HeartRate2Go

Matthias Böffel
Patrick Mathias
Markus Nebel
Janina Sauer

15. Januar 2015

Hochschule Kaiserslautern
University of Applied Sciences

Betreuer: Prof. Dr.-Ing. Jan Conrad

Inhaltsverzeichnis

1 Einleitung

1.1 Vorstellung des Projektes

Im Rahmen der Veranstaltung *Frameworkbasierte GUI-Entwicklung* wurde das Projekt *HeartRate2Go* ausgearbeitet. *HeartRate2Go* ist ein Softwarepaket bestehend aus einer Desktop-PC Anwendung und einem Android Smartphone/Smartwatch App-Bundle zur Erfassung der Herzfrequenz für Ruhe- oder Aktivitätsmessungen. Die von der Smartwatch per Pulsoxymetrie gemessenen Herzfrequenz wird über das gekoppelte Smartphone zur Desktop-PC Anwendung übertragen und dort übersichtlich zusammen mit bereits getätigten Messungen dargestellt. *HeartRate2Go* unterstützt damit den Anwender bei der Kontrolle seines Pulses, auch über längere Zeiträume hinweg.

1.2 Ablauf einer Messung

Um *HeartRate2Go* verwenden zu können sind zunächst drei Hardware-Komponenten notwendig:

- Eine Android-Smartwatch
- Ein Android-Smartphone und
- Ein Desktop-PC (mit Windows, Linux oder OS X)

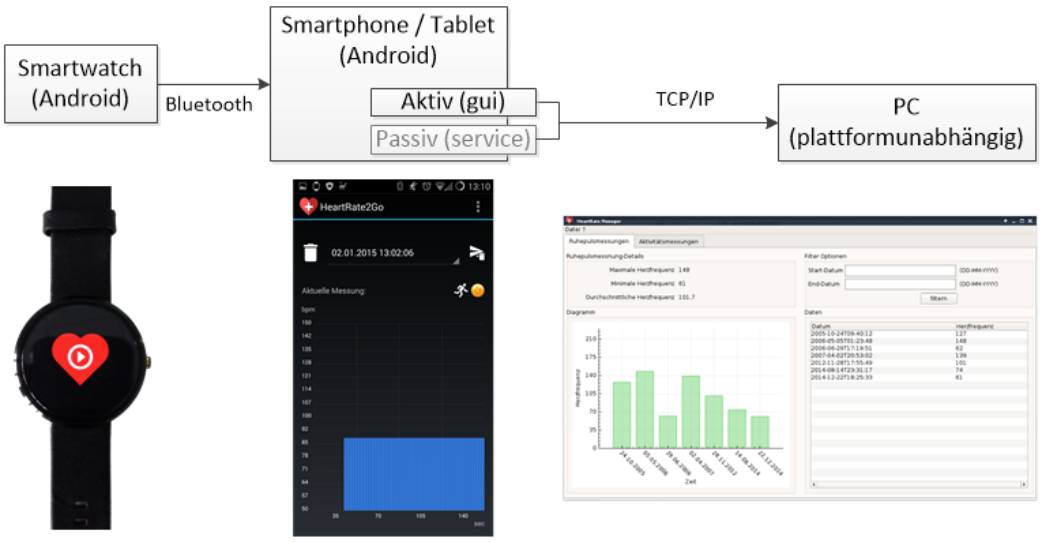

Abbildung 1: Ablauf der Datenübertragung

Für eine Messung trägt der Anwender die Smartwatch am Handgelenk und startet die *HeartRate2Go*-App. Er wählt nun aus, ob er eine Ruhe- oder eine Aktivitätsmessung durchführen möchte. Anschließend wird der Messvorgang gestartet. Bei einer Ruhemessung wird die Messung automatisch nach einer festgelegten Zeit beendet, bei der Aktivitätsmessung muss der Benutzer die Messung manuell beenden. Nach Abschluss des Messvorgangs wird der Anwender nach seiner Stimmung während der Messung gefragt. Er kann zwischen "gut", "okay" und "schlecht" wählen. Die Übertragung der Messwerte von der Smartwatch an das gekoppelte Smartphone wird automatisch per Bluetooth durchgeführt, sobald sich das Smartphone in Reichweite befindet. Auf der Smartphone-App werden dann die Messungen bereits in einem vorläufigen Balkendiagramm angezeigt. Der Anwender ist dann jederzeit in der Lage die auf dem Smartphone gespeicherten Messungen über das Heimnetzwerk per Knopfdruck an die PC-Anwendung zu übertragen. In Abbildung 1 ist der Ablauf der Datenübertragung nochmals visualisiert.

1.3 Medizinische Apps

Im Laufe der letzten Jahre wurde der Markt mit Apps, die einen medizinischen Hintergrund besitzen, überschüttet. Wenn man im deutschen iTunes-Store nach „Medizin“ sucht, erhält man mehrere hundert Einträge, dies gilt genauso für den Google-Play Store.

Im vergangen Jahr sind die Absatzzahlen von medizinischen Apps in Großbritannien, Frankreich, Niederlande und Deutschland um 42 Prozent gestiegen, vermeldet das GfK (Marktforschungszentrum).
Diese Apps decken nahezu jeden Bereich der Medizin ab, egal ob es um die Speicherung von Vitaldaten, die Messung von Vitaldaten mit einem zusätzlichen Messgerät und die Auswertung der Daten geht. Des Weiteren sind auch viele Nachschlagewerke darunter enthalten.

Zu beachten ist allerdings, dass keiner der Apps den Arztbesuch ersetzt. Sie geben lediglich eine erste Einschätzung und sind dadurch eine große Erleichterung für den Nutzer. Allerdings ist es auch so, dass jeder Programmierer eine App mit medizinischem Hintergrund in die verschiedenen Stores hochladen darf. Diese werde nicht auf ihren Nutzen hin überprüft, so sind auch viele Apps zu finden, die mehr als Spielerei gelten.

Kaum eine App ist ein Medizinprodukt nach dem Medizinproduktegesetzt, sie gelten lediglich als Wellness- beziehungsweise Lifestyle-Apps.

1.4 Medizinische Kenntnisse - Pulsoxymetrie

Für die Messung des peripheren Pulses per Android-Uhr wird das Prinzip der Reflexions-Pulsoxymetrie genutzt (Abbildung 2).

Dieses Verfahren benötigt zwei Sensoren: zum einen eine Lichtquelle, zum anderen ein Lichtsensor. Die Lichtquelle sendet Infrarot-Lichtwellen aus, die durch die Haut dringen. Der Sensor misst die Lichtanteile, die absorbiert wurden.

Die Lichtabsorption im Blut ist abhänig von der Hämoglobinkonzentration und der Sättigung des Hämoglobins mit Sauerstoff. Oxigeniertes und desoxigeniertes Hämoglobin schwächen das Licht jeweils charakteristisch ab.

Mit diesem Prinzip ist es auch möglich, die Sauerstoffsättigung im kapillären Blut zu messen [1].

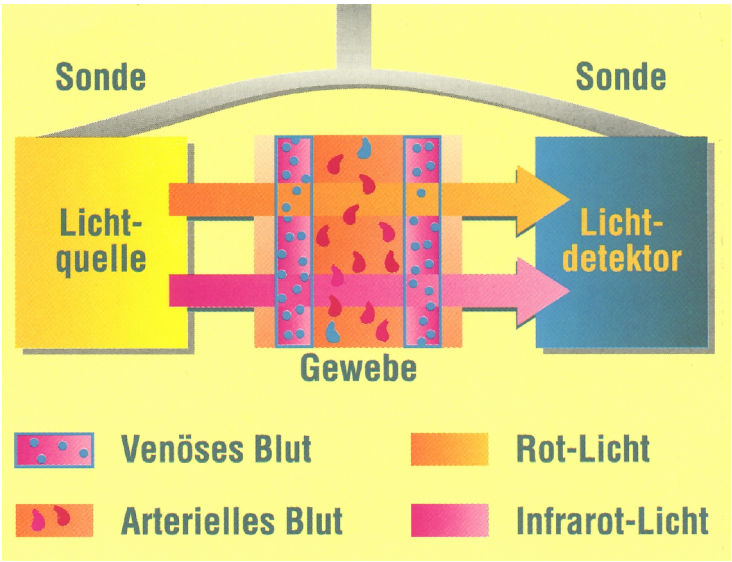

Abbildung 2: Veranschaulichung der Pulsoxymetrie[2]

2 Wearable & Handheld

2.1 Android (Standard)

Das Android-Betriebsystem erfreut sich weltweit größter Beliebtheit. Android ist mit über 75% auf dem Markt das meist verbreitete Mobil-Betriebssystem für Smartphones und Tablets. Ein großer Vorteil des mobilen Betriebssystems ist die Möglichkeit, die Funktionalität durch Installation zusätzlicher Anwendungen (Apps) zu erweitern.

Erstellt werden Apps i.d.R. mit Hilfe des Android-Frameworks in der Programmiersprache Java. Das Android-Framework passt in die Kategorie der modernen GUI-Frameworks, da die Programmlogik strikt von den Definitionen für das Layout getrennt ist. Das Layout wird durch Dateien mit XML-Struktur festgelegt und kann so unabhängig vom Code angepasst werden.

2.2 Android Wear

Android Wear als noch recht neues Betriebssystem stellt eine ressourcenschonende Version des Standard-Android-Betriebssystems für Wearables (Smartwatches, Armbänder, etc.) dar. Die Wearables bringen in den meisten Fällen Sensoren für Fitness-Tracking (z.B. Pulsmesser, und Schrittzähler) mit, die durch die Android-API bereits unterstützt werden. Das Wearable-Gerät kann zwar selbstständig agieren, ist jedoch ohne entsprechende Hardware zur Nutzung von Internet, W-LAN oder anderen Ressourcen auf ein gekoppeltes Handheld-Gerät angewiesen. Auch Hersteller Google betont, dass das Betriebssystem grundlegend zur Kopplung mit einem Smartphone bzw. Tablet (im Folgenden allgemein: Handheld) ausgelegt ist. Notifications vom Handheld werden beispielsweise bequem auf das Wearable-Gerät weitergeleitet, während in die andere Richtung Spracheingaben auf dem Wearable-Gerät interpretiert und zum Handheld-Gerät zur weiteren Verarbeitung übermittelt werden können. Die Kommunikation findet dabei i.d.R. über eine spezielle Wearable-Bluetooth-API statt.

Die Design-Prinzipien, die grundlegend auf den Entwicklerseiten von Android Wear[3] empfohlen werden, unterstreichen ebenfalls die enge Verbundenheit zum Handheld-Gerät. So sollen rechen- bzw. zeitintensive Tasks auf das leistungsfähigere Handheld-Gerät ausgelagert werden und Konfigurationen für Wearable-Apps weitestgehend auf dem Handheld-Gerät vorgenommen werden. So bietet es sich an für eine Wearable-App gleich eine zugehörige Handheld-App mitzuliefern. Die Installation einer Wearable-App erfolgt dabei auch über das Handheld-Gerät: Eine APK-Installations-Datei kann mehrere Apps für verschiedene Geräte enthalten, die dann automatisch verteilt werden. Apps für Android Wear werden unter den gleichen Bedingungen erstellt wie Apps für das Standard-Android-Betriebssystem. Zusätzlich unterstützt Android Wear sowohl runde als auch quadratische Display-Typen, was bei der Gestaltung des Layouts zu beachten ist.

2.3 Anforderungen

Im vorliegenden Projekt soll ein Wearable-Gerät, das über einen Pulsmesser- und Schrittzähler-Sensor verfügt, die jeweiligen Werte über einen begrenzten Zeitraum auslesen und temporär verwalten. Dieser Vorgang wird folgend als Messung bezeichnet. Dabei wird zwischen Aktivitäts- und Ruhemessung unterschieden. Ersteres soll die Daten solange aufzeichnen, bis die Messung vom Benutzer beendet wird und Letzteres soll festgelegt eine einmütige Messung durchführen und den Median-Wert bilden. Die Auswahl über die Art der Messung soll unmittelbar vor der Messung durch den Benutzer stattfinden.

Im Anschluss an die Messung soll eine Dialog-Abfrage die Stimmung des Benutzers während der Messung erfassen. Die Messdaten sollen nach Bearbeitung dieses Dialogs ohne zusätzliche Interaktion per Bluetooth zum Handheld-Gerät übertragen werden, sofern dieses verfügbar ist. Wenn das Handheld-Gerät zu diesem Zeitpunkt nicht zur Verfügung steht, soll der Benutzer die Möglichkeit haben die Messung erneut zu versenden oder zu verwerfen. Eine persistente Speicherung der Messdaten soll dabei auf dem Wearable-Gerät nicht stattfinden. Abbildung 3 zeigt die bisher beschriebene Struktur auf.

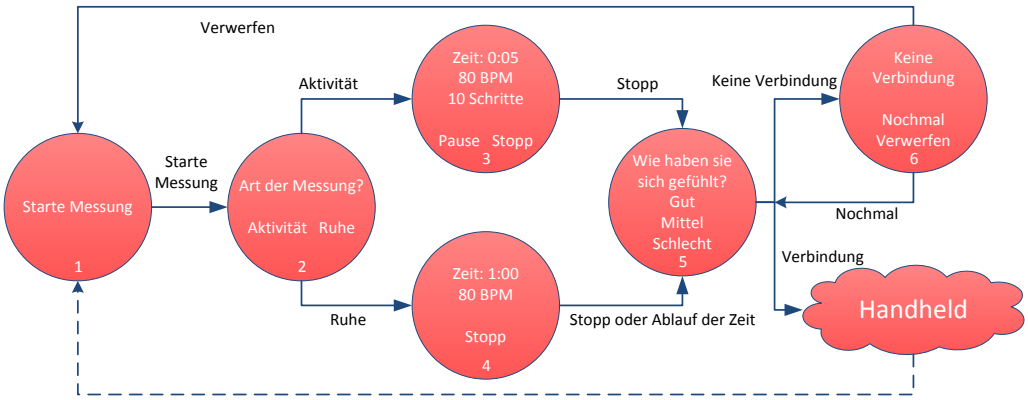

Abbildung 3: Struktur der Wearable-App

Das Handheld-Gerät soll sich kontinuierlich in Bereitschaft zum Empfang von Messdaten befinden und diese verarbeiten können. Zusätzlich soll die App auf dem Handheld-Gerät über eine grafische Oberfläche zur Konfiguration und zur temporären Verwaltung von Messdaten verfügen. Ein Graph soll die Messungen übersichtlich visualisieren. Eine Bewertung der Daten ist an dieser Stelle nicht erforderlich.

Das Handheld-Gerät soll die Messdaten per Bluetooth an ein Bluetooth-fähiges Endgerät weiter versenden können (z.B. PC oder Notebook). Es soll sowohl möglich sein, die Daten automatisch durch den Hintergrund-Dienst versenden zu lassen, als auch die Daten manuell über die grafische Oberfläche zu versenden. Die geplante Struktur der Handheld-App wird in Abbildung 4 dargestellt.

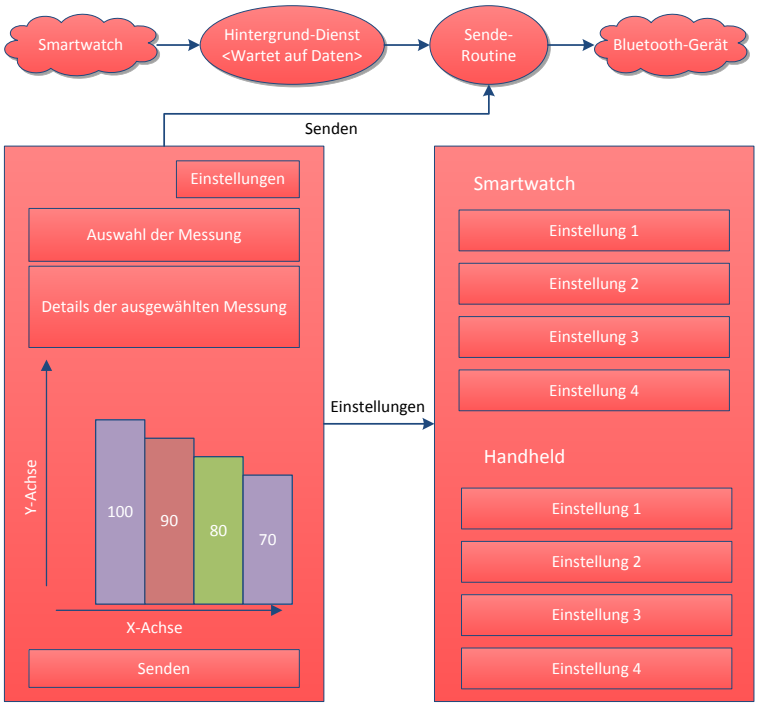

Abbildung 4: Struktur der Handheld-App

2.4 Vorbereitung

Im Rahmen des Projektes werden eine Motorola Moto 360 Smartwatch und ein Samsung Galaxy S4 Smartphone als Testgeräte verwendet. Auf der Moto 360 läuft Android Wear in der Version 5.0 und auf dem Galaxy S4 läuft Android in der Version 4.4 (Cyanogenmod 11 Custom-Rom). Die Geräte sind grundsätzlich gekoppelt und die gleichnamige Google App „Android Wear“ verwaltet auf dem Smartphone (im Hintergrund) die Verbindung zur Smartwatch. Die Entwicklung erfolgt mit Hilfe der Entwicklungsumgebung „Android Studio“, die ebenfalls von Google veröffentlicht wurde. Dabei kann die Smartphone-App direkt über USB „debugged“ werden, während der Debugging-Vorgang bei der Smartwatch über das gekoppelte Smartphone per Bluetooth gebrückt wird.

2.5 Einschränkungen

Bei Verwendung der Bluetooth-Library in der Qt-Framework Anwendung konnten Inkompatibilitäten nicht umgegangen werden (Näheres im Abschnitt QT Framework Anwendung). Aus diesem Grund soll die von der Handheld-App ausgehende Verbindung als TCP/IP Verbindung implementiert werden. Dabei soll ein UDP-Broadcast verwendet werden, um die Handheld-App im Netzwerk bekannt zu machen. Alternativ kann aber auch eine feste Adressierung angegeben werden. Da das Software-Paket sich letztlich an den gewöhnlichen Heimanwender richtet, ist grundsätzlich ein Heimnetzwerk mit Router und verbundenen Endgeräten zu erwarten und diese Anpassung sogar der Bluetooth-

Variante vorzuziehen, da Bluetooth-Hardware zwar an Notebooks, jedoch vergleichsweise selten an feststehenden PCs verfügbar ist.

2.6 Implementierung

Die Implementierung teilt sich in drei Unterabschnitte zur Beschreibung: Die gemeinsame Bibliothek, die Wearable-App und die Handheld-App.

2.6.1 Common-Library

Die beiden Apps für das Wearable- und das Handheld-Gerät teilen sich eine Bibliothek um gemeinsame Datenstrukturen einfacher zu organisieren. *HeartRateData* bildet einen Datensatz einer Messung ab, während *HeartRateMeasure* eine umfassende Messung inklusive Modus, Stimmung und Durchschnittswert repräsentiert. Hier finden sich zusätzlich Methoden zur Übertragung und Konvertierung der Daten. So werden die Messdaten über die Android Wearable-Message-API als String im CSV-Format gesendet. Die statische Klasse *HeartRateFile* bietet Methoden zum Laden und persistenten Speichern der genannten Datenstrukturen an. Diese Methoden werden nur auf dem Handheld-Gerät verwendet, jedoch könnte das Wearable-Gerät in einer möglichen Erweiterung auch eine persistente Speicherung der Daten vornehmen. In *ByteCodes* werden Prefix-Byte-Codes für die serielle Übertragung per TCP angegeben. Der Empfänger der Daten muss dabei ebenfalls diese Byte-Codes kennen. In Abbildung 5 werden diese Klassen dargestellt. Zusätzlich werden innerhalb der Common-Library alle verwendeten Grafiken zentral angelegt, um Redundanzen zu vermeiden.

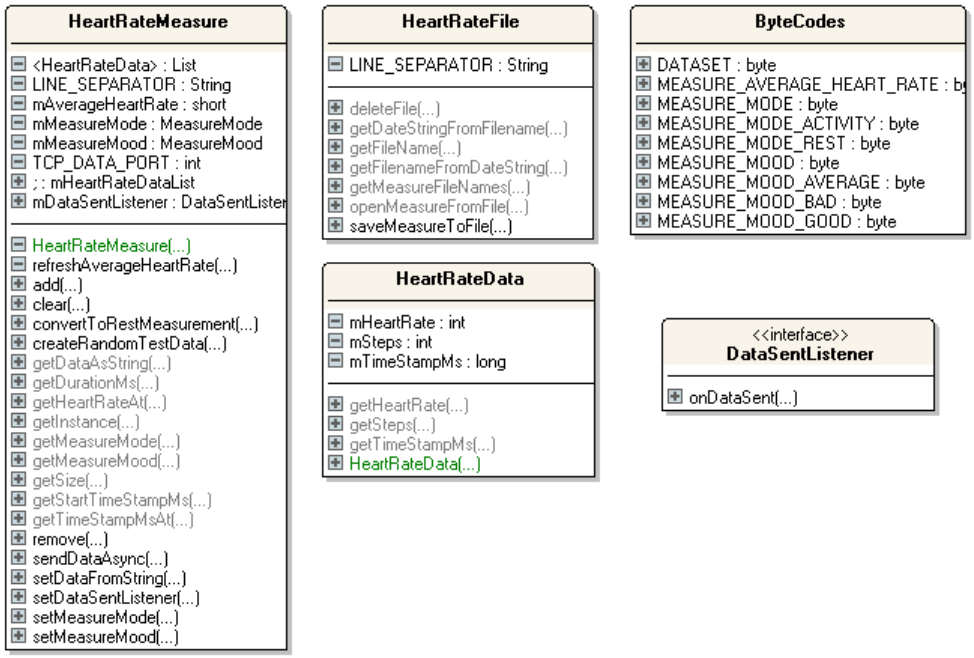

Abbildung 5: Klassendiagramm: Common-Library

2.6.2 Wearable-App

Die Wearable-App teilt sich in verschiedene Grundkomponenten. Dabei bildet die Klasse *WearActivity* das Zentrum der Ausführung. Sie wird von der Android-Klasse *Activity* abgeleitet und dazu verwendet um eine Programmlogik mit Layout-Inhalten zu verknüpfen. Das WearActivity-Layout (siehe Abbildung 6) enthält alle grafischen Elemente, die zur Funktionalität benötigt werden:

- Zeitzähler inkl. Symbol
- Schrittzähler inkl. Symbol
- Puls
- Startbutton / Pausebutton
- Stoppbutton
- Hintergrundanimation

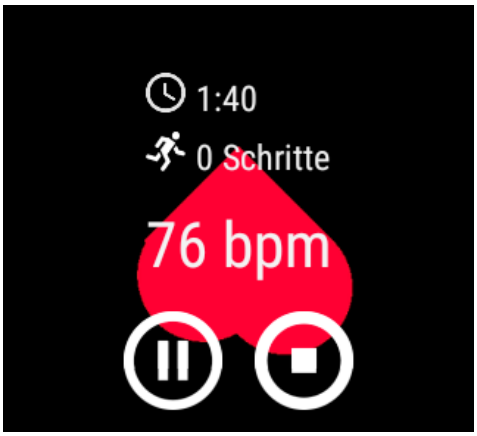

Abbildung 6: Layout: WearActivity

Die Hintergrundanimation (ein animiertes Herz-Symbol) besteht aus 32 Einzelbildern, die per XML-Animationsdatei aufeinander folgend abgespielt werden. Die Programmlogik der Hauptklasse *WearActivity* basiert auf einem Zustandsautomaten. Elemente, die im jeweiligen Zustand nicht benötigt werden, werden ausgeblendet und umgekehrt. Analog zur Nummerierung in Abbildung 3 finden sich die Zustände hier wieder. Die Zustände 2, 5 und 6 zur manuellen Abfrage von Informationen werden allerdings in externe Dialoge ausgelagert, da das Layout sich dort grundlegend ändert.

Diese Dialoge werden wiederum durch eigene Activitys repräsentiert und von der *WearActivity* mittels Aufruf der Methode *startActivity* gestartet. Sie liefern bei Beendigung einen entsprechenden Return-Wert. Abbildung 7 zeigt das Layout der drei Dialoge.

Abbildung 7: Layout: Dialoge

Bei der Gestaltung der Wearable-Layouts wird das *BoxInsetLayout* als Container verwendet, um eine äquivalente Darstellung auf quadratischen und runden Geräten zu erzielen und nur ein Layout für beide Display-Typen bereitstellen zu müssen.

Abgesehen von den Activity-Klassen gibt es noch andere Klassen, die an der Umsetzung der Programmlogik innerhalb der Wearable-App beteiligt sind. So gibt es beispielsweise die Klasse *RunningTimer*, die den Timer für Aktivitätsmessungen implementiert. Dabei kann eine beliebige Zeitspanne zum Auslösen des Events *onTimerUpdate* angegeben werden. Das Event wird innerhalb der *WearActivity*-Klasse genutzt um die Timer-Information auf der grafischen Oberfläche zu aktualisieren. Mit dem Event verbunden ist auch die regelmäßige Erinnerung (Vibrationsmuster bzw. Meldung), dass die App noch aktiv ist.

Daneben gibt es noch die Klasse *RestTimer*, die die Ruhemessung unterstützt. Hier wird eine festgelegte Zeitspanne rückwärts gezählt und im Anschluss das Event *onTimerFinished* ausgelöst. Analog zur Klasse *RunningTimer* gibt es ebenfalls ein Event *onTimerUpdate* zur Aktualisierung der Oberfläche.

Die wichtigste Komponente zur Erfassung der Sensor-Daten ist die Klasse *SensorLogger*, die den Zugriff auf den Puls- und Schrittsensor verwaltet. Innerhalb des internen Events *onSensorChanged* werden für beide Sensoren die entsprechenden Werte gespeichert und ein neues Event *onSensorLog* zur Verwendung in der *WearActivity*-Klasse abstrahiert. *SensorLogger* bietet Methoden zum Starten, Stoppen und Pausieren des Logging-Vorgangs an. Auch das Messungsintervall kann festgelegt werden.

Eine weitere wichtige Komponente ist die Singleton-Klasse *HeartRateDataSync*, die mit Hilfe der Wearable-Message-API einen String zu allen verbundenen Handheld-Geräten überträgt. Durch die Pfad-Angabe („/heartrate2go-message“) kann der Listener-Service am anderen Ende die App der Nachricht zuordnen.

Den letzten Bestandteil der Wearable-App bilden die Klassen *DataLayerListenerService* und *Settings*. Die Klasse *DataLayerListenerService* wird von der Android-Klasse *WearableListenerService* abgeleitet, die zur Verarbeitung eingehender Kommunikation seitens Wearable-API verwendet wird. Eine gleichnamige Klasse wird auf dem Handheld-Gerät ebenfalls verwendet, um die von *HeartRateDataSync* ausgehenden Nachrichten zu empfangen. Auf dem Wearable-Gerät wird die Implementierung dieser Struktur lediglich für den Empfang von Settings-Werten benötigt. Dabei werden die Einstellungen innerhalb einer Hash-Map übertragen, da die Einstellungen selbst auch als Key-Value-Paare vorliegen. Die Klasse *Settings* wird verwendet, um Zugriff auf die Klasse *SharedPreferences* und die enthalten Einstellungen zu abstrahieren.

Die Zusammenhänge werden im Klassendiagramm der Abbildung 8 noch einmal verdeutlicht.

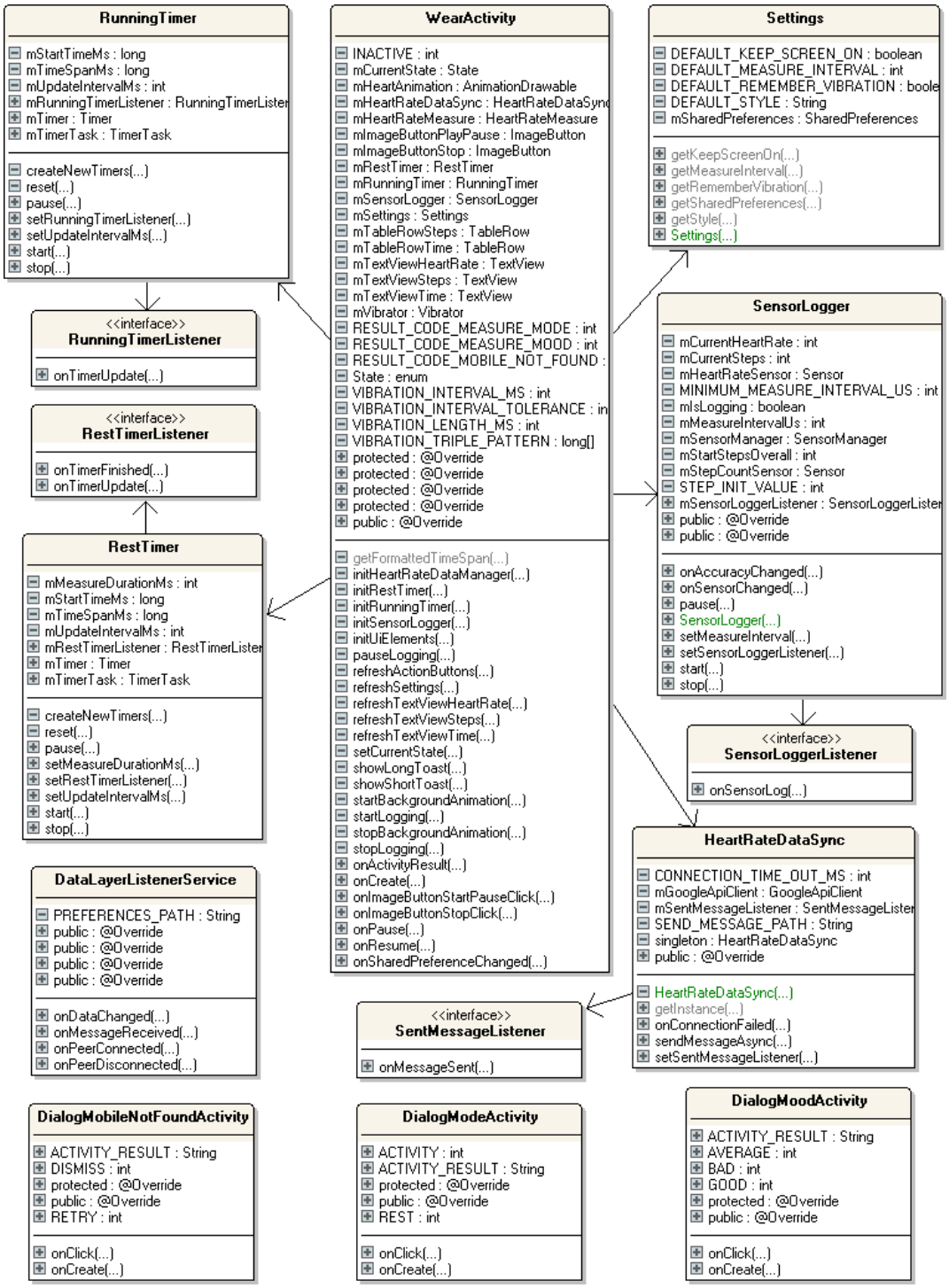

Abbildung 8: Klassendiagramm: Wearable-App

2.6.3 Handheld-App

Die Handheld-App dient primär zur Weiterübertragung der über Bluetooth empfangenen Mess-Daten an eine externe Anwendung über TCP. Daneben werden auch Daten grafisch als Diagramm dargestellt und persistent vorrätig gehalten. Zur Diagramm-Darstellung wird sich der freien Library *GraphView*[4] bedient. Über eine Drop-Down-Liste kann eine Messung zur Anzeige

ausgewählt werden. Eine ausgewählte Messung kann manuell über TCP versendet werden, gelöscht werden oder Beides gleichzeitig. Der Settings-Dialog der Handheld-App lässt verschiedene Einstellungen für Handheld-App und Wearable-App vornehmen, wie Abbildung 10 zeigt. Abbildung 11 zeigt ein Klassendiagramm der Handheld-App.

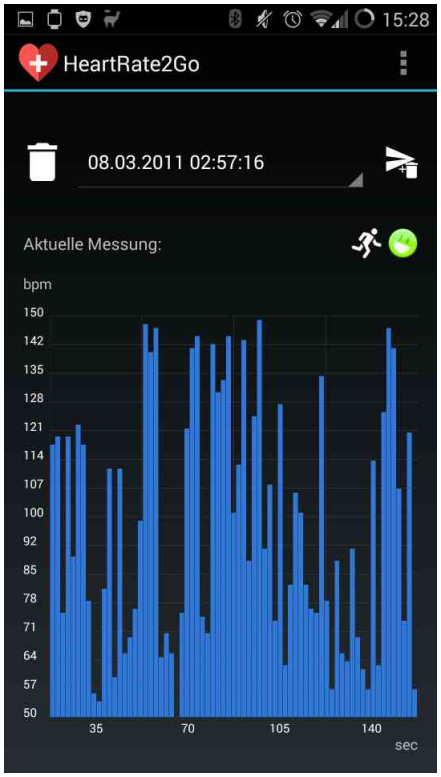

Abbildung 9: Layout: Handheld

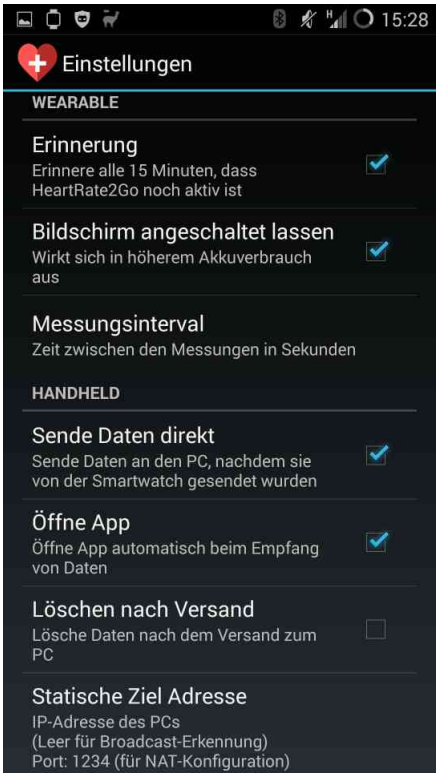

Abbildung 10: Layout: Settings

Die Handheld-App verfügt neben der Settings-Activity über eine Haupt-Activity *HandheldActivity* zur Darstellung der Oberflächeninhalte. Die beschriebenen, überschaubaren Funktionen sind hier zentral verfügbar. Abbildung 9 zeigt das zugehörige Layout.

Der Vorgang zum Senden der Messdaten über TCP ist innerhalb der Common-Klasse *HeartRateMeasure* als Methode zu finden. Als Parameter muss lediglich die IP-Adresse übergeben werden. Innerhalb dieser Methode werden die Objektdaten mit Hilfe der Prefix-Byte-Codes serialisiert und als Byte-Stream versendet. Im Anschluss an den Sende-Vorgang berichtet das Event *onDataSent* über den Status.

Die Klasse *NetworkBroadcast* dient zur einfachen Identifizierung vor TCP-Gegenstellen im Netzwerk. Hierfür wird ein Magic-Packet per UDP-Broadcast in das lokale Netz gesendet und das Event *onBroadcastFinished* ausgelöst, das bei Erfolg die IP-Adresse des Empfängers und dessen Antwort übergibt. Die Verwendung des Broadcasts ist einstellungsabhängig. Die Angabe einer statischen IP-Adresse ist ebenfalls möglich, womit der Broadcast umgangen wird.

Letztlich ist die Klasse *DataLayerListenerService* für den Empfang der Messdaten von der Wearable-App verantwortlich und bildet das zentrale Element der Handheld-App. Bei eingehenden Messdaten wird der Service gestartet und sendet - je nach Einstellung - die Daten im Hintergrund unmittelbar zur TCP-Gegenstelle weiter. Entsprechend wird, bei aktivierter Einstellung zum Starten der App, die Oberflächen-Activity beim Empfangen von Daten mit gestartet und zeigt die aktuelle Messung an.

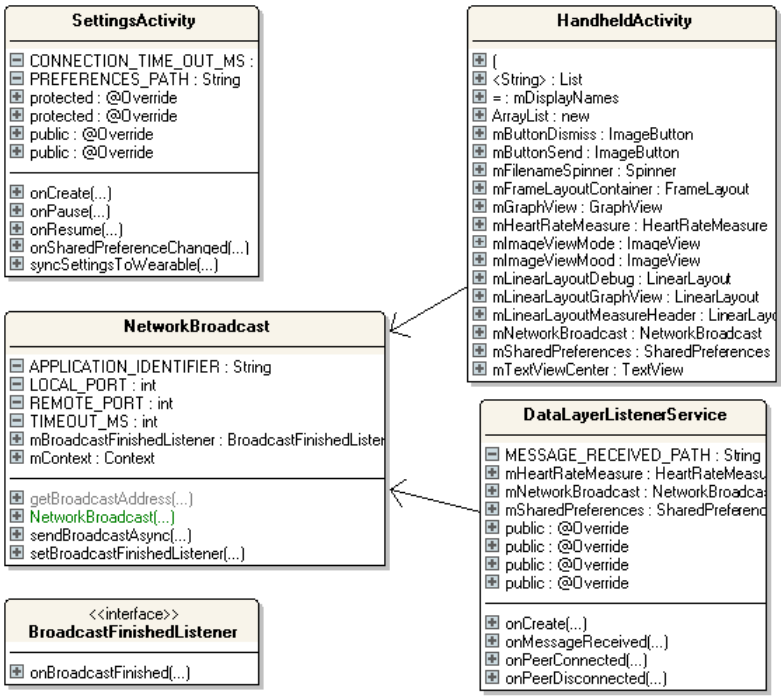

Abbildung 11: Klassendiagramm: Handheld-App

2.7 Evaluierung

Es werden unterschiedliche Testfälle hinsichtlich Fehlerverhalten bei den vorgesehenen Ausführungsroutinen angefertigt und geprüft. Die Testfälle und deren Ergebnisse werden tabellarisch aufgelistet.

2.7.1 Wearable-App

Im Vordergrund steht bei der Wearable-App die ständige Definiertheit des Zustandsautomaten. In folgender Tabelle 1 werden alle Zustände, Zustandsänderungen und Folgezustände aufgelistet und verifiziert. Die in Abbildung 3 gezeigte Zustandsabfolge gilt hier als Referenz.

Zustand	Änderung	Folgezustand	
Start (1)	Starte Messung	Messungsabfrage (2)	✓
Messungsabfrage (2)	Aktivität	Aktivitätsmessung (3)	✓
Aktivitätsmessung (3)	Pause	Aktivitätsmessung (3)	✓
Aktivitätsmessung (3)	Stopp	Stimmungsabfrage (5)	✓
Messungsabfrage (2)	Ruhe	Ruhemessung (4)	✓
Ruhemessung (4)	Stopp	Stimmungsabfrage (5)	✓
Stimmungsabfrage (5)	+ Verbindung	Start (1)	✓
Stimmungsabfrage (5)	- Verbindung	Retry-Abfrage (6)	✓
Retry-Abfrage (6)	Retry + Verbindung	Start (1)	✓
Retry-Abfrage (6)	Retry - Verbindung	Retry-Abfrage (6)	✓

Tabelle 1: Testfälle: Wearable

2.7.2 Handheld-App

Die Handheld-App muss in der Rolle des Daten-Vermittlers zwischen Wearable-Gerät und IP-Endgerät die Messdaten zuverlässig und unverändert weiterleiten. Die korrekte Darstellung von Messdaten ist ebenfalls ein Testkriterium. Außerdem wird die Anwendung und Übertragung von Einstellungen geprüft. Die nachstehende Tabellen 2 und 3 zeigen die Testkriterien auf.

Testfall	
Daten auf grafischer Oberfläche korrekt anzeigen	✓
Daten mit Hintergrund-Dienst versenden	✓
Daten mit grafischer Oberfläche versenden	✓
Toast über Versende-Status anzeigen	✓
Daten löschen	✓
Zufällige Daten erzeugen (Debugmodus)	✓
Alle Datensätze versenden (Debugmodus)	✓
Daten bei nur bei erfolgreicher Verbindung löschen	✓

Tabelle 2: Testfälle: Handheld-Verhalten

Einstellung	Typ	
Erinnerung	Wearable	✓
Bildschirm angeschaltet lassen	Wearable	✓
Messungsintervall	Wearable	✓
Sende Daten direkt	Handheld	✓
Öffne App automatisch	Handheld	✓
Löschen nach Versand	Handheld	✓
Statische Ziel-Adresse	Handheld	✓
Debug-Modus	Handheld	✓

Tabelle 3: Testfälle: Handheld-Einstellungen

3 Desktop Anwendung mittels Qt-Framework

Qt ist eine C++ Klassenbibliothek für die plattformunabhänige Entwicklung von Benutzeroberflächen, sowie normalen Konsolenapplikationen. Das primäre Ziel von Qt liegt in der Unterstützung zur Gestaltung von graphischen Oberflächen, sowie eine reibungslose Portierung der Applikation auf verschiedenen Betriebssystemen. Diesbezüglich stellt Qt zum einen ein Widget-System und zum anderen die Beschreibungssprache QML zur Verfügung. Ein weiteres wichtiges Merkmal von Qt liegt in der guten Unterstützung zur Internationalisierung von Applikationen. In diesem Zusammenhang stellt Qt mehrere kleine Toolkits (Qt Linguist, lupdate, lrelease) dem Entwickler bereit. Neben den zuvor genannten Punkten besitzt Qt eine Reihe weitere Funktionen, wie Beispielsweise die Netzwerkschnittstellen, Datenbankanbindungen, OpenGL, XML und die Anbindung zu anderen Programmiersprachen. Bezüglich der Anbindung zu anderen Programmiersprachen, besteht die Möglichkeit, das Qt Framework mit Hilfe der Programmiersprache Java zu verwenden. Abschließend sollte noch erwähnt werden, dass Qt mit der Erweiterung QtQuick versucht, die Entwicklung von Benutzeroberflächen auf mobilen Endgeräten dem Entwickler zu ermöglichen.
[5] [6] [7]

3.1 Auswahl des Qt-Framemworks

Zu Beginn des Projektes musste eine Entscheidung bezüglich des verwendeten Frameworks getroffen werden. In diesem Zusammenhang sind mehrere Frameworks analysiert und miteinander verglichen worden. In diesem Kontext ist die Entscheidung zugunsten des Qt-Frameworks gefallen. Im Nachfolgenden werden die wichtigsten Aspekte für die Entscheidung pro Qt erörtert.

Der wichtigste Aspekt für die Wahl des Qt-Frameworks war die plattformunabhägige Entwicklung und die daraus resultierende Portierung auf mehrere Endsysteme. Des Weiteren hat sich die leicht umsetzbare Internationalisierung durch die zu Hilfenahme der von Qt bereitgestellten Toolkits positiv auf die Entscheidungsfindung ausgewirkt. Ebenfalls positiv zu bewerten ist die sehr gute Online Dokumentation von Seiten Qt's und die reibungslose Kompatibilität zu C/C++. Ein weiterer Punkt der für das Qt Framework gesprochen hat, liegt in dem Umstand, dass ein Großteil der von Qt bereitgestellten Funktionalitäten sich mit den Anforderungen an die Applikation überschneidet. Abschließend sollte noch erwähnt werden, dass ein großes Interesse von Seiten der Teammitglieder bestand, eine Applikation mit Hilfe des Qt-Frames zu realisieren.
[5] [6] [7]

3.2 Model-View Konzept

In der Praxis werden viele User Interfaces mit dem MVC Pattern realisiert. Dieses Pattern besteht aus der View, dem Controller und dem Model. In diesem Zusammenhang wird eine strikte Trennung der einzelnen Schichten angestrebt.

Das Ziel dieses Ansatzes, liegt in dem Austausch der View, ohne eine Anpassung der internen Datenstruktur durchführen zu müssen. Eine weitere Variante dieses Konzeptes ist das Model/View Konzept. Hierbei wird die View mit dem Controller kombiniert. Die Aufgabe der View es ist mit Hilfe des Models dem Benutzer die Informationen anzuzeigen. Der Controller reagiert lediglich auf Interaktion des Benutzers mit der View. Zusätzlich kann bei einer Model/View Architektur das Konzept eines „delegates“ eingeführt werden. Dieser besitzt die Aufgabe, die einzelnen Datenelemente des Models benutzerspezifisch anzuzeigen oder auf bestimmte Veränderungen des Datenbestandes von Seiten des Benutzers auf der View zu reagieren. Das Zusammenspiel der einzelnen Komponenten wird in Abbildung 12 nochmals graphisch veranschaulicht.

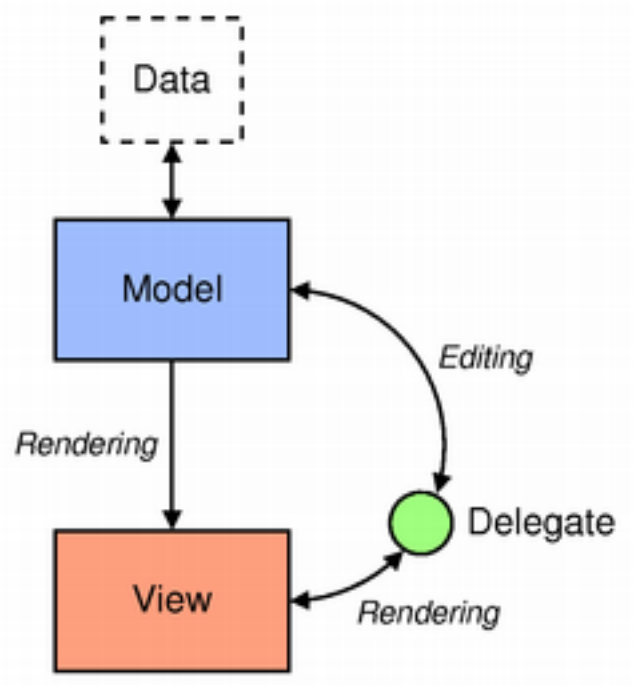

Abbildung 12: Veranschaulichung des Model-View-Konzepts

In Abbildung 12 wird deutlich, dass eine Trennung zwischen der Speicherung und der Darstellung der Daten besteht. Die Aufgabe des Models besteht darin, der View und dem Delegate minimalistische Schnittstellen für die Kommunikation bereitzustellen. Die Kommunikation zwischen den einzelnen Komponenten wird in Qt mit Hilfe des Signal und Slot Konzeptes realisiert. Findet eine Änderung am Datenbestand des Models statt, wird ein Signal an die View und den entsprechenden Delegate geschickt. Diese rufen die entsprechenden Slots auf und aktualisieren die View. Im umgekehrten Fall, wenn der Benutzer die Daten via View verändert, schickt diese ein Signal an das Model und den Delegate. Ein weiteres Hilfsmittel der View ist der Modelindex. Dieser Index wird verwendet, um die einzelnen Informationen aus dem Datenbestand zu lesen.

Ein Modelindex ist eine Referenz auf einen einzelnen Datensatz des Models. Durch die Zuhilfenahme eines Delegates, kann dieser benutzerspezifisch auf der View dargestellt werden. Diesbezüglich muss erwähnt werden, dass es mehrere Möglichkeiten für die Erstellung der View unter Qt existieren. Eine Möglichkeit

ist die Erstellung mittels Qt Widgets. Diese können durch Qt bereitgestellte Klassen erzeugt werden. Hierbei ist eine eindeutige Trennung zwischen View und Model schwer möglich. Eine andere Variante ist die Erstellung mittels QML. QML ist eine Art Beschreibungssprache für Benutzeroberflächen. Hierbei muss lediglich eine Beschreibung der GUI angegeben werden. Diese ist so allgemein gehalten, dass Designer oder UI-Entwickler ohne irgendwelche Vorkenntnisse einer Programmiersprache eine View anfertigen können. Der Vorteil hiervon liegt in der klaren Trennung der einzelnen Aufgaben. Ein Designer kann sich ausschließlich um das UI kümmern und ein Software-Entwickler um das dazugehörige Model und den Controller. Diesbezüglich existiert eine klare Trennung zwischen dem Model und der View.

In Qt sind alle Model Klassen von der Abstrakten Basisklasse QAbstractItemModel abgeleitet. Diese Basisklasse bietet eine Vielzahl an Schnittstellen für die Kommunikationen mit der View an. Um auf eine gegeben Datenstruktur besser reagieren zu können, bietet Qt 3 besondere Model-Typen an. Diese Modeltypen sind: QListModel, QTableModel und QTreeModel. Mit Hilfe dieser 3 Modeltypen, kann eine Vielzahl der Anwendungsbereiche abgedeckt werden. In diesem Projekt ist ausschließlich ein QAbstractListModel verwendet worden. Zur Umsetzung des Model View/Konzeptes wurde für die Bereitstellung der UI die Beschreibungssprache QML verwendet. Für die Implementierung des Models wurde die bereits von Qt bereitgestellte Klasse QAbstractListModel verwendet. Hierbei ist eine neue Unterklasse von QAbstractListModel erzeugt und die entsprechenden Methoden für die Kommunikation mit der View neu implementiert worden. Als Datenstruktur wurde eine QList mit entsprechenden Datenobjekten gewählt. Die Datenobjekte besitzen die Aufgabe, die einzelnen Messwerte einer Messung zu kapseln. Eine Vielzahl der QML Elemente wie beispielsweise eine ListView oder TableView bieten standardmäßig eine Property „model“ an, welche die Verknüpfung mit dem entsprechenden Model realisiert. Des Weiteren kann mittels QML ein delegate Objekt erzeugt werden. Mit dessen Hilfe, können beispielsweise die Einträge in einer ListView bestmöglich auf die Wünsche des End-Benutzers angepasst werden.

Der Vorteil des Qt Frameworks ist die automatische Anpassung der GUI an eine Änderung des internen Datenbestandes. Der Entwickler muss lediglich dem Model mitteilen, wann eine Änderung am Datenbestand des Models durchgeführt wurde. Infolgedessen übernimmt das Framework die komplette Aktualisierung der GUI. Der umgekehrte Fall, dass der Benutzer die Daten mittels GUI verändern kann, ist im Projektverlauf nicht implementiert worden.

Für die bestmögliche Darstellung der gesammelten Daten, mussten mehrere Diagramme erstellt werden. Hierbei bestand die Möglichkeit, alle Diagramme über die von Qt bereitgestellten Klassen zu erzeugen, oder ein bereits vorhandenes auf Qt basierendes Modul namens QCustomPlot zu verwenden. Dieses Modul kapselt die von Qt bereitgestellten „Paint“ Klassen und gibt dem Benutzer eine Vielzahl an bereits vorimplementierten Diagramm-Typen. Ursprünglich wurde diese Third-Party Modul für den Einsatz mit Qt Widgets konzipiert. Diesbezüglich musste eine Portierung in QML durchgeführt wer-

den. Folglich konnten einige Funktionalitäten wie beispielsweise das Zoomen nicht in QML überführt werden. Die Portierung ist mittels der QCustomPlot Support Seite durchgeführt worden.

[8] [9] [10] [11] [12] [13] [14]

3.3 Umsetzung mittels QML

Die Qt-Meta-Language kurz QML ist eine deklarative Programmiersprache für die Erstellung von Benutzeroberflächen. QML ist ein Teil von QtQuick und stellt eine Alternative zum Qt-Widget System dar. Die Hauptintention von QML ist die Erstellung von Benutzeroberflächen für Mobile-Systeme oder Desktop-Anwendungen. Hierfür ist eine einfache Syntax gewählt worden, sowie eine Integration von externem JavaScript Code. Infolgedessen, kann eine Trennung der Back-End und Front-End Entwicklung einer Applikation realisiert werden. Dieser Umstand zeigt einen großen Vorteil von QML gegenüber dem Qt-Widget System. Das Qt-Widget System, ermöglicht die Erstellung von Benutzeroberflächen mit Hilfe von C++ Code. Der Nachteil hiervon ist zum einen die fehlende Trennung von Applikation und Darstellungsschicht und zum anderen die mangelnde Übersicht im Quellcode bezüglich der View. Darüberhinaus ermöglicht QML die Erstellung von Animationen ohne zusätzliche Hilfsmittel. Des Weiteren können 3D- Animationen mittels OpenGl performanter und weniger fehleranfälliger als beim Qt-Widget System gezeichnet werden.

Durch die Verwendung von QML wird dem Entwickler eine Vielzahl an Freiheiten angeboten. Diese können beispielsweise für die Erstellung von eigenen Komponenten oder die benutzerdefinierte Anpassung von standardisierten Elementen genutzt werden. Im Verlauf des Projektes, sind mehrere QML Elemente benutzerspezifisch angepasst worden. Im Nachfolgenden wird eine QML spezifische Technik vorgestellt, wie ein QML Element durch den Entwickler manuell angepasst werden kann.

Für die Realisierung eines eigenen benutzerspezifischen QML Elements bietet QML eine Property namens „style“ an. Mit Hilfe dieser, kann die Standardimplementierung eines QML Elements überschrieben werden. In Abbildung 13 ist ein einfaches Tab Element mit Hilfe einer TabViewStyle, sowie einer neuen Tab-Komponente benutzerspezifisch angepasst worden.

Abbildung 13: Customized TabBar einer TabView

Hierfür ist zu Beginn eine neue QML-Datei mit dem Namen der neuen Tab-Komponenten angelegt worden. Anschließend wurde der „style“ Property der TabView Komponente, welche die einzelnen Tab-Elemente zusammenfasst, ein neues TabViewStyle Element zugewiesen. Dieses Element enthält alle spezifischen Anpassungen bezüglich des angepassten Tabs. In diesem Fall ist die Farbe, Größe, Schrift, Höhe bzw. Breite des Standard Tabs angepasst worden.

Ebenfalls ist ein neues Icon dem Tab hinzugefügt worden, um dem Anwender einen besseren Überblick auf die Applikation zu geben.

Abschließend sollte noch erwähnt werden, dass die Third-Party Komponente QCostumPlot (Erstellung der Diagramme) mit Hilfe einer eigenen QML Komponenten in die bestehende View integriert wurde. Hierfür musste ein neuer QML Type registriert und an der View angemeldet werden. Zusätzlich wurde das QML State System für das Aktualisieren der einzelnen Diagramme verwendet.
[15] [16] [17] [18]

3.4 Datenübertragung

3.4.1 Verzicht auf Bluetooth

Ursprünglich war geplant die Datenübertragung vom Smartphone zur Desktop-PC-Anwendung mittels Serial-Port-Profile oder RFCOMM über Bluetooth zu implementieren. Allerdings zeigte sich sehr früh beim Implementierungsversuch mittels der Qt-Bluetooth Klassen-Bibliothek *QBluetooth* einige Schwierigkeiten bezüglich der Plattformunabhängigkeit. *QBluetooth* unterstützt lediglich Linux, BlackBerry und Android[19]. Die nativen Bluetooth-Stacks von Windows und OS X werden nicht unterstützt. Da zu erwarten ist, dass der Großteil der Zielgruppe von HeartRate2Go Windows als Betriebssystem einsetzt, wurde die Alternative TCP/IP gewählt.

3.4.2 TCP-Server

Die Qt-Komponente der HeartRate2Go Anwendung agiert für die Datenübertragung als TCP-Server. Die Umsetzung wird mit Hilfe der *QTcpServer*-Klasse der Qt-Bibliothek vorgenommen[20]. Eingehende Verbindungen werden in der TcpConnection-Klasse verarbeitet, die sich von der *QThread*-Klasse ableitet und dadurch die asynchrone Verarbeitung mehrerer Verbindungen sicherstellt.

Wenn der TcpServer eine eingehende Verbindung annimmt, erzeugt er ein Objekt der TcpConnection-Klasse und übergibt ihr den Connection-Socket. Anschließend erhält der Thread des TcpConnection-Objekts die Kontrolle über diese Verbindung.

Die Thread-Loop der TcpConnection-Klasse wartet auf eingehende Bytes und hängt diese an ein *QByteArray*-Objekt an. Nach dem Trennen der Verbindung wird dieses *QByteArray*-Objekt zur weiteren Auswertung an den DataReceiver übergeben.

3.4.3 Server-Discovery

Um bei der Verwendung einer TCP/IP-Datenübertragung eine sichere Konnektivität zwischen Smartphone und Desktop-PC zu gewährleisten, muss der Anwender einen zusätzlichen Konfigurationsaufwand zur Ermittlung und Konfiguration der IP-Adresse in Kauf nehmen. Die HeartRate2Go Desktop-Anwendung

verfügt deshalb über einen UDP-Server, der mittels eines einfachen Broadcast-Protokolls die Server-Discovery automatisiert und so dem Anwender den Konfigurationsaufwand gänzlich erspart[1].

Die Implementierung des UDP-Servers in der BroadcastReceiver-Klasse erfolgt durch Ableitung von *QThread* und einem auf Port 45454 gebundenen *QUdpSocket*-Objekt[21]. Die Thread-Loop wartet anschließend auf eingehende Datagramme.

Beim Eintreffen eines Datagramms wird zunächst überprüft, ob es sich dabei um das erwartete Magic-Packet handelt. Ist dies der Fall, wird das Datagramm an den Absender an Port 45455 zurück gesendet. Der ursprüngliche Absender kann dadurch beim Empfang der Antwort die IP-Adresse des Servers ermitteln und eine TCP-Verbindung herstellen. Hat sich das Magic-Packet als falsch herausgestellt, wird das Datagramm verworfen.

Bei dem Magic-Packet handelt es sich um den ASCII-String der UUID "86417ce4-4f19-4a59-ae27-f404653e9751".

3.4.4 DataReceiver

Die DataReceiver-Klasse nimmt das eigentliche Parsing des vom Smartphone erhaltenen Byte-Vektors unter Beachtung der Network-Byte-Order vor. Sie abstrahiert lediglich die Byte-Vektor-Darstellung der Daten von der vom Modell benötigten Listen-Repräsentation. Die Klasse ist als Singleton implementiert, da sie aufgrund des verwendeten Signal-Slot-Konzepts instanziiert werden muss.

Der Parsevorgang wird mit Hilfe eines Zustandsautomaten realisiert, da die zu erwartenden Daten einem festgelegten Schema folgen müssen.

Nach einem erfolgreichen Parsevorgang werden die so gewonnenen Datensätze an die ImportExport-Klasse übermittelt und der FilterController mit einem Signal über das Eintreffen neuer Daten benachrichtigt.

In den Tabellen 4 und 5 wird das für die Datenübertragung verwendete Protokoll erläutert. Das *Data*-Datenfeld darf dabei als einziges Datenfeld mehrfach vorkommen und dient zur Übertragung der einzelnen vorgenommenen Messungen während eines Gesamt-Messvorgangs. Bei einer Ruhemessung ist hier ein einziger Datensatz zu erwarten. Bei einer Aktivitätsmessung ist die Anzahl der *Data*-Datensätze abhängig von der Dauer der Messung und des konfigurierten Messintervalls. Weiterhin dient ein leeres *Data*-Feld als Sentinel um das korrekte Ende der Übertragung (EOT) zu signalisieren.

[1] Für Netze in denen Broadcast-Messages blockiert, oder über Netz-Grenzen hinweg geroutet werden, ist weiterhin eine manuelle Konfiguration notwendig.

Bezeichnung	Byte	Länge	Beschreibung
Mode	`0x00`	1	Typ der Messung
Mood	`0x01`	1	Gefühlszustand bei der Messung
Average	`0x02`	2	Durchschnitt aller Messwerte
Data	`0xff`	12/0	Datensatz eines Messpunktes/EOT

Tabelle 4: Protokoll-Datenfelder der Datenübertragung (Länge in Byte)

Bezeichnung	Länge	Typ	Beschreibung
Zeitstempel	8	ulong	Zeitpunkt in Millisekunden des aktuellen Messpunktes
Puls	2	ushort	Pulswert dieses Messpunktes
Schritte	2	ushort	Anzahl der Schritte (inkrementell)

Tabelle 5: Aufbau des *Data*-Datenfeldes aus Tabelle 4

3.5 Datenhaltung

3.5.1 SQLite

Zur Speicherung sämtlicher Messungen, und damit verbundenen Datensätzen, wird auf die SQLite-Datenbankengine zurückgegriffen. Das Qt-Framework bietet mit der *QSqlDatabase*-Klasse eine Abstraktion zu vielen SQL-basierenden Datenbankengines an[22]. Unter anderem unterstützt sie die SQLite-Engine. Der SQLite-Datenbank-Container wird plattformunabhängig im Benutzerverzeichnis des die Anwendung ausführenden Benutzers im Verzeichnis "HeartRate" gespeichert.

3.5.2 Datenbankschema

Das Datenbankschema besteht aus den vier in Abbildung 14 dargestellten Entitäten *Measurement*, *Data*, *Mood* und *Type*.

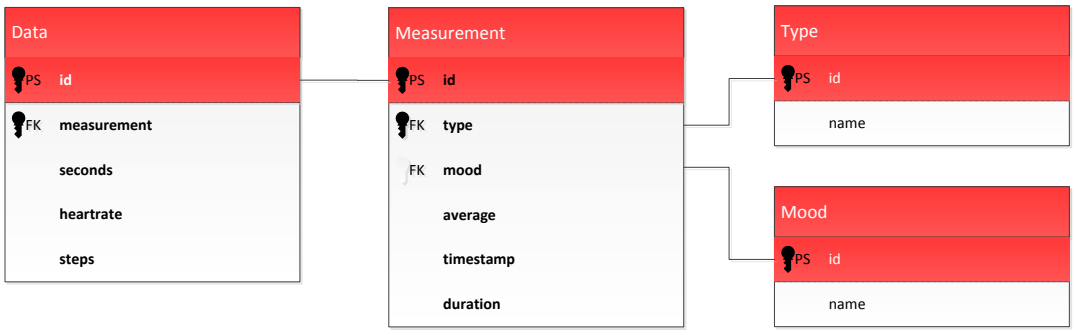

Abbildung 14: Das SQLite Datenbankschema

Tabelle 6 erläutert deren Aufgaben im Einzelnen. Zu einem *Measurement*-Eintrag gehören ein oder mehrere *Data*-Einträge. *Mood* und *Type* enthalten die möglichen Gefühlszustände und Arten von Messungen. In der aktuellen

Ausführung enthält *Mood* nur ”gut”, ”durchschnittlich” und ”schlecht”. *Type* enthält Ruhe- und Aktivitätsmessung.

DB-Tabelle	Beschreibung
Measurement	Speichert allgemeine Informationen **einer** Messung
Data	Speichert die einzelnen Messwerte zu einem *Measurement*-Eintrag
Mood	Speichert die möglichen Gefühlszustände, die jeder Messung zugeordnet werden
Type	Speichert die möglichen Messungstypen, die jeder Messung zugeordnet werden

Tabelle 6: Aufbau des *Data*-Datenfeldes aus Tabelle 4

3.5.3 Datenbankabfragen

Die Datenbankabfragen sind in der aktuellen Version recht einfach gehalten. Es gibt lediglich für das Eintragen neuer Messungen ein SQL-Insert Statement, welches innerhalb einer Transaktion alle Messwerte einer Messung in die Datenbank hinzufügt. Alle weiteren Datenbankabfragen sind als SQL-Select realisiert. Welche Anfragen abgesetzt werden ist abhängig von den vom Anwender definierten Filtern und seiner Auswahl einer expliziten Messung.

Abstrakt formuliert finden die in Tabelle 7 Abfragen Verwendung.

3.6 Evaluierung

In Bezug auf die Evaluierung der Desktop-Anwendung, werden unterschiedliche Testfälle hinsichtlich Fehlerverhalten bei den vorgesehenen Ausführungsroutinen angefertigt und geprüft. Die Testfälle und die dazugehörigen Ergebnisse der Evaluierung werden in tabellarischer Form aufgelistet.

3.6.1 Desktop Anwendung

Es ist wichtig, dass bei der Desktop Anwendung alle empfangen Daten in den verschiedenen Grafiken und Tabellen dargestellt und gespeichert werden. Die Tabelle 8 zeigt die Testkriterien im einzelnen auf. Da die Anwendung plattfromunabhänig ist, wird auf drei Betriebssystemen getestet, auf Windows 8.1, Mac OS Version 10.9.4 und Gentoo Linux 32 Bit Kernel 3.17.4-gentoo.

3.7 Probleme bezüglich Qt

Das Hauptaugenmerk dieses Kapitels liegt in der Beschreibung der aufgetreten Probleme bezüglich des Qt-Frameworks.
Das größte Problem das während des Projektverlaufes aufgetaucht ist, befasst sich mit der Druckfunktion. Um eine solche Funktion realisieren zu können, bietet Qt einen standardisierten Druckdialog an. Dieser Druckdialog soll laut Online-Dokumentation plattformunabhängig arbeiten. Im Weiteren Projektverlauf musste jedoch erkannt werden, dass diese Aussage nur teilweise

Abfrage	Beschreibung
Messung eintragen	Fügt zunächst die allgemeinen Informationen der Messung der Datenbank hinzuu. Anschließend alle einzelnen damit verbundenen Messwerte.
Messungen	Alle allgemeinen Informationen über vorhandene Messungen *eines Messungstyps* werden abgefragt. Wird verwendet um die Tabelle "Verfügbare Messungen" zu füllen wenn kein Filter gesetzt ist.
Messungen von-bis	Alle allgemeinen Informationen über vorhandene Messungen innerhalb eines definierten Zeitraums. Wird verwendet um die Tabelle "Verfügbare Messungen" zu füllen wenn ein Filter gesetzt ist.
Jahre	Liefert alle Jahreszahlen zurück, in denen Messungen vorliegen.
Monate	Liefert alle Monatsnamen zurück, in denen Messungen vorliegen.
Messwerte einer Messung	Liefert alle einzelnen Messwerte einer Messung zurück. Wird verwendet um den Graphen zu erzeugen.

Tabelle 7: Die verwendeten Datenbankabfraqen

korrekt ist. Diesbezüglich sollte erwähnt werden, dass unter dem Betriebssystem Windows 8.1 viele offene Bugs in Bezug auf den QPrintDialog vorhanden sind. Dies Ursache hatte einen kompletten Programmabsturz zur Folge. Des Weiteren konnte während der Evaluierungsphase verifiziert werden, dass unter speziellen Linuxdistributionen wie beispielsweise Gentoo, ebenfalls Probleme bezüglich der Druckfunktion existieren. Aus diesem Grund, kann lediglich auf dem Betriebssystem MacOs 10.9.4 eine korrekte Ausführung der Druckfunktion garantiert werden. Weitere Informationen zu dieser Thematik sind in der Evaluierung der Desktop-Anwendung zu finden.
Das nächste Problem befasst sich mit der Initialisierung der Controller-Klassen in Verbindung mit den QML-TabView-Elementen. In diesem Kontext sollte erwähnt werden, dass die Controller Klassen auf spezielle Signale oder Benutzereingaben von der View reagieren. In diesem Zusammenhang werden alle Controller-Klassen nach der Erstellung, aber vor der Anzeige der View initialisiert. Die einzelnen Controller-Klassen verwenden für den Zugriff von C++ Code auf einzelne QML Elemente den QObjectTree. Bezugnehmend auf das QML-Element TabView wird dieser QObjectTree erst erzeugt, wenn der entsprechende Tab durch den Benutzer selektiert wird. Infolgedessen manifestierte sich das Problem, dass bei der Initialisierung der einzelnen Controller-Klassen nicht auf die entsprechenden Signale von Seiten der View zugegriffen werden konnte, da der hierfür benötigte QObjetTree noch nicht vollständig vorhanden war. Zur Lösung dieses Problems, wurde vor der eigentlichen Initialisierung der einzelnen Controller, eine manuelle Selektion der einzelnen Tabs von Sei-

Testfall	Windows 8.1	Mac OS	Gentoo
Daten über TCP von Smartphone empfangen	✓	✓	✓
Unterscheidung der Daten zwischen Ruhe- und Aktivitätsmessung	✓	✓	✓
Anzeigen der Daten als Kurvendiagramm bei Aktivitätsmessung	✓	✓	✓
Anzeigen der Daten als Balkendiagramm bei Ruhemessung	✓	✓	✓
Dauerhafte Speicherung der aller Daten	✓	✓	✓
Darstellung in tabellarischer Form für beide Messtypen	✓	✓	✓
Mehrsprachigkeit	✓	✓	✓
Drucken der Daten	✗	✓	✗
Selektierung der Daten nach Datum	✓	✓	✓

Tabelle 8: Testfälle: Desktop Anwendung

ten der Applikation vorgenommen. Dieser Trick hatte zur Folge, dass bei der Bereitstellung der einzelnen Controller-Klassen, die benötigten Signale von Seiten der View mit den korrespondierenden Controller-Slots verknüpft werden konnten.

4 Fazit

Das Ziel dieses Kapitels ist eine kritische Stellungnahme bezüglich des Projektverlaufs. Hierfür wird ein Vergleich zwischen den geplanten und erreichten Zielen durchgeführt. Des Weiteren ist es von Vorteil, die positiven sowie negativen Aspekte, die während des Projektverlauf aufgetaucht sind, ausführlich zu diskutieren. Zum Abschluss des Kapitels werden gezielte Erweiterungsmöglichkeiten in Bezug auf spätere Projekte diskutiert.

4.1 Retrospektive

Bei der Umsetzung des geplanten Projektes traten nur wenige Schwierigkeiten auf. Die Arbeitsgruppe konnte durch Internet-Messaging zeitunabhängig kommunizieren und Gedanken, Anregungen, Kritik sowie Ideen austauschen. Die Projektdateien unterlagen vollständig einer Online-Versionskontrolle, um gemeinsame Zugriffe zu koordinieren und Änderungen angemessen protokollieren zu können. Durch die Trennung von Design und Programmlogik innerhalb der beiden verwendeten Frameworks konnten die Zuständigkeiten von Beginn an klar geregelt werden (siehe Tabelle 9).

Teilnehmer	Arbeitsaufgabe
Matthias Böffel	Android Programmlogik
Patrick Mathias	Qt Design
Markus Nebel	Qt Programmlogik
Janina Sauer	Android Design, medizinisches Wissen

Tabelle 9: Arbeitsaufgaben der Teilnehmer

Es konnten alle im Konzeptpapier geforderten Grundfunktionalitäten implementiert werden. Darüber hinaus wurden die meisten optionalen Features umgesetzt. Lediglich die Bewertung der Messdaten durch Benutzerprofile mit Anamnese-Werten und die Berechnung des Kalorienverbrauchs flossen aufgrund mangelnder Zeit nicht mehr in das Projekt mit ein.Außerdem musste die TCP-Übertragung als Alternative zur ausgehenden Bluetooth-Verbindung verwendet werden.

Das QT-Framework und das Android Framework wurden von den Teilnehmern bereits in anderen Projekten verwendet. Die QML-Komponente und das Android Wear Framework waren allerdings bis dahin unbekannt und erforderten Einarbeitung in die Materie. In den jeweiligen Abschnitten (Android: Abschnitt 2.5 / QT: Abschnitt 3.7) werden weitere Probleme bzw. komplexe Punkte aufgezeigt.

Die im Modulhandbuch angegebene Zeit von 102 Stunden (Selbststudium) wurde nicht eingehalten. Es wurden schätzungsweise im pro Teilnehmer Durchschnitt 150 Stunden zur Realisierung des Projektes aufgewendet. Neben dem Umfang der gesetzten Ziele, war auch der Enthusiasmus der Arbeitsgruppe für die aufgewendete Zeit verantwortlich.

4.2 Ausblick

Wie schon im Konzeptpapier erwähnt, besteht die Möglichkeit, das Projekt *HeartRate2Go* zu publizieren und es so anderen Anwendern zugänglich zu machen. Hierzu könnte es durch weitere Funktionen erweitert werden. Einige dieser Anwendungen finden sich schon im Konzeptpapier unter 2.b. Optionale Funktionen, zum Beispiel: das Anlegen von Benutzerprofilen.
Dies geschieht derzeit nur ansatzweise, die gesendeten Werte werden für jedes Benutzerprofil des Betriebssystems separat abgespeichert. Jedoch ist eine Anamneseabfrage noch nicht möglich. In dieser würde nach Alter, Geschlecht, Größe, Gewicht, maximaler und minimaler Pulswert für die beiden Messwerttypen gefragt werden. So wäre auch eine erste Einschätzung der gemessenen Werte möglich.

Ein anderer Punkt, ist die Berechnung des Kalorienverbrauchs. Zwar wird während einer Aktivitätsmessung die Anzahl der Schritte angezeigt, jedoch war es leider in der vorgegeben Zeit nicht priorisiert, der dadurch resultierende Kalorienverbrauch zu errechnen. Hierfür ist auch die Schrittlänge nötig, die mit einem Benutzerprofil einhergeht.

Des Weiteren stand zur Diskussion, ob dem Nutzer die Möglichkeit gegeben werden soll, Marker zu setzen, die eine besondere Situation kennzeichnen und in der späteren Ansicht speziell angezeigt werden. Dies ist bei einer, vom Hausarzt angeordneten, Langzeit-EKG-Messung ein wichtiger Teil, auch für die spätere Bewertung der Messung.

Da das *HeartRate2Go-Programm* auch auf dem Betriebssystemen MAC OS X läuft, wäre eine App für das iOS-Betriebssystem auch praktisch. Derzeit existiert die *HeartRate2Go-App* nur für Android. Die Erstellung einer iOS-App war jedoch leider nicht möglich, da hierfür keine passende Apple-Komponenten zur Verfügung standen.

Die Umsetzung der genannten Punkte scheiterte an der begrenzten Zeit, die für dieses Projekt zur Verfügung stand.

Literaturverzeichnis

[1] Dr. Uta Groger. Behandlungsassistenz in der Arztpraxis. 1. Auflage. 2007.

[2] Thomas Erler. *Meßprinzip der Pulsoximetrie*. http://edoc.hu-berlin.de. 2004. http://edoc.hu-berlin.de/habilitationen/erler-thomas-2002-12-03/HTML/chapter3.html (besucht am 10. 01. 2015).

[3] Google Inc. *Android Wear | Android Developers*. https://developer.android.com. 2015. https://developer.android.com/wear/index.html (besucht am 13. 01. 2015).

[4] Thomas Gehring. *Graph View*. http://www.android-graphview.org/. 2015. http://www.android-graphview.org/ (besucht am 13. 01. 2015).

[5] Helmut Herold. *Das Qt-Buch Portable GUI-Programmierung unter Linux/Unix/Windows*. 2014. http://www.millin.de/lp-3-89990-122-3.pdf (besucht am 27. 10. 2014).

[6] Helmut Herold. *C++, UML und Design Patterns: Grundlagen und Praxis der Objektorientierung Kapitel 29*. 2014. https://books.google.de/books?id=6NhV153imi4C&pg=PA777 (besucht am 27. 10. 2014).

[7] Prof. Dr. rer. nat. Gerhard Dikta. *Einführung in die GUI-Programmierung mit Qt 4*. 2014. https://www.matse.itc.rwth-aachen.de/dienste/public/show_document.php?id=8048 (besucht am 27. 10. 2014).

[8] Qt Documentation. *Model/View Concept*. http://doc.qt.io. 2014. http://qt-project.org/doc/qt-4.8/modelview.html (besucht am 01. 11. 2014).

[9] Qt Documentation. *Model/View Programming*. http://doc.qt.io. 2014. http://qt-project.org/doc/qt-4.8/model-view-programming.html (besucht am 01. 11. 2014).

[10] Qt Documentation. *Qt and QML Interaction*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html (besucht am 02. 11. 2014).

[11] Qt Documentation. *Exchanging Data between Qml and C++*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html (besucht am 03. 11. 2014).

[12] QCustomPlot. *Integrate QCustomPlot to Qml*. http://doc.qt.io. 2014. http://www.qcustomplot.com/index.php/support/forum/172 (besucht am 12. 11. 2014).

[13] QCustomPlot. *Setting up QCostumPlot*. http://doc.qt.io. 2014. http://www.qcustomplot.com/index.php/tutorials/settingup (besucht am 11. 11. 2014).

[14] Daniel Molkentin. The Book of QT 4.0. 1. Auflage. 2006.

[15] Qt Documentation. *Qml Events*. http://doc.qt.io. 2014. http://qt-project.org/doc/qt-4.8/qmlevents.html (besucht am 10. 11. 2014).

[16] Qt Documentation. *Qml Documentation*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qtqml-index.html (besucht am 10. 11. 2014).

[17] Qt Documentation. *Qml Tutorial*. http://doc.qt.io. 2014. http://qt-project.org/doc/qt-4.8/qml-tutorial.html (besucht am 10. 11. 2014).

[18] Qt Documentation. *TabViewStyle*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qml-qtquick-controls-styles-tabviewstyle.html.

[19] Qt Documentation. *Qt Bluetooth*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qtbluetooth-index.html (besucht am 10. 01. 2015).

[20] Qt Documentation. *QTcpServer Class*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qtcpserver.html (besucht am 23. 12. 2014).

[21] Qt Documentation. *QUdpSocket Class*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qudpsocket.html (besucht am 23. 12. 2014).

[22] Qt Documentation. *QSqlDatabase Class*. http://doc.qt.io. 2014. http://doc.qt.io/qt-5/qsqldatabase.html (besucht am 23. 12. 2014).