Softwareanforderungen zugeordnet.

Kapitel 4: Architektur / Implementierung

Die durch die Ableitung aus der Anforderungsanalyse entwickelte Architektur der Software wird in Kapitel 4 dargestellt. Dazu werden alle relevanten Komponenten aufgeschlüsselt, Umsetzungentscheidungen erklärt und dargelegt, warum diese zum Einsatz kommen. Außerdem wird die Implementierung erläutert. Zuletzt werden Möglichkeiten zur Erweiterung des Frameworks aufgezeigt.

Kapitel 5: Test und Evaluation

Die entwickelte Test- und Nachweisstrategie wird in diesem Kapitel erläutert. Dazu werden die verschiedenen Testarten, Testfälle und das Konzept beschrieben, um sicherzustellen, dass die Anforderungen korrekt und vollständig umgesetzt wurden. Außerdem werden die Beispielimplementerierungen beschrieben und die Vorteile des Frameworks evaluiert.

Kapitel 6: Zusammenfassung

Das letzte Kapitel beinhaltet eine abschließende Reflexion der Arbeit. Wichtige Bestandteile und Entscheidungen werden dabei zusammenfassend bewertet.

Anhang

Im Anhang findet sich zum einen die Anleitung zum Einbinden des Frameworks in andere Projekte. Zum anderen werden die verwendeten Tools beschrieben und die Implementierungen für die Umfrage der Vorteilsevaluation dargestellt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, mit denen sich die Arbeit befasst. Zum einen wird hierbei Bluetooth als Technologie vorgestellt und auf die Besonderheiten der Version 4, insbesondere der Bereich Low Energy, eingegangen. Zum besseren Verständnis werden dabei auch die wichtigsten Begriffe erläutert, die im Verlauf der Arbeit genutzt werden. Des Weiteren werden die Besonderheiten im Umgang mit Bluetooth unter Apples Betriebssystem iOS erläutert. Außerdem wird auf das Event-basierte Progarmmierparadigma und auf dessen Spezialfall, die funktionale-reaktive-Programmierung, eingegangen, die in dieser Arbeit weite Anwendung finden.

2.1. Bluetooth

Mit dem primären Ziel, störende Kabelverbindungen zu umgehen, wurde die Funktechnologie Bluetooth entwickelt. Diese ermöglicht es, bei einer Vielzahl von Anwendungsfeldern Daten zwischen verschiedenen Geräten auszutauschen, ohne dass Sichtkontakt bestehen muss.

Bluetooth wurde im Jahr 1999 vorgestellt. Durch die Zusammenarbeit der Firmen Ericsson, IBM, Intel, Nokia und Toshiba wurde 1998 ein Interessenverband gegründet, welche an der Fortentwicklung der Technologie arbeitet. Mittlerweile gehören mehr als 24000 Unternehmen dieser so genannten Bluetooth Special Interest Group (SIG) an [BS14]. Seitdem wird durch stetige Weiterentwicklung das Ziel verfolgt, Datenübertragung zwischen Geräten zu erleichtern und zu verbessern. Dabei wird sich an aktuellen Nutzungsanforderungen orientiert, wie z.B. vernetzte Kleingeräte mit niedrigem Energiebedarf oder IP-Kommunikation [BS15b].

2.1.1. Bluetooth Versionen

Seit der Veröffentlichung entstanden bisher 4 Hauptversionen, die teilweise in Unterversionen erweitert worden sind. In der folgenden Tabelle ist ein kurzer Überblick über die Hauptentwicklungsschritte dargestellt:

Bluetooth Version	Beschreibung
1.x	Erste Version mit späterer Beseitigung von Sicherheitslücken und Verbesserung der Störempfindlichkeit
2.x	Erhöhung der Datenrate und zusätzliche von Verbesserungen beim Pairing
3.x	Erweiterung um einen HighSpeed-Kanal
4.x	Einführung von Bluetooth LE, Verbesserung der Fehlerkorrektur, Einführung von IP-Verbindungen

Tabelle 2.1.: Hauptversionen der Bluetooth-Entwicklung [Wik15a]

Während Bluetooth anfangs in ganz verschiedenen Anwendungsfällen genutzt wurde, so zeichnete sich in den letzten Jahren ein Trend zum hauptsächlichen Einsatz von 3 Anwendungsbereichen unter der Nutzung von Bluetooth 2.1 ab [Sau13, S. 355]:

- Kabellose Verbindung zwischen einem Smartphone und Audioendgeräten (Headsets, Freisprecheinrichtungen, Lautsprecher)
- Austausch von Dateien (z. B. Bildern) zwischen verschiedenen Endgeräten.
- Anbindung von kabellosen Tastaturen und anderen Eingabegeräten an Notebooks, PCs und Smartphones.

Mittlerweile entwickelt sich aber ein aufstrebender, neuer Einsatzbereich. Durch die Erweiterung von Bluetooth durch den Teil Low Energy können selbst kleine, batteriebetriebene elektronischen Geräte per Funk kommunizieren. Die Besonderheiten dieses neuen Teils der Bluetooth Spezifikation, die Funktionsweise und die Unterschiede zum klassischen Teil von Bluetooth, im folgenden Bluetooth classic genannt, werden im weiteren Verlauf dieses Kapitels erläutert.

2.1.2. Bluetooth Low Energy

Mit dem Ziel der Erschaffung von neuen Nutzungsszenarien im Bereich Sport, Fitness, Werbung, Entertainment und Sicherheit wurde 2010 Bluetooth LE als Teil der

Version 4.0 veröffentlicht. Dabei war essenzieller Bestandteil der Aufgabe, möglichst energieeffizient Daten, insbesondere von kleinen elektronischen Geräten, zu übertragen. Dazu wurden die bestehenden Konzepte von Bluetooth aufgegriffen, abgespeckt und überarbeitet [BS15b]. Es wurden dabei technische Details, die im folgenden Unterabschnitt 2.1.3 genauer erklärt sind, als auch der Aufbau und die Interaktionsweise, beschrieben in Unterabschnitt 2.1.4, verändert. Es entstand somit ein neuer Markt für drahtlose Sensoren, zum Beispiel im Bereich von Fitness und Gesundheit (z.B. drahtlose Blutdruckmessgeräte oder Herzfrequenzmessgeräte). Aber auch die Werbebranche und der Einzelhandel erhält mit der von Bluetooth LE möglichen Indoor-Lokalisierung weitere Möglichkeiten. Dazu werden Bluetooth LE Geräte, sogenannte Beacons, in Räumen verteilt und durch die Signalstärke kann der Standpunkt zum Smartphone bestimmt werden. Damit lassen sich zusätzliche Informationen zu Produkten oder Rabatte an Kunden senden und es könnte auf diese Weise sogar bezahlt werden [Val14]. In der Abbildung 2.1 ist ein solches Konzept zu erkennen. Ein weiterer aufstrebender Sektor ist die Hausautomatisierung. Heizungsregler, Lampen oder auch Hausgeräte können somit drahtlos über das Smartphone bedient werden.

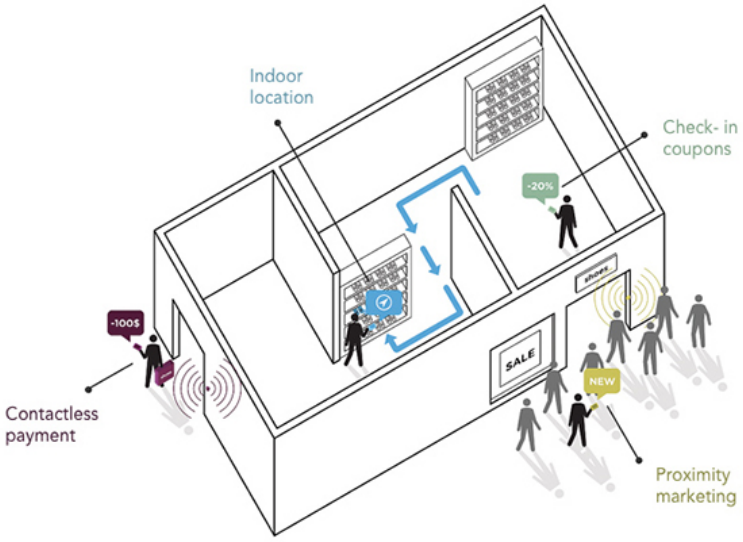

Abbildung 2.1.: Anwendungsszenario von Bluetooth LE im Einzelhandel [Val14]

2.1.3. Technische Daten

Seit der Einführung von Bluetooth 4.0 wird Hardware in verschiedenen Konfigurationen angeboten. Dabei können Geräte entweder eines der beiden oder beide Teile von Bluetooth 4.0 gleichzeitig unterstützen. Geräte, die beide Teile unterstützen, werden

mit dem „Bluetooth Smart Ready“ Logo gekennzeichnet. Geräte, die nur den Low Energy Bereich beherrschen, werden mit mit dem „Bluetooth Smart“ Logo versehen. Diese können sich dann nur mit einem anderen Bluetooth Smart oder Smart Ready Gerät verbinden. Die Verbindung zu Bluetooth classic ist nicht möglich [BS15a].

In der folgenden Tabelle 2.2 sind die wichtigsten technischen Daten der beiden Bluetooth-Typen zusammengefasst:

Technische Spezifikation	Bluetooth Classic	Bluetooth LE
Reichweite	100 m	$>$ 100 m
Maximale Datenrate	1–3 Mbit/s	1 Mbit/s
Nutzbare Datenrate	0.7–2.1 Mbit/s	0.27 Mbit/s
Latenz	100 ms	6 ms
Energieverbrauch	1 W (Referenzwert)	0.01W - 0.5 W
Maximale Geräteverbindungen	7	unbegrenzt

Tabelle 2.2.: Technische Daten von Bluetooth classic und Bluetooth LE [Wik14a]

Wie man anhand der Tabelle sehen kann, ähneln sich viele Daten. Besonderes auffällig sind aber die Unterschiede bei der Datenrate und der Latenz. Sowohl die Übertragungsrate, als auch die Latenz sind bei Bluetooth LE sehr gering. Dies ist damit begründet, dass die Technologie das Ziel hat, kleine Daten möglichst verzögerungsfrei und stromsparend zu übertragen und danach wieder in einen Ruhezustand zu gehen. Einen großen Unterschied kann man auch beim Stromverbrauch von Bluetooth LE beobachten. Durch das strikte und vorrangige Ziel der effizienten Datenübertragung wurde der Stromverbrauch, besonders im Ruhezustand, auf ein Minimum reduziert. Um dieses Konzept beizubehalten ist auch bei der Implementierung in dieser Arbeit darauf zu achten, möglichst schnell wieder in den Ruhezustand zu wechseln. Auch die Begrenzung auf maximal 7 aktive Verbindungen entfällt, sodass die Implementierung eine performante Verwaltung von vielen Geräten gleichzeitig ermöglicht werden muss.

2.1.4. Aufbau und Terminologie

Bluetooth LE unterscheidet sich neben den technischen Daten auch im Aufbau und der Verwendung von Bluetooth classic. Im Folgenden werden der Aufbau und weitere wichtige Begriffe erklärt, die in der Arbeit verwendet werden.

2.1.4.1. Betriebsarten

Bei der Verbindung von Bluetooth Geräten bildet das *Generic Access Profile (GAP)* die Grundlage, die eine Zusammenfassung von Funktionalitäten darstellt. Dieses *Profile* ist dafür zuständig, auf welche Art Geräte miteinander kommunizieren und sich miteinander verbinden. Des Weiteren dient dieses *Profile* auch als Grundlage für alle anderen Profile bei Bluetooth. Bei der Nutzung von Bluetooth LE sind damit vier verschiede Betriebsarten verfügbar. Zum einen gibt es den *Advertise*-Modus. In diesem Modus kann ein Gerät kontinuierlich Daten aussenden, um Geräte, die im zweiten Modus, dem *Scan*-Modus sind, zu benachrichtigen. Damit kann dem Ziel eines Verbindungsaufbaus nachgegangen werden. Ist dieser erfolgreich, bekommen die anderen beiden Betriebsarten Relevanz. Der dritte Modus nennt sich *Peripheral*-Modus. Dabei verhält sich das Gerät als Server, welcher Informationen zur Verfügung stellt. Normalerweise sind *Peripheral*-Geräte, im folgenden *Peripheral*-Module genannt, kleine batteriebetriebene Geräte, die sich mit anderen Geräten im *Central*-Modus verbinden, um Informationen wie z. B. die Temperatur zu übertragen. Ein *Central*-Module ist dabei der Client, dass diese Informationen empfängt und ggf. auswertet, wie beispielsweise das Smartphone oder Tablet. Die Betriebsarten sind auch parallel nutzbar. Semantisch sind *Advertise*- mit *Peripheral*-Modus und *Scan*- mit *Central*-Modus zusammengehörig [Tow14].

2.1.4.2. Konzept

Während die Nutzer bei Bluetooth classic die verschiedenen Anwendungsszenarien über die verschiedenen Profile ausführen (Beispielsweise auf das *Advanced Audio Distribution Profile (A2DP)*, um Musik zu übertragen), so hat man bei Bluetooth LE nur die Möglichkeit, auf das *Generic Attribute Profile (GATT)* zuzugreifen [Pfü14]. Dieses *Profile* definiert das Konzept, wie Daten ausgetauscht werden. *GATT* kommt dabei aber erst zum Einsatz, sobald eine Verbindung mittels *GAP* zwischen zwei Geräten zustande gekommen ist. Wie in [Pfü14] und [Tow14] beschrieben definiert *GATT* dabei ein Konzept auf 3 aufeinander aufbauenden Objekten, genannt *Attributes*:

Service

Ein *Service* stellt eine Sammlung von zusammengehörigen *Characteristics* dar und kann zusätzlich noch weitere *Services*, sogenannte *secondary Services*, beinhalten.

Characteristic

Eine *Characteristic* stellt einen bestimmten Wert wie zum Beispiel die Temperatur dar. Die Daten einer *Characteristic* selbst sind gewöhnliche Bytefolgen, die mithilfe der *Descriptors* oder durch das Einhalten von Konventionen richtig interpretiert werden können, um so zum Beispiel Text zu übertragen. Eine *Characteristic* kann dabei mehrere *Descriptors* besitzen.

Descriptor

Ein *Descriptor* beinhaltet eine zusätzliche Information zu einer *Characteristic*. Für die Temperatur-*Characteristic* könnte das beispielsweise die Einheit (z.B. Celsius) sein.

GATT benutzt intern das *Attribute Protocol (ATT)*. Dieses dient zur Speicherung der *Attributes (Services, Characteristics und Descriptors)* in einer simplen Lookup-Tabelle[1]. Damit wird das Suchen und Finden von Services und das Schreiben und Lesen von *Characteristics* und *Descriptors* ermöglicht. Alle *Attributes* werden bei der Kommunikation über *GATT* immer mit zuweisbaren Indikatoren angesprochen. Diese sind nach dem *Universally Unique Identifier (UUID)* Standard definiert, müssen aber nicht einzigartig sein, da die Attributes alle in der Lookup-Tabelle von *ATT* mit einem einzigartigen 16Bit Identifizierer, genannt *Handle*, abgespeichert werden [Pfü14]. Es gibt allerdings eine Liste der Bluetooth Special Interest Group, in der spezielle *UUID*s für definierte Zwecke vorgegeben sind.

2.1.4.3. Datenaustausch

Ein Peripheral-Module, das Daten bereitstellen möchte, muss mindestens ein *Service* mit einer *Characteristic* bereitstellen. Damit kann dann eine *Central*-Module nach diesem *Service* suchen und die *Characteristic* auslesen. In der Abbildung 2.2 kann man den Aufbau eines *Peripheral*-Modules sehen, das einen Thermometer darstellt.

Ein Central-Module hat verschiedene Möglichkeiten die Daten des Peripheral-Modules zu erhalten. Zum einen kann es über das *GATT*-Protokoll mittels einer Leseaufforderung den Wert erhalten, zum anderen kann das Peripheral-Modul auch über eine

[1] Tabelle mit statischem Inhalt zur Refenzierung von Datensätzen und zur Vermeidung von Neuberechnungen.

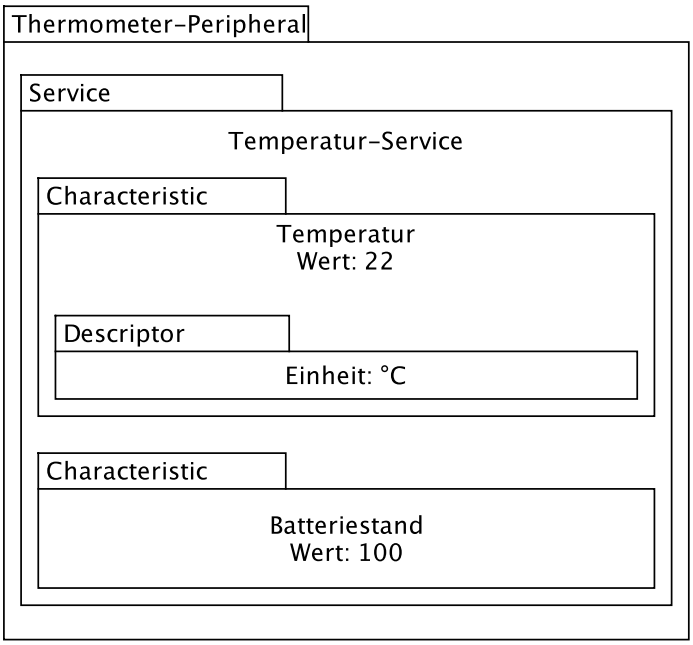

Abbildung 2.2.: Attributes-Struktur eines Thermometer-Peripheral

Wertveränderung informieren. Dazu abonniert sich das Central-Modul die *Characteristic*. Mittels einer so genannten *Indication* oder *Notification* kann das Peripheral-Module den Wert dann weitergeben. Bei einer *Indication* verlangt das *Peripheral* eine Empfangsbestätigung, bei einer *Notification* nicht [Wik14a].

Auch das Verändern des Wertes einer *Characteristic* kann von beiden Modulen aus geschehen. Das Peripheral-Modul kann dabei den Wert einfach aktualisieren. Das Central-Module kann eine Schreibaufforderung senden. Dies kann mit und ohne einer Bestätigung erfolgen.

2.2. Bluetooth unter iOS: CoreBluetooth

Das Unternehmen Apple ist in vielen Bereichen sehr restriktiv, was die Nutzung der Hardware bei deren iOS Geräten angeht. Damit wird das Ziel verfolgt, durch Beschränkungen mehr Kontrolle über die Entwicklungen für die Geräte zu erhalten, um sicherzustellen, dass der Nutzer immer eine gleichbleibend gute und erwartungskonforme User Experience erhält. Dies hat den Vorteil, dass besser auf die Funktionalitäten eingegangen werden und Optimierungen, wie beispielsweise im Strombedarf, getroffen werden können. Allerdings entstehen dadurch unter Umständen größere

Probleme spezielle Funktionalitäten umzusetzen, die nicht im Fokus des Herstellers liegen. Eine Möglichkeit, diese Beschränkungen zu umgehen, ist das Betriebsystem mittels eines so genannten Jailbreakes[2] zu öffnen und somit vollen Zugriff auf Dateisystem und Hardware zu erlangen. Projekte wie BTstack[3] ermöglichen dabei den vollständigen Zugriff auf die Bluetooth Hardware, sodass man sehr viel freier entwickeln kann und nicht an die Vorgaben von Apple gebunden ist. Der Nachteil hierbei ist allerdings, dass Apps, die einen Jailbreak voraussetzen, nicht in den AppStore[4] aufgenommen werden, da dies von Apple nicht gestattet wird [Sad09, S. 58f].

Aus diesem Grund wird auch in dieser Arbeit auf die Verwendung eines Jailbreaks verzichtet und es werden nur öffentlich freigegebene Schnittstellen genutzt, um auf die Hardware zuzugreifen. Durch diese Entscheidung sind insbesondere die Funktionen von Bluetooth classic eingeschränkt. Man hat bis auf wenige, spezielle Anwendungsfälle (wie zum Beispiel Musik-Streaming) noch immer kaum Zugriff auf Hardware und Steuerung [App13]. Die einzige Möglichkeit auf mehr Kontrolle ist die Inanspruchnahme des MFi[5] Programm von Apple. Dieses Programm bietet Zugriff auf weitere Tools, Dokumentationen und Support, ist allerdings kostenpflichtig und dessen Inhalte unterliegen einem Geheimhaltungsvertrag. Welche Funktionalitäten genau mit diesem Programm verfügbar werden, ist daher öffentlich nicht bekannt [App14c].

Glücklicherweise hat das aber auf die Funktionalität des zu entwickelnden Frameworks kaum Einfluss, da Apple seit dem Realase von iOS Version 5 das so genannte CoreBluetooth Framework eingeführt hat. Dieses öffentlich verfügbare Framework bietet die Möglichkeit, auf die grundlegenden Bluetooth LE Funktionalitäten zuzugreifen. Damit kann man auf einem höheren Level Funktionen umsetzen. CoreBluetooth ist also eine Abstraktionsstufe über den *Profiles GAP, GATT* und *ATT*, um auf die Grundfunktionen von Bluetooth LE zuzugreifen. In der ersten Version wurden zunächst die Möglichkeit eingeführt, sich mit anderen Bluetooth LE Geräten zu verbinden und diese zu steuern. Mit iOS Version 6 wurde dann noch hinzugefügt, dass das iOS-Gerät selbst als Peripheral-Modul fungieren kann [App14b]. Somit sind beide Betriebsarten von Bluetooth abgedeckt, womit das Framework für diese Arbeit nutzbar ist. Umgegangen wird dabei mit den Hauptkomponenten (*Peripheral, Central, Service, Characteristic* und *Descriptor* der Bluetooth LE *Profiles*), die als Objekte abgebildet werden. Mithilfe von Managern kann man diese Objekte steuern.

[2] Beim Jailbreak wird eine Schwachstelle im Betriebsystem ausgenutzt, um nicht autorisierte Funktionen auszuführen.
[3] https://code.google.com/p/btstack/
[4] Plattform zum Vertrieb von Anwendungen
[5] https://developer.apple.com/programs/mfi/

Wie auch bei Bluetooth classic ergeben sich aber auch mit der Nutzung von CoreBluetooth einige Beschränkungen, die allerdings weit weniger kritisch ausfallen. So lässt sich kein unbegrenzter Scan im Hintergrund durchführen und auch die Scan- oder Advertise-Zeitintervalle sind nicht einstellbar. Außerdem hat CoreBluetooth einige konzeptionelle Schwächen. Genauer wird dies in Abschnitt 3.5 noch einmal erläutert.

Insgesamt muss zur Umsetzung von Bluetooth LE Funktionalitäten zwangsweise das CoreBluetooth-Framework verwendet werden, wenn man nicht auf mithilfe eines Jailbreaks entsperrte Geräte setzt.

2.3. Event-basierte Programmierung / Funktional-reaktive Programmierung

Das Konzept der Event-basierten Programmierung beruht darauf, dass die Programmsteuerung durch Ereignisse, den Events, gesteuert wird. Diese können z.B. durch Benutzereingaben, Netzwerkanfragen oder durch Nachrichtenaustausch anderer Programmteile ausgelöst werden [Wik14d]. Dabei bestehen solche Systeme immer aus zwei Arten von Bausteinen: Ereignisquellen -und Senken. Die Kommunikation erfolgt dabei unidirektional von Quelle zur Senke, wobei die Kommunikationbeziehung meist erst zur Laufzeit hergestellt wird [Sta14a, S. 153].

Eine gute Beschreibung für ein Event-basiertes Programm wird in [PBB04] definiert.

> „Der Kontrollfluss (d.h. die Reihenfolge der Ausführungsschritte) eines Informatik-Systems wird hauptsächlich durch Ereignisse und nicht durch Kontrollstrukturen oder Funktionsaufrufe bestimmt."
>
> (Prof. Bernd Brügge, Ph.D. in [PBB04])

Dieses Konzept kann auch verschiedenen Ausprägungen eingesetzt werden. Zum besseren Verständnis des Frameworks wird im Folgenden deshalb zunächst auf die Eventsysteme von iOS eingegangen.

2.3.1. Eventsysteme bei iOS

Besonders seit dem Aufkommen von grafischen Benutzeroberflächen hat die Verwendung von Events stark zugenommen. Auch bei der Entwicklung von Software für die

Plattform iOS kommt man bereits sehr schnell in Kontakt mit Events. Dabei sind diese als verschiedene Ausprägungen implementiert. In [Ebe14] sind dabei häufige Umsetzungen aufgezählt und werden im Folgenden zum besseren Verständnis kurz erläutert:

Callbacks Aufruf einer übergebenen Funktion sobald das Ereignis eintritt [Wik15b].

Delegates Übergabe einer Zuständigkeit in ein anderes Objekt bei Eintritt eines Ereignisses [Wik14c].

Key-Value-Observing Mechanismus zur Überwachung einer Wertänderung eines Property[6] [App12a].

Actions Kommunikationsmechanismus zwischen User-Interface-Elementen und Objekten [App12b].

Diese beschriebenen Umsetzungen werden aber bewusst nicht für die Implementierung des Frameworks gewählt. Verwendet wird stattdessen das Objective-C Framework ReactiveCocoa[7], das noch weitere Vorteile bietet. Es basiert auf dem funktional-reaktiven Programmierparadigma, ein Spezialfall der beschriebenen Event-basierten Programmierung. Im folgenden Unterkapitel wird dieser Spezialfall erläutert, das Framework an sich erklärt und die für diese Arbeit relevanten und verwendeten Komponenten beschrieben.

2.3.2. ReactiveCocoa

ReactiveCocoa wurde von GitHub[8] entwickelt. Es ist ein sehr mächtiges Framework, dessen Hauptaufgabe es ist, funktional-reaktive Programmierung bei iOS und OS X zu unterstützen. Dabei bietet es als größten Vorteil die Möglichkeit, eventbasierte Umsetzungen, wie in Unterabschnitt 2.3.1 beschrieben, zu vereinheitlichen und komfortabel zu nutzen [Git15]. So können beispielsweise sowohl *Delegates* als auch *Key-Value-Observing* in das Event-System von ReactiveCocoa umgewandelt werden.

Das Besondere der funktional-reaktiven Programmierung ist die Kombination des Datenflusses der reaktiven Programmierung mit der Nutzung von Elementen der funk-

[6] Ein Property ist ein Klassenattribut bei Objective-C mit automatisch generierten Getter/Setter-Methoden und anpassbarer Zugriffsregelung und Speicherverwaltung.
[7] https://github.com/ReactiveCocoa/ReactiveCocoa
[8] https://github.com/

tionalen Programmierung. Spezieller ist dabei der Kontrollfluss und dessen Event-Bearbeitung kombiniert mit Funktionen höherer Ordnung gemeint, also Funktionen, die eine Funktion als Argument erhält [Wik15c].

Als Grundstein dient dabei immer ein sogenanntes Signal, das `RACSignal`. Ein Signal stellt ein Objekt dar, das drei verschiedene Event-Typen versenden kann: *next*, *error* und *completed*. Diese Typen repräsentieren gleichzeitig auch den Zustand des Signals. Mit einem *next*-Event sendet das Signal immer einen Wert mit, der sich verändert hat. Mit *error* wird ein Fehlerfall dargestellt und ein *completed* sagt aus, dass das Signal beendet wurde und keine Werte mehr folgen [Tho13].

Durch das anteilig funktionale Konzept können die *next*-Werte modifiziert werden. Dabei können Signale und ihre Werte verkettet, gesplittet oder umgewandelt werden. Außerdem können Seiteneffekte zu einem Signal hinzugefügt werden, was bedeutet, dass man sich bei einem Signal registriert und dann Aktionen ausführt, wenn Event-Typen bei dem Signal auftreten.

In Abbildung 2.3 ist dieses Konzept schematisch abgebildet.

In dieser Abbildung sind 2 Signale zu sehen. Diese senden zwei unterschiedliche Objekte zu indeterminitschen Zeitabständen aus. Mit der `-combineLatest:reduce:`-Methode werden diese beiden Signale zusammengefasst und zusätzlich noch mit `-filter:`gefiltert. Diese beiden Operationen werden durch das funktionale Prinzip ermöglicht. Zusätzlich wird danach noch ein reaktiver Seiteneffekt auf das *completed*-Event gelegt, sodass, sobald beide Signale in den diesen Zustand übergehen, „*Signals completed*" ausgegeben wird.

ReactiveCocoa ist durch das beschriebene Konzept besonders gut geeignet, um zum einen asynchrone Netzwerkkommunikationen zu implementieren. Zum anderen kann es ein stabiles und vor allem einheitliches Eventsystem abbilden und wird deshalb zur Umsetzung des Bluetooth Frameworks genutzt. Genauer wird diese Auswahl in der Analyse (Abschnitt 3.5) beschrieben.

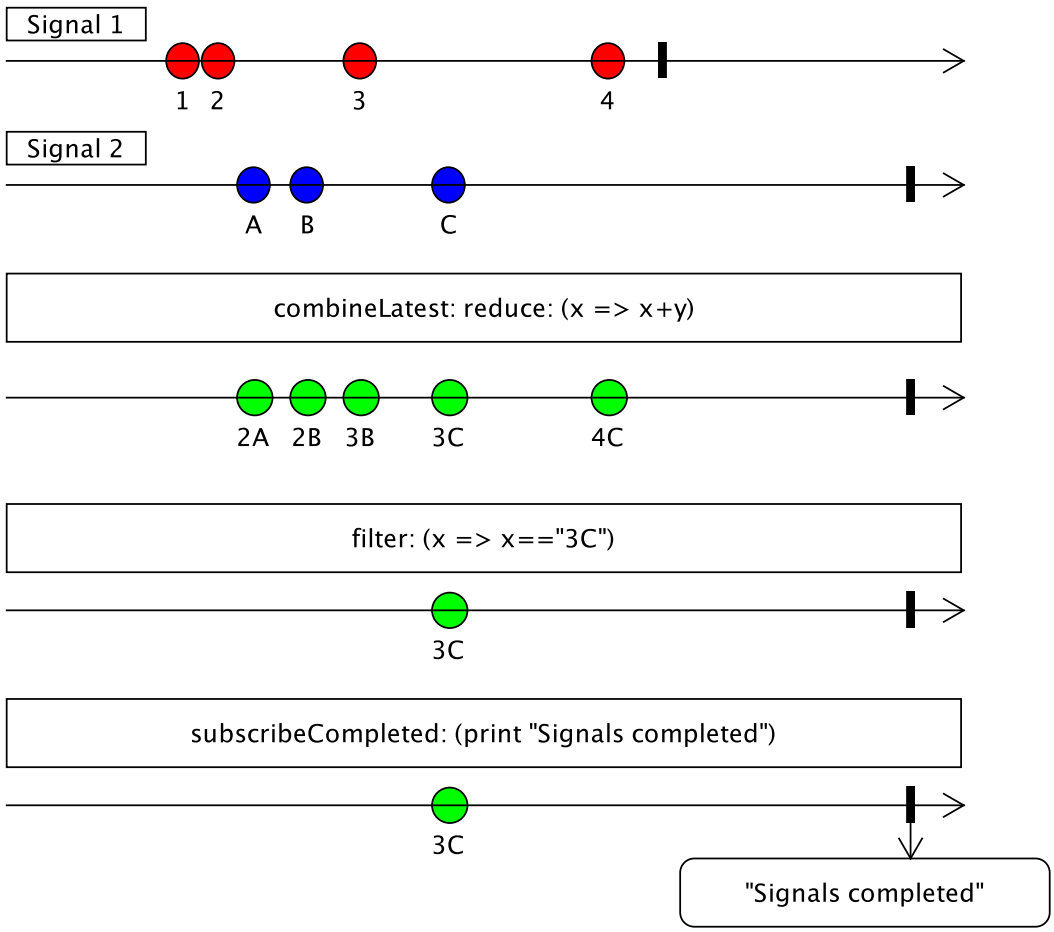

Abbildung 2.3.: Beispielhafte Nutzung von Signalen mit ReactiveCocoa

3 Anforderungsanalyse

Durch Analyse des Problemraums wird im Folgenden die Software konzipiert und spezifiziert. Die daraus gewonnen Produktfunktionen werden an Softwareanforderungen gekoppelt. So erhält man eine vollständige Liste an umzusetzenden Anwendungsteilen. Jede Produktfunktion sowie jede Softwareanforderung wird durch eine eindeutige Kennung versehen, damit auch in späteren Kapiteln darauf Bezug genommen werden kann. Im weiteren Verlauf werden außerdem generelle Entscheidungen beschrieben und auch auf nicht-funktionale Anforderungen eingegangen.

3.1. Anforderungen an das Framework

Bluetooth LE ist eine Technologie bei der, ohne Eingaben des Nutzers, Änderungen im Systemkontext auftreten können. Das liegt zum einen daran, dass nutzungszielbedingt möglich ist, dass Geräte sich verbinden, die Verbindung trennen, oder sich die Daten der *Attributes* ändern. Zum anderen liegt das an der Funkverbindung und dessen relativ hohe Störanfälligkeit gegenüber kabelbasierten Verbindungen. So kann es schnell zu Schwankungen der Signalstärke bis hin zu Verbindungsabbrüchen kommen. Um mit diesen verteilten Systemen umgehen zu können stellt die größte Herausforderung dar, ein geeignetes Format zum Nachrichtenaustausch zu finden. Auf den unteren Schichten ist dies bereits durch die Bluetooth-Spezifikation definiert. Um nun das Ziel der komfortablen Nutzung von Bluetooth LE zu erreichen, ist die Hauptaufgabe, auf den oberen Schichten effizient kommunizieren zu können. Wichtig ist hierbei, Komplexität zu reduzieren und zu ermöglichen, dass mit wenig Aufwand die Verbindungen gesteuert werden.

Das zu implementierende Framework soll jegliche Verbindungssteuerung übernehmen. Dabei sollen alle Möglichkeiten der Steuerung, die iOS bei Bluetooth LE unterstützt, aufgegriffen werden. Dabei stellt man fest, dass es eine begrenzte Anzahl an Aktionen

gibt, die eintreten oder ausgelöst werden können. Diese sind zudem noch eindeutig identifizierbar. Sie sind in Unterabschnitt 3.3.1 aufgelistet.

Eine gute Methode zur Steuerung ist, das Eintreten solcher Ereignisse als Events auszudrücken, wie in Abschnitt 2.3 beschrieben. Bei der Nutzung von Events geben sich viele verschiedene Möglichkeiten zur Umsetzung. Die Wahl fiel dabei auf die Nutzung des Frameworks ReactiveCocoa. Die genaue Analyse, wieso genau diese Variante zur Umsetzung genommen wurde, ist im Kapitel Abschnitt 3.5 beschrieben. Eine genauere Erklärung zur ReactiveCocoa findet sich im Unterabschnitt 2.3.2 der Grundlagen.

3.2. Produktfunktionen

Das oberste Ziel des Frameworks ist die komfortable Nutzung der Bluetooth Low Energy Technologie. Dabei sollte allerdings nicht außer Acht gelassen werden, dass vielseitige Anwendungsfälle auch zu verschiedenen Anforderungen führen. Mit einem generischen Blick auf alle Schnittstellen des Frameworks soll somit sichergestellt werden, dass keine Einsatzzwecke ausgeschlossen werden.

Insgesamt soll sich bei der Umsetzung auf die Betriebsarten konzentriert werden, um von diesem Standpunkt alle Funktionen abzuleiten.

PF01: Advertise-Modus

Geräte können Datenpakete versenden, um auf die eigene Präsenz hinzuweisen. Dabei ist es auch möglich, benutzerdefinierte Daten mitzusenden, ohne dass eine eigentliche Verbindung aufgebaut werden muss.

PF02: Scan-Modus

Im *Scan*-Modus können Geräte gefunden werden, die sich im *Advertise*-Modus befinden. Dabei soll es auch möglich sein, die benutzerdefinierten Daten auszulesen.

PF03: Peripheral-Modus

Das *Peripheral*-Modul hat Daten, die anderen Geräten zur Verfügung gestellt werden können. Dazu muss die *Attributes*-Hierarchie aufgebaut werden können auf der *Services*, *Characteristics* und *Descriptors* erzeugt und gruppiert

werden. Außerdem soll man die Berechtigungen setzen und mit Verbindungsanfragen umgehen können oder auch *Indications* beziehungsweise *Notifications* versenden.

PF04: Central-Modus

Das *Central*-Modul stellt sich als Client auf und verwaltet die Verbindungen zu anderen *Peripheral*-Modulen. So soll es nach einem Verbindungsaufbau möglich sein, diese zu steuern. Dazu ist es nötig, die *Services*, *Characteristics* und *Descriptors* zu durchsuchen und abzurufen und diese ggf. auszulesen. Außerdem soll die Datenübertragung gesteuert werden, indem man sich auf *Characteristics* registriert oder den Wert liest oder setzt.

3.3. Funktionale Anforderungen

Die funktionalen Anforderungen spiegeln die eigentlich nutzbaren Funktionen des Frameworks nach außen wider. Diese sind durch Analyse der Produktfunktionen und durch Identifizierung der Interaktionsmöglichkeiten von Bluetooth LE entstanden.

3.3.1. Anforderungsliste

Es ist somit eine Liste mit 15 Anforderungen entstanden, die in dem Framework umgesetzt werden.

FA01 Bluetooth Hardware Status überwachen

FA02 Scannen nach Bluetooth LE Geräten

FA03 Bekannte Bluetooth LE Geräte verwalten

FA04 Verbindungen aufbauen und trennen

FA05 *Advertise*-Daten senden

FA06 *Services* hinzufügen und entfernen

FA07 *Characteristics* hinzufügen und entfernen

FA08 *Descriptors* hinzufügen und entfernen

FA09 *Indications/Notifications* versenden

FA10 *Services* auslesen

FA11 *Characteristics* und dessen Werte auslesen

FA12 Werte von *Characteristics* ändern

FA13 Bei *Characteristics* registrieren

FA14 *Descriptors* und dessen Werte auslesen

FA15 Signalstärke und Geräteabstand bestimmen

3.3.2. Zuordnungstabelle

Zur besseren Übersicht und zum Prüfen der Vollständigkeit wurde die folgende Zuordnungstabelle 3.1 erstellt.

Funktionale Anforderung → Produktfunktion	PF01	PF02	PF03	PF04
FA01 (Bluetooth Hardware Status überwachen)	x	x	x	x
FA02 (Scannen nach Bluetooth LE Geräten)		x		
FA03 (Bekannte Bluetooth LE Geräte verwalten)				x
FA04 (Verbindungen aufbauen und trennen)				x
FA05 (*Advertise*-Daten senden)	x			
FA06 (*Services* hinzufügen und entfernen)			x	
FA07 (*Characteristics* hinzufügen und entfernen)			x	
FA08 (*Descriptors* hinzufügen und entfernen)			x	
FA09 (*Indications/Notifications* versenden)			x	
FA10 (*Services* auslesen)				x
FA11 (*Characteristics* und dessen Werte auslesen)				x
FA12 (Werte von *Characteristics* ändern)				x
FA13 (Bei *Characteristics* registrieren)				x
FA14 (*Descriptors* und dessen Werte auslesen)				x
FA15 (Signalstärke und Geräteabstand bestimmen)		x		x

Tabelle 3.1.: Zuordnung der funktionalen Anforderungen an die Produktfunktion

Daraus ist zu erkennen, dass [FA01] und [FA15] mehreren Produktfunktionen zugeordnet sind. Das ist damit begründet, dass die Überwachung des Hardwarestatus

[FA01] Grundvoraussetzung des Betriebs aller Produktfunktionen darstellt und das Bestimmen der Signalstärke bzw. des Geräteabstands [FA15] sowohl beim Scannen nach Geräten, als auch bei einem verbundenen Gerät möglich sein soll. Alle weiteren Anforderungen sind jeweils einer Produktfunktion zugeordnet.

3.4. Nicht-Funktionale Anforderungen

Nicht nur die funktionale Umsetzung ist wichtig um sicherzustellen, dass effizient und effektiv mit dem Framework gearbeitet werden kann. Auch die nicht-funktionalen Bereiche der Software legen fest, wie gut eine Software arbeitet. Dazu wurden im Folgenden 4 wichtige Anforderung aus diesem Bereich aufgestellt:

NF01: Einfache Integration in Projekte

Das Framework soll effizient in andere Projekte integrierbar sein. Dazu soll es mithilfe eines Dependency Managers[1] eingebunden werden können. Alle Abhängigkeiten sollen dabei aufgelöst und mit in das Projekt geladen werden.

NF02: Gute Performance

Durch die Nutzung des Frameworks sollen die Geschwindigkeiten beim Verbindungsmanagement, der Datenübertragung, der Datenüberwachung, etc. gegenüber der reinen Nutzung von CoreBluetooth nicht verschlechtert werden. Außerdem sollen Programme niemals durch die Nutzung des Frameworks einfrieren oder instabil werden.

NF03: Generische Nutzbarkeit

Wie bereits erwähnt sollen mit dem Framework keine Nutzungsszenarien ausgeschlossen werden. Das Framework soll so modular aufgebaut werden, dass jeder Nutzer auf alle Funktionen Zugriff hat und keine Anforderungen blockiert werden.

NF04: Gute Dokumentation

Alle zur Verfügung stehenden Funktionen sollen durch Kommentare im Quellcode aussagekräftig dokumentiert werden, sodass immer leicht nachvollziehbar bleibt, welche Auswirkungen jede Funktion hat.

[1] Anwendung zum automatischen Auflösen zu externen Abhängigkeiten, wie z.B. externe Frameworks

3.5. Generelle Designentscheidungen

Eine der wichtigsten Entscheidungen bei der Entwicklung des Frameworks ist die Auswahl eines geeigneten Eventsystems. Wie bereits in den Grundlagen (Abschnitt 2.3) beschrieben, werden bei iOS bereits viele verschiedene Arten eingesetzt, die auch dieses Event-basierte Framework einsetzen könnte. CoreBluetooth nutzt beispielsweise das *Delagate*-Pattern, um die asynchronen Abläufe umzusetzen, was aber zu schlecht wartbarem und unstrukturiertem Quellcode führt, da so alle Operationen in einem *Delegate*-Objekt verarbeitet werden und man ständig Fallunterscheidungen durchführen muss. Auch der eigentliche Programmfluss ist nicht einfach nachzuvollziehen, da selbst einfache Verbindungsanfragen in verschiedenen Bereichen verteilt sind.

Deshalb wurde das Framework ReactiveCocoa ausgewählt.

Durch die Architektur von Bluetooth LE und die Kombination mit ReactiveCocoa hat man noch eine Möglichkeit, das einfache Event-Prinzip geschickt aufzuwerten. Aufgrund dessen, dass die Ausprägungen von Bloototh LE Geräten einem nutzungsabhängigen *Attributes*-Schema entsprechen, könnte man diese einzelnen *Attribute*-Objekte und deren Werte als Signale von ReactiveCocoa betrachten, die sich in unbestimmten Zeitintervallen verändern. Der große Vorteil hierbei ist noch, dass sich diese Signale auch kombinieren, splitten oder umwandeln, also funktional verändern lassen. Somit wird noch mehr Flexibilität erzielt und man kann mit komplexeren Abhängigkeiten leichter und effizienter umgehen. Damit wären dann von außen getriggerte Operationen abgedeckt. Aber auch die von innen gesteuerten Methoden, wie z.B. ein Verbindungsaufbau, könnten so Abgebildet werden, dass ein Signal den Zustand des Verbindungsaufbaus repräsentiert und die *completed-* oder *error*-Events das Ergebnis abbilden.

Außerdem können dadurch noch folgende Vorteile erreicht werden. Es sollen dabei die Zuständigkeiten an die eigentlich zuständigen Objekte zurückgegeben werden. Dabei wäre zum Beispiel eine *Characteristic* zu nennen, bei der man einen Wert direkt bei dem *Characteristic*-Objekt setzen soll und nicht erst über einen den zentralen `CBPeripheralManager` von CoreBluetooth. Des Weiteren lässt das funktionale Konzept das Kombinieren oder Filtern von Events zu. Denn häufig ist es nötig, dass mehrere Vorbedingungen gegeben sein müssen, damit eine Aktion erfolgreich ausgeführt werden kann. Auch dafür stehen mit ReactiveCocoa gute Lösungen, z.B. das Kombinieren von Signalen, bereit.

Ein nennenswerter Nachteil bei der Verwendung von ReactiveCocoa ist, dass das Konzept eher speziell und noch nicht sehr weit verbreitet ist und daher eine Einarbeitungszeit voraussetzt, sowie das richtige Verständnis der Entwickler erfordert. Dies sollte allerdings kein Problem darstellen, da sowohl das zu entwickelnde Framework als auch ReactiveCocoa gut dokumentiert ist.

Auf Grund der genannten Vorteile werden überall, wo es sinnvoll erscheint, also zumeist bei asynchronen Netzwerkoperationen (z.B. Verbindungsaufbau) oder von außen ausgelösten Ereignissen (z.B. Verbindungsanfragen), die Komponenten von ReactiveCocoa verwendet.

4 Architektur / Implementierung

Durch Ableitung der vorangegangenen Analyse wird im folgenden Kapitel die Architektur und die daraus resultierende Implementierung beschrieben. Dabei wird der Aufbau und das entwickelte Konzept erklärt, es wird auf die eingesetzten Methoden eingegangen und beschrieben, wie die Anforderungen umgesetzt wurden. Außerdem wird noch die Erweiterbarkeit berücksichtigt.

4.1. Generelle Architekturentscheidungen

Das Framework wird in der Programmiersprache Objective-C erstellt. Diese ist seit der Veröffentlichung des iPhones die offizielle von Apple genutzte Sprache. Die auf der Entwicklerkonferenz WWDC 2014 neu vorgestellte Programmiersprache Swift wird aufgrund der nicht vollständigen Kompatibilität zu ReactiveCocoa nicht verwendet. Zur Umsetzung wurden die Entwicklungsumgebungen AppCode 3.1[1] von Jetbrains und Xcode 6[2] von Apple genutzt.

iOS mit der Version 7 wird als Mindestvoraussetzung zum Betrieb des Frameworks angenommen. Obwohl CoreBluetooth schon mit iOS 5 eingeführt wurde, ergaben sich in den folgenden Versionen teils massive Änderungen der API[3]. Erst seit Version 6 ist es möglich, als *Peripheral*-Gerät zu fungieren. Seit iOS 7 gab es Änderungen beim Zugriffsschutz der Hardware, die eine wichtige Rolle spielen. Mit dem SDK 7 werden bereits 96% der Geräte abgedeckt [App14a]. In der momentan aktuellen Version 8 gibt es zudem kaum signifikante Änderungen, sodass Version 7 ein guter Kompromiss zwischen Aktualität und Kompatibilität darstellt.

[1] https://www.jetbrains.com/objc/
[2] https://developer.apple.com/xcode/
[3] Schnittstelle eines Programmteils, das zur Interaktion mir dem System zu Verfügung gestellt wird.

Wie in Abschnitt 3.1 beschrieben, wird ReactiveCocoa genutzt, um die reaktiven Bestandteile des Frameworks zu erstellen. Es werden alle Datenflüsse, die sich kontinuierlich ändern können, oder Ereignisse, die auftreten können, mit den Möglichkeiten von ReactiveCocoa abgedeckt.

Zudem wird ein großes Augenmerk auf Kapselung gelegt. Alle nur intern verwendeten Daten werden mithilfe von Anonymous Categories [4] so gekapselt, dass nach außen (in den öffentlichen Header-Dateien) nur das für den Nutzer des Frameworks verwendbare Interface gezeigt wird.

Alle Klassen erhalten das Präfix „RBT“, da Objective-C keine Namespaces[5] unterstützt. So soll verhindert werden, dass bei der Verwendung mehrerer Frameworks doppelte Klassennamen entstehen.

Des Weiteren werden Funktionalitäten, die nicht in jedem Anwendungsfall gebraucht werden (z.B. die Abstandsbestimmung), in Class Categories[6] ausgelagert.

Als Strukturgrundlage für die Entwicklung wird CoreBluetooth genutzt. Allerdings sollen die bereits erwähnten Schwächen durch das funktional-reaktive Konzept ausgemerzt werden.

Um [NF01] (Einfache Integration in Projekte) umzusetzen, wird der Dependency Manager Cocoapods genutzt. So lässt sich das Framework einfach in andere Projekte einbinden, und die Abhängigkeit zu ReactiveCocoa wird gleichzeitig elegant mit aufgelöst. Außerdem werden dadurch auch eventuelle Bugfixes von ReactiveCocoa automatisch mit eingebunden. Zudem erhält man auch im eigenen Framework eine gute Versionsverwaltung beim Verteilen von Updates. Dazu wird ein sogenannter Podspec[7] erzeugt. Zusätzlich wird noch ein Header erstellt, der alle weiteren Klassen importiert, sodass nur eine Zeile notwendig ist, um das Framework in eine Klasse einzubinden. Wie man das Framework in Projekte einbindet, wird im Anhang (Abschnitt A.1) erklärt.

Zur besseren Verständlichkeit des Frameworks wird zudem noch eine Klassendokumentation in HTML-Form generiert. Diese basiert auf den Kommentaren, die auf-

[4] Sprachkonstrukt in Objective-C, um ein interne Implementierungen zu erweitern, ohne Implementierungdetails im öffentlichen Header zu deklarieren.
[5] Pfadstruktur zu eindeutigen Unterscheidung von Namen
[6] Sprachkonstrukt in Objective-C, um Klassen zu erweitern oder Funktionalitäten semantisch zu trennen
[7] Beschreibungsdatei zum Konfigurieren eines Frameworks für Cocoapods

grund der nichtfunktionalen Anforderung NF04 umgesetzt werden. Diese Dokumentation ist auf der beiliegenden CD im Verzeichnis `Framework/Documentation/` zu finden.

4.2. Systemarchitektur

Das Framework wird intern auf CoreBluetooth zugreifen, um alle Bluetooth LE Funktionalitäten umzusetzen. Dazu wird ein Wrapper[8] aufgebaut, der die nach außen reaktive Steuerung zulässt und trotzdem die Apple Richtlinien erfüllt, um für den App-Store zugelassen zu werden, da nur die offizielle API zur Steuerung von Bluetooth verwendet wird.

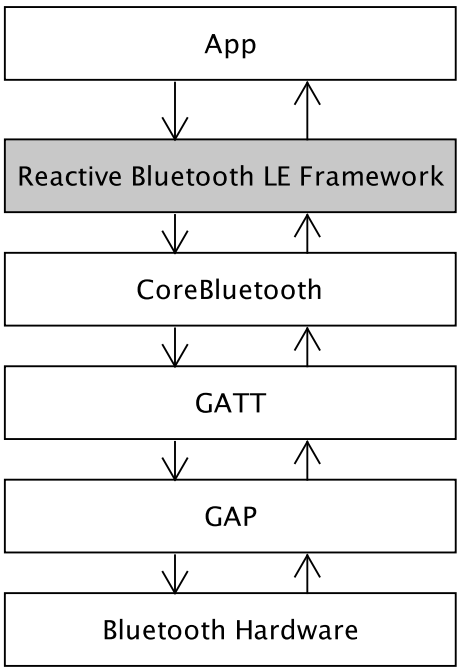

Abbildung 4.1.: Grundlegende Systemarchitektur des Frameworks

In Abbildung 4.1 ist die Struktur zu sehen, mit der das Framework umgesetzt wird. Die ganze Kommunikation zwischen App und Bluetooth LE verläuft über das Framework. Intern wird dann, wie beschrieben, auf die API von CoreBluetooth zugegriffen, welche hardwarenah über die Bluetooth LE *Profiles* mit anderen Geräten kommunizieren kann.

[8] Als Wrapper wird eine Software bezeichnet, wenn sie eine andre Software umschließt und somit eine Schnittstelle in eine andere Schnittelle umwandelt. [Wik15d]

Zur Verwendung des Frameworks werden zwei Ansatzpunkte zur Verfügung gestellt. Als Einstiegspunkt der Produktfunktionen [PF02] und [PF04] wird die Klasse `RBTCentralModule` verwendet. Um die Funktionalitäten von [PF01] und [PF03] zu nutzen, steht die Klasse `RBTPeripheralModule` zur Verfügung.

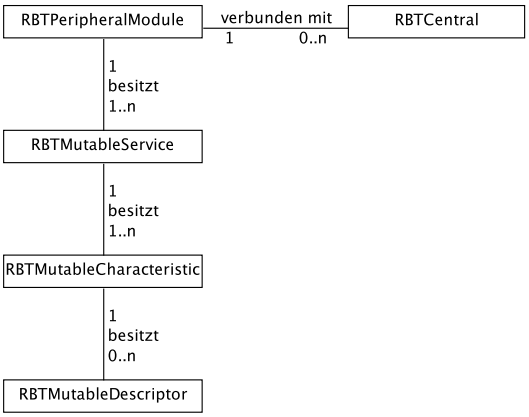

Abbildung 4.2.: Klassendiagramm für PF01 und PF03

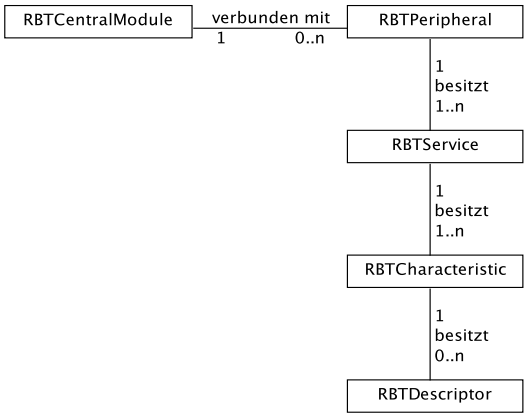

Abbildung 4.3.: Klassendiagramm für PF02 und PF04

Die Klassendigramme in Abbildung 4.2 und Abbildung 4.3 stellen eine grundlegende Übersicht über die Klassen-Struktur des Frameworks dar, welche auf den Erkenntnissen der Anforderungsanalyse entstanden sind. Auch sind die eben erwähnten 2 Ansatzpunkte zu erkennen. Die Objekt-Struktur ist sowohl an CoreBluetooth und damit auch an die Bluetooth-*Attributes*-Struktur angelehnt. Das `RBTPeripheralModule` erstellt eine *Attributes*-Struktur und bekommt Verbindungsanfragen von externen *Centrals*. Im Gegenzug dazu verbindet sich das `RBTCentralModule` zu externen *Peripherals* und steuert diese.

4.3. Umsetzung

In den folgenden Unterkapiteln werden die erstellten Klassen näher beschrieben. Dabei wird vor allem auf das API des Frameworks eingegangen.

4.3.1. Central-Module

Das *Central*-Module stellt den Einstiegspunkt der Betriebsarten *Scan* und *Central* bereit, die zu den Produktfunktionen [PF02] und [PF04] gehören. Diese Klasse ermöglicht es, Verbindungen zu externen Bluetooth-*Peripherals* aufzunehmen und diese zu verwalten [FA02, FA03]. Gesucht werden kann nach allen verfügbaren Geräten [FA02] mit der Methode `-(RACSignal *)scan`. Zudem kann die Suche mit weiteren Methoden eingeschränkt werden. Beispielsweise liefert `-(RACSignal *)scanWithMaxRSSI:(NSNumber *)RSSI` nur Geräte mit der angegebenen maximalen Signalstärke. Als Rückgabewert bekommt man ein Objekt vom Typ `RACSignal`. Mit diesem Signal werden die gefundenen Geräte kontinuierlich übergeben.

Die Klasse bietet zudem mehre Properties, die den Zustand der Bluetooth-Hardware [FA01] und den Scanzustand abbilden. Außerdem verwaltet das *Central*-Module verbundene und auch ehemals verbundene *Peripherals* [FA03]. Dazu kann man die zur Verfügung gestellten `-(RACSequence *)retrieve[...]` - Funktionen nutzen. Beim Aufruf der Methode `-(RACSequence *)retriveConnectedPeripherals` erhält man nacheinander alle mit dieser Klasseninstanz verbundenen Geräte und kann diese mittels der funktionalen Bestandteile von ReactiveCocoa verändern, um z.B. eine Filterung nach Namen durchzuführen.

4.3.2. Peripheral

Die Klasse `RBTPeripheral` repräsentiert ein entferntes Bluetooth LE Gerät im *Advertise*- oder *Peripheral*-Modus. Mit dem *Central*-Module kann man durch Nutzung der `-scan[...]` oder `-retrive[...]`-Methoden dieses Objekt erhalten und kann das entfernte Gerät nun steuern.

Es werden die Methoden `-(RACSignal *)connect` und `-(RACSignal *)disconnect` zum Verbinden und Trennen des Gerätes angeboten [FA04].

Im folgenden Codeausschnitt ist die Implementierung der connect-Methode zu sehen:

Listing 4.1: Connect-Methode der Klasse RBTPeripheral

```
1 - (RACSignal *)connect {
2     [self.centralModule.cbCentralManager connectPeripheral:self.cbPeripheral
          options:nil];
3
4     @weakify(self)
```

```
5     RACSignal *connectSignal = [RACSignal createSignal:^RACDisposable *(id <
          RACSubscriber> subscriber) {
6         @strongify(self)
7
8         // Check for error
9         RACDisposable *failDisposable =
10                [[[self.centralModule.peripheralConnectionFailedSignal filter:^
                      BOOL(RACTuple *failedTuple) {
11                    @strongify(self)
12                    RACTupleUnpack(CBCentralManager *manager, CBPeripheral *
                          failedPeripheral) = failedTuple;
13                    return (manager == self.centralModule.cbCentralManager &&
                          failedPeripheral == self.cbPeripheral);
14                }] take:1] subscribeNext:^(RACTuple *failedTuple) {
15                    NSError *error = [failedTuple third];
16                    [subscriber sendError:error];
17                }];
18
19        // Check for success
20        RACDisposable *successDisposable =
21                [[[self.centralModule.peripheralConnectedSignal filter:^BOOL(
                      RACTuple *connectedTuple) {
22                    @strongify(self)
23                    RACTupleUnpack(CBCentralManager *manager, CBPeripheral *
                          connectedPeripheral) = connectedTuple;
24                    return (manager == self.centralModule.cbCentralManager &&
                          connectedPeripheral == self.cbPeripheral);
25                }] take:1] subscribeNext:^(RACTuple *connectedTuple) {
26                    [subscriber sendCompleted];
27                }];
28
29        return [RACDisposable disposableWithBlock:^{
30            // cleanup signals
31            [failDisposable dispose];
32            [successDisposable dispose];
33        }];
34    }];
35
36    return connectSignal;
37 }
```

Wie man sehen kann, werden die durch Signale ersetzten *Delegate*-Methoden für die Ergebnisse des Verbindungsaufbaus (Erfolg- oder Fehlerfall) zusammengefasst und abonniert. Danach wird ein neues Signal mit dem Ergebnis erzeugt und zurückgegeben. So wird nach Nutzung der Methode asynchron das Ergebnis als Signal zurückgeben. Mit dem Erstellen von Seiteneffekten für Signal-Events *Completed* oder *Error* kann auf dieses Ergebnis reaktiv reagiert werden. Die Klasse enthält außerdem Methoden, um die Signalstärke auszulesen. Mit der Class Extension `RBTPeripheral+Beacon` ist es zudem noch möglich, den Geräteabstand in Metern auszulesen [FA15].

4.3.3. Peripheral-Module

Zum Erzeugen eines Bluetooth-*Peripherals*, mit dem sich *Centrals* verbinden können, wurde die Klasse `RBTPeripheralModule` implementiert. Diese Klasse bietet die Möglichkeit zum Erzeugen einer *Attributes*-Struktur (siehe Konzept). Dafür wird die Methode `-(RACSignal *)addService:(RBTMutableService *)service` genutzt, um einen *Service* mit *Characteristics* hinzuzufügen [FA06, FA07]. Nachdem man diese Struktur erzeugt hat, ist es möglich, mit der Methode `-(RACSignal *)startAdvertising` in die Betriebsart *Advertise* zu schalten [FA05] und somit mit dieses *Peripheral* bekannt zu machen.

Des Weiteren ist das *Peripheral*-Module dafür zuständig, Verbindungsanfragen manuell aufzulösen. Um dem Anwender des Frameworks die unnötige wiederkehrende Bearbeitung von Standardverbindungsanfragen, wie z.B. das Auslesen eines Wertes einer *Characteristic*, abzunehmen, werden diese automatisch erledigt. Wenn allerdings eine manuelle Auflösung der Verbindungsanfragen gewünscht ist, kann dies mit der Methode `-(void)respondToRequest:(CBATTRequest *)request withResult:(CBATTError)result` erledigt werden. Dabei wird der `request` bearbeitet und mit einem `result` (z.B. `CBATTErrorRequestNotSupported` bei nicht unterstützten Anfragen) wieder an den Sender geleitet.

4.3.4. Central

Die Klasse `RBTCentral` stellt ein Bluetooth LE Gerät in der Betriebsart *Central* dar. Es bietet die Methode `-(void)setDesiredConnectionLatency:(CBPeripheralManagerConnectionLatency)latency` um die Latenz zwischen den Verbindungen explizit anzugeben. Eine automatische Auswahl der Latenz wird allerdings bereits durch iOS getroffen und wurde deshalb nur aufgrund der Vollständigkeit implementiert. Außerdem enthält diese Klasse das Property `maximumUpdateValueLength`, mit dem die maximal unterstütze Bytelänge beim Senden einer *Indication* oder einer *Notification* erhalten werden kann.

4.3.5. Service

Um den Produktfunktionen zu entsprechen, wurde bei der Implementierung des *Service* zwischen 2 Arten unterschieden. Zum einen wird der *Service* eines entfernten

Bluetooth-*Peripherals* von dem *Service* für den Einsatz der Klasse `RBTPeripheral-Module` unterschieden.

Die Klasse `RBTService` stellt einen *Service* eines entfernten *Peripherals* dar. Dabei ist es möglich, die *Characteristics* mit der Methode `-(RACSignal *)discover-AllCharacteristics` auszulesen [FA11]. Außerdem werden bereits ausgelesene *Characteristics* verwaltet. Die Klasse `RBTMutableService` dagegen entspricht einem *Service*, der auf dem lokalen Gerät für das *Peripheral*-Module erzeugt wird. Der Unterschied dabei ist, dass Methoden implementiert wurden, die es ermöglichen, *Characteristics* oder *secondary Services* hinzuzufügen oder zu löschen [FA07]. Außerdem kann man die UUID des *Service* festlegen.

4.3.6. Characteristic

Characteristics sind ebenfalls in 2 Klassen geteilt. Mit der Klasse `RBTCharacteristic` kann der Wert einer *Characteristic* eines entfernten Bluetooth-*Peripherals* ausgelesen und verändert werden [FA11, FA12]. Unterschieden wird bei dem Ändern des Wertes zwischen einer Änderung mit und ohne Bestätigung. Beim Ändern eines Wertes mit Bestätigung erhält man ein Signal, das im Erfolgsfall ein *completed*-Event sendet. Außerdem kann die *Characteristic* mit der Methode `-(RACSignal *)setNotifyingStatus:(BOOL)notifying` abonniert werden, sodass Wertänderungen automatisch übertragen werden [FA13].

`RBTCharacteristic` besitzt zudem das Property `valueSignal`, welches ein Signal für den Wert darstellt, um ggf. Seiteneffekte auf eine Wertänderung zu legen.

Die Klasse `RBTMutableCharacteristic` wurde implementiert, um eine *Characteristic* für das *Peripheral*-Module zu erzeugen. Beim Erzeugen kann man eine UUID wählen und die Zugriffsberechtigungen des Wertes festlegen. Außerdem können *Descriptors* hinzugefügt und entfernt werden [FA08]. Mit der Methode `-(BOOL)notifySubscribedCentrals` lassen sich verbundene *Centrals* über eine Wertänderung benachrichtigen (*Indication/Notification* [FA09]).

4.3.7. Descriptor

Auch das *Attribute*-Objekt *Descriptor* lässt sich in zwei Implementierungen aufteilen. Die Klasse `RBTDescripor` entspricht einem *Descriptor* eines entfernten *Peri-*

pherals. Sowohl der Wert als auch die *UUID* eines solchen *Descriptors* kann beliebig sein. Zum Auslesen des Wertes wurde die Methode `-(RACSignal *)readValue` und zum Setzen die Methode `-(RACSignal *)writeValue:(NSData *)data` implementiert [FA14]. Beide Methoden liefern ein Signal zurück, bei dem das Ergebnis der Operation als Event-Type gesendet wird. Bei der Auslese-Methode wird zusätzlich auch der ausgelesene Wert als `next`-Event zurückgegeben. Im Gegensatz dazu gibt es beim Erstellen eines *Descriptors* für ein *Peripheral*-Module Einschränkungen. Bei der Erstellung eines `RBTMutableDescriptor`s werden aufgrund von iOS-Restriktionen nur zwei verschiedene selbst erstellbare *Descriptors* (Characteristic User Description oder Characteristic Presentation Format[9]) unterstützt. Zum Erstellen wurde die Methode `-(instancetype)initWithUUID:(CBUUID *)UUID value:(id)value` implementiert.

4.4. Erweiterbarkeit

Das entwickelte Framework deckt zunächst alle Anwendungsszenarien ab, die mit Bluetooth LE unter iOS realisiert werden können. Zur einfacheren Nutzbarkeit von standardisierten Anwendungsfällen ergibt sich aber eine Verbesserungsmöglichkeit.

Standardisierte *Services*, *Characteristics* und *Descriptors* können vordefiniert werden. Die Bluetooth SIG hat bereits eine ganze Reihe solcher Anwendungsfälle spezifiziert. So könnte beispielsweise ein Human Interface Device (HID[10])-*Service*[11] soweit vorbereitet sein, dass man keine eigene *Attributes*-Struktur aufbauen muss, sondern nur Methoden zum Daten setzen nutzt und somit das HID steuern kann. So würde die interne Bluetooth Struktur weiter abstrahiert werden. Es ist aber mit einem hohen Implementierungsaufwand verbunden, sodass sich diese Erweiterung aus Zeitmangel in dieser Arbeit nicht umsetzen ließ.

Diese und andere Erweiterungsmöglichkeiten würden sich aber durch den modularen Aufbau des Frameworks, beispielsweise durch Vererbung oder mit Class Extensions, gut integrieren lassen.

[9] Weitere Beschreibung dazu: https://developer.apple.com/library/prerelease/ios/documentation/CoreBluetooth/Reference/CBMutableDescriptor_Class/index.html#//apple_ref/occ/instm/CBMutableDescriptor/initWithType:value:

[10] Bezeichnung für ein Eingabegerät zur Computersteuerung (z.B. eine Tastatur)

[11] https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.human_interface_device.xml

5 Tests und Evaluation

In diesem Kapitel wird erläutert, wie und welche Tests genutzt werden, um das Framework zu prüfen. Des Weiteren wird durch eine Beispielimplementierung festgestellt, wie gut sich das Framework in externen Projekten einsetzen lässt. Zuletzt wird evaluiert, welche Vorteile das entwickelte Framework bietet, und zusätzlich dargestellt, dass das Framework zielerfüllend einen Mehrwert bietet.

5.1. Test des Frameworks

Der Nachweis zur Erfüllung der Spezifikation stellt zunächst die Grundvorraussetzung fehlerfreier Software dar. Die im Folgenden näher beschriebe Teststrategie basiert zum einen auf Unit-Tests und zum anderen auf halbautomatischen System-Tests. Dies ist sinnvoll, um die einzelnen Komponenten und die unterschiedlichen Bestandteile genau und vollständig zu testen und auch die Besonderheiten der Bluetooth Funktechnologie zu berücksichtigen. Mit diesen beiden Testarten kann eine komplette Testabdeckung der Spezifikation des Framework sichergestellt werden. Dabei basieren die Tests auf den in Unterabschnitt 3.3.1 definierten Anforderungen. Besondere Spezialfälle, wie zum Beispiel Hardwareinkompatibilitäten, können aus zeitlichen Gründen in dieser Arbeit allerdings nicht getestet werden.

Alle Tests sind im Verzeichnis `/Framework/Tests/` auf der beiliegenden CD enthalten.

5.1.1. System Tests

Bluetooth ist eine Technologie, die davon lebt, Verbindungen bei unterschiedlichen Umgebungssituationen aufzubauen oder Daten per Funk zu übertragen. Deshalb kann

man sich auf keinen Fall nur auf Unit-Tests verlassen, da verteilte Komponenten zusammenarbeiten müssen und so Unit-Testfälle keine zufriedenstellende Aussagekraft beinhalten würden. Viele Probleme zeigen sich erst bei der eigentlichen Kommunikation wie bei Datenübertragungen, Verbindungauf- und abbau oder Zustandsänderungen. Aus diesem Grund werden Systemtests durchgeführt, um die Anforderungen zu testen, die nicht mit Unit-Tests abgedeckt werden können. Man könnte diese Tests vollautomatisierten, was den Vorteil mit sich bringen würde, sehr effizient und schnell alle Testfälle durchführen zu können. Dabei müsste man auch die externen Geräte steuern können. Durch den hohen Aufwand ist dies aber in dieser Bachelorarbeit nicht umsetzbar, weshalb auf halbautomatische Tests zurückgegriffen wird. Hierbei werden alle Testfälle genau spezifiziert. Ein Gerät als Kommunikationspartner wird dann von Hand in vorgegebene Zustände gebracht. Mit dem Framework werden diese Testfälle in geeigneter Weise umgesetzt. So ergeben sich verifizierbare und vergleichbare Testergebnisse.

Um die halbautomatischen System-Tests umzusetzen, wurden zunächst die Tests spezifiziert. Die Tabelle 5.1 zeigt übersichtlich alle Testfälle, die mit den Systemtests umgesetzt wurden.

Testnummer	Beschreibung	Anforderung
ST01	Bluetooth Hardware Status prüfen	FA01
ST02	Scannen nach Peripherals	FA02
ST03	Verbinden/Verbindung trennen zu Peripherals	FA04
ST04	Advertise Daten senden	FA05
ST05	Indication/Notification versenden	FA09
ST06	Services durchsuchen	FA10
ST07	Characteristic und dessen Wert auslesen	FA11
ST08	Wert einer Characteristic ändern	FA12
ST09	Bei einer Characteristic registrieren	FA13
ST10	Descriptor und dessen Wert auslesen	FA14
ST11	RSSI und Geräteabstand bestimmen	FA15

Tabelle 5.1.: Testfälle der System-Tests

Danach wurde die Durchführung geplant. Dabei stellte sich heraus, dass eine Anwendung zur Ausführung und ein Kommunikationspartner für die Durchführung der Tests notwendig ist. Es wurde daraufhin eine iOS App für die Testfälle entwickelt. Außerdem wurde eine Anwendung, genannt *SystemTestHelper*, geschrieben, die den System-Tests als Kommunikationspartner dient und alle Vorraussetzungen für alle Testfälle enthält. Beide Anwendungen werden im Folgenden weiter erläutert.

SystemTestHelper

Der *SystemTestHelper* wurde für die Platform MacOS X entwickelt, damit der Entwickler kein zweites iOS Gerät benötigt und so die Tests auf seinem Entwicklungscomputer ausführen kann. Es ist nur ein Mac mit Bluetooth 4.0 Unterstützung notwendig. Die Anwendung (in der Abbildung 5.1 zu sehen) enthält nur einen Button. Wenn dieser geklickt wird, wird der Computer so konfiguriert, dass dieser als Kommunikationspartner für die im Folgenden beschriebenen System-Test App fungiert und die Vorraussetzungen aller System-Testfälle gegeben sind.

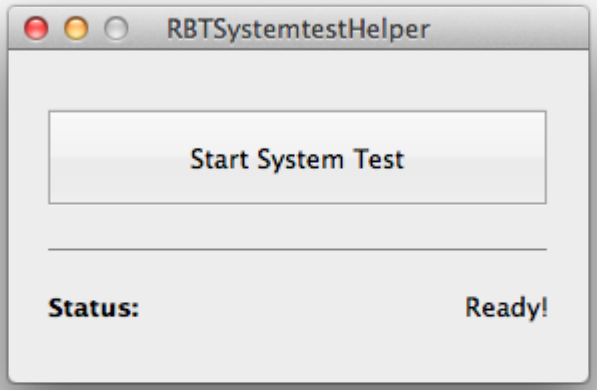

Abbildung 5.1.: User Interface des SystemTestHelpers

System-Test App

Zum Ausführen der eigentlichen System-Tests wurde eine iOS Anwendung erstellt. Diese stellt alle Testfälle übersichtlich in einer Tabelle dar (Abbildung 5.2). Nach Auswahl eines Tests erhält man eine Beschreibung über den Testfall und dessen Anforderungen. Anschließend kann man den Test starten und erhält eine Rückmeldung, ob der Test erfolgreich war. Außerdem hat man noch die Möglichkeit, die Log-Ausgabe eines Tests anzusehen, um z.B. einen Fehlschlag eines Tests zu analysieren. Die Oberfläche der Anwendung nach der erfolgreichen Ausführung ist in Abbildung 5.3 zu sehen.

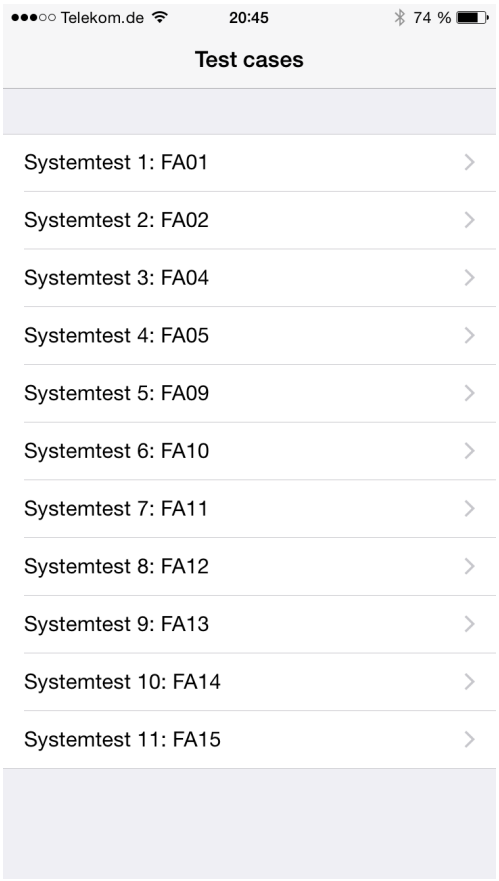

Abbildung 5.2.: System-Test Applikation mit Auswahlliste der Tests

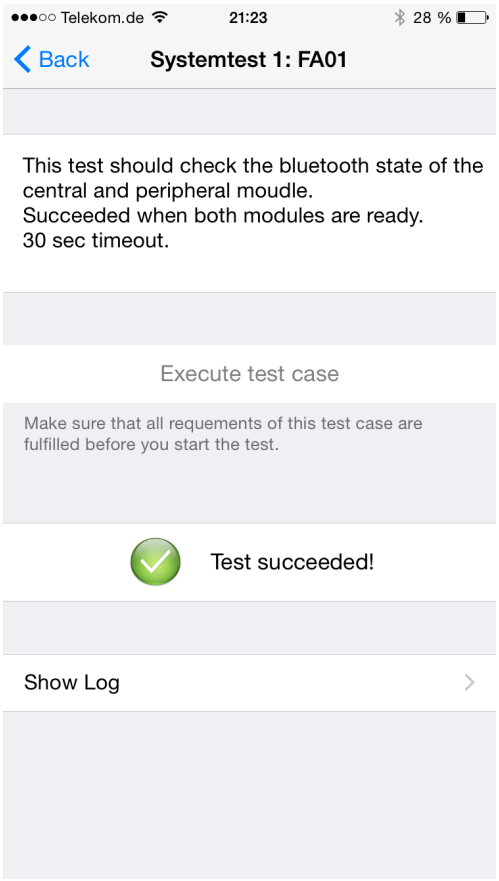

Abbildung 5.3.: User Interface der System-Test App nach der erfolgreichen Ausführung von ST01

Intern ist die App sehr generisch aufgebaut, damit sich schnell neue Testfälle hinzufügen lassen. Dazu wurde ein Protocol[1] erstellt, welcher jeder Testfall implementieren muss. In Abbildung 5.3 ist ein UML-Klassendiagramm dieses Protocols zum besseren Verständnis zu sehen und in Systemtest ST03 ist der Test für ST03 dargestellt.

[1]Ein Protocol bezeichnet bei der Programmiersprache Objective-C einen speziellen Header mit Deklarationen von Methoden, die eine Klasse implementieren muss, wenn die diesen nutzt. (Das Konzept ähnelt dem des *Interface* von Java)

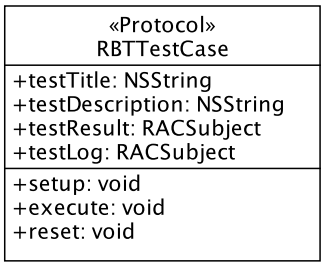

Abbildung 5.4.: UML-Klassendiagramm Test-Protocol

Listing 5.1: Systemtest ST03

```
1  @implementation RBTTest3
2  //[...]
3  - (void) setup {
4      self.testTitle = @"Systemtest 3: FA04";
5      self.testDescription = @"Connect and disconnect to a peripheral. 30sec
           timeout. Req: There should be a peripheral within range.";
6      //[...]
7  }
8  - (void) execute {
9      @weakify(self)
10     [self.centralModule.bluetoothState subscribeNext:^(NSNumber *state) {
11         @strongify(self)
12         if (state.integerValue == 5) {
13             [self.testLog sendNext:@"Bluetooth state changed to Ready"];
14             [[[self.centralModule scanWithDuplicates:YES]take:1] subscribeNext:^(
                   RBTPeripheral *peripheral) {
15                 @strongify(self)
16                 [self.testLog sendNext:@"Found peripheral"];
17                 self.peripheral = peripheral;
18                 [self.testLog sendNext:@"Try to connect to Peripheral"];
19                 [[self.peripheral connect] subscribeError:^(NSError *error) {
20                     [self.testResult sendError:error];
21                 }completed:^{
22                     @strongify(self)
23                     [self.testLog sendNext:@"Connected successfully"];
24                     //[...]
25                 }];
26             }];
27         }
28     }];
29 }
30 - (void)reset {
31     if (self.peripheral) {
32         [self.peripheral disconnect];
33     }
34 }
35 @end
```

In der Methode `-(void)setup` werden der Test initialisiert und die Properties für den Testtitel (`testTitle`) und die Testbeschreibung (`testDescription`) gesetzt. In der folgenden Methode `-(void)execute` wir der eigentliche Test ausgeführt. In Systemtest ST03 wird dabei nach *Peripherals* gesucht und danach eine Verbindung aufgebaut. Mit dem Property `testLog` kann man Logs senden (beispielsweise in Zeile 13 zu sehen) und mit dem Property `testResult` kann man den Test fertigstellen oder fehlschlagen (in Zeile 20 zu sehen) lassen. Die Methode `-(void)reset` wird nach jeder Beendigung jedes Tests aufgerufen und kann zum Aufräumen oder Wiederherstellen von Zuständen genutzt werden.

Zum Hinzufügen eines Testfalls muss man nur eine neue Klasse erzeugen, die das Protocol implementiert, und diese der Klasse `RBTTestCasesController` hinzufügen. Im User Interface der System-Test App erscheint dann eine neue Zeile mit dem neuen Test.

Die Ergebnisse der Testfälle sind inklusive Logausgaben im Verzeichnis `/Evaluation/Test-Results` auf der beiliegenden CD enthalten.

5.1.2. Unit Tests

Die Teile des Frameworks, die nicht den Zweck der Funkverbindung oder der Kommunikation erfüllen, wurden durch Unit-Tests abgedeckt. Dabei wurde besonders der richtige Aufbau der Datenstruktur und die richtige Interaktion mit dem gekapselten, darunterligenden Framework CoreBluetooth getestet.

Objekte, die nicht manuell erzeugt werden können, wurden durch Mocks ersetzt. Dazu wurde das TestMock-Framework OCMock[2] genutzt. Als anschauliches Beispiel ist Unit-Test UT01 im folgenden Listing 5.2 zu sehen. Dabei sieht man wie OCMock genutzt wurde, um eine `CBPeripheral`-Objekt zu mocken, bei dem nur der Identifier einen echten Wert enthält (siehe Zeile 6). In diesem Test wird vorgegeben, dass sich ein Peripheral verbunden hat. Geprüft wird, ob die Verwaltung der verbundenen Geräte funktioniert.

[2] http://ocmock.org/

Listing 5.2: Unit Test UT01

```
1 -(void)testRetriveConnectedPeripherals {
2     RBTCentralModule *centralModule = [[RBTCentralModule alloc]init];
3
4     // mock CBPeripheral
5     id peripheral = OCMClassMock([CBPeripheral class]);
6     OCMStub([peripheral identifier]).andReturn([NSUUID UUID]);
7
8     // fake call successful connect
9     [centralModule.cbCentralManager.delegate centralManager:centralModule.
          cbCentralManager didConnectPeripheral:peripheral];
10
11    // check retrieving
12    XCTAssertTrue([[centralModule retrieveConnectedPeripherals] array].count ==
          1, "There should be one connected peripheral.");
13    [peripheral verify];
14 }
```

In der folgenden Tabelle 5.2 sind alle Unit-Testfälle übersichtlich aufgegliedert.

Testnummer	Beschreibung	Anforderung
UT01	Verbundene Peripherals verwalten	FA03
UT02	Bekannte Peripherals verwalten	FA03
UT03	CoreBluetooth Peripherals wrappen	FA03
UT04	Service hinzufügen	FA06
UT05	Service mit inkl. secondary Service hinzufügen	FA06
UT06	Service entfernen	FA06
UT07	Service mit inkl. secondary Service entfernen	FA06
UT08	Characteristic hinzufügen	FA07
UT09	Characteristic veröffentlichen	FA07
UT10	Characteristic entfernen	FA07
UT11	Descriptor hinzufügen	FA08
UT12	Descriptor veröffentlichen	FA08
UT13	Descriptor entfernen	FA08

Tabelle 5.2.: Testfälle der Unit-Tests

5.1.3. Testabdeckung und -ergebnisse

In der folgenden Tabelle 5.3 sind die Anforderungen nochmals den Testarten gegenübergestellt, um übersichtlich darzustellen, welche Anforderungen mit welchen Tests abgedeckt wurden. Wie in der Tabelle zu sehen ist, sind [FA02, FA06, FA07, FA08] durch Unit-Tests abgedeckt. Dabei ist auch zu sehen, dass für jeweils eine Anforderung mehrere Tests durchgeführt wurden. Das ist damit begründet, dass Unit-Tests

immer nur einen kleinen Bereich testen und nur durch verschiedene Tests die vollständige Anforderung abgedeckt werden kann. Alle weiteren funktionalen Anforderungen sind durch System-Tests abgedeckt. Da diese Tests einen größeren Umfang haben und mehrere Komponenten zusammen getestet werden, reicht dabei jeweils ein Test pro Anforderung aus.

		Funktionale Anforderung (FA)														
	Nr.	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Unit-Test	UT01			x												
	UT02			x												
	UT03			x												
	UT04						x									
	UT05						x									
	UT06						x									
	UT07						x									
	UT08							x								
	UT09							x								
	UT10							x								
	UT11								x							
	UT12								x							
	UT13								x							
System-Test	ST01	x														
	ST02		x													
	ST03				x											
	ST04					x										
	ST05									x						
	ST06										x					
	ST07											x				
	ST08												x			
	ST09													x		
	ST10														x	
	ST11															x

Tabelle 5.3.: Zuordnung der Testfälle auf die Testarten

Nach Auswertung der Tests sind nachfolgend die Ergebnisse der Testfälle dargestellt. Wie zu sehen ist, wurden alle Testfälle erfolgreich abgeschlossen. Die dokumentierten Ergebnisse sind auf der CD im Verzeichnis `/Evaluation/Test-Results/` zu finden.

	Testnummer	Beschreibung	Ergebnis
Unit-Test	UT01	Verbundene Peripherals verwalten	erfolgreich
	UT02	Bekannte Peripherals verwalten	erfolgreich
	UT03	CoreBluetooth Peripherals wrappen	erfolgreich
	UT04	Service hinzufügen	erfolgreich
	UT05	Service mit inkl. secondary Service hinzufügen	erfolgreich
	UT06	Service entferen	erfolgreich
	UT07	Service mit inkl. secondary Service entfernen	erfolgreich
	UT08	Characteristic hinzufügen	erfolgreich
	UT09	Characteristic veröffentlichen	erfolgreich
	UT10	Characteristic entfernen	erfolgreich
	UT11	Descriptor hinzufügen	erfolgreich
	UT12	Descriptor veröffentlichen	erfolgreich
	UT13	Descriptor entfernen	erfolgreich
Systemtest	ST01	Bluetooth Hardware Status prüfen	erfolgreich
	ST02	Scannen nach Peripherals	erfolgreich
	ST03	Verbinden/Verbindung trennen zu Peripherals	erfolgreich
	ST04	Advertise Daten senden	erfolgreich
	ST05	Indication/Notification versenden	erfolgreich
	ST06	Services durchsuchen	erfolgreich
	ST07	Characteristic und dessen Wert auslesen	erfolgreich
	ST08	Wert einer Characteristic ändern	erfolgreich
	ST09	Bei einer Characteristic registrieren	erfolgreich
	ST10	Descriptor und dessen Wert auslesen	erfolgreich
	ST11	RSSI und Geräteabstand bestimmen	erfolgreich

Tabelle 5.4.: Beschreibung und Auswertung der Testfälle

5.2. Beispielimplementierung

Wie in der Aufgabenstellung festgelegt, wurden Anwendungsfälle beispielhaft implementiert. Dazu wurden zwei iOS Anwendungen geschrieben, die über Bluetooth LE unter Nutzung des Frameworks miteinander interagieren können. Für die Umsetzung des Anwendungsfalls des Peripheral Moduls wurde eine Senderapplikation geschrieben. Diese stellt Daten für die Empfängerapplikation bereit, welche den Anwendungsfall des *Central*-Moduls repräsentiert. Bei der Beispielimplementierung handelt es sich

um ein Sicherheitskonzept zum Versenden von Feueralarmen. Die genaue Umsetzung und Funktion wird in den nächsten beiden Kapiteln erläutert.

Die beiden Projekte der Anwendungen der Beispielsimplementierung sind im Verzeichnis `/Implementation` auf der beiliegenden CD enthalten.

5.2.1. Senderapplikation

Die Senderapplikation ist eine iOS App, die dazu dient, einen Feueralarm zu versenden. Dazu fungiert die App als Bluetooth-*Peripheral*, sodass sich Geräte mit der Empfängerapplikation verbinden können. Die Bluetooth-Funktionlitäten wurden dabei vollständig mit dem entwickelten Framework umgesetzt. In der App ist es möglich, einen Alarmtext zu verfassen und ein Bild des Etagenplans zu versenden. Dazu wurde folgender Bluetooth *Service* erstellt:

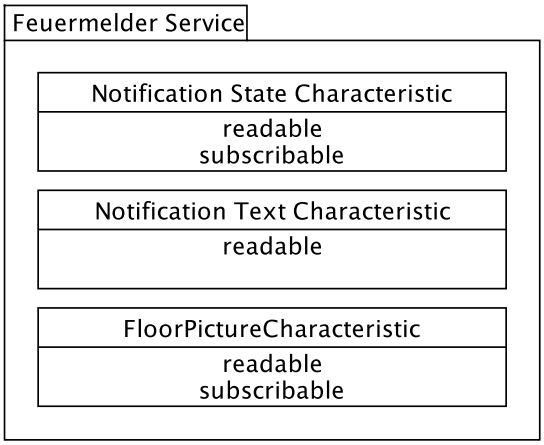

Abbildung 5.5.: Struktur des Bluetooth Service der Sendeapplikation

Sobald der Nutzer Senden betätigt, werden alle Geräte, die sich bei der `Notification State Characteristic` registriert haben per *Indication* benachrichtigt. Die Senderapplikation ist dabei zusätzlich immer in der *Advertise*-Betriebtsart, sodass sich alle in der nähe befindlichen Empfängerapplikationen immer mit dem Sender verbinden können.

5.2.2. Empfängerapplikation

Die Empfängerapplikation stellt eine App dar, die Feueralarme empfangen kann. Nach dem Start der App hat der Nutzer keine Interaktionsmöglichkeit. Es wird nur der Verbindungstatus zu einem Sender angezeigt, wie in Abbildung 5.6 dargestellt. Sobald eine Sender-App in Reichweite ist, wird man mit dieser verbunden. Dabei registriert sich das Gerät sofort bei der `Notification State Characteristic`.

Sobald eine Feueralarmmeldung empfangen wird, erscheint eine Benachrichtigung auf dem iOS-Gerät. Beim Öffnen dieser erhält man dann den Meldungstext und es wird die Reichweite zum Sender bestimmt. Das ist in Abbildung 5.7 gut zu sehen. Außerdem ist es möglich, den Etagenplan vom Sender herunterzuladen und die Meldung auszublenden.

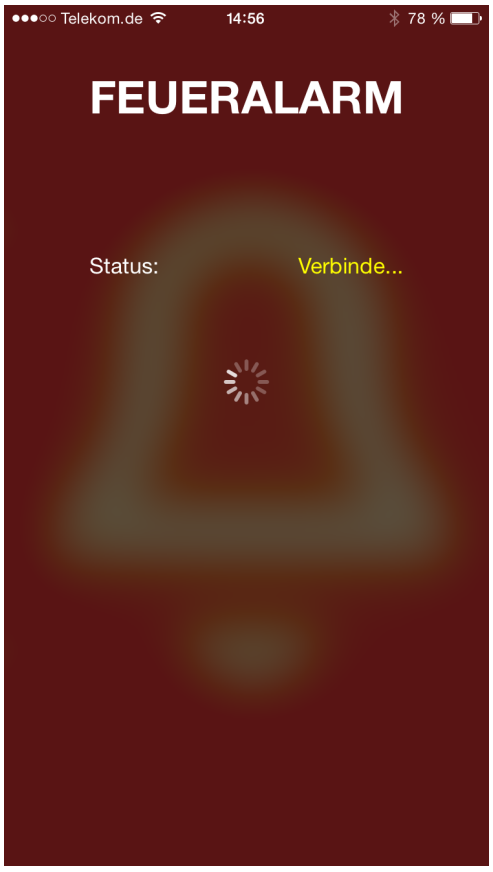

Abbildung 5.6.: Empfängerapplikation beim Verbindungsaufbau

Abbildung 5.7.: Empfängerapplikation bei der Anzeige einer Alarmmeldung

Dieses Verhalten ist mit der reaktiven Umsetzung mittels des implementierten Frame-

works entstanden. Sobald ein Alarm durch die Indication der `Notification State Characteristic` empfangen wurde, wird der Meldungstext mit der `Notification Text Characteristic` abgerufen. Außerdem wird kontinuierlich der Geräteabstand bestimmt. Das Übertragen des Etangenplans wird ausgelöst, indem sich bei der `Floor Picture Characteristic` registriert. Die Senderapplikation sendet daraufhin das Bild per *Indication*.

5.3. Vorteilsevaluation

Das Ziel dieser Arbeit ist es, die Integration von Bluetooth LE effizienter und effektiv zu gestalten. Die funktionalen Anforderungen wurden dazu in der Anforderungsanalyse spezifiziert und in Abschnitt 5.1 getestet. Die Integration und Nutzbarkeit wurde mithilfe der Beispielimplementierung aufgezeigt. Zusätzlich werden in der folgenden Aufzählung nochmals die Vorteile der Nutzung des Frameworks erklärt.

- Alle Zustandsänderungen oder asynchrone Netzwerkoperationen sind über ein gemeinsames Interface, den Signalen, abgebildet.
- Alle Aktionen können bei dem zuständigen Objekt ausgeführt werden und es wird kein Umweg über einen Manager genutzt.
- Standardverbindungsanfragen werden automatisch verarbeitet.
- Signale können kombiniert, geschachtelt und verknüpft werden.
- Signale können von verschiedenen Programmteilen simultan empfangen werden.
- Zusammengehörige Implementierungen werden nicht geteilt.
- Implementierungen können durch Verknüpfungen der Signale wartbarer und nachvollziehbarer werden. Außerdem wird der Programmfluss besser erkennbar.

5.3.1. Umfrage

Zur Verifikation wurde dazu eine Umfrage bei iOS Entwicklern durchgeführt. Es wurden den Entwicklern dabei zwei verschiedene Implementierungen einer typischen Bluetooth LE Verbindung vorgelegt. Eine der beiden Implementierungen nutzt dabei das Framework, die andere die Standard CoreBluetooth Implementierung. Die Implementierungen sind im Anhang (Section Abschnitt A.3) abgebildet.

Die Auswertung dieser Umfrage ergab, dass acht von acht Entwicklern die Nutzung des Frameworks bevorzugen. Ebenso stuften es alle als besser testbar ein. Sieben der acht Befragten bezeichneten das Framework zudem als besser verständlich und besser änderbar.

Auch wenn diese Umfrage nicht repräsentativ ist, so spricht es trotzdem für die Vorteile des Frameworks.

Die Umfrage, sowie die Auswertung sind auf der beiligenden CD im Verzeichnis `/Evaluation/Survey` bzw. `/Evaluation/Survey-Results` enthalten.

5.3.2. Geschwindigkeitstest

Es wurde außerdem ein Geschwindigkeitstest durchgeführt. Dafür wurden die Implementierungen der Umfrage auf die Ausführungsdauer hin überprüft. Dabei wurden 10 Testdurchführungen durchlaufen. Die Tabelle 5.5 zeigt die gemessen Zeiten. Es ist zu erkennen, dass die Zeiten bei beiden Implementierungen sehr schwanken. Allerdings ist sowohl der absolut niedrigste Wert mit dem entwickelten Framework entstanden, als auch der die durchschnittliche Ausführungsdauer niedriger, sodass auch dieser Test für das entwickelte Framework spricht.

Testnummer	CoreBluetooth	Reactive Bluetooth
SP01	1,870569 s	1,435333 s
SP02	1,203404 s	1,798383 s
SP03	1,85205 s	1,274656 s
SP04	1,506652 s	1,449603 s
SP05	1,392993 s	1,587649 s
SP06	1,211287 s	1,290256 s
SP07	1,407293 s	1,381540 s
SP08	1,523331 s	1,475207 s
SP09	1,418941 s	1,076057 s
SP10	1,355479 s	1,301121 s

Durchschnitt	1,4742000 s	1,4069805 s
Differenz	0,0672195s	

Tabelle 5.5.: Protokoll des Geschwindigkeitstests

Das Projekt zur Ausführung des Geschwindigkeitstest ist auf der beiliegenden CD im Verzeichnis `/Framework/Tests/SpeedTest` enthalten.

6 Zusammenfassung

Bluetooth Low Energy ist eine aufstrebende Technologie mit hohem Potential zur Entwicklung neuer Konzepte der Gerätevernetzung. Eine hohe Verbreitung und sinnvolle Anwendungsmöglichkeiten ergeben gute Voraussetzungen für einen erfolgreichen Einsatz in neuen Produkten.

Im Rahmen dieser Arbeit wurde ein Framework zur reaktiven Steuerung von Bluetooth LE unter iOS konzeptioniert und implementiert. Zunächst wurden die Struktur und die Besonderheiten dieser Technologie analysiert. Dabei wurde auch festgestellt, welche Bluetooth-Funktionen unter iOS nutzbar sind, wo Grenzen erkennbar sind und wo Probleme auftreten. Abgeleitet wurden davon Produktfunktionen, aus denen eine Anforderungsliste mit eindeutigen Zuständigkeiten erstellt wurde. Mit dieser Grundlage wurde eine modulare Architektur zur reaktiven Steuerung geschaffen, mit der generisch alle Bluetooth LE Funktionen von iOS umgesetzt werden können.

Mit dem Framework ist man in der Lage, auf Übertragungen, Wertveränderungen, Benachrichtigungen oder Zustandsänderungen aller Bluetooth-Komponenten zu reagieren und den Programmfluss dementsprechend zu steuern. Genutzt wird dafür das Open Source Framework ReactiveCocoa. Damit wurden auch alle asynchronen Verbindungsoperationen implementiert, sodass eine einheitliche Struktur vorherrscht. Auch wenn sich die Restriktionen von iOS nicht vollständig umgehen lassen, wurde mithilfe des reaktiven Konzeptes dieser Arbeit die Steuerung und das Verbindungsmanagement erleichtert und deutlich verbessert.

Zur Sicherstellung einer guten Softwarequalität und zur Prüfung der korrekten Umsetzung der Anforderungen wurden Unit-Tests und Integrations-Tests durchgeführt. Dabei wurden zum einen die Datenstruktur, zum anderen auch ganze Verbindungsabläufe und dessen Zusammenarbeit getestet. Für die Integrations-Tests wurde dafür eine iOS Anwendung geschrieben, die ein halbautomatisiertes Testen ermöglicht und dabei eine gute Protokollierung der Testschritte sowie eine einfache Testdurchführung ermöglicht.

Mithilfe von zwei Beispielimplementierungen wurde das Framework bei einem aktiven Einsatz getestet. Dabei wurden die einfache Integration sowie die gute Nutzbarkeit dargestellt. Außerdem wurde dabei der reaktive Programmfluss und die reaktive Interaktion mehrerer Bluetooth-Geräte analysiert.

Nicht zuletzt durch die Vorteilsevaluation wurde gezeigt, dass das Ziel dieser Arbeit, eine effiziente und effektive Steuerung von Bluetooth Low Energy unter iOS, erreicht wurde. In einer Umfrage gaben alle befragten Personen an, die Nutzung des entwickelten Frameworks generell zu bevorzugen.

A Anhang

A.1. Anleitung zur Einbindung des Frameworks

Zum Einbinden des Frameworks und zum leichten Auflösen der Abhängigkeiten kann CocoaPods genutzt werden. Dieses Programm lässt sich mit folgenden Befehlen im Terminal installieren.

Listing A.1: Shell-Befehle zum Installieren von CocoaPods

```
1 [sudo] gem install cocoapods
2 pod setup
```

Danach ist es nötig, eine Datei mit dem Namen *„Podfile"* zu erstellen und dieses im Projektverzeichnis abzulegen. Der Inhalt sollte mit folgendem Listing übereinstimmen.

Listing A.2: Beispielinhalt eines Podfile

```
1 source 'https://github.com/CocoaPods/Specs.git'
2
3 target '<TARGET>' do
4     pod 'ReactiveBluetoothLE', :path => '<PATH (e.g. ../../Framework)>'
5 end
6
7 target '<TARGET>Tests' do
8     pod 'ReactiveBluetoothLE', :path => '<PATH (e.g. ../../Framework)>'
9 end
```

Dabei sollte `<TARGET>` mit dem Apptarget ersetzt werden. Außerdem sollte der Pfad zum Framework bei `<PATH>` ersetzt werden.

Nun navigiert man zu dem Projektverzeichnis und aktiviert Cocoapods mit dem Befehl `[sudo] pod install`. Dabei wird das Framework und dessen Abhängigkeiten

installiert und es wird ein Workspace für das Projekt angelegt, welches alle Einstellungen enthält.

Mit dem Import-Befehl `#import <ReactiveBluetoothLE/ReactiveBluetoothLE.h>` hat man nun Zugriff auf alle Klassen des entwickelten Frameworks.

A.2. Verwendete Software

Im Nachfolgenden wird die verwendete Software beschrieben, die bei der Erstellung dieser Arbeit genutzt wurde.

A.2.1. Programmierung

AppCode

Zum Erstellen des Frameworks wurde die Entwicklungsumgebung AppCode verwendet. Diese bietet eine gute Auto-Vervollständigung, gute Refactoring-Möglichkeiten und unterstützt den Programmierer mit sinnvollen Anmerkungen.

Xcode

Da AppCode nicht alle Funktionen zur Erstellung von User Interfaces bietet, wurde für die Entwicklung der Beispielsanwendungen und Testanwendungen die Entwicklungsumgebung Xcode verwendet.

Instruments

Das Entwickler-Programm Instruments wurde genutzt, um den Speicherverbrauch sowie das Speichermanagement des Frameworks zu testen und die Leitung zu überwachen.

OCMock

Das Mocking-Framework OCMock wurde genutzt, um die Unit-Tests umzusetzen. Dabei wurden vor allem die CoreBluetooth-Objekte gemockt, die sich auf andere Weise nicht erstellen lassen.

ReactiveCocoa

Zur reaktiven Umsetzung des Frameworks wurde, wie in der Arbeit beschrieben, ReactiveCocoa genutzt.

CocoaPods

Zum Auflösen der externen Abhängigkeiten, wie z.B. die Nutzung von ReactiveCocoa, wurde der Dependency Manager CocoaPods genutzt. Außerdem wird es dazu genutzt, das Framework in andere Projekte oder auch in die erstellten Test- und Beispielsanwendungen einzubinden.

A.2.2. Bachelordokument

Diese Arbeit wurde mit der Textverarbeitung Lyx geschrieben. Die Dokumentvorlage wurde mir freundlicherweise von Oliver Mader bereitgestellt, die im Zuge seiner Bachelorarbeit [Mad12] entstanden ist. Diese habe ich zudem nach meinen eigenen Wünschen angepasst. Zudem wurde JabRef genutzt, um die Literaturverweise zu verwalten. Richtlinien und Empfehlungen zur Erstellung dieser Arbeit wurde vom Dokument [Kra12] entnommen, welches vom betreuenden Professor dieser Arbeit, Prof. Dr. rer. nat. Nane Kratzke, zur Verfügung gestellt wurde.

A.3. Umfrage zur Vorteilsevaluation

Zur Vorteilsevaluation wurden an iOS-Entwickler die folgenden Programm-Listings verteilt. Listing A.3 und Listing A.4 zeigen dabei die Referenzimplementierung mit CoreBluetooth. Listing A.5 und Listing A.6 zeigen dagegen die gleiche Implementierung mit dem entwickelten Framework. Dazu haben die Entwickler folgende Tabelle A.1 zum Ausfüllen erhalten.

	Standard CoreBluetooth Implementierung	Reactive Bluetooth LE Framework for iOS Implementierung
Verständlichkeit		
Änderbarkeit		
Testbarkeit		
Generelle Bevorzugung		

Tabelle A.1.: Umfrage zur Vorteilsevaluation

Das vollständige Umfrageblatt, sowie dessen Ergebnisse sind auf der beiliegenden CD im Verzeichnis `/Evaluation/Survey` bzw. `/Evaluation/Survey-Results` zu finden.

Listing A.3: CoreBluetooth Peripheral Implementierung

```
#import <CoreBluetooth/CoreBluetooth.h>


@interface CoreBluetoothImplementationPeripheral : NSObject <CBPeripheralManagerDelegate>

@property(nonatomic) CBPeripheralManager *peripheralManager;
@property(nonatomic) CBMutableService *testService;
@property(nonatomic) CBMutableCharacteristic *testCharacteristic;

@end


@implementation CoreBluetoothImplementationPeripheral

- (instancetype)init {
    self = [super init];
    if (self) {
        _peripheralManager = [[CBPeripheralManager alloc] initWithDelegate:self queue:nil];
        [self setup];
    }
    return self;
}

- (void)setup {
    CBUUID *testServiceUUID = [CBUUID UUIDWithString:@"AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA"];
    CBUUID *testCharacteristicUUID = [CBUUID UUIDWithString:@"BBBBBBBB-BBBB-BBBB-BBBB-
        BBBBBBBBBBBB"];

    // create attributes structure
    self.testService = [[CBMutableService alloc] initWithType:testServiceUUID primary:YES];
    self.testCharacteristic = [[CBMutableCharacteristic alloc] initWithType:
        testCharacteristicUUID properties:(CBCharacteristicPropertyRead |
        CBCharacteristicPropertyWrite) value:nil permissions:(CBAttributePermissionsReadable
        | CBAttributePermissionsWriteable)];
    self.testService.characteristics = @[self.testCharacteristic];
}

- (void)peripheralManagerDidUpdateState:(CBPeripheralManager *)peripheral {
    if (peripheral.state == 5) {
        // publish attributes
        [self.peripheralManager addService:self.testService];
    }
}

- (void)peripheralManager:(CBPeripheralManager *)peripheral didAddService:(CBService *)service
    error:(NSError *)error {
    if (error) {
        return;
    }

    // set characteristic value
    self.testCharacteristic.value = [@"TEST-VALUE" dataUsingEncoding:NSUTF8StringEncoding];

    // set device name & start advertising
    [self.peripheralManager startAdvertising:@{CBAdvertisementDataLocalNameKey : @"TEST-DEVICE
        ",
            CBAdvertisementDataServiceUUIDsKey : @[self.testService.UUID]}];
}

// read request handling
- (void)peripheralManager:(CBPeripheralManager *)peripheral didReceiveReadRequest:(
    CBATTRequest *)request {
    if (request.offset > self.testCharacteristic.value.length) {
        [self.peripheralManager respondToRequest:request withResult:CBATTErrorInvalidOffset];
        return;
    }
    request.value = [self.testCharacteristic.value subdataWithRange:NSMakeRange(request.offset
        ,
```

```
61              self.testCharacteristic.value.length - request.offset)];
62      [self.peripheralManager respondToRequest:request withResult:CBATTErrorSuccess];
63  }
64
65  // write request handling
66  - (void)peripheralManager:(CBPeripheralManager *)peripheral didReceiveWriteRequests:(NSArray
        *)requests {
67      for (CBATTRequest *request in requests) {
68          self.testCharacteristic.value = request.value;
69          [self.peripheralManager respondToRequest:request withResult:CBATTErrorSuccess];
70      }
71  }
72
73  @end
```

Listing A.4: CoreBluetooth Central Implementierung

```
1   #import <CoreBluetooth/CoreBluetooth.h>
2
3
4   @interface CBSpeedTester : NSObject <CBCentralManagerDelegate, CBPeripheralDelegate>
5
6   @property(nonatomic) CBCentralManager *centralManager;
7   @property(nonatomic) CBPeripheral *peripheral;
8   @property(nonatomic) CBService *service;
9   @property(nonatomic) CBCharacteristic *characteristic;
10
11  @end
12
13
14  @implementation CBSpeedTester
15
16
17  - (instancetype)init {
18      self = [super init];
19      if (self) {
20          _centralManager = [[CBCentralManager alloc] initWithDelegate:self queue:
                dispatch_get_main_queue()];
21      }
22      return self;
23  }
24
25
26  - (void)centralManagerDidUpdateState:(CBCentralManager *)central {
27      if (central.state == 5) {
28          self.startDate = [NSDate date];
29
30          // scan for peripherals
31          [self.centralManager scanForPeripheralsWithServices:nil options:nil];
32      }
33  }
34
35  - (void)centralManager:(CBCentralManager *)central didDiscoverPeripheral:(CBPeripheral *)
        peripheral advertisementData:(NSDictionary *)advertisementData RSSI:(NSNumber *)RSSI {
36      if (RSSI.integerValue > -15) {
37          return;
38      }
39
40      self.peripheral = peripheral;
41      self.peripheral.delegate = self;
42      [central stopScan];
43
44      // connect to peripheral
45      [central connectPeripheral:peripheral options:nil];
46  }
47
48  - (void)centralManager:(CBCentralManager *)central didConnectPeripheral:(CBPeripheral *)
        peripheral {
49      if (peripheral == self.peripheral) {
50
51          // discover services
52          [self.peripheral discoverServices:nil];
53      }
```

```
54 }
55
56 - (void)peripheral:(CBPeripheral *)peripheral didDiscoverServices:(NSError *)error {
57     if (!error && peripheral == self.peripheral) {
58         CBUUID *testServiceUUID = [CBUUID UUIDWithString:@"AAAAAAAA-AAAA-AAAA-AAAA-
               AAAAAAAAAAAA"];
59         for (CBService *service in peripheral.services) {
60             if ([service.UUID isEqual:testServiceUUID]) {
61                 self.service = service;
62
63                 // discover characteristic
64                 [self.peripheral discoverCharacteristics:nil forService:self.service];
65             }
66         }
67     }
68 }
69
70 - (void)peripheral:(CBPeripheral *)peripheral didDiscoverCharacteristicsForService:(CBService
       *)service error:(NSError *)error {
71     if (!error && service == self.service) {
72         CBUUID *testCharacteristicUUID = [CBUUID UUIDWithString:@"BBBBBBBB-BBBB-BBBB-BBBB-
               BBBBBBBBBBBB"];
73         for (CBCharacteristic *characteristic in self.service.characteristics) {
74             if ([characteristic.UUID isEqual:testCharacteristicUUID]) {
75                 self.characteristic = characteristic;
76
77                 // read characteristic
78                 [self.peripheral readValueForCharacteristic:self.characteristic];
79             }
80         }
81     }
82 }
83
84 - (void)peripheral:(CBPeripheral *)peripheral didUpdateValueForCharacteristic:(
       CBCharacteristic *)characteristic error:(NSError *)error {
85     if (!error) {
86         NSLog(@"Characteristic Value: %@", characteristic.value);
87
88         // write characteristic
89         [self.peripheral writeValue:[@"Test" dataUsingEncoding:NSUTF8StringEncoding]
               forCharacteristic:self.characteristic type:CBCharacteristicWriteWithResponse];
90     }
91 }
92
93 - (void)peripheral:(CBPeripheral *)peripheral didWriteValueForCharacteristic:(CBCharacteristic
       *)characteristic error:(NSError *)error {
94     if (!error && characteristic == self.characteristic) {
95
96         // disconnect peripheral
97         [self.centralManager cancelPeripheralConnection:self.peripheral];
98     }
99 }
100
101
102 @end
```

Listing A.5: Reactive Peripheral Implementierung

```
#import <ReactiveBluetoothLE/ReactiveBluetoothLE.h>


@interface ReactiveImplementationPeripheral : NSObject

@property(nonatomic) RBTPeripheralModule *peripheralModule;
@property(nonatomic) RBTMutableService *testService;
@property(nonatomic) RBTMutableCharacteristic *testCharacteristic;

@end


@implementation ReactiveImplementationPeripheral

- (instancetype)init {
    self = [super init];
    if (self) {
        _peripheralModule = [[RBTPeripheralModule alloc] init];
        [self setup];
    }
    return self;
}

- (void)setup {
    CBUUID *testServiceUUID = [CBUUID UUIDWithString:@"AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA"];
    CBUUID *testCharacteristicUUID = [CBUUID UUIDWithString:@"BBBBBBBB-BBBB-BBBB-BBBB-
        BBBBBBBBBBBB"];

    // set device name
    self.peripheralModule.name = @"TEST-DEVICE";

    // create attributes structure
    self.testService = [[RBTMutableService alloc] initPrimaryServiceWithUUID:testServiceUUID];
    self.testCharacteristic = [[RBTMutableCharacteristic alloc]
                        initWithUUID:testCharacteristicUUID
                          properties:(CBCharacteristicPropertyRead |
                                      CBCharacteristicPropertyWrite)
                               value:nil
                         permissions:(CBAttributePermissionsReadable |
                                      CBAttributePermissionsWriteable)];

    @weakify(self)
    [[self.peripheralModule.peripheralState filter:^BOOL(NSNumber *state) {
        return state.intValue == 5;
    }] subscribeNext:^(id x) {
        @strongify(self)

        // publish attributes
        [self.testService addCharacteristic:self.testCharacteristic];
        [[self.peripheralModule addService:self.testService] subscribeCompleted:^{
            @strongify(self)

            // set characteristic value
            self.testCharacteristic.value = [@"TEST-VALUE" dataUsingEncoding:
                NSUTF8StringEncoding];

            // start advertising
            [self.peripheralModule startAdvertising];
        }];
    }];
}

@end
```

Listing A.6: Reactive Central Implementierung

```
#import <ReactiveBluetoothLE/ReactiveBluetoothLE.h>

@interface RBTSpeedTester : NSObject

@property(nonatomic) RBTCentralModule *centralModule;
```

```
@property(nonatomic) RBTPeripheral *peripheral;
@property(nonatomic) RBTService *service;
@property(nonatomic) RBTCharacteristic *characteristic;

@end


@implementation RBTSpeedTester

- (instancetype)init {
    self = [super init];
    if (self) {
        _centralModule = [[RBTCentralModule alloc] init];
        [self initTest];
    }
    return self;
}

- (void)initTest {
    @weakify(self)
    [[self.centralModule.bluetoothState filter:^BOOL(NSNumber *state) {
        return ([state isEqual:@(5)]);
    }] subscribeNext:^(id x) {
        @strongify(self)
        self.startDate = [NSDate date];
        [self startTest];
    }];
}

- (void)startTest {
    CBUUID *testServiceUUID = [CBUUID UUIDWithString:@"AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA"];
    CBUUID *testCharacteristicUUID = [CBUUID UUIDWithString:@"BBBBBBBB-BBBB-BBBB-BBBB-
        BBBBBBBBBBBB"];
    @weakify(self)

    // scan for peripherals
    [[[self.centralModule scan] take:1] subscribeNext:^(RBTPeripheral *peripheral) {
        @strongify(self)
        self.peripheral = peripheral;

        // conntect to peripheral
        [[self.peripheral connect] subscribeCompleted:^{
            @strongify(self)

            // discover services
            [[self.peripheral discoverServices] subscribeCompleted:^{
                @strongify(self)
                self.service = [self.peripheral serviceWithUUUID:testServiceUUID];

                // discover characteristics
                [[self.service discoverAllCharacteristics] subscribeCompleted:^{
                    @strongify(self)
                    self.characteristic = [self.service characteristicWithUUUID:
                        testCharacteristicUUID];

                    [self readAndWrite];
                }];
            }];
        }];
    }];
}

- (void)readAndWrite {
    @weakify(self)

    // read characteristic
    [[self.characteristic readValue] subscribeCompleted:^{
        @strongify(self)
        NSLog(@"Characteristic Value: %@", self.characteristic.value);

        // write characteristic
        [[self.characteristic writeValue:[@"TEST" dataUsingEncoding:NSUTF8StringEncoding]
            withResponse:YES] subscribeCompleted:^{
            @strongify(self)

```

```
78              // disconnect peripheral
79              [self.peripheral disconnect];
80          }];
81      }];
82  }
83
84  @end
```

Literaturverzeichnis

[App12a] Apple: *Introduction to Key-Value Observing Programming Guide*. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html. Version: 07 2012. – zuletzt aufgerufen: 15.02.2015

[App12b] Apple: *Target-Action*. https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Target-Action/Target-Action.html. Version: 01 2012. – zuletzt aufgerufen: 15.02.2015

[App13] Apple: *iOS: Unterstützte Bluetooth-Profile*. http://support.apple.com/de-de/HT3647. Version: 10 2013. – zuletzt aufgerufen: 20.12.2014

[App14a] Apple: *App Store Distribution*. https://developer.apple.com/support/appstore/. Version: 12 2014. – zuletzt aufgerufen: 29.12.2014

[App14b] Apple: *What's New in iOS*. https://developer.apple.com/library/ios/releasenotes/General/WhatsNewIniOS/Articles/iOS5.html. Version: 10 2014. – zuletzt aufgerufen: 26.02.2015

[App14c] Apple, Inc.: *MFI - Frequently Asked Questions*. https://mfi.apple.com/MFiWeb/getFAQ.action. Version: 2014. – zuletzt aufgerufen: 16.01.2015

[BS14] Bluetooth SIG, Inc.: *History of the Bluetooth Special Interest Group*. http://www.bluetooth.com/Pages/History-of-Bluetooth.aspx. Version: 12 2014. – zuletzt aufgerufen: 20.12.2014

[BS15a] Bluetooth SIG, Inc.: *Frequently Asked Questions - Bluetooth Brand*. Online. http://www.bluetooth.com/Pages/Smart-Logos-FAQ.aspx. Version: 2015. – zuletzt aufgerufen: 02.02.2015

[BS15b] Bluetooth SIG, Inc.: *Specification Adopted Documents*. https://www.bluetooth.org/en-us/specification/adopted-specifications. Version: 2015. – zuletzt aufgerufen: 20.02.2015

[Ebe14] Eberhardt, Colin: *ReactiveCocoa Tutorial - The Definitive Introduction*. http://www.raywenderlich.com/62699/reactivecocoa-tutorial-pt1. Version: 03 2014. – zuletzt aufgerufen: 28.02.2015

[Git15] GitHub, Inc.: *ReactiveCocoa*. https://github.com/ReactiveCocoa/ReactiveCocoa. Version: 02 2015. – zuletzt aufgerufen: 28.02.2015

[Hei14] Heise: *E-Commerce-Studie: iPhone-Nutzer generieren mehr Einnahmen*. http://www.heise.de/mac-and-i/meldung/E-Commerce-Studie-iPhone-Nutzer-generieren-mehr-Einnahmen-2171382.html. Version: 04 2014. – zuletzt aufgerufen: 20.12.2014

[Kra12] Kratzke, Nane: *How to write a Bachelor, Master, Diploma Thesis? Commented Template for Bachelor, Master or Diploma Thesis*. https://github.com/nkratzke/BAMA-Template/blob/master/BAMA-Template.pdf. Version: August 21 2012

[Mad12] Mader, Alexander O.: *Berührungslose Benutzerinteraktion für .NET-basierte Software-Anwendungen*. September 2012. – Bachlor-Thesis, Fachhochschule Kiel

[PBB04] Prof. Bernd Bruegge, Ph.D: *Einfuehrung in die Informatik II: Ereignisbasierte Programmierung*. http://wwwbruegge.informatik.tu-muenchen.de/twiki/pub/Lehrstuhl/Informatik2SoSe2004/S04_06_EreignisBasierteProg.pdf. Version: 2004. – zuletzt aufgerufen: 19.12.2014

[Pfü14] Pfützenreuter, Elvis: *Bluetooth: ATT and GATT*. https://epx.com.br/artigos/bluetooth_gatt.php. Version: 2014. – zuletzt aufgerufen: 23.12.2014

[Sad09] Sadun, Erica: *Das iPhone-Entwicklerbuch - Rezepte für Anwendungsprogrammierung mit dem iPhone SDK (Apple Software)*. Addison-Wesley Verlag, 2009

[Sau13] Sauter, Martin: *Grundkurs Mobile Kommunikationssysteme - UMTS, HSPA und LTE, GSM, GPRS, Wireless LAN und Bluetooth*. Springer, 2013

[Sta14a] Starke, Gernot: *Effektive Software-Architekturen*. Hanser, 2014

[Sta14b] Statista: *Vergleich der Marktanteile von Android und iOS am Absatz von Smartphones in Deutschland von Januar 2012 bis November 2014*. http://de.statista.com/statistik/daten/studie/256790/umfrage/marktanteile-von-android-und-ios-am-smartphone-absatz-in-deutschland/. Version: 12 2014. – zuletzt aufgerufen: 10.12.2014

[Tho13] Thompson, Mattt: *ReactiveCocoa*. http://nshipster.com/reactivecocoa/. Version: 02 2013. – zuletzt aufgerufen: 20.2.2015

[Tow14] Townsend, Adafruit K.: *Introduction to Bluetooth Low Energy - A basic overview of key concepts for BLE*. https://learn.adafruit.com/introduction-to-bluetooth-low-energy/. Version: 03 2014. – zuletzt aufgerufen: 13.01.2015

[Val14] Valle, Andrea: *Die Verknüpfung der digitalen mit der physischen Welt ermöglicht das optimale Shopping-Erlebnis im Laden*. http://blogs.adobe.com/digitaleurope/files/2014/01/customer-journey.jpg. Version: 02 2014. – zuletzt aufgerufen: 01.03.2015

[Wik14a] Wikipedia: *Bluetooth Low Energy*. http://en.wikipedia.org/wiki/Bluetooth_low_energy. Version: 03 2014. – zuletzt aufgerufen: 02.02.2015

[Wik14b] Wikipedia: *Bluetooth Special Interest Group*. http://en.wikipedia.org/wiki/Bluetooth_Special_Interest_Group. Version: 10 2014. – zuletzt aufgerufen: 30.12.2014

[Wik14c] Wikipedia: *Delegation (Softwareentwicklung)*. http://de.wikipedia.org/wiki/Delegation_(Softwareentwicklung). Version: 05 2014. – zuletzt aufgerufen: 30.01.2015

[Wik14d] Wikipedia: *Event-driven programming*. http://en.wikipedia.org/wiki/Event-driven_programming. Version: 11 2014. – zuletzt aufgerufen: 15.02.2015

[Wik15a] WIKIPEDIA: *Bluetooth*. `http://de.wikipedia.org/wiki/Bluetooth`. Version: 01 2015. – zuletzt aufgerufen: 30.01.2015

[Wik15b] WIKIPEDIA: *Callback (computer programming)*. `http://en.wikipedia.org/wiki/Callback_(computer_programming)`. Version: 02 2015. – zuletzt aufgerufen: 15.02.2015

[Wik15c] WIKIPEDIA: *Higher-order function*. `http://en.wikipedia.org/wiki/Higher-order_function`. Version: 01 2015. – zuletzt aufgerufen: 15.02.2015

[Wik15d] WIKIPEDIA: *Wrapper (Software)*. `http://de.wikipedia.org/wiki/Wrapper_%28Software%29`. Version: 02 2015. – zuletzt aufgerufen: 15.02.2015