ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Realisierung eines Sensor-Frameworks für mobile Anwendungen und Integration von Sensorinformationen am Beispiel einer mobilen Fragebogen-Applikation

Masterarbeit an der Universität Ulm

Vorgelegt von:
Patrick Zeller
patrick.zeller@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Betreuer:
M.Sc. Johannes Schobel

2013

Fassung 19. November 2013

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Es gibt eine Vielzahl möglicher Einsatzbereiche mobiler Anwendungen mit Sensorunterstützung. Einige Anwendungen können durch die zusätzliche Integration von Sensordaten verbessert werden, während viele neue Anwendungsbereiche durch den Zugriff auf Sensordaten überhaupt erst ermöglicht werden. Viele unterschiedliche interne sowie externe Sensoren sind für die Verwendung mit mobilen Geräten verfügbar. Es existiert jedoch keine einheitliche Schnittstelle zum Zugriff auf diese Sensoren. Sensoren unterscheiden sich dabei nicht nur in ihrer Verbindungsart, d.h. ob es sich zum Beispiel um Bluetooth- oder USB-Sensoren handelt. Vielmehr sind darüber hinaus für jeden einzelnen Sensor unterschiedliche Kommunikationsprotokolle, Interaktionsmuster und Datenformate zu beachten. Dies erschwert insbesondere die Entwicklung von Anwendungen, die mehrere verschiedene Sensoren unterstützen sollen. Um dies zu ermöglichen wurde in dieser Arbeit ein Sensor-Framework konzipiert und entwickelt, das es Entwicklern mobiler Anwendungen ermöglicht über eine abstrakte, einheitliche Schnittstelle auf unterschiedliche interne und externe Sensoren zuzugreifen. Außerdem wurde eine mobile Anwendung aus dem medizinisch-psychologischen Bereich unter Berücksichtigung zusätzlicher Anforderungen auf Android portiert. Diese stellt ein Beispiel einer mobilen Anwendung dar, die durch die zusätzliche Integration von Sensordaten profitieren würde.

Inhaltsverzeichnis

1

Einleitung

Mobile Endgeräte wie Smartphones und Tablets erleben in der heutigen Zeit einen regelrechten Boom. So wurde beispielsweise auf der diesjährigen Google I/O verkündet, dass bereits 900 Millionen Android-Geräte aktiviert wurden, ein großer Teil davon Ende 2012 bzw. Anfang 2013 [Goo13]. Durch die Verfügbarkeit günstiger, immer leistungsfähigerer mobiler Endgeräte können diese für viele neue Anwendungsbereiche genutzt werden. Am Institut für Datenbanken und Informationssysteme der Universität Ulm entstanden so bereits einige Arbeiten, die sich mit mobilen Anwendungen im medizinischen und psychologischen Bereich beschäftigten. Im Projekt *MEDo* [Lan12] wurde eine iPad-Anwendung entwickelt, die die Visite im Krankenhaus durch eine prozessunterstützte, mobile Aufgabenverwaltung unterstützt. Darüber hinaus befassten sich einige Arbeiten mit der Umsetzung von papierbasierten Fragebögen aus dem Bereich der klinischen Psychologie als mobile Anwendung. Hier wurden die durch den Touchscreen ermöglich-

ten neuen Eingabemethoden untersucht und die automatische Auswertung und digitale Weiterverarbeitung der Antwortdaten ermöglicht [Sch13; Lie12; Mai12].

Im Rahmen dieser Projekte wurde klar, dass viele Anwendungen durch die zusätzliche Integration von Sensordaten profitieren würden. Außerdem gibt es viele weitere Einsatzbereiche für mobile Anwendungen, die durch den Zugriff auf Sensordaten überhaupt erst erschlossen werden können. In [Sch+13] werden einige realistische Beispiele für den Einsatz mobiler, sensorunterstützter Anwendungen im medizinischen Bereich dargestellt.

UC1 Ärztliche Visite im Krankenhaus: Im Rahmen der Visite muss zum Beispiel der Blutdruck oder Blutzucker eines Patienten gemessen werden. In diesem Fall können Ärzte durch eine mobile Anwendung mit Sensorintegration unterstützt werden, die diese Daten sammelt und beispielsweise auch in einer elektronischen Patientenakte abspeichert, um sie zu archivieren und zu dokumentieren. Es wäre weiter vorstellbar, dass diese Daten automatisch an ein System zur klinischen Entscheidungsunterstützung weitergeleitet werden, das aufgrund der gemessenen Daten notwendige Maßnahmen vorschlägt.

UC2 Rettungsdienst: In einem Rettungseinsatz können Vitaldaten mithilfe einer mobilen, sensorunterstützten Anwendung gemessen werden. Das Gerät kann hier beispielsweise anzeigen, ob bestimmte kritische Zustände vorliegen, zum Beispiel wenn der Patient hyperventiliert oder der Puls zu stark ansteigt oder abfällt.

UC3 Psychologische Fragebögen: Während ein Patient einen psychologischen Fragebogen auf einem mobilen Endgerät ausfüllt, können gleichzeitig Vitalparameter durch Sensoren aufgezeichnet werden. Hierdurch können weitere Informationen erfasst werden, die bei der späteren Auswertung des Fragebogens behilflich sein können. Steigt beispielsweise der Puls eines Patienten beim Beantworten einer Frage zu potentiellen Gewalterfahrungen während der Kindheit an, so lassen sich hierdurch zusätzliche Erkenntnisse gewinnen, die möglicherweise weit über die auf die Frage gegebene Antwort hinausgehen.

Im Zusammenhang mit dem *Open Data Kit Sensors Framework* [Ope13b], welches in Abschnitt 2.2 genauer betrachtet wird, werden weitere Einsatzbereiche sichtbar. Für

diese wurden an der University of Washington schon konkrete mobile Anwendungen implementiert [Cha+12].

UC4 Medizinische Untersuchungen: Für die Messung bestimmter Vitaldaten durch Personal ohne spezielle medizinische Qualifikation und außerhalb medizinischer Einrichtungen wurde eine entsprechende Anwendung für mobile Endgeräte entwickelt. Die gemessenen Daten können zur Diagnose automatisch an einen Arzt weitergeleitet und für weitere Zwecke archiviert werden. Die Oberfläche der Anwendung kann die Untersuchung dabei durch spezielle Hilfen, zum Beispiel in Form von Tutorial-Videos, unterstützen.

UC5 Überwachung physikalischer Prozesse: Für den Einsatz in Afrika wurde eine mobile Anwendung entwickelt, die den Pasteurisationsprozess von Muttermilch mithilfe von Temperatursensoren überwacht. Da HIV-infizierte Mütter das Virus über die Muttermilch auf ihr Kind übertragen können, wird so sichergestellt, dass der Erhitzungsprozess korrekt abläuft und somit alle Viren zuverlässig inaktiviert werden.

UC6 Überwachung der Kühlkette von Impfstoffen: Durch eine mobile Anwendung wurde überwacht, dass die Kühlkette beim Transport von Impfstoffen nicht unterbrochen wird. Werden Temperaturen außerhalb des zulässigen Bereichs festgestellt, kann automatisch eine entsprechende Benachrichtigung an verantwortliches Personal versandt werden.

UC7 Datensammlung in Feldstudien: In Äthiopien wurde eine Feldstudie durchgeführt, um den täglichen Zeitbedarf für die Beschaffung von Trinkwasser zu untersuchen. Im Rahmen dieser Studie wurden Wasserkanister mit Bewegungssensoren ausgestattet, die konstant Daten aufzeichnen. Mithilfe eines mobilen Endgeräts wurden diese Daten dann später ausgelesen, analysiert und weiterverarbeitet.

Auch im kommerziellen bzw. Fitness-Bereich haben bereits viele Unternehmen die Bedeutung von mobilen Anwendungen mit Sensorintegration erkannt. Beispiele hierfür sind *Nike+*, *Runtastic* und *Fitbit* [Nik13; run13; Fit13]. Anhand der vielfäligen möglichen Anwendungsbereiche zeigt sich, dass mobile, sensorunterstützte Anwendungen ein sehr großes Potenzial besitzen.

Viele unterschiedliche interne sowie externe Sensoren sind für die Verwendung mit mobilen Endgeräten verfügbar. Als interne Sensoren werden hier diejenigen Sensoren bezeichnet, die im Gerät fest verbaut sind. Mit externen Sensoren sind zusätzliche kabelgebundene oder kabellose Sensorgeräte gemeint. Dadurch, dass keine einheitliche Schnittstelle zum Zugriff auf interne und externe Sensoren existiert, ist die Integration verschiedener Sensoren für einen Anwendungsentwickler kompliziert. Sensoren unterscheiden sich dabei nicht nur in ihrer Verbindungsart, d.h. ob es sich zum Beispiel um Bluetooth- oder USB-Sensoren handelt. Vielmehr sind darüber hinaus für jeden einzelnen Sensor unterschiedliche Kommunikationsprotokolle, Interaktionsmuster und Datenformate zu beachten. Dies erschwert insbesondere die Entwicklung von Anwendungen, die nicht nur genau einen bestimmten Sensor unterstützen sollen.

In dieser Arbeit wurde daher ein Sensor-Framework entwickelt, das es mobilen Anwendungen ermöglicht über eine abstrakte, einheitliche Schnittstelle auf unterschiedliche interne und externe Sensoren zuzugreifen. Darüber hinaus wurde eine mobile Anwendung aus dem medizinisch-psychologischen Bereich unter Berücksichtigung zusätzlicher Anforderungen auf Android portiert. Diese stellt ein Beispiel einer mobilen Anwendung dar, die sich für die zusätzliche Integration von Sensordaten eignet.

1.1 Ziele

Ziel dieser Arbeit ist es ein Sensor-Framework zu entwickeln, das einfach in bereits bestehende oder zukünftig zu entwickelnde mobile Anwendungen integriert werden kann und diesen ermöglicht auf eine Vielzahl von verschiedenen internen und externen Sensoren zuzugreifen. Die Schnittstelle zum Zugriff auf Sensoren sollte dabei möglichst von sensorspezifischen Details abstrahieren. Der Kommunikationskanal des Sensors, sowie dessen spezifisches Kommunikationsprotokoll sollte durch das Sensor-Framework gekapselt werden. Das Ziel ist eine Schnittstelle zu entwerfen, die es einer Anwendung ermöglicht auf Sensoren nach bestimmten Klassifikationen zuzugreifen. Damit ist gemeint, dass das Framework beispielsweise eine Anfrage nach einem Sensor, der den Puls messen kann, durch die Angabe von passenden Sensoren beantworten kann.

Damit das Sensor-Framework praktisch beliebige Sensoren ansprechen kann, soll eine *Plug-and-Play*-Architektur entwickelt werden, durch die für jeden Sensor möglichst einfach ein Treiber geschrieben werden kann, um ihn so in das Framework zu integrieren. Das Sensor-Framework soll dabei so aufgebaut sein, dass das Schreiben eines Sensortreibers möglichst nur die Implementierung wirklich sensorspezifischer Logik erfordert, sowie keine Eingriffe im eigentlichen Sensor-Framework notwendig sind. Insbesondere Funktionaliät, die den Aufbau und die Verwaltung eines bestimmten Kommunikationskanals eines Sensors, also beispielsweise einer Bluetooth-Verbindung, betreffen, sollen bereits durch das Sensor-Framework auf einer semantisch höheren Ebene bereitgestellt werden.

1.2 Aufbau der Arbeit

Zunächst wird auf einige Grundlagen eingegangen und der Kontext dieser Arbeit durch die Diskussion verschiedener verwandter Arbeiten beleuchtet. Daraufhin wird auf die Entwicklung eines Sensor-Frameworks anhand der Anforderungen, einer darauf basierenden Konzeption und Architektur, und der konkreten Implementierung dargestellt. Anschließend wird die mobile Fragebogenanwendung vorgestellt, die als Beispielanwendung für die Integration von Sensorinformationen dient. Deren Entwicklung aus den Anforderungen und wenige wichtige Implementierungsaspekte werden kurz besprochen bevor die Verwendung des Sensor-Frameworks in mobilen Anwendungen diskutiert wird. Zuletzt werden die Ergebnisse dieser Arbeit zusammengefasst und einige über den Rahmen dieser Arbeit hinausgehende Thematiken in Form eines Ausblicks präsentiert.

2

Grundlagen und Stand der Technik

In diesem Kapitel werden zunächst einige Grundlagen erläutert. Um den Kontext der vorliegenden Arbeit darzustellen werden daraufhin verschiedene, mit dieser verwandte Arbeiten und Projekte aus dem kommerziellen und wissenschaftlichen Bereich diskutiert.

2.1 Grundlagen

Es folgen einige Erläuterungen zum Bluetooth-Standard und eine Darstellung der für diese Arbeit relevanten Sensoren. In diesem Zuge wird das Kommunikationsprotokoll eines Bluetooth-Sensors aus dem medizinischen Bereich genauer betrachtet.

2.1.1 Bluetooth

Das im Rahmen dieser Arbeit erstellte Sensor-Framework unterstützt unter anderem die Kommunikation mit Bluetooth-Sensoren. Daher werden an dieser Stelle einige zum besseren Verständnis der Arbeit notwendige Grundlagen im Bezug auf Bluetooth dargestellt.

Der Bluetooth-Standard wird von der *Bluetooth Special Interest Group (SIG)* verwaltet. Bluetooth wird für die kabellose Kommunikation über kurze Strecken eingesetzt. Aktuell ist diese Technologie sehr weit verbreitet und wird in vielen verschiedenen Bereichen eingesetzt. Unter anderem wird sie bereits seit Jahren von praktisch jedem Mobiltelefon bzw. Smartphone oder Tablet unterstützt. Grundsätzlich ist Bluetooth dazu gedacht Kabel zwischen verschiedenen Geräten zu ersetzen [Blu13b]. Die Hauptmerkmale der Bluetooth-Technologie sind dabei der geringe Stromverbrauch, sowie die niedrigen Kosten der benötigten Hardware. Gerade diese Eigenschaften führen dazu, dass in kleinen, günstigen Sensor-Geräten sehr häufig Bluetooth-Hardware steckt. Im Umfeld der Bluetooth-Technologie gibt es viele verschiedene Spezifikationen, Versionen, Protokolle und in dem jeweiligen Zusammenhang wichtige Begriffe. Im Folgenden wird auf diejenigen näher eingegangen, die im Rahmen dieser Arbeit relevant sind. Dies sind zum Beispiel Protokolle, die tatsächlich von den betrachteten Sensoren unterstützt werden und damit verbundene Begriffe, deren Verständnis daher im späteren Verlauf dieser Arbeit wichtig ist.

Bluetooth-Protokollstack: Abbildung 2.1 zeigt die für diese Arbeit relevante Form des Bluetooth-Protokollstacks. *Radio*, *Baseband*, *LMP* und *L2CAP* sind im *OSI-Modell* [Int94] auf den Schichten 1 und 2 einzuordnen[1]. Details zu diesen Protokollen werden hier nicht weiter diskutiert. *RFCOMM* stellt das verwendete Transport-Prototokoll dar. Die darüber liegende Schicht stellt die API für Anwendungen dar, die über das *Serial Port Profile* interagieren. Oberhalb befinden sich dann die Anwendungen, die miteinander kommunizieren wollen.

RFCOMM: *Radio Frequency Communication (RFCOMM)* stellt ein Transport-Prototokoll dar, über das im Prinzip eine serielle Schnittstelle emuliert wird. Wichtig ist in

[1] Schicht 1: Bitübertragungsschicht; Schicht 2: Sicherungsschicht

diesem Zusammenhang, dass auf dieser Ebene ein zuverlässiger Datenstrom bereitgestellt wird, ähnlich wie beim *Transmission Control Protocol (TCP)*. Baut eine Anwendung auf RFCOMM auf, so treten in empfangenen Datenströmen daher insbesondere keine Bit-Fehler auf und auch die Ordnung der übertragenen Bits bleibt erhalten.

Bluetooth-Profile: In Bluetooth-Profilen sind die Dienste festgelegt, die Geräte nutzen bzw. bereitstellen. Ein Bluetooth-Profil stellt im Prinzip einen vertikalen Schnitt durch den Bluetooth-Protokollstack dar [Pal13]. In einem Profil sind außerdem Optionen und Parameter für die darunterliegenden Protokollschichten definiert um Interoperabilität zwischen Geräten verschiedener Hersteller sicherzustellen. Alle in dieser Arbeit betrachteten Bluetooth-Sensoren setzen das *Serial Port Profile* und die damit verbundene zuverlässige Kommunikation über Bluetooth-Sockets ein. Es liegt daher nahe, dass dies auch bei vielen anderen Bluetooth-Sensoren der Fall sein wird und diese somit besonders leicht in das hier vorgestellte Sensor-Framework integriert werden können.

SDP: Mithilfe des *Service Discovery Protocol* wird erkannt, welche Dienste von einem Gerät angeboten werden. Die Dienste sind dabei durch einen *Universally Unique Identifier (UUID)* bezeichnet. Handelt es sich um offizielle Dienste, so nennt man diese Dienste Bluetooth-Profile.

Pairing: Bevor Geräte über Bluetooth miteinander kommunizieren können, ist es in der Regel notwendig ein *Pairing* durchzuführen, bei dem die Geräte permanent miteinander verknüpft werden. Bei diesem Prozess muss typischerweise eine PIN-Nummer an einem oder beiden zu verknüpfenden Geräten eingegeben werden. Daraufhin werden zwischen den Geräten Schlüssel ausgetauscht, die fortan für die gegenseitige Authentifizierung und Verschlüsselung der Kommunikation genutzt werden kann. Das Pairing ist ein einmaliger Prozess, der von dem tatsächlichen Verbindungsaufbau zwischen den Geräten zu unterscheiden ist. Sind die Geräte einmal auf diese Art miteinander verknüpft worden, kann eine Verbindung zwischen den Geräten anschließend beliebig oft und ohne weitere Nutzerinteraktion durchgeführt werden.

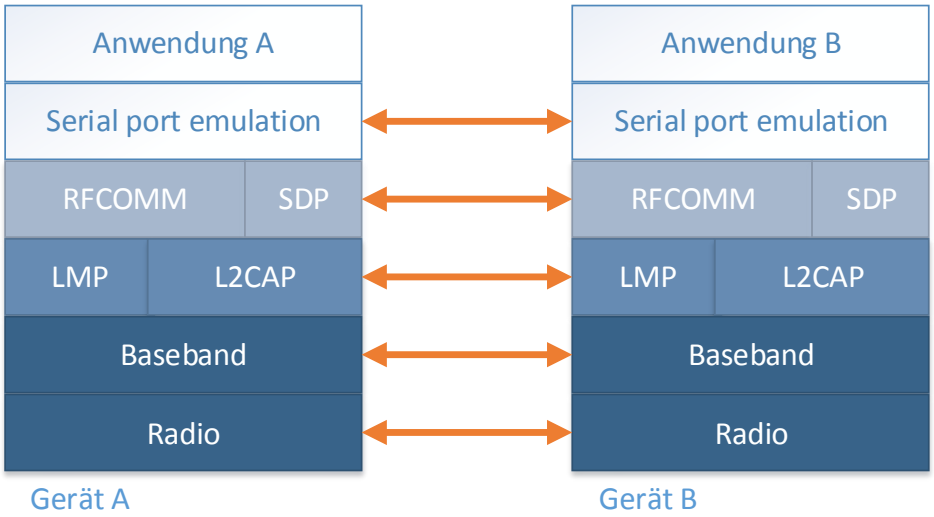


Abbildung 2.1: Vereinfachte Form des Bluetooth-Protokollstacks. Abbildung frei nach [Blu13c].

2.1.2 Sensoren

Es gibt sehr viele unterschiedliche Sensoren, die für die Integration in mobile Anwendungen infrage kommen. Diese können nach ihrer Art, d.h. was gemessen wird, und nach dem jeweils unterstützten Kommunikationskanal unterschieden werden. Nachfolgend werden beispielhaft einige externe Sensoren nach ihrer Art aufgezählt.

Sensoren zur Messung bestimmter Vitalparameter bzw. Sensoren aus dem tendenziell medizinischen Bereich sind beispielsweise Herzfrequenzmesser, EKG- und EEG-Messgeräte, Atmungs- und Atemfrequenz-Messgeräte, Alkohol-Tester, Blutzuckermesser, Oximeter und Blutdruckmesser. Darüber hinaus gibt es Umgebungssensoren zur Messung von Temperatur, Gas oder CO2, Licht und Luftfeuchtigkeit. Im Fitness-Bereich gibt es, neben einigen bereits unter den medizinischen Sensoren genannten Geräten, beispielsweise Schrittzähler, Aktivitätssensoren und Waagen zur Messung von Körperfett und Gewicht. Außerdem sind Gyrometer, Accelerometer, Kartenleser, GPS-Geräte und viele weitere externe Sensoren verfügbar.

Betrachtet man Sensoren nach dem jeweils verwendeten Kommunikationskanal, so sind dabei folgende Technologien zu unterscheiden:

Bluetooth: Bluetooth wurde aufgrund der hohen Relevanz bezüglich der in dieser Arbeit betrachteten Sensoren bereits in Abschnitt 2.1.1 vorgestellt.

USB: Für die Verwendung von USB-Sensoren ist in Bezug auf Android Folgendes zu beachten: Zwar besitzen praktisch alle Android-Geräte eine USB-Schnittstelle, jedoch wird nicht immer der *Host-Modus* unterstützt. Dies hat zur Folge, dass diese Geräte nur USB-Hardware unterstützen, die das *Android Accessory Communication Protocol* implementieren, die also explizit als Android-Peripherie ausgelegt sind. Diese eingeschränkte USB-Unterstützung betrifft selbst aktuelle Referenzgeräte aus der Nexus-Serie, wie beispielweise das Nexus 4.

NFC: *Near Field Communication (NFC)* ist eine Technologie zur kabellosen Datenübertragung über Entfernungen im Zentimeterbereich. Die geräteinterne NFC-Hardware kann als interner Sensor betrachtet werden, mit dem Daten von NFC-Tags gemessen werden. NFC könnte aber beispielsweise auch dazu genutzt werden, Daten von externen Sensoren auszulesen. In aktuellen Android-Geräten ist typischerweise NFC-Hardware vorhanden und auch Windows Phone unterstützt NFC seit Version 8. Bisher wird NFC nicht von Apple-Geräten unterstützt.

WiFi (IEEE 802.11): *WiFi* ermöglicht die kabellose Datenübertragung im 2,4- oder 5-GHz-Band. Diese Übertragungsart wird von sehr vielen mobilen Endgeräten unterstützt und kann auch für die Kommunikation mit externen Sensoren eingesetzt werden. Ein Beispiel eines externen Sensors, der Daten mit dieser Technik überträgt, ist eine intelligente Waage aus dem Projekt *Fitbit*, welches in Abschnitt 2.2.1 kurz vorgestellt wird.

ZigBee: *ZigBee* ist ein Standard, der sich vor allem für drahtlose Ad-Hoc-Netzwerke bzw. Sensor-Netzwerke eignet. Um ZigBee mit mobilen Endgeräten zu verwenden wird zum jetzigen Zeitpunkt meist zusätzliche Hardware benötigt. Mit ZigBee könnte ein mobiles Endgerät beispielsweise direkt mit Sensornetzwerken interagieren. Ein solches Szenario steht in dieser Arbeit jedoch nicht direkt im Fokus.

ANT: *ANT* ist ein weiterer kabelloser Übertragungsstandard im 2,4-GHz-Band. Einige wenige Smartphones, beispielsweise die aktuellen High-End-Geräte von Samsung und einige Sony-Geräte, unterstützen diesen Standard direkt durch

entsprechende integrierte Hardwarekomponenten. Ansonsten kann ANT über zusätzliche Hardware nachgerüstet werden. Für Android wird von *ANT Wireless* ein *Software Development Kit (SDK)* für die Entwicklung von Anwendungen mit ANT-Unterstützung bereitgestellt. *ANT+* erweitert den Standard durch eine zusätzliche Interoperabilitätsschicht, in der es beispielsweise Profile für Sensoren wie Herzfrequenz-Messgeräte gibt. Da externe Sensoren explizit als Anwendungsszenario von ANT+ dargestellt werden, wäre die Unterstützung von ANT als eine mögliche Erweiterung des hier vorgestellten Sensor-Frameworks zu berücksichtigen.

Interne Gerätesensoren: Neben externen Sensoren können auch interne Sensoren über die einheitliche Schnittstelle des in dieser Arbeit vorgestellten Frameworks angesprochen werden. Dabei tritt die gewünschte Vereinfachung des Zugriffs auf Sensoren jedoch vor allem bei den externen Sensoren ein, da die Android-API bereits eine Schnittstelle zum Zugriff auf interne Sensoren bietet. Dennoch ist die zusätzliche Integration der internen Sensoren in das Sensor-Framework sinnvoll, da hierdurch eine einheitliche Schnittstelle für alle Sensoren entsteht. Es ist zudem zu beachten, dass auch interne Sensoren wie Kamera und Mikrofon in das Sensor-Framework integriert wurden. Diese zählen in der Android-API nicht zu den Sensoren und werden über separate Schnittstellen angesprochen. Aktuell unterstützt Android bereits 20 verschiedene interne Sensoren. Android 4.4, welches wenige Tage vor Abschluss dieser Arbeit[2] veröffentlicht wurde, unterstützt neben einigen neuen Sensorarten erstmals die Kommunikation über eine geräteinterne Infrarotschnittstelle, welche im Prinzip einen weiteren Sensor darstellt, der ähnlich wie die Kamera in der Android-API nicht zu den Sensoren gezählt wird.

Als ein Beispiel eines Bluetooth-Sensors aus dem medizinischen Bereich wurde in dieser Arbeit das *MedChoice Oximeter MD300C318T* betrachtet. Es handelt sich dabei um ein sogenanntes *Pulsoximeter* in Form eines kleinen Finger-Clips, mit dem sowohl die Sauerstoffsättigung als auch gleichzeitig die Herzfrequenz gemessen werden kann. Die Messung am Finger erfolgt durch einen Lichtsensor. Sie basiert auf dem Prinzip, dass mit Sauerstoff gesättigtes Hämoglobin Licht im Rot- bis Infrarotbereich im Ver-

[2] Android 4.4 wurde am 31.10.2013 veröffentlicht.

gleich zu ungesättigtem Hämoglobin unterschiedlich stark absorbiert [Cho11a]. Zwei Dioden im Sensor emittieren Licht in unterschiedlichen Wellenlängen in Richtung eines Lichtsensors. Zur Messung wird der Finger in den Sensor gesteckt, wodurch das Licht der Dioden teilweise absorbiert wird. Durch den Grad dieser Absorption lässt sich die Sauerstoffsättigung des Bluts messen.

Im Sensor-Framework, das zuvor im Rahmen einer Bachelorarbeit entwickelt wurde [Lie12], wurde das MedChoice Oximeter bereits verwendet. Über den Hersteller konnte hier eine sehr ausführliche Dokumentation über das genaue Kommunikationsprotokoll des Sensors bezogen werden, die auch für die hier vorgestellte Arbeit von großem Nutzen war. Als Beispiel eines sensorspezifischen Kommunikationsprotokolls wird dieses Protokoll im Folgenden genauer dargestellt. Das Protokoll ist im bereits in Abbildung 2.1 gezeigten Bluetooth-Stack auf der Anwendungsebene einzuordnen.

MD300C318T Bluetooth-Kommunikationsprotokoll

Das Gerät kann entweder im *Real-Time-Modus* oder im *Non-Real-Time-Modus* betrieben werden. In dieser Arbeit wird ausschließlich der Real-Time-Modus betrachtet, da dieser sich aufgrund der Echtzeit-Messung für eine medizinische Anwendung besser eignet. In diesem Modus werden nach erfolgreicher Synchronisation jede Sekunde neue Messdaten übermittelt. Generell gibt es hier drei verschiedene Nachrichten- bzw. Paketformate.

Abbildung 2.2: Format der 6-Byte-Kommandonachricht des Masters.

Abbildung 2.2 zeigt den Aufbau einer Kommandonachricht, die im 6-Byte-Format vorliegt. Dieses Format wird für Nachrichten vom *Master* zum Oximeter verwendet. Der Master ist dabei das Gerät mit dem das Oximeter verbunden ist, also in unserem Fall beispielsweise ein Tablet. Mit der Kommandonachricht werden Befehle zum Oximeter gesendet,

beispielsweise kann die Synchronisierung gestartet oder das Oximeter abgeschaltet werden.

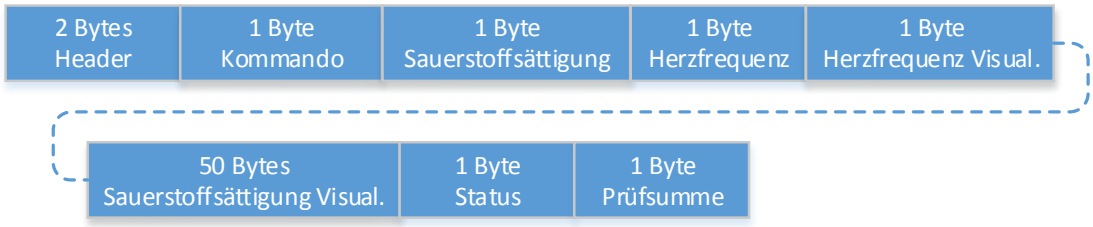


Abbildung 2.3: Nachricht vom Oximeters im 58-Byte-Format.

Der Aufbau des zweiten Paketformats wird in Abbildung 2.3 gezeigt. Dieses Format wird verwendet um im Real-Time-Modus Messdaten vom Sensor zum Master zu senden. Am Anfang des Pakets befindet sich ein *Header* (immer 0x55 0xAA) und das Kommando-Feld (immer 0xF6). Darauf folgen mit der gemessenen Sauerstoffsättigung und Herzfrequenz die eigentlichen Messdaten. Außerdem enthält das Paket noch weitere Daten wie die Pulsamplitude, die auch auf dem Display des Oximeters sichtbar sind. Das dritte Paketformat wird unter anderem für Bestätigungsnachrichten des Oximeters und für Messdaten im Non-Real-Time-Modus verwendet. Der genaue Aufbau dieses Formats wird hier nicht näher dargestellt, da es im Real-Time-Modus hauptsächlich für Bestätigungsnachrichten eingesetzt wird.

In Abbildung 2.4 wird gezeigt, wie mithilfe der soeben beschriebenen Nachrichten die Kommunikation eines Tablets mit dem Oximeter erfolgt. Zunächst muss einmalig ein Pairing zwischen dem Oximeter und dem Tablet stattfinden. Darauf folgt der Bluetooth-Verbindungsaufbau, wodurch der Kommunikationskanal zwischen Sensor und Tablet für das sensorspezifische Kommunikationsprotokoll hergestellt wird. Daraufhin sendet das Tablet eine Synchronisierungsanfrage im 6-Byte-Kommandoformat. Das Oximeter antwortet mit einer Bestätigungsnachricht. Nach Empfang dieser Nachricht sendet das Tablet eine Anfrage nach Test-Daten. Das Oximeter Antwortet mit den Test-Daten im 58-Byte-Format. Der Austausch dieser Test-Daten ist lediglich ein notwendiger Schritt in einer Art weiterem Verbindungsaufbau im Rahmen des Kommunikationsprotokolls. Es werden hier noch keine tatsächlich gemessenen Daten ausgetauscht. Das Tablet

bestätigt daraufhin den Empfang der Test-Daten. Von nun an sendet das Oximeter jede Sekunde eine Nachricht mit den gemessenen Daten zur Sauerstoffsättigung und der Herzfrequenz. Das Tablet muss alle 5 Sekunden eine weitere Bestätigungsnachricht senden. Empfängt das Oximeter 30 Sekunden lang keine Bestätigung, trennt es die Bluetooth-Verbindung automatisch. Wird das Oximeter nicht mehr benötigt, kann das Tablet eine entsprechende Kommandonachricht senden, durch die sich das Oximeter abschalten lässt.

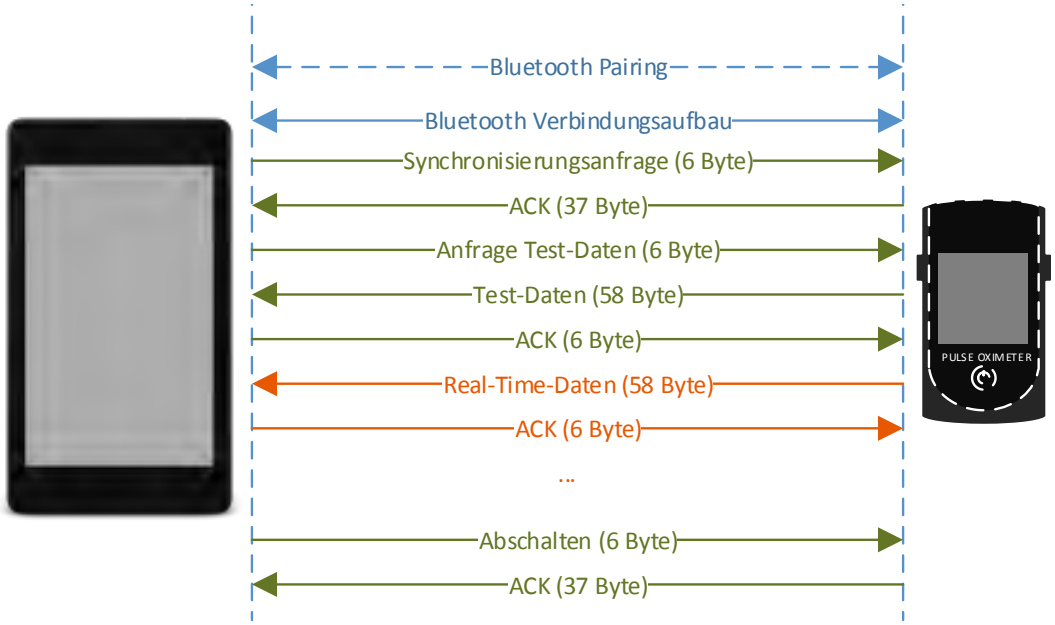


Abbildung 2.4: Kommunikationsprotokoll MedChoice Oximeter MD300C318T

2.2 Verwandte Arbeiten

Um den Kontext in dem die vorliegende Arbeit zu sehen ist besser verstehen zu können, werden nachfolgend einige verwandte Arbeiten angesprochen. Hierbei wird zwischen Projekten aus dem kommerziellen und dem wissenschaftlichen Bereich unterschieden.

2.2.1 Kommerzieller Bereich

Ein Beispiel für eine mobile Anwendung mit Unterstützung für externe Sensoren aus dem kommerziellen Bereich ist *Nike+* [Nik13]. Unter diesem Namen werden unter anderem verschiedene mobile Anwendungen angeboten, über die mithilfe von Sensoren Trainingsaktivitäten bzw. allgemein alle Bewegungen aufgezeichnet werden können. Über den Vergleich der eigenen Aktivität mit der von Freunden und mit virtuellen Trophäen sollen so Trainingsanreize geschaffen werden. Unter iOS können von Nike angebotene externe Sensoren mit der Anwendung verwendet werden. Mithilfe eines in den Schuh eingesetzten Sensors können so während des Laufens Zeit, Entfernung und Geschwindigkeit aufgezeichnet werden. Außerdem können durch ein Armband für alle Arten von Aktivitäten Bewegungen aufgezeichnet werden. Mit der von Nike angebotenen Android-Anwendung können diese externen Sensoren nicht verwendet werden. Diese greift lediglich auf die internen Gerätesensoren zurück.

Ein ähnliches System bietet *runtastic* an [run13]. Hier können verschiedene externe Pulssensoren mit einem Smartphone verbunden werden, um so während des Trainings die GPS-Position und die Herzfrequenz aufzuzeichnen. Die aufgezeichneten Daten und erzielten Erfolge können anschließend analysiert und mit anderen Nutzern verglichen werden.

Ein weiteres Beispiel stellt *Fitbit* dar [Fit13]. Für dieses System wird ein *Fitbit-Tracker* benötigt, ein externer Sensor, der direkt über den Online-Shop von Fitbit gekauft werden kann. Es handelt sich dabei um einen Beschleunigungsmesser, modellabhängig mit zusätzlichem Höhenmesser, der Daten wie die gegangenen Schritte, zurückgelegte Strecke, verbrannte Kalorien, Minuten mit Aktivität, geschlafene Stunden und Schlafqualität aufzeichnet. Mit der Fitbit-App können die so aufgezeichneten Daten automatisch per Bluetooth mit einer beschränkten Auswahl von Apple- und Samsung-Smartphones synchronisiert werden. Anschließend können auch hier die aufgezeichneten Daten analysiert und erreichte Erfolge mit Freunden verglichen werden.

All diesen kommerziellen Projekten liegt ein mehr oder weniger geschlossenes Ökosystem von Sensoren und Anwendungen zugrunde. Nur vereinzelt lassen sich Sensoren etablierter Fremdanbieter, wie zum Beispiel Pulssensoren von Polar, mit dem jeweiligen

System verwenden. Aufgrund der proprietären Natur der mobilen Anwendungen und ihrer geschlossenen Kommunikation mit den externen Sensoren lassen sich hier nicht viele Erkenntnisse für das in dieser Arbeit zu erstellende Sensor-Framework gewinnen. Es ist jedoch festzuhalten, dass es zum Beispiel im Fitness-Bereich bereits einige etablierte mobile Anwendungen mit Unterstützung für externe Sensoren gibt. Der Fokus liegt in dieser Arbeit jedoch eher auf Anwendungen im medizinischen bzw. psychologischen Bereich.

2.2.2 Wissenschaftlicher Bereich

Am Institut für Datenbanken und Informationssysteme der Universität Ulm entstanden bereits einige wissenschaftliche Arbeiten, die sich mit mobilen Anwendungen im medizinischen und psychologischen Bereich beschäftigten. Als potenzielle Kandidaten für die zusätzliche Integration von Sensordaten wurden mobile Fragebogenanwendungen identifiziert. Einige papierbasierte Fragebögen aus der klinischen Psychologie wurden bereits als mobile Anwendung realisiert. So zum Beispiel der *MACE-Test (Modified Adverse Childhood Experiences)*, ein psychologischer Fragebogen zum Erfassen und Auswerten belastender Kindheitserlebnisse [Sch13]. Weitere Arbeiten beschäftigten sich mit dem *KINDEX Mum Screen* [Lie12; Mai12]. Mit diesem Fragebogen können bestimmte Risikofaktoren der kindlichen Entwicklung bereits während der Schwangerschaft erkannt werden, um eine Schwangere entsprechend der erkannten Risikofaktoren unterstützen zu können. Außerdem wurde eine mobile Fragebogenanwendung zur Untersuchung der Traumatisierung burundischer Soldaten durch Kampfeinsätze entwickelt [Lie12].

Es existieren auch bereits wissenschaftliche Arbeiten, die sich mit der Entwicklung eines Sensor-Frameworks für mobile Geräte befassen. Dandelion [LRZ10] ist ein System, durch das die Entwicklung sensorunterstützter Anwendungen erleichtert werden soll. Sensorhersteller müssen dabei eine Laufzeitumgebung bereitstellen, über die sogenannte *Senselets* auf den Sensoren selbst ausgeführt werden können. Ein Anwendungsentwickler kann dann solche plattformunabhängige Senselets schreiben und muss sich dadurch nicht mit plattformabhängigen Programmiersprachen und weiteren plattformspezifischen Details beschäftigen. Dieser Ansatz wird in dieser Arbeit nicht

verfolgt, da hierdurch nur Sensoren unterstützt werden können, auf denen die benötigte Laufzeitumgebung vorhanden ist. Dabei ist fraglich, ob Sensorhersteller dazu veranlasst werden könnten ihre Sensoren entsprechend anzupassen.

Im Rahmen des Projekts *Open Data Kit (ODK)* [Ope13a], einer Sammlung von Werkzeugen zur mobilen Datenerfassung, wurde ein anderes Sensor-Framework [Bru+12] entwickelt. Ähnlich wie in dieser Arbeit ist das Ziel mit dem Open Data Kit Sensors Framework, Entwicklern von mobilen, sensorunterstützten Anwendungen eine einheitliche Schnittstelle zum Zugriff auf interne und externe Sensoren bereitzustellen. Einige Anwendungen, die bereits mit diesem Framework realisiert wurden, wurden kurz in Abschnitt 1 angesprochen. Das Framework ist in mehrere Android-Apps unterteilt. So gibt es eine Framework-App, jeweils eine App für jeden Sensortreiber und eine App für jede Anwendung, die das Sensor-Framework verwendet. Letztere enthält also die für die jeweilige Anwendung benötigte Logik und eine entsprechende Benutzeroberfläche und verwendet die Framework-App um auf externe Sensoren zuzugreifen. Für die Kommunikation zwischen den verschiedenen Applikationen werden diverse androidspezifische IPC-Mechanismen (*Interprocess Communication*) verwendet.

Die Auslagerung der Treiber in separate Apps wird damit begründet, dass hierdurch Treiberentwickler, bzw. idealerweise die Hersteller von Sensoren, Treiber über einen *App Store* wie *Google Play* verteilen können. Der modulare Aufbau soll so ermöglichen, dass Endanwender neue Treiber installieren können, ohne die Anwendung neu kompilieren zu müssen. Diese Anforderung steht für das in dieser Arbeit entwickelte Sensor-Framework nicht im Fokus. Das Nutzungsszenario des hier präsentierten Sensor-Frameworks sieht vor, dass das Framework in eine Anwendung, die Sensorunterstützung benötigt, als Bibliothek eingebunden und gemeinsam mit dieser Anwendung und den Treiberklassen kompiliert wird. Die Sensortreiber als separate Apps bereitzustellen bietet für den Einsatzbereich des in dieser Arbeit entwickelten Sensor-Frameworks nicht genügend Vorteile, um die dadurch verursachte zusätzliche Komplexität und Einschränkungen in Bezug auf die Schnittstelle zwischen Framework und Treibern zu rechtfertigen. Dass sich durch diese Art der Modularisierung auch einige auf den ersten Blick nicht offensichtliche Probleme ergeben, wird im Folgenden an einem Beispiel gezeigt.

Im Open Data Kit Sensors Framework sind Treiber zustandslos. Wie jedoch anhand des in Abschnitt 2.1.2 dargestellten Kommunikationsprotokolls klar wird, ist für die Verarbeitung von Nachrichten im Sensortreiber durchaus ein Zustand zu beachten. Dies wird im ODK-Framework so gelöst, dass in `getSensorData(...)`, einer Methode der Sensortreiber-Schnittstelle, vom Framework gespeicherte Zustandsinformationen des Sensors als Parameter übergeben werden müssen. Der durch den Aufruf resultierende neue Zustand des Sensors wird als Rückgabewert an das Framework zurückgesendet. Dies erscheint umständlicher, als das Speichern des Zustands eines Sensors im Treiber zu erlauben. Es kann vermutet werden, dass die Zustandslosigkeit in Sensortreibern weniger ein Feature, sondern eine Folge der Auslagerung der Treiber in eigene Apps darstellt. Unter Android ist es nicht ohne weiteres möglich, mehrere Instanzen der selben App zu öffnen. Da im ODK-Framework mehrere gleiche Sensoren den selben Treiber verwenden können, kommunizieren diese daher mit der selben Instanz der Treiber-App. Da man einem Treiberentwickler nicht die gleichzeitige Zustandsverwaltung mehrerer Sensoren zumuten will, musste dieses Zustands-Multiplexverfahren genutzt werden, in dem das Sensor-Framework je nach dem in einem Aufruf angesprochenen Sensor den zugehörigen Zustand an die Treiber-App weitergibt. In dem im Rahmen dieser Arbeit entwickelten Sensor-Framework kann für jeden Sensor eine eigene Instanz in Form eines Objekts erzeugt werden, sodass diese Probleme nicht auftreten, da jede Instanz den Zustand eines Sensors speichern kann.

Eine weitere verwandte Arbeit stellt ein Sensor-Framework dar, welches im Rahmen einer Bachelorarbeit an der Universität Ulm entstand [Nie13]. Tabelle 2.1 stellt die wichtigsten konzeptionellen Unterschiede dieses Frameworks zu dem in dieser Arbeit entwickelten Framework dar. Zusammenfassend lässt sich sagen, dass die Aufgabenstellung enger gefasst war und das Ergebnis als eine Art Prototyp für diese Arbeit diente. Auf Basis dieser Vorarbeit wurden jedoch weitere, den Problembereich erweiternde Überlegungen angestellt, die zusammen mit dem Prototyp-Framework die Grundlage für diese Arbeit darstellen [Sch+13].

Sensor-Framework aus [Nie13]	In dieser Arbeit entwickeltes Framework
Es werden nur Bluetooth-Sensoren betrachtet.	Für jede Verbindungsart, z.B. Bluetooth, gibt es einen Sensor-Manager. Eine darüberliegende Architekturebene bietet eine einheitliche Schnittstelle zum Zugriff auf interne und externe Sensoren jedweder Art an.
Berücksichtigt nur die Interaktion, die in dieser Arbeit als *Event Pattern* bezeichnet wird.	Es werden verschiedene *Interaction Patterns* unterstützt.
Die benötigte Sensorklasse muss manuell instanziiert werden.	Ein Typensystem unterstützt den Zugriff auf Sensoren nach bestimmten Klassifikationen.
Es werden explizit Methoden wie `connect()` und `disconnect()` aufgerufen.	Die Schnittstelle zu den Sensoren abstrahiert von Vorgängen wie beispielsweise einem Verbindungsaufbau.
Es gibt keine Treiber im engeren Sinn. Ein „Treiber“ wird mit genau einem konkreten Sensor gleichgesetzt. Beispielsweise wird in einer Treiberklasse eine feste Bluetooth-Geräteadresse verwendet.	Ein Treiber unterstützt mehrere gleichartige Sensoren, beispielsweise Sensoren eines bestimmten Modells.
Das Framework bietet gewisse Erweiterungsmöglichkeiten, die es einem Anwendungsentwickler erleichtern mehrere verschiedene Bluetooth-Sensoren zu unterstützen.	Zusätzliche Treiber können hinzugefügt werden ohne Änderungen am Code des Sensor-Frameworks vorzunehmen. Durch zusätzlich vorhandene Treiber werden zusätzliche Sensoren unterstützt.
In einer das Framework nutzenden Anwendung wird explizit eine bestimmte Treiberklasse verwendet. Es kann keine Anwendung geschrieben werden, die beispielsweise einen beliebigen Pulssensor benutzt.	Es exitiert eine strikte Trennung von Sensortreibern und das Framework nutzenden Anwendungen. Werden weitere Treiber hinzugefügt, kann eine Anwendung weitere Sensoren verwenden, ohne dass die Anwendung verändert werden muss.

Tabelle 2.1: Vergleich des Sensor-Frameworks, das in einer verwandten Arbeit enwtickelt wurde, zum in dieser Arbeit entwickelten Framework.

3

Sensor-Framework

Dieses Kapitel zeigt die Konzeption und Entwicklung des Sensor-Frameworks. Hierzu werden zunächst die funktionalen und nicht-funktionalen Anforderungen diskutiert. Anschließend wird die Architektur des Frameworks dargestellt und dabei auf verschiedene Konzepte eingegangen, die zur Erfüllung der Anforderungen entworfen wurden. Zuletzt werden einige Implementierungsdetails gezeigt, um die konkrete Funktionsweise des Sensor-Frameworks zu veranschaulichen.

3.1 Anforderungen

Im Folgenden werden die Anforderungen an das zu erstellende Sensor-Framework diskutiert. Diese ergaben sich durch die Betrachtung und Recherche verschiedener Sensoren, sowie die unterschiedlichen Anwendungsszenarien, in denen ein solches

Sensor-Framework eingesetzt werden kann. Einige Anforderungen kamen dabei hinzu, als das Sensor-Framework in einer anderen studentischen Arbeit eingesetzt werden sollte.

3.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die konkrete Funktionalität, die das Sensor-Framework bereitstellen soll.

A1 Plug and Play: Eine der Kernanforderungen an das Sensor-Framework, welches es zu entwickeln galt, war das *Plug-and-Play*-Konzept. Im Kontext dieser Arbeit wird darunter die Möglichkeit verstanden, neue Treiber für Sensoren möglichst einfach hinzufügen zu können. Dies soll möglich sein, ohne dass Änderungen am Code des Sensor-Frameworks nötig werden. Außerdem sollte möglichst kein Spezialwissen über das eigentliche Sensor-Framework nötig sein, um einen neuen Treiber zu schreiben und damit einen neuen Sensor einzubinden. In einem Sensortreiber soll nur die jeweils sensorspezifische Logik neu geschrieben werden müssen, während gemeinsam benötigte Funktionalität bereits vom Framework bereitgestellt wird.

Diese spezifische Funktionalität eines Sensors kann etwa aus dem Senden von speziellen Bytefolgen bzw. Paketen bestehen, die durch das jeweilige Kommunikationsprotokoll des Sensors bestimmt werden. Hier kann die Aufgabe des Sensor-Frameworks unter anderem darin bestehen, einen einfachen Zugriff auf den jeweiligen Kommunikationskanal bereitzustellen.

A2 Unterstützung verschiedener Interaktionsabläufe: Das Sensor-Framework muss verschiedene Arten von Sensoren unterstützen. An dieser Stelle wird nach dem Ablauf der Interaktion mit dem jeweiligen Sensor unterschieden. So gibt es beispielsweise Sensoren, die nach einem Startbefehl kontinuierlich Messungen durchführen und die Ergebnisse jeder Messung sofort übertragen, bis sie wieder gestoppt werden. Dies trifft zum Beispiel auf verschiedene Sensoren zu, die Vitalparameter (Puls, Sauerstoffsättigung) eines Patienten messen. Es gibt aber auch Sensoren, die nach expliziter Aufforderung nur genau eine Messung durchführen. Ein

Beispiel hierfür wäre ein Kamerasensor, der nach einer Anfrage ein Bild macht. Darüber hinaus gibt es Sensoren, die von einem Start- bis zu einem Stopbefehl eine Messung durchführen und erst nach dem Stopbefehl die gemessenen Daten zurückliefern, beispielsweise ein Sensor, der Audio-Daten aufzeichnet.

Der Zugriff auf diese verschiedenen Arten von Sensoren muss mit dem Sensor-Framework möglich sein. Es muss also überlegt werden, wie die Schnittstelle zu den Sensoren und nach außen zur Anwendung gestaltet werden muss, damit diese verschiedenen Abläufe unterstützt werden können. Dabei ist zu beachten, dass es noch weitere Arten von Interaktionsabläufen mit Sensoren geben kann, die zum Zeitpunkt der Konzeptionierung und Implementierung noch nicht bekannt waren und eventuell erst beim Hinzufügen neuer Sensoren auffallen.

A3 Klassifizierung der Sensoren: Es soll möglich sein auf die Sensoren über bestimmte Klassifizierungen zuzugreifen. So soll eine Anwendung, die das Sensor-Framework nutzt, beispielsweise eine Anfrage nach einem Pulsoximeter stellen können. Ebenso könnte das Sensor-Framework nach einem Sensor gefragt werden, der den Puls messen kann. Zu beachten ist hier, dass es sich dabei um unterschiedliche Arten von Anfragen handelt. Die erste bezieht sich auf eine Klassifizierung des Sensors, die zweite auf den Typ der gemessenen Daten.
Eine weitere mögliche Klassifizierung, die von einer das Framework nutzenden Anwendung abfragbar sein sollte, ist die Art der Verbindung des Sensors. Das Sensor-Framework muss darüber Auskunft geben können, ob es sich zum Beispiel um einen Bluetooth- oder einen USB-Sensor handelt. Denn abhängig von dieser Information könnte ein Sensor bevorzugt verwendet oder in einer Oberfläche entsprechend angezeigt werden.

A4 Serialisierung: Im Zusammenhang mit der KINDEX-Anwendung ergab sich eine weitere Anforderung. Da hier die Sensordaten für die Speicherung und den Export in *XML (Extensible Markup Language)* vorliegen müssen, ist es notwendig, dass das Sensor-Framework die jeweiligen Sensordaten in ein serialisiertes Format wie XML oder JSON konvertieren kann. Alternativ hierzu kann eine Serialisierung, sofern nötig, auch in der jeweiligen Anwendung, die die Sensordaten entgegennimmt,

erfolgen. Es ist trotzdem sinnvoll diese Funktion über das Sensor-Framework anzubieten. Wenn jeder Sensor eine Funktion anbietet, die die jeweils gemessenen Daten serialisiert, kann auf diese in verschiedenen Anwendungen zurückgegriffen werden. Darüber hinaus kann eine Standardserialisierung durch das Framework angeboten werden, auf die zurückgegriffen wird, wenn der Sensor selbst keine spezifischere Serialisierung der eigenen Daten anbietet.

A5 Abstraktion der konkreten Schnittstelle: Um einen Sensor anzusprechen ist einiges an spezifischem Code nötig. Es kann nötig sein eine Verbindung über den jeweiligen Kommunikationskanal auf- und abzubauen oder diese zu verwalten. Darüber hinaus gibt es oft ein sensorspezifisches Kommunikationsprotokoll, das die jeweiligen Paketformate bzw. Bytefolgen definiert, die notwendig sind, um mit dem jeweiligen Sensor zu kommunizieren. Einer Anwendung, die auf dem Sensor-Framework aufbaut, soll eine Schnittstelle bereitgestellt werden, in der von dieser Verbindungsverwaltung und der konkreten Schnittstelle zum Sensor abstrahiert wird. Das hat zur Folge, dass beispielsweise ein Bluetooth-Sensor auf die selbe Art angesprochen wird wie ein USB-Sensor oder ein interner Sensor, der direkt im Gerät verbaut ist.

A6 Unterschiedliche Kommandos für Sensoren: Manche Sensoren unterstützen unterschiedliche Kommandos oder Betriebsmodi. Dies zeigte sich beispielsweise bei Betrachtung des in dieser Arbeit verwendeten Pulsoximeters und bei einem Blutzuckersensor, der im Rahmen einer weiteren studentischen Arbeit mit dem Sensor-Framework verwendet werden sollte. Das Sensor-Framework sollte daher unterschiedliche Kommandos an einen Sensor weiterleiten können, um die Nutzung verschiedener Betriebsmodi möglich zu machen.

3.1.2 Nicht-funktionale Anforderungen

Zusätzlich zu den funktionalen Eigenschaften sind einige Anforderungen zu beachten, die sich auf die Qualität des Frameworks beziehen.

A7 Zuverlässigkeit: Gerade in der Interaktion mit verschiedenen externen Geräten kann es zu Fehlern kommen. Daher sollte eine gewisse Fehlertoleranz bzw. Fehlererholung vorhanden sein. So sollte es etwa zu keinem Systemabsturz kommen, wenn die Verbindung zu einem Sensor abbricht. Als optimale Zielvorstellung gilt es einen Mechanismus bereitzustellen, der in diesem Fall beispielsweise automatisch versucht, die Verbindung wiederherzustellen.

A8 Benutzbarkeit: Das Sensor-Framework soll die Entwicklung von mobilen, sensorunterstützten Anwendungen in zukünftigen Projekten erleichtern. Damit das Framework einfach in diese Anwendungen integriert werden kann, sollte die Schnittstelle, die das Sensor-Framework anbietet, leicht verständlich sein. Jedoch sollte im Rahmen der Benutzbarkeit bzw. Verständlichkeit auch darauf geachtet werden, dass leicht zu verstehen ist, wie das Hinzufügen eines neuen Sensortreibers funktioniert. Dies ist erforderlich, damit das als funktionale Anforderung diskutierte *Plug-and-Play*-Prinzip möglich wird.

A9 Erweiterbarkeit: Da das Sensor-Framework konzeptionell fähig sein soll mit vielen unterschiedlichen Sensoren zu interagieren, muss an verschiedenen Punkten auf eine einfache Erweiterbarkeit geachtet werden. Das Hinzufügen weiterer Sensoren wurde bereits diskutiert. Außerdem wurde bereits erwähnt, dass es weitere Interaktionsabläufe mit Sensoren geben kann, die hinzugefügt werden müssen. Ein weiterer Punkt für mögliche Erweiterungen sind weitere Sensor-Manager. Diese sind jeweils für eine Verbindungsart zuständig und verwalten die entsprechenden Sensoren. Die Sensor-Manager werden im späteren Verlauf der Arbeit genauer vorgestellt.

3.2 Konzeption und Architektur

Die zuvor behandelten Anforderungen werfen einige Probleme auf, für die Lösungskonzepte gefunden werden müssen. Daher wird im Folgenden die Konzeption bzw. Architektur des Sensor-Frameworks vorgestellt und dabei gezeigt, welche Überlegungen und Entscheidungen jeweils zu dem gewählten Ansatz führen.

3.2.1 Architekturüberblick

Abbildung 3.1 zeigt einen groben Überblick über die Architektur des Sensor-Frameworks. Dabei ist die Aufteilung in drei Ebenen zu sehen.

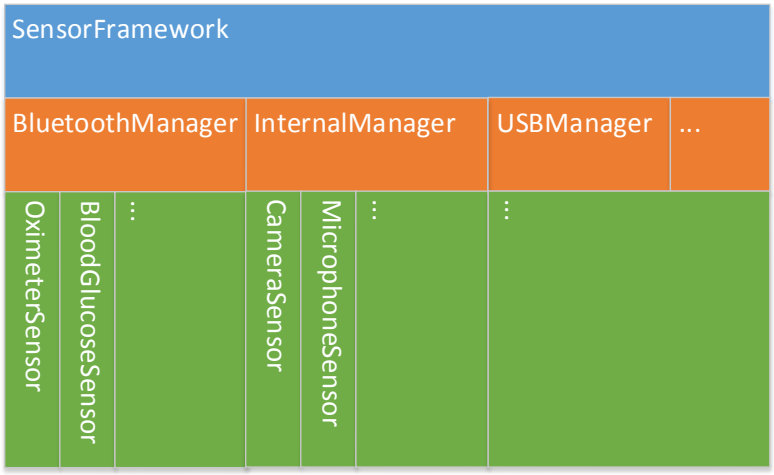


Abbildung 3.1: Architektur des Sensor-Frameworks in drei Ebenen

`SensorFramework` bildet den nach außen hin sichtbaren Teil bzw. die Schnittstelle, die anderen Anwendungen angeboten wird. Diese Ebene übernimmt folgende Aufgaben:

Zugriff auf Sensoren: Es werden Funktionen zum Zugriff auf Sensoren entsprechend ihrer Klassifizierung bereitgestellt. Hier kann nach Typ des Sensors, sowie nach gemessenem Datentyp gefiltert werden.

Anfragen an Sensoren: Anfragen an einen bestimmten Sensor werden an den zugehörigen Sensor-Manager weitergeleitet.

Weiterleiten der Daten: Gemessene Daten werden vom jeweiligen Sensor-Manager in Empfang genommen und an den entsprechenden Empfänger nach außen weitergegeben.

Die *Sensor-Manager* verwalten jeweils eine Art von Sensoren. Die Aufteilung erfolgt nach Verbindungstyp, da hierbei jeweils unterschiedliche Logik notwendig ist um die Verbindungen aufzubauen oder diese zu verwalten. Die Aufgaben eines Sensor-Managers sind:

Verwalten der Verbindungen: Die Verbindung zum Sensor wird verwaltet d.h. bei Bedarf wird der Auf- und Abbau angestoßen.

Verwaltung der vorhandenen Sensoren: Es erfolgt die Zuordnung von vorhandenen Geräten zu Sensortreibern und damit Sensoren, die einer das Framework nutzenden Anwendung zur Verfügung gestellt werden.

Delegation der Anfragen: Anfragen von `SensorFramework` werden zu den Sensoren weitergeleitet, Ergebnisse werden auf umgekehrtem Weg zurückgeliefert.

Die unterste Ebene bilden die *Sensoren* bzw. deren Sensortreiber. Ein Treiber ist dabei immer einem *Sensor-Manager* zugeordnet. Pro Sensortreiber kann es theoretisch auch mehrere gleiche Sensoren geben, die von diesem Treiber unterstützt werden. Es folgen die Aufgaben eines Sensortreibers:

Kommunikation mit dem Sensor: Der tatsächliche Verbindungsaufbau des Kommunikationskanals zum Sensor erfolgt auf dieser Ebene. Das bedeutet jedoch nicht, dass diese Logik für jeden Sensor neu geschrieben werden muss. Bestimmte gemeinsam benötigte Funktionalität wird vom Sensor-Framework bereitgestellt.

Sensorspezifisches Kommunikationsprotokolls: Das Kommunikationsprotokoll mit dem Sensor muss hier implementiert werden. Dieses läuft als höhere Protokollebene über den Kommunikationskanal zum Sensor und kann hier einen weiteren Verbindungsaufbau erfordern. Darauf folgen meist bestimmte Kommandos an den Sensor und eventuell Bestätigungsnachrichten.

Spezifische Serialisierung: Reicht die Standardserialisierung des Sensor-Frameworks nicht aus, kann hier die Serialisierung der Sensordaten gesteuert werden.

Bereitstellen von Sensor-Informationen: Der Sensortreiber stellt bestimmte Informationen über den Sensor bereit. Darunter z.B. einen Namen des Sensors, den Typ und den Typ der gemessenen Daten. Das hierfür verwendete Typensystem wird in 3.2.3 vorgestellt.

3.2.2 Interaction Patterns

In Abschnitt 3.1.1 wurde bereits erwähnt, dass es unterschiedliche Arten von Sensoren gibt, die sich nach dem Ablauf der Interaktion mit dem jeweiligen Sensor unterscheiden. Im Folgenden wird dargestellt, wie die Anforderung diese unterschiedlichen Arten von Sensoren zu unterstützen in der Konzeption des Sensor-Frameworks umgesetzt wurde und welche Überlegungen dafür entscheidend waren.

Im Umfeld von Geschäftsprozessen gibt es den Begriff der *Service Interaction Patterns* [BDH05; Bee+08]. Hierbei geht es um die Dienstkomposition im Umfeld von Web-Services. Es besteht daher kein direkter Zusammenhang mit dieser Arbeit, es kann sich jedoch an grundlegenden Konzepten und Begriffen aus diesem Bereich orientiert werden.

Beim der Entwicklung des Prinzips der *Interaction Patterns* wurden außerdem die Konzepte aus der Android-API zum Zugriff auf Sensoren [And13b] in Betracht gezogen. Hier wird zwischen verschiedenen Arten von Sensoren unterschieden, jedoch zeigt sich diese Unterscheidung nicht besonders klar, wenn man die API betrachtet. Es gibt den Begriff *Streaming Sensor*, der Sensoren bezeichnet, die Daten nach bestimmten regelmäßigen Zeitintervallen liefern. Demgegenüber gibt es auch Sensoren, die nur bei Änderung des Messwerts Daten liefern. Für diese gibt es keine bestimmte Bezeichnung. Außerdem gibt es noch *Trigger Sensors*. Diese warten nach ihrem Start auf ein bestimmtes Trigger-Ereignis. Tritt dieses auf, wird der entsprechende Aufrufer hiervon in Kenntnis gesetzt, woraufhin sich der Sensor selbstständig wieder deaktiviert. Der Sensor vom Typ `TYPE_SIGNIFICANT_MOTION` ist der einzige derzeit definierte Trigger-Sensor.

Der Zugriff bzw. die Interaktion mit internen Sensoren funktioniert in der Android-API so, dass zunächst eine Liste von verfügbaren Sensoren erfragt wird. Diese ist nach dem verlangten Sensortyp gefiltert. Danach können durch den Aufruf von `registerListener()` und `unregisterListener()` Empfänger für die Sensordaten registriert bzw. entfernt werden, wodurch der Sensor im Hintergrund unter Umständen aktiviert oder deaktiviert wird. Handelt es sich um einen Trigger-Sensor, muss eine andere Schnittstelle verwendet werden. Mittels `requestTriggerSensor()` wird der Sensor zum warten auf das Trigger-Ereignis veranlasst. Mithilfe von `cancelTriggerSensor` kann dies vor-

zeitig abgebrochen werden. Wenn die Methoden der Schnittstelle für Trigger-Sensoren für andere Sensoren verwendet werden, führt dies zu einem Fehler (*Exception*). Die Android-API bietet jedoch keine Möglichkeit an, dies vor dem Aufruf zu erkennen. Dafür muss auf die Dokumentation zurückgegriffen werden, in der entsprechende Sensortypen als Trigger-Sensor gekennzeichnet sind. Für das zu erstellende Sensor-Framework wurde sich zum Ziel gesetzt solche Unterscheidungen auch über die API zugänglich zu machen und damit expliziter darzustellen.

Ausgehend von den Ideen aus dem Geschäftsprozess-Umfeld, der Sensor-API in Android und der Betrachtung verschiedener interner und externer Sensoren wurde deutlich, dass sich das im Folgenden dargestellte Konzept von Interaction Patterns am besten für das Sensor-Framework eignet. Ein Sensor unterstützt dabei jeweils genau ein Interaction Pattern. Prinzipiell wäre es jedoch auch möglich, dass ein Sensor mehrere Interaction Patterns unterstützt. Dies war jedoch bei den hier betrachteten Sensoren nicht notwendig und wird daher nicht weiter diskutiert.

Event Pattern

Das erste hier definierte Interaction Pattern wird als *Event Pattern* bezeichnet. Dies entspricht einem Listener-Konzept, welches ein weit verbreitetes Entwurfsmuster darstellt. Die Anwendung registriert dabei beim Sensor-Framework einen Listener für einen bestimmten Sensor. Falls nötig wird dadurch eine Verbindung zum Sensor aufgebaut. Sensordaten werden über den Listener zurückgeliefert, bis der Listener wieder abgemeldet wird. Das im Rahmen dieser Arbeit verwendete Pulsoximeter ist ein Beispiel für einen Sensor, für den sich dieses Muster eignet. Aufgrund der auch in anderen Arbeiten betrachteten Sensoren [Nie13; Sch+13] ist davon auszugehen, dass dies ein typisches Interaktionsmuster für Sensoren aus dem medizinischen Bereich darstellt.

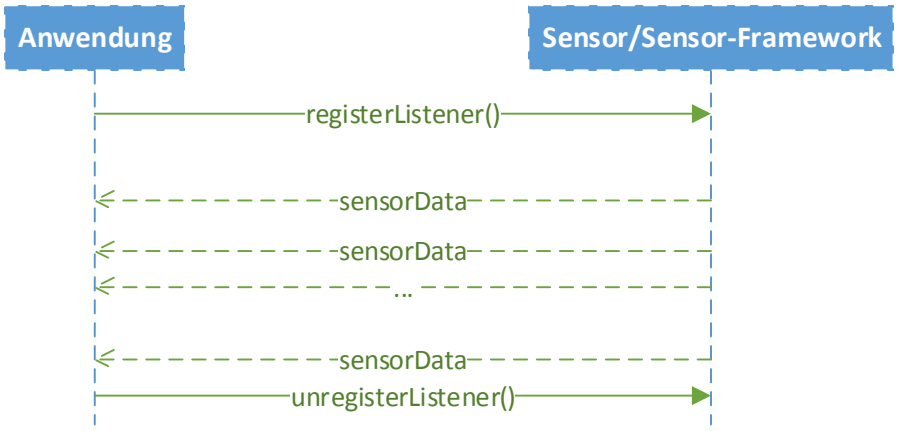


Abbildung 3.2: Das Event-Pattern entspricht einem Listener-Konzept.

Synchronous Request Pattern

Die simpelste Variante stellt das *Synchronous Interaction Pattern* dar. Auf Anfrage werden die Sensordaten zurückgeliefert, oder es wird ein Fehler signalisiert.

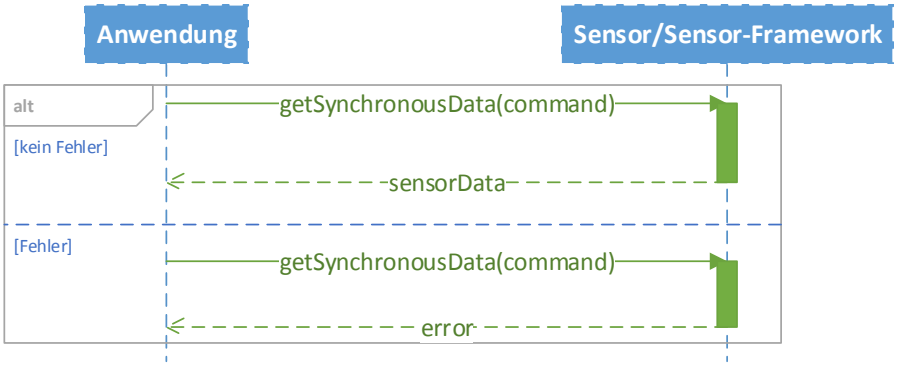


Abbildung 3.3: Synchronous Request Pattern

Asynchronous Request Pattern

Ein weiteres Interaction Pattern wird hier als *Asynchronous Request Pattern* bezeichnet. Der Aufrufer wird durch die Anfrage nach Sensordaten nicht blockiert. Es wird ein

Callback übergeben, über das die Daten zurückgegeben werden können sobald diese vorliegen. In Abschnitt 3.4 ist zusätzlich die Signalisierung von Fehlern angedeutet, sowie die Möglichkeit verschiedene Kommandos an den Sensor zu senden. Es ist vorgesehen, dass genau eine Antwortnachricht erwartet wird. Dies reicht für die bisher betrachteten Sensoren aus. Falls für einen gewissen Sensor mehr als eine Antwort nötig werden sollte, müsste überlegt werden dieses Muster zu ändern oder ein neues hinzuzufügen. Das Asynchronous Request Pattern wird zum Beispiel bei der Anbindung des Kamerasensors verwendet.

Der Grund für die Existenz dieses asynchronen Musters neben einer synchronen Variante ist, dass verschiedene Faktoren bei der Abfrage von Daten eines Sensors zu einem Blockieren einer synchronen Abfrage führen würden. Dies wird beispielsweise durch den Aufbau einer Bluetooth-Verbindung und Empfang von Daten über einen Socket verursacht. Ein weiterer Grund sind längere Verarbeitungszeiten eines Sensors, wie sie beispielsweise für den Kamerasensor nötig sind, bis die Bilddaten tatsächlich zur Verfügung stehen.

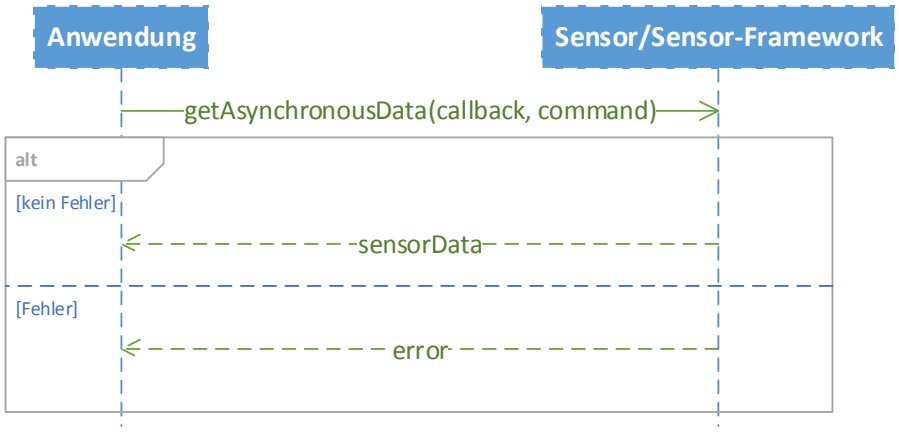


Abbildung 3.4: Asynchronous Request Pattern

Recording Pattern

Bei der Analyse verschiedener Sensoren wurde zudem das *Recording Pattern* identifiziert. Hier wird mithilfe von `start()` und `stop()` eine Messung gestartet bzw. gestoppt. Die Daten liegen jedoch erst nach dem Ende der Messung vor, worauf eventuell noch eine Verarbeitungszeit folgen kann. Üblicherweise können nicht mehrere Messungen parallel durchgeführt werden und die Messung muss durch den selben Aufrufer gestartet und gestoppt werden, was über einen Vergleich des Callbacks sichergestellt wird. Über das Callback können auch zur Laufzeit auftretende Fehler signalisiert werden.

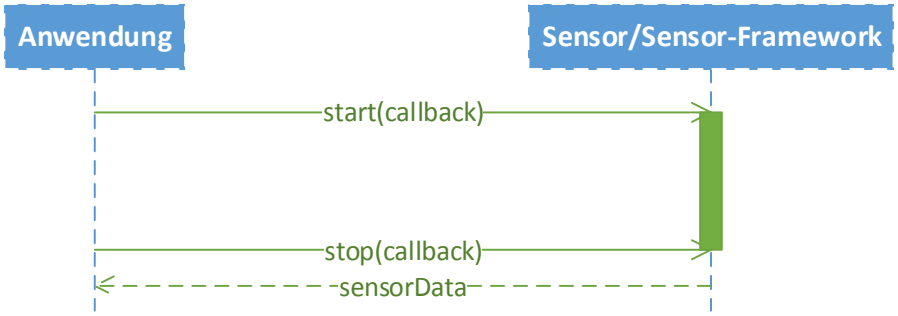


Abbildung 3.5: Recording Pattern

3.2.3 Typensystem

Als eine der Anforderungen[1] an das Sensor-Framework wurde genannt, dass es möglich sein soll auf die Sensoren über bestimmte Klassifizierungen zuzugreifen. Eine Anwendung könnte so zum Beispiel eine Anfrage nach verfügbaren Kamerasensoren stellen, nach Sensoren mit denen ein Bild gemacht werden kann oder auch feststellen, ob es sich bei einem Sensor um einen Bluetooth-Sensor handelt. Wäre solch eine Klassifizierung nicht vorhanden, müsste sich eine Anwendung beim Zugriff auf einen Sensor jeweils auf einen konkreten Sensor beziehen. Dies könnte im Fall eines Bluetooth-Sensors etwa über die Angabe einer Hardware-Adresse passieren. Dies wäre jedoch in Anbetracht der angestrebten Abstraktion der konkreten Schnittstelle zu den Sensoren zu vermeiden.

[1] siehe Anforderung **A3** in Abschnitt 3.1.1

Erst durch ein Typensystem wird auch die strikte Trennung von Sensortreibern und das Framework nutzenden Anwendungen im Rahmen der *Plug-and-Play*-Architektur ermöglicht.

Betrachtet man die Protokolle, die im Zusammenhang mit Bluetooth oder USB verwendet werden, stößt man auf dort verwendete Typensysteme. Auf Ebene des *Baseband* aus dem Bluetooth-Protokoll-Stack sind bei der Bluetooth Special Interest Group (SIG) bestimmte *Service Classes*, *Major Device Classes* und *Minor Device Classes* definiert [Blu13a]. Die *Service Class* gibt dabei eine Gerätefamilie, wie beispielsweise *Audio*, an. Bei den Major Device Classes gibt es 32 Klassen, die die Hauptfunktionalität des Geräts angeben. Hier gibt es eine Klasse *Health*, welche in Bezug auf die im Rahmen dieser Arbeit betrachteten Sensoren eventuell relevant wäre. Unter den *Minor Device Classes* von *Health* befinden sich Klassen wie *Pulse Oximeter*, *Glucose Meter* und *Heart/Pulse Rate Monitor*. Da dies genau die Arten von Sensoren aus dem medizinischen Umfeld sind, auf die sich hier hauptsächlich konzentriert wird, wäre zu überprüfen, ob diese Informationen in das Typensystem des Sensor-Framework mit einbezogen werden könnten. Tests mit dem hier verwendeten Pulsoximeter zeigten jedoch, dass dieses über die *Device Classes* keine verwertbaren Informationen preisgibt. Die Frage nach der Device Class liefert hier lediglich *Uncategorized device*. Daher wurde die Idee aufgegeben, diese Informationen in das Typensystem des Sensor-Frameworks einzubeziehen. Ähnlich verhält es sich bei den Bluetooth-Profilen. Unterstützt ein Gerät z.B. das *Health Device Profile* sind dafür bestimmte Datentypen definiert, darunter abermals Typen für ein Pulsoximeter und andere medizinische Sensoren. Ein solches spezielles Bluetooth-Profil wird aber von keinem der betrachteten Sensoren unterstützt.

Ein ähnliches System gibt es bei USB-Geräten mit den *Communication Device Classes (CDC)* und den Informationen im *Device-Deskriptor* und *Interface-Deskriptor* [USB11]. Da der Fokus in dieser Arbeit zunächst auf die drahtlose Kommunikation mit Sensoren gerichtet ist, werden USB-Sensoren im derzeitigen Implementierungsstand noch nicht unterstützt, d.h. der entsprechende Sensor-Manager ist nicht implementiert. Daher bleibt vorerst offen, inwieweit diese Informationen in einem Typensystem des Sensor-Frameworks verwendet werden könnten.

In der Android-API beschränkt sich die Klassifizierung der internen Gerätesensoren darauf, dass es einige fest definierte Konstanten für Sensoren gibt, z.B. `TYPE_GRAVITY`. Ein solches System genügt den Anforderungen an das Sensor-Framework nicht. Um wirklich ein *Plug-and-Play*-Konzept auf Ebene der Sensortreiber im Sinne der Anforderung A1 zu unterstützen wird ein System benötigt, bei dem neue Typen von Sensoren hinzugefügt werden können, die zum Zeitpunkt der Implementierung des Sensor-Frameworks noch nicht bedacht wurden. Dennoch können auch fest durch Konstanten bzw. Enums definierte Typen von Vorteil sein. Hier kann unter Umständen schon durch den Compiler sichergestellt werden, dass nur gültige Typen für zu implementierende oder zu verwendende Sensoren angegeben werden. Es ist auch sowohl für eine das Sensor-Framework nutzende Anwendung als auch für einen Sensortreiber weit weniger kompliziert mit solchen vordefinierten Typen zu arbeiten, als es mit dynamisch zu definierenden der Fall wäre. Ein weiterer Vorteil von vom Sensor-Framework vordefinierten Typen besteht darin, dass Typkollisionen bzw. Mehrfachdefinitionen vermieden werden. Denn die Alternative zu vom Sensor-Framework fest vordefinierten Typen stellt die dynamische Definition der Typen durch die jeweiligen Sensortreiber dar. Existieren nun jedoch zwei Sensoren wie z.B. ein Pulsgurt und ein Pulsoximeter, wird hier ein Datentyp wie *Puls* möglicherweise mehrfach definiert. Dadurch wäre es einer Anwendung, die das Sensor-Framework nutzt, nicht mehr so einfach möglich diese Daten oder die dazugehörigen Sensoren als gleichartig zu erkennen und entsprechend in einer Oberfläche anzuzeigen und die Daten ihrem Typ nach zu verarbeiten. Wie bei den Datentypen kann es auch zu einer Mehrfachdefinition bei den Sensortypen kommen. Durch solche Mehrfachdefinitionen würde der Nutzen eines Typensystems infrage gestellt werden. Denn dieses sollte darauf abzielen, dass einmal definierte Typen auch durch alle entsprechenden Sensoren verwendet werden, damit eine das Framework nutzende Anwendungen einen Nutzen aus dem Typensystem ziehen kann.

Aus diesen Überlegungen folgt die Idee, die Vorteile von vordefinierten Typen mit denen der dynamischen Typdefinitionen zu verbinden. Im Sensor-Framework gibt es daher bereits einige vordefinierte Typen. Die Absicht ist hiermit bereits möglichst viele Typen, bei denen es bereits absehbar ist, dass sie benötigt werden könnten, bereitzustellen. Somit könnte in den meisten Fällen die erhöhte Komplexität und das Problem der

Mehrfachdefinitionen bei dynamisch definierten Typen vermieden werden. Wenn die vordefinierten Typen bei der Anbindung eines neuen Sensors nicht ausreichen, ist es jedoch auch möglich neue zu definieren.

Wenn bisher von Typen gesprochen wurde bezog sich dies sowohl auf den Typ eines Sensors, als auch auf die Typen der von einem Sensor gemessenen Werte. Beide dieser Typkonzepte werden im Sensor-Framework dieser Arbeit berücksichtigt. Abbildung 3.6 zeigt dabei den Zusammenhang zwischen Sensortypen und Datentypen. Ein Sensortyp besitzt einen beschreibenden Namen und eine eindeutige Identifikationsnummer. Jedem Sensortyp sind Datentypen zugeordnet. Der Sensortyp `PULSE_OXIMETER` besitzt beispielsweise die Datentypen `SATURATION_OF_PERIPHERAL_OXYGEN` und `PULSE_RATE`. Jeder Datentyp besteht wiederum aus einem beschreibenden Namen und einer Identifikationsnummer. Außerdem besitzt ein Datentyp eine Klasse wie etwa `Integer`, der den Typ der enthaltenen Daten angibt. Sowohl Sensortyp als auch Datentyp können vordefinierte oder dynamisch definierte Typen sein. Es is möglich einen Sensortyp zu definieren, der gleichzeitig einige neue Datentypen definiert, sowie auch auf vordefinierte Datentypen zurückgreift. Details wie die Definition neuer Typen konkret umgesetzt wird werden in 3.3.6 dargestellt.

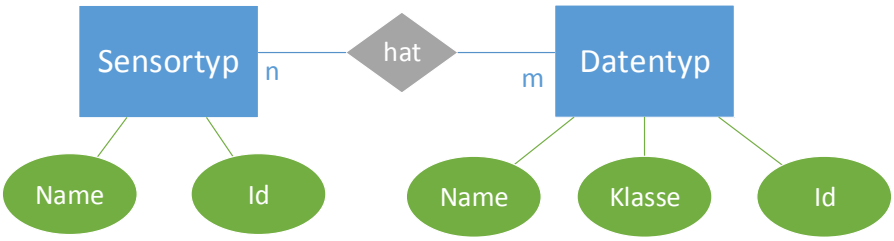


Abbildung 3.6: Sensortypen und Datentypen im Typensystem des Sensor-Frameworks.

3.3 Implementierung

Bisher wurde das Sensor-Framework auf einer konzeptionellen Ebene betrachtet, in der spezifische Implementierungsdetails weitestgehend nicht diskutiert wurden. Im Folgenden werden daher verschiedene Details zur Implementierung näher dargestellt.

Dabei soll gezeigt werden, wie die vorher im Rahmen der Konzeption und Architektur dargestellten Konzepte konkret in Code umgesetzt wurden.

3.3.1 Hinzufügen von Treibern zum Sensor-Framework

Wie bereits erwähnt ist es im Rahmen der *Plug-and-Play*-Architektur möglich Treiber zum Sensor-Framework hinzuzufügen. Beim Initialisieren des Sensor-Frameworks können daher Treiberklassen übergeben werden. Diese werden anhand ihres Typs an den entsprechenden `SensorManager` weitergereicht. Mit *Typ* ist an dieser Stelle der Java-Klassentyp gemeint, der durch den Einsatz von *Reflection* zur Laufzeit erkannt werden kann. Dies wird in Listing 3.1 gezeigt. Erbt ein Sensortreiber beispielsweise von der Klasse `InternalSensor`, so wird er an den `InternalSensorManager` übergeben.

Nicht alle Treiber müssen bei der Initialisierung übergeben werden. Die Treiber, die zum jetzigen Zeitpunkt bereits in das Sensor-Framework integriert sind werden automatisch hinzugefügt. Aufgrund der Anforderungen wurde jedoch ein Mechanismus benötigt, über den auch neue Treiber zum Sensor-Framework hinzugefügt werden können. Die anfängliche Idealvorstellung für diesen Mechanismus war, dass Treiber allein durch das Hinzufügen des Treibers zum entsprechenden *Package* automatisch zur Laufzeit erkannt werden können. So sollte ein Bluetooth-Sensor in einem Package `*.bluetooth.sensors` durch Reflection zur Laufzeit als Bluetooth-Sensor erkannt und zum Bluetooth-Manager hinzugefügt werden. Dieser Ansatz warf jedoch einige Probleme auf. Die Java-API bietet auch mithilfe von Reflection keine Methoden an um etwa alle Klassen eines Pakets aufzulisten. Arbeitet man mit einer richtigen *Java Virtual Machine* ist es möglich diese Beschränkung zum Beispiel durch den Zugriff auf das Dateisystem zu umgehen. In Android sind solche Hilfskonstruktionen aufgrund der dort genutzten *Dalvik Virtual Machine* jedoch nicht ohne weiteres möglich bzw. ist schwer sicherzustellen, dass eine solche Lösung plattform- und versionsunabhängig ist. Aufgrund der Besonderheiten der Android-Plattform sind auch externe Bibliotheken, die zur Lösung dieses Problems nützlich wären, nicht unbedingt nutzbar. Die Bibliothek *Reflections* [ron13] erweitert die Java-Reflection-API beispielsweise unter anderem um die Möglichkeit zur Laufzeit auf Klassen nach deren Paket zuzugreifen. Diese Bibliothek

ist jedoch nicht zu Android und der Dalvik Virtual Machine kompatibel. Unter diesen Voraussetzungen wurde die bereits angesprochene Lösung gewählt, zusätzliche Treiber bei der Initialisierung manuell an das Sensor-Framework zu übergeben. Somit ist es trotzdem möglich neue Treiber ohne Änderungen am Code des Sensor-Frameworks einzubinden. Nachdem die Treiberklassen einmal an das Framework übergeben wurden, greift eine Anwendung nicht mehr direkt auf einen Treiber zu, sodass hierdurch eine Trennung des Codes der Anwendung von den Treibern gegeben ist.

```
1  /**
2   * Returns a sublist of Class objects containing all the Class
       objects of the given list that are of a subtype of t
3   *
4   * This is a utility method to get the drivers that a specific
       sensor manager is responsible for, from of a list of sensor
5   * drivers.
6   */
7  private static <T extends Sensor> List<Class<? extends T>>
       getDriversOfType(List<Class<? extends Sensor>> drivers,
       Class<T> t) {
8    List<Class<? extends T>> driversOfType = new ArrayList<Class
       <? extends T>>();
9
10   for (Class<? extends Sensor> driver : drivers)
11     if (t.isAssignableFrom(driver))
12       driversOfType.add(driver.asSubclass(t));
13
14   return driversOfType;
15 }
```

Listing 3.1: Übergebene Treiber werden durch Reflection nach ihrem Sensor-Klassentyp aufgeteilt, um sie dem passenden Sensor-Manager zuzuordnen.

3.3.2 Verarbeitung einer Anfrage durch das Sensor-Framework

Um die Funktionsweise des Sensor-Frameworks näher zu beschreiben, wird der Ablauf einer konkreten Anfrage einer Anwendung an das Sensor-Framework hier dargestellt. Listing 3.2 zeigt dazu den konzeptionellen Ablauf der Interaktion einer Anwendung mit einem Sensor, der das Event-Pattern unterstützt.

```
1  // add additional sensor drivers that are to be used
2  List<Class<? extends Sensor>> sensorDrivers = new ArrayList<
       Class<? extends Sensor>>();
3  sensorDrivers.add(OximeterSensor.class);
4
5  // initialize sensor framework
6  SensorFramework.init(this, sensorDrivers);
7  SensorFramework framework = SensorFramework.getInstance();
8  List<Sensor> sensorList = framework.getSensorsByType(Sensor.
       Type.PULSE_OXIMETER);
9
10 SensorEventListener listener = new SensorEventListener() {
11   @Override
12   public void onDataReceived(SensorData sensorData) {
13     if (sensorData.getType() == SensorData.Type.PULSE_RATE) {
14       int pulseRate = (Integer) sensorData.getData();
15       // process received pulse rate
16     }
17     else if (sensorData.getType() == SensorData.Type.
           SATURATION_OF_PERIPHERAL_OXYGEN) {
18       int oxigenSaturation = (Integer) sensorData.getData();
19       // process received oxygen saturation
20     }
21   }
22
23   @Override
24   public void onError(Sensor sensor) {
25     // display error
26   }
27 };
28
29 // register listener to start receiving data
30 for (Sensor sensor : sensorList) {
31   framework.registerListener(listener, sensor);
32 }
33 // ...
34 // after some time...
35 // unregister listener, sensor stops
36 for (Sensor sensor : sensorList) {
37   framework.unregisterListener(listener, sensor);
38 }
```

Listing 3.2: Konzeptioneller Ablauf der Interaktion einer Anwendung mit einem Sensor, der das Event-Pattern unterstützt.

Wie bereits diskutiert werden zunächst eventuell zusätzlich benötigte Treiberklassen an das Framework übergeben. Diese müssen von einer der Sensor-Oberklassen (z.B. `BluetoothSensor`) erben. Außerdem muss eines der Interaction Patterns implementiert werden. Alternativ steht zum Beispiel für Bluetooth-Sensoren schon eine abstrakte

Klasse bereit, die dies bereits sicherstellt und darüber hinaus auch weitergehende Funktionalität für Bluetooth-Sensoren bereitstellt. Nach der Initialisierung des Sensor-Frameworks kann eine Referenz auf das `SensorFramework` im Stil des *Singleton Pattern* erfragt werden. Dies ist wegen dem *Activity Lifecycle* in Android nützlich, da so von mehreren *Activities* bzw. auch nach deren Neuerzeugung im Rahmen dieses Lebenszykluses auf das Framework zugegriffen werden kann. Als nächster Schritt werden die verfügbaren Sensoren zum Beispiel einem Typ nach abgefragt. In Listing 3.2 wird hierbei ein vordefinierter Typ verwendet. Anschließend wird über die durch das jeweilige Interaction Pattern des Sensors unterstützten Methoden mit dem Sensor interagiert. Im Beispiel funktioniert dies durch die Methoden `registerListener(...)` bzw. `unregisterListener(...)`. Es ist zu beachten, dass in dem dargestellten Beispiel als bekannt vorausgesetzt wird, dass ein Sensor vom Typ `PULSE_OXIMETER` die gezeigten beiden Integer-Datentypen liefert. Informationen zu einem Sensortyp und dessen Datentypen lassen sich jedoch auch zur Laufzeit über die entsprechenden Methoden in `Sensor.Type` und `SensorData.Type` gewinnen.

Im Folgenden wird dargestellt, wie die Anfrage aus Listing 3.2 auf den verschiedenen Ebenen des Sensor-Frameworks verarbeitet wird. Zuerst erreicht der Aufruf der Methode `registerListener(...)` die Ebene bzw. Klasse `SensorFramework`. Hier wird festgestellt, ob für diesen Sensor bereits ein Listener registriert ist. Ist dies der Fall, ist der betroffene Sensor bereits aktiv und sendet Daten, die nur noch zusätzlich an den neu registrierten Listener weitergeleitet werden müssen. Daher kann in diesem Fall die Anfrage direkt von `SensorFramework` durch bloßes Speichern des neuen Listeners verarbeitet werden. Der entsprechende Sensor-Manager und der Sensor müssen in diesem Fall nicht informiert werden. Gibt es noch keinen Listener bzw. Interessenten für die Daten des entsprechenden Sensors, wird die Anfrage an den für den Sensor verantwortlichen Sensor-Manager weitergereicht. Auf Ebene der Sensor-Manager entfällt auf diese Weise die Verwaltung mehrerer Listener, da alle Sensordaten lediglich an `SensorFramework` weitergeleitet werden, von wo aus sie dann an alle registrierten Listener gehen.

In dem dargestellten Beispiel erreicht die Anfrage auf nächster Ebene die Klasse `BluetoothSensorManager`. Diese interagiert mit dem Sensor über die Schnittstelle

`BluetoothEventPattern` mithilfe der Methoden `connect()`, `setListener(...)` und `disconnect()` und verwaltet hiermit die Verbindung zum Sensor bzw. baut sie in diesem Fall auf. Der Sensor startet daraufhin seine Messungen und liefert die gesammelten Daten in Form von *Events* an den Sensor-Manager zurück. Von hier fließen die Daten zunächst wieder an `SensorFramework`, von wo aus sie an die registrierten Listener, also zur das Sensor-Framework nutzenden Anwendung, weitergereicht werden. Abbildung 3.7 veranschaulicht den soeben beschriebenen Ablauf. Zu beachten ist dabei, dass die gemessenen Daten nicht direkt vom Sensor an die Anwendung zurückgegeben werden. Sie durchlaufen alle Ebenen des Sensor-Frameworks von unten nach oben, sodass eine weitere Verarbeitung der Daten an zentraler Stelle, zum Beispiel zur Pufferung oder Archivierung, innerhalb des Sensor-Frameworks möglich wäre.

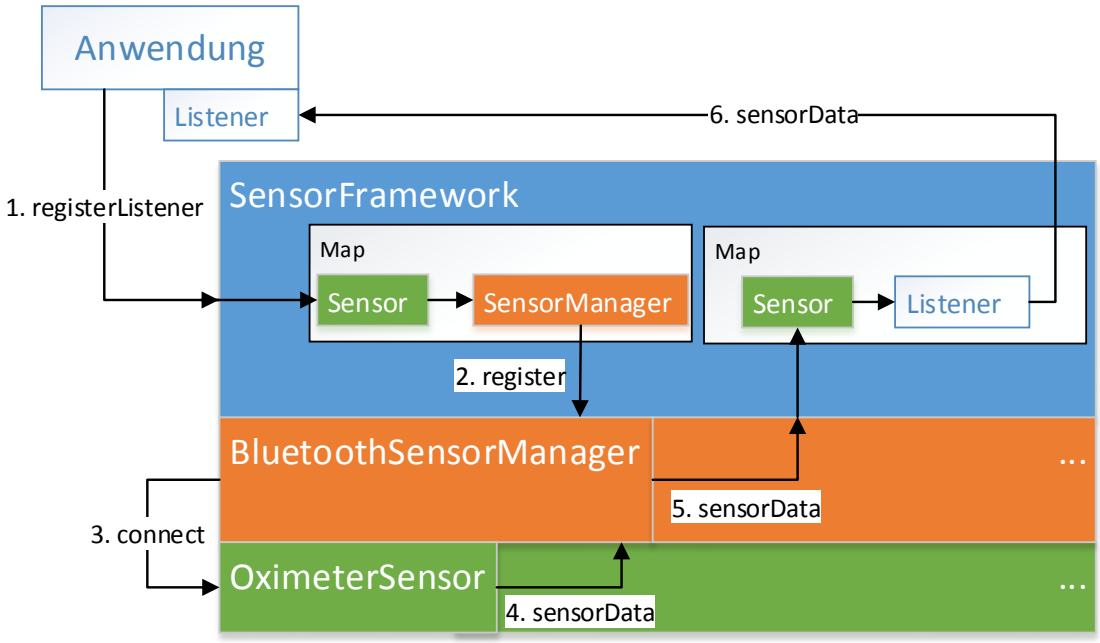


Abbildung 3.7: Anfrageverarbeitung im Sensor-Framework am Beispiel des Event Patterns. Die Anfrage und die Sensordaten durchlaufen die drei Ebenen.

3.3.3 Abstrakte Sensorklassen

Bestimmte Funktionalitäten werden nicht nur für genau einen Treiber benötigt. Beispielsweise kommunizieren viele Bluetooth-Sensoren in irgendeiner Form über RFCOMM Bluetooth-Sockets. Tatsächlich war dies sogar bei allen Bluetooth-Sensoren, die während dieser Arbeit betrachtet wurden, der Fall. Hierzu muss eine Verbindung aufgebaut und verwaltet werden. Im Sinne der Anforderung A1 sollte das Implementieren neuer Sensortreiber möglichst einfach sein. Das Sensor-Framework sollte also schon möglichst viele solcher Funktionalitäten, die in mehreren Sensortreibern benötigt werden, bereitstellen. Dadurch soll erreicht werden, dass in einem Treiber nur noch die wirklich sensorspezifischen Teile neu implementiert werden müssen. Dieses Prinzip wird im Folgenden anhand der Klasse `AbstractBluetoothSensor` näher erläutert.

Abbildung 3.8 zeigt ein vereinfachtes Klassendiagramm der Klassen in Zusammenhang mit `AbstractBluetoothSensor` und deren Vererbungshierarchie. Dabei ist zu sehen, dass konkrete Sensortreiber nicht direkt von `AbstractBluetoothSensor` erben. Es existiert noch eine weitere Ebene in Form einer abstrakten Klasse, die die Methoden des jeweiligen *Interaction Patterns* auf die von `AbstractBluetoothSensor` bereitgestellten Methoden abbildet. Gezeigt ist dies hier am Beispiel des *Event Patterns*. Analog dazu gibt es eine ähnliche abstrakte Klasse auch für das *Asynchronous Request Pattern*.

In Tabelle 3.1 wird Funktionalität, die den Treibern von Bluetooth-Sensoren bereitgestellt wird, anhand der geerbten Methoden näher dargestellt. Abstrakte Methoden müssen von der jeweiligen Treiberklasse überschrieben werden. Diese stellen den Anknüpfungspunkt für die sensorspezifischen Codeteile dar. Andere als *protected final* markierte Methoden stehen für die Nutzung in den Treiberklassen bereit.

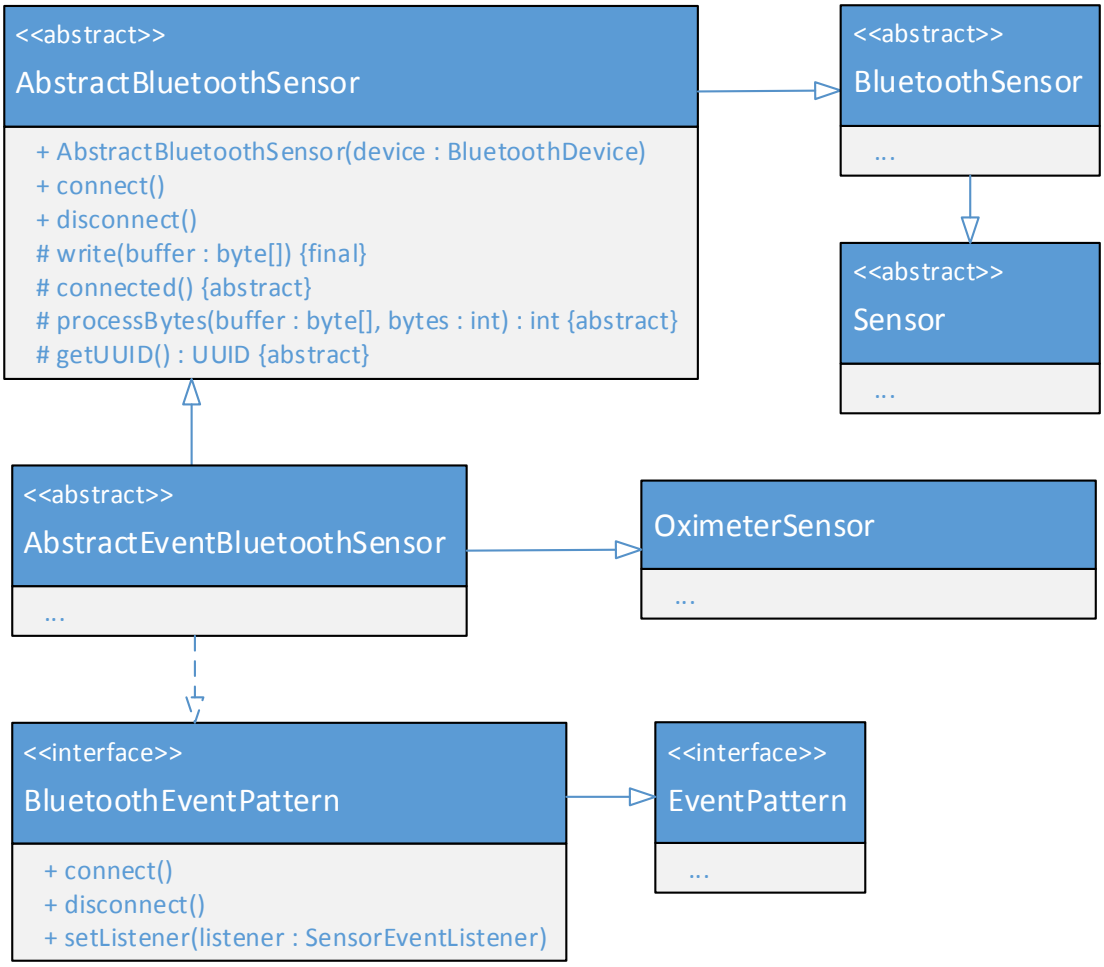


Abbildung 3.8: Vereinfachtes Klassendiagramm der Klassen in Zusammenhang mit `AbstractBluetoothSensor` und deren Vererbungshierarchie.

connect() / disconnect()	Hier findet der Bluetooth-Verbindungsaufbau statt. Wurde dieser erfolgreich abgeschlossen, wird `connected()` aufgerufen um so in Subklassen über den Verbindungsaufbau informiert zu werden.
connected()	Wird nach erfolgreichem Verbindungsaufbau aufgerufen. Subklassen bzw. Treiber überschreiben diese Methode und senden hier typischerweise im Rahmen des sensorspezifischen Kommunikationsprotokolls bestimmte Anfragenachrichten bzw. Kommandos zum Bluetooth-Sensor.
write(byte[])	Diese Methode ermöglicht in Subklassen das Schreiben auf den Bluetooth-Socket.
processBytes(byte[], int)	Über diese Methode werden die vom Bluetooth-Socket gelesenen Bytes an die Subklasse bzw. den Treiber geliefert. Aufgrund der Erfahrungen mit den im Rahmen dieser Arbeit betrachteten Sensoren ist der Zugriff auf den Bytestrom für eine paketbasierte Verarbeitung im Treiber optimiert. Da bei einem direkten Lesen von einem Bluetooth-Socket nicht sichergestellt ist, dass die durch einen Leseaufruf erhaltenen Daten genau mit den Paketgrenzen des Sensor-Kommunikationsprotokolls übereinstimmen bietet diese Methode die Möglichkeit nur eine bestimmte Anzahl an Bytes zu verarbeiten und die restlichen Bytes für die spätere Verarbeitung zurückzustellen.
getUUID()	Für den Verbindungsaufbau über Bluetooth-Sockets ist eine *UUID* notwendig, die den gewünschten Dienst bezüglich des *Service Discovery Protocol (SDP)* bezeichnet. Durch Überschreiben dieser Methode in einem Treiber, kann die vom Sensor unterstütze UUID festgelegt werden.

Tabelle 3.1: Methoden von `AbstractBluetoothSensor`. Treiber von Bluetooth-Sensoren können diese Klasse erweitern und erhalten damit einen einfachen, paketbasierten Kommunikationskanal zum Sensor.

3.3.4 Konkrete Realisierung der Interaction Patterns

In Abschnitt 3.2.2 wurde das Prinzip der Interaction Patterns bereits auf konzeptioneller Ebene vorgestellt. Im Folgenden wird darauf eingegangen, wie dieses Prinzip konkret auf Code- bzw. Interface-Ebene realisiert wurde.

Ein Interaction Pattern stellt den konzeptionellen Interaktionsablauf mit einem Sensor dar. Aus diesem ergibt sich jeweils eine Schnittstelle, die für diese Interaktion genutzt werden kann. Die Interaction Patterns wurden daher als *Interfaces* implementiert. Hier ist jedoch zu beachten, dass eine Anwendung, die das Sensor-Framework nutzt, nicht direkt mit dem Sensor über die Interaction-Pattern-Schnittstelle interagiert, sondern nur indirekt mithilfe der Klasse `SensorFramework`, welche gewissermaßen den von außen sichtbaren Teil des Sensor-Frameworks darstellt. Aus Sicht der Anwendung besteht ein Interaction Pattern dadurch aus bestimmten Methoden, die an `SensorFramework` aufgerufen werden können. So besteht etwa das Event Pattern aus den Methoden `registerListener(SensorEventListener, Sensor)` und `unregisterListener(SensorEventListener, Sensor)` in `SensorFramework`. Welche Methoden hier zu welchem Interaction Pattern gehören ist der Dokumentation der Methoden zu entnehmen. Welches Interaction Pattern ein Sensor unterstützt lässt sich zur Laufzeit herausfinden. Listing 3.3 zeigt die hierfür verwendete Methode. Diese befindet sich in der Klasse `Sensor`, der Basisklasse für alle Treiber. Die Methode ist als `final` markiert, was ein Überschreiben in Treibern verhindert und somit sicherstellt, dass diese tatsächlich ein Interaction Pattern implementieren.

Das konkrete Interface, das ein Sensor für die Unterstützung eines Interaction Patterns implementieren muss, ist nicht nur vom Interaction Pattern selbst abhängig. Es spielt auch eine Rolle, ob der Sensor zum Beispiel ein Bluetooth-Sensor, oder ein interner Gerätesensor ist. Denn für Bluetooth-Sensoren ist beispielsweise der `BluetoothSensorManager` für die Verwaltung der Bluetooth-Verbindung zum Sensor zuständig. Dafür benötigt er ein Interface zum Sensor mit bluetoothspezifischen Methoden, wie zum Beispiel `connect()`. Daher gibt es zu einem Interaction Pattern jeweils ein Interface, von dem wiederum für jede Verbindungsart speziellere Interfaces erben.

Dies ist in Abbildung 3.8 zu sehen. Hier gibt es das Interface `EventPattern` und das speziellere `BluetoothEventPattern`.

```
1  public final Sensor.InteractionPattern getInteractionPattern()
       throws ClassCastException {
2    if (this instanceof EventPattern)
3      return InteractionPattern.EVENT;
4    else if (this instanceof AsynchronousRequestPattern)
5      return InteractionPattern.ASYNC_REQUEST;
6    else if (this instanceof SynchronousRequestPattern)
7      return InteractionPattern.SYNC_REQUEST;
8    else if (this instanceof RecordingPattern)
9      return InteractionPattern.RECORDING;
10   // can already be checked by the sensor manager when loading
       drivers
11   throw new ClassCastException("sensor driver does not
       implement any interaction pattern");
12 }
```

Listing 3.3: Es kann zur Laufzeit erfragt werden, welches Interaction Pattern von einem Sensor unterstützt wird.

Wird in einer Anwendung das *Event Pattern* verwendet, sollte darauf geachtet werden den verwendeten *Listener* nicht im Kontext der *Activity* zu speichern, da dann im Rahmen des Lebenszyklus der Listener beispielsweise bei jeder Rotation des Bildschirms deregistriert, neu erzeugt und wieder registriert werden muss. Wird hierbei der letzte bzw. einzige für einen Sensor registrierte Listener abgemeldet und wieder angemeldet, wird im Falle eines Bluetooth-Sensors dann jedes mal auch die Bluetooth-Verbindung zum Sensor neu aufgebaut. Um dies zu verhindern sollte der Listener in einem statischen Kontext gespeichert und damit unabhängig von Activity-Lebenszyklen wiederverwendet werden. Dieses Verhalten ist eine Folge der abstrakten Schnittstelle des Sensor-Frameworks, die Methoden wie `connect()` und `disconnect()` vor einer Anwendung versteckt. Solche Verbindungslogik ist im entwickelten Framework der Ebene der Sensor-Manager zugeordnet. An diesem Beispiel lassen sich Trade-offs bei der Schnittstellengestaltung zwischen Abstraktion, Einheitlichkeit und Vereinfachung auf der einen Seite und funktionaler Mächtigkeit, der Sichtbarkeit verbindungsspezifischer Logik und höherer Komplexität auf der anderen Seite erkennen.

3.3.5 Implementieren eines Sensortreibers

Im Folgenden wird am Beispiel des in dieser Arbeit betrachteten Pulsoximeters dargestellt, wie ein Treiber für einen Sensor geschrieben werden kann. Abbildung 3.9 zeigt den Treiber `OximeterSensor` und dessen Oberklassen. Es ist zu sehen, dass ein Sensortreiber, einige abstrakte Methoden implementieren muss. Diese stammen zum einen aus der Klasse `Sensor`, aber auch aus dem Interface des jeweiligen Interaction Patterns. Zum Teil werden diese abstrakten Methoden durch abstrakte Klassen auf dem Vererbungspfad zwischen `Sensor` und `OximeterSensor` implementiert. So steht beispielsweise auf Ebene des `BluetoothSensor` der `ConnectionType` als `ConnectionType.BLUETOOTH` fest und die entsprechende Methode ist hier implementiert. Da bei Bluetooth-Sensoren außerdem, wie bereits in Abschnitt 3.3.3 diskutiert, auf die Klasse `AbstractBluetoothSensor` zurückgegriffen werden kann, müssen hier nur noch wenige Methoden tatsächlich implementiert werden. Methoden wie `getName()` erfordern dabei unter Umständen nur eine Zeile Code. Da es für ein Pulsoximeter bereits einen vordefinierten Typ gibt gilt dies ebenso für `getType()`. Ähnlich simpel ist die Implementierung von `getUUID()`, da hier lediglich die UUID des Serial Port Profile zurückgegeben werden muss. Etwas interessanter ist die Methode `isDeviceSupported(BluetoothDevice device)`. Deren Implementierung in `OximeterSensor` ist in Listing 3.4 zu sehen. Der `BluetoothSensorManager` muss Bluetooth-Geräte den Treibern zuordnen, durch welche sie unterstützt werden. Deshalb muss jeder `BluetoothSensor` diese Methode besitzen. Leider lässt sich deren Existenz nicht durch den Compiler erzwingen, da statische Methoden in Java nicht vererbt werden können und hier sozusagen eine abstrakte statische Methode benötigt würde. Fehlt diese Methode in einem Treiber für einen Bluetooth-Sensor führt dies dazu, dass der Treiber kein Gerät unterstützt. Dem Treiber stehen in `isDeviceSupported(...)` verschiedene Informationen zur Verfügung, anhand derer er unterstützte Geräte erkennen kann.

- **Gerätename:** Der Gerätename kann einem gerätespezifischen Namensschema entsprechen, anhand dessen das Gerät erkannt werden kann.

- **UUIDs:** Durch einen SDP-Request, der während des Pairing-Prozesses durchgeführt wird, sind die vom Gerät gemeldeten UUIDs bekannt und können hier zur Identifikation genutzt werden.
- **Hardware-Adresse:** Ähnlich wie bei der im Ethernet verwendeten MAC-Adresse gibt es auch hier einen herstellerspezifischen Teil, der zur Erkennung des Geräts beitragen kann.

```
1  public static boolean isDeviceSupported(BluetoothDevice device)
2  {
3    // check name
4    String name = device.getName();
5    if (!name.matches("BCOXM-\\d{9}"))
6      return false;
7
8    // check supported uuid
9    ParcelUuid[] uuids = device.getUuids();
10   if (uuids.length != 1)
11     return false;
12   if (!uuids[0].getUuid().equals(SPP_UUID))
13     return false;
14
15   return true;
16 }
```

Listing 3.4: Diese Methode wird benötigt um einem Treiber eines Bluetooth-Sensors die unterstützten Geräte zuzuordnen. Implementierung aus `OximeterSensor`.

3.3.6 Umsetzung des Typensystems

In Abschnitt 3.2.3 wurde das für die Klassifizierung der Sensoren verwendete Typensystem vorgestellt. Dabei gibt es bereits einige vordefinierte Typen im hier implementierten Sensor-Framework. Es lassen sich jedoch auch neue Typen in den Sensortreibern definieren. Dies soll im Folgenden anhand der konkreten Implementierung gezeigt werden.

Listing 3.5 zeigt, wie einige vordefinierte Sensortypen als Aufzählungstyp (`Enum`) umgesetzt wurden. Es finden sich hier Referenzen auf `SensorData.Type`, einen weiteren Aufzählungstyp, der ähnlich wie die Sensortypen umgesetzt wurde und dessen Implementierung daher hier nicht gezeigt wird. Dieser repräsentiert die Datentypen der von den Sensoren gemessenen Daten. Beim Schreiben eines Sensortreibers wird der

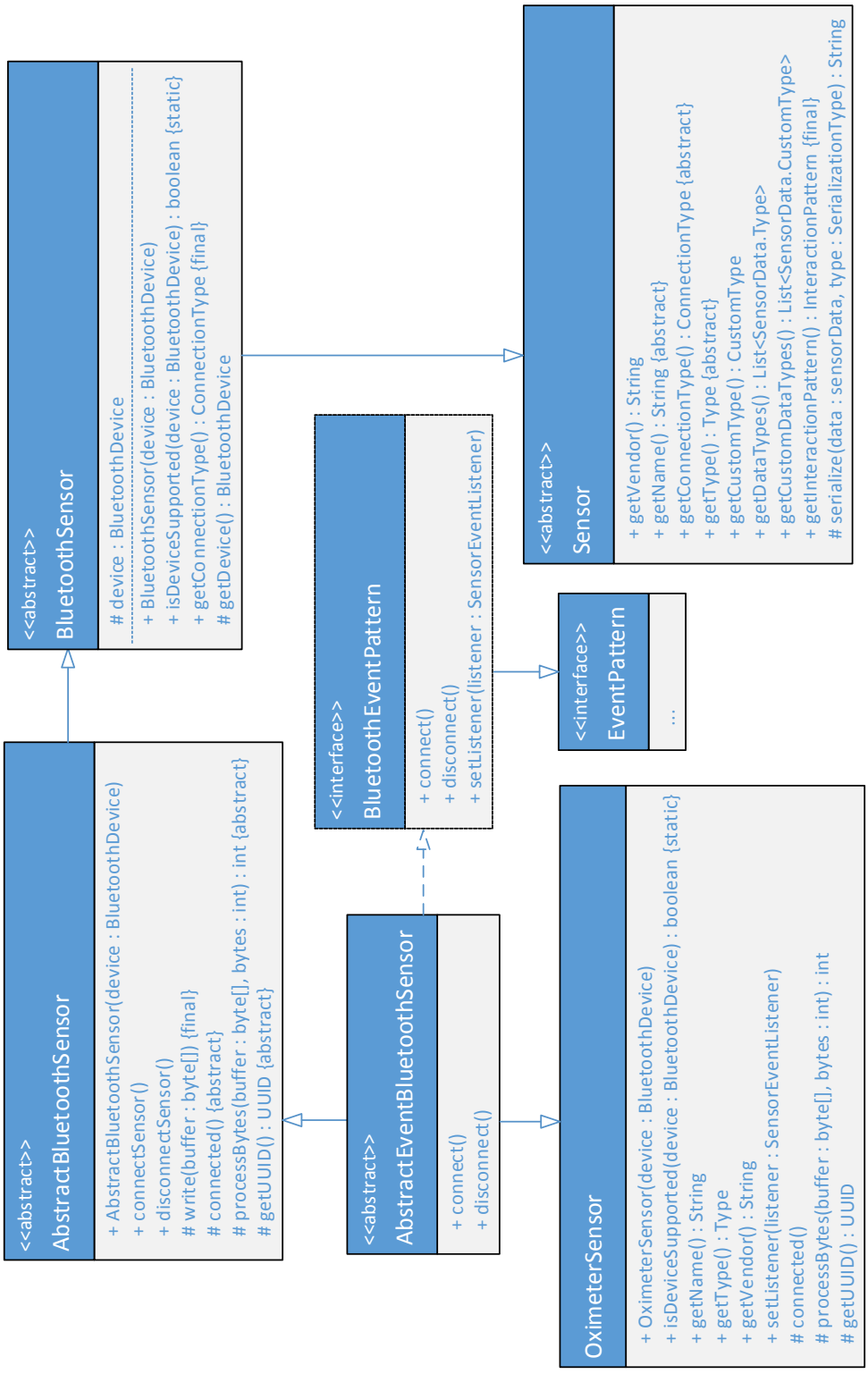


Abbildung 3.9: Vererbungshierarchie eines Bluetooth-Sensortreibers

Sensortyp und die gemessenen Datentypen durch Überschreiben von Methoden aus der Basisklasse `Sensor` festgelegt. Wie dies genau funktioniert, wird im Folgenden anhand der jeweiligen Methoden in `Sensor` gezeigt.

```
1  public enum Type {
2    PULSE_OXIMETER("pulse oximeter", SensorData.Type.
       SATURATION_OF_PERIPHERAL_OXYGEN, SensorData.Type.
       PULSE_RATE),
3    CAMERA("camera", SensorData.Type.IMAGE_JPEG),
4    MICROPHONE("microphone", SensorData.Type.AUDIO_THREE_GPP),
5    BLOOD_GLUCOSE_METER("blood glucose meter", SensorData.Type.
       GLUCODE_RECORD, SensorData.Type.GLUCODE_RECORD_LIST),
6    CUSTOM_SENSOR("custom"), UNKNOWN("unknown");
7
8    private final String name;
9    private final List<SensorData.Type> dataTypes;
10
11   private Type(String name, SensorData.Type... dataTypes) {
12     this.name = name;
13     this.dataTypes = new ArrayList<SensorData.Type>(Arrays.
         asList(dataTypes));
14   }
15
16   public String getName() {
17     return name;
18   }
19
20   public int getId() {
21     return this.ordinal();
22   }
23
24   public List<SensorData.Type> getDataTypes() {
25     return Collections.unmodifiableList(dataTypes);
26   }
27 }
```

Listing 3.5: Umsetzung der vordefinierten Typen.

public abstract Sensor.Type getType()

Im einfachsten Fall existiert bereits ein vordefinierter Sensortyp, der für den zu implementierenden Sensortreiber verwendet werden kann. In diesem Fall ist dies die einzige Methode, die überschrieben werden muss. Die Implementierung der restlichen typenbezogenen Methoden in `Sensor`, wie beispielsweise `getDataTypes()`, nutzen dann den hier gewählten Typ, um ebenfalls passende Typinformationen zurückzugeben. Um einen neuen Sensortyp zu definieren, muss hier der Wert `Sensor.Type.CUSTOM_SENSOR` zurückgegeben werden. Es

ist möglich einen Sensortyp zu definieren, der nur vordefinierte Datentypen benutzt. Es kann aber auch ein neuer Sensortyp definiert werden, der auch neue Datentypen definiert, oder der zum Teil vordefinierte und zum Teil neu definierte Datentypen verwendet. Soll ein neuer Sensortyp mit vordefinierten Datentypen definiert werden, müssen `getCustomType()` und `getDataTypes()` überschrieben werden. Sollen auch neue Datentypen definiert werden, muss zusätzlich `getCustomDataTypes()` überschrieben werden.

public Sensor.CustomType getCustomType()

Diese Methode muss überschrieben werden, wenn in `getType()` der Wert `Sensor.Type.CUSTOM_SENSOR` zurückgegeben wird. Hier kann ein neuer Sensortyp definiert werden. Der hierfür verwendete `Sensor.CustomType` besteht, wie bereits in Abbildung 3.6 gezeigt, im Wesentlichen aus einem Namen und einer ID (Identifikationsnummer). Um Kollisionen mit den vordefinierten Typen zu vermeiden sollte diese hoch genug gewählt werden, zum Beispiel größer als tausend. Die zugehörigen Datentypen werden durch `getDataTypes()` festgelegt.

public List<SensorData.Type> getDataTypes()

Diese Methode muss überschrieben werden, wenn in `getType()` der Wert `Sensor.Type.CUSTOM_SENSOR` zurückgegeben wird. Falls neue Datentypen definiert werden sollen, kann `SensorData.Type.CUSTOM_DATA` verwendet werden. In diesem Fall muss auch `getCustomDataTypes()` überschrieben werden.

public List<SensorData.CustomType> getCustomDataTypes()

Falls neue Datentypen definiert werden sollen, muss diese Methode überschrieben werden. Hier werden die Datentypen dann durch Instanzen der Klasse `SensorData.CustomType` beschrieben. Ein Datentyp besteht aus einem Namen, einer eindeutigen ID und einem Klassentyp (`Class<?>`). Sollen für einen Sensortyp sowohl vordefinierte als auch neu definierte Datentypen eingesetzt werden, lassen sich Instanzen von `SensorData.CustomType` auch einfach aus einem vordefinierten `SensorData.Type` erzeugen.

Entsprechend der Anforderung A3 kann das Sensor-Framework mithilfe des so umgesetzten Typensystems Anfragen einer Anwendung nach einem bestimmten Typ

von Sensoren beantworten. Es kann auf verfügbare Sensoren nach dem Sensortyp, nach den vom Sensor gemessenen Datentypen und nach der Art der Verbindung zugegriffen werden. Listing 3.6 zeigt die Methoden der Klasse `SensorFramework` durch die dies ermöglicht wird. Die beiden Methoden `getSensorsByType(...)` und `getSensorsByDataType(...)` werden dabei im Zusammenhang mit vordefinierten Typen benutzt, während für dynamisch definierte Typen die beiden Methoden `getSensorsByTypeId(...)` und `getSensorsByDataTypeId(...)` verwendet werden können, in denen die Typen über die ID angegeben werden.

```
1 public List<Sensor> getSensorsByType(Sensor.Type sensorType);
2
3 public List<Sensor> getSensorsByTypeId(int sensorTypeId);
4
5 public List<Sensor> getSensorsByDataType(SensorData.Type
     sensorDataType);
6
7 public List<Sensor> getSensorsByDataTypeId(int dataTypeId);
8
9 public List<Sensor> getSensorsByConnectionType(ConnectionType
     cType);
```

Listing 3.6: Schnittstelle für den Zugriff auf Sensoren in `SensorFramework` nach Sensortyp, Datentyp und Art der Verbindung.

Neben dieser Art des Zugriffs auf die Sensoren ermöglicht das Typensystem außerdem einer Anwendung, die das Sensor-Framework nutzt, zur Laufzeit weitere Informationen über den Sensor und die gemessenen Daten eines Sensors zu erhalten. Abrufbar sind so beispielsweise der Typenname zur entsprechenden Anzeige in einer Oberfläche oder Klassentypen der Daten für eine entsprechende weitere Verarbeitung.

3.3.7 Threading in Sensor-Managern

In Android-Anwendungen muss immer darauf geachtet werden, den *UI- (User Interface)* bzw. *Main-Thread* nicht für längere Zeit zu blockieren, da ansonsten die Benutzeroberfläche nicht mehr auf Eingaben reagiert, was nach einer gewissen Zeit zudem zu einer Fehlermeldung und dem Beenden der Anwendung führen kann. Daher stellt die Android-API verschiedene Konzepte bereit, wie längere Aufgaben mithilfe von separaten Threads im Hintergrund ausgeführt werden können. Dies ist zum Beispiel dann nötig,

wenn Daten über das Netzwerk nachgeladen werden müssen oder umfangreiche Speicherzugriffe durchgeführt werden. Ist die Verarbeitung im Hintergrund abgeschlossen oder liegen Zwischenergebnisse vor, gibt es Mechanismen um von Hintergrund-Threads aus mit dem Main-Thread zu kommunizieren. Eine ähnliche Kommunikation zwischen verschiedenen Threads lässt sich auch innerhalb des Sensor-Frameworks nutzen.

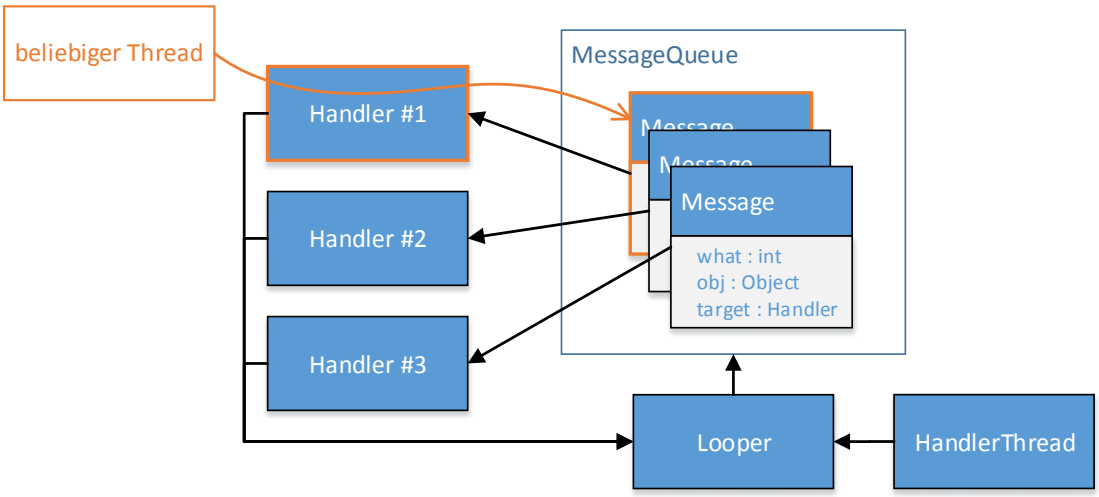


Abbildung 3.10: Aufbau einer Message Queue in Android. Nachrichten werden in eine Warteschlange eingereiht und sequentiell verarbeitet. Abbildung frei nach [HP13].

Für die Kommunikation zwischen Sensoren und Sensor-Manager wird eine *Message Queue* verwendet. Diese wird in der Android-API durch die Klassen `Looper`, `Handler`, `HandlerThread` und `Message` realisiert. Abbildung 3.10 zeigt wie diese miteinander zusammenhängen. Nachrichten können über einen `Handler` in die Warteschlange eingereiht werden. Dieser ist an den `Looper` gebunden, in dessen Thread er erzeugt wurde. Die Warteschlange wird vom `Looper` verwaltet. Der `HandlerThread` verarbeitet Nachrichten aus der Warteschlange sequentiell.

Sensor-Manager können, falls gewünscht, von `AbstractLooperSensorManager` erben und besitzen dadurch bereits solch eine Message Queue mit einem Thread, der die erhaltenen Nachrichten verarbeitet. Dieses Konzept wird dazu verwendet, um Daten von den Sensoren in Empfang zu nehmen und im Thread des Sensor-Managers weiter-

zuverarbeiten bzw. auf die nächsthöhere Ebene weiterzugeben. Dies wird in Abbildung 3.11 gezeigt. Im Sensortreiber muss dabei nicht direkt mit dem `Handler` bzw. der Message Queue gearbeitet werden. Diese Logik wird auf Ebene des Sensor-Managers gekapselt und so vor dem Sensortreiber versteckt. Im Sensortreiber wird einfach eine Listener-Methode aufgerufen, um Daten an den Sensor-Manager zu übergeben. Diese fügt dann eine Nachricht in die Message Queue ein (siehe Listing 3.7).

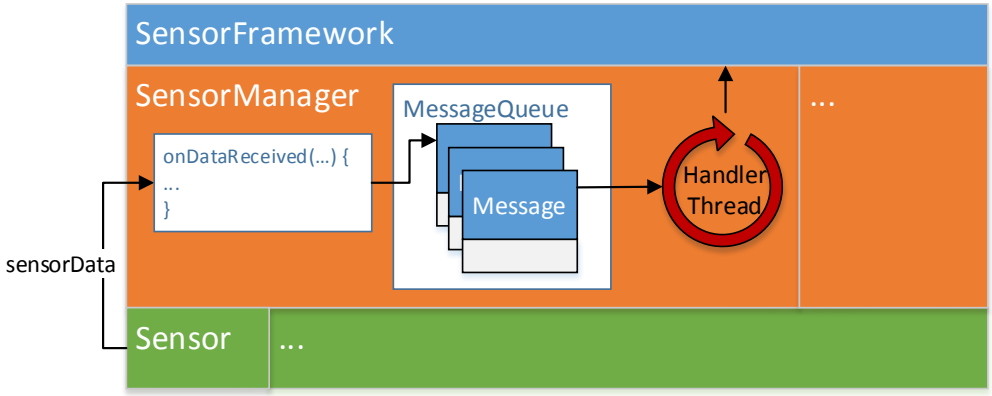


Abbildung 3.11: Sensor-Manager mit Message Queue und eigenem Thread.

```
1 /** called in sensor's thread */
2 public void onDataReceived(SensorData snsrData) {
3   hndlr.obtainMessage(DATA_RECEIVED, snsrData).sendToTarget();
4 }
```

Listing 3.7: Schnittstelle eines Sensor-Managers zur Datenübergabe an dessen Thread.

3.3.8 Implementierte Sensor-Manager und Treiber

Derzeit ist ein Bluetooth-Manager und ein Manager für interne Sensoren implementiert. Das Framework ist jedoch für das Hinzufügen weiterer Sensor-Manager ausgelegt. Es wurde ein Treiber für das in Abschnitt 2.1.2 diskutierte Oximeter implementiert, sowie für das integrierte Mikrofon und die Kamera. Ein Treiber für einen Blutzuckersensor, der über Bluetooth angesprochen wird, wurde im Rahmen einer weiteren studentischen Arbeit implementiert. Außerdem wurde ein in einer früheren Arbeit [Nie13] verwendeter Bluetooth-Pulsgurt berücksichtigt und damit sichergestellt, dass auch für weitere Bluetooth-Sensoren sehr einfach entsprechende Treiber implementiert werden können.

4

Integration des Sensor-Frameworks in mobile Anwendungen

Bevor an der Konzeption und Implementierung des Sensor-Frameworks gearbeitet wurde, wurde zu Beginn dieser Arbeit eine mobile Fragebogenanwendung aus dem medizinisch-psychologischen Bereich auf der Android-Plattform realisiert. Diese stellt ein Beispiel einer mobilen Anwendung dar, die durch die zusätzliche Integration von Sensordaten[1] profitieren würde. Durch die Realisierung dieses Projekts konnten zudem einige wichtige Erkenntnisse darüber gewonnen werden, welche plattformspezifischen Aspekte, d.h. mögliche Problemfelder und Chancen, bei der darauf folgenden Entwicklung des Sensor-Frameworks zu beachten sind.

Der *KINDEX Mum Screen* ist ein papierbasierter Fragebogen aus der klinischen Psychologie, mit dem bestimmte Risikofaktoren der kindlichen Entwicklung bereits während der

[1] siehe hierzu **UC3** in Abschnitt 1

Schwangerschaft erkannt werden können, um eine Schwangere anhand des Ergebnisses passend zu unterstützen. Im Rahmen einer Diplomarbeit, die an der Universität Ulm durch eine Kooperation mit der Universität Konstanz entstand, wurde dieser Fragebogen bereits als iOS-Anwendung umgesetzt [Lie12]. Es ergaben sich jedoch zusätzliche Anforderungen an die bereits realisierte mobile Anwendung, sowie den Wunsch nach einer entsprechenden Android-Umsetzung.

4.1 Anforderungen

Auf eine detailierte Diskussion der Anforderungen an die zu entwickelnde mobile Anwendung wird hier verzichtet. In der entsprechenden Diplomarbeit [Lie12] werden die Anforderungen und deren Hintergrund bereits in ausführlicher Form dargestellt, sodass diese dort bei Bedarf nachgeschlagen werden können. Um dennoch einen grobe Vorstellung von der zu realisierenden Anwendung und deren Umfang zu ermöglichen, werden im Folgenden einige Anforderungen kurz in Aufzählungsform aufgeführt.

Funktionale Anforderungen: Automatische Auswertung, Zeiterfassung, Mehrbenutzerverwaltung, Mehrsprachigkeit, Verschlüsselung, Speicherung, Personalisierung durch Arztdaten, Export, Pausierfunktion, Ansehen ausgefüllter Fragebögen, Administratorbereich

Nicht-Funktionale Anforderungen: Usability, Design, Datenschutz/Sicherheit

4.2 Realisierte Anwendung

Im Folgenden wird die entwickelte Anwendung vorgestellt. Abbildung 4.1 zeigt in vereinfachter Form den Aufbau der Navigation durch die Anwendung. Es handelt sich dabei um eine Art logischen Aufbau der Applikation auf Bildschirm-Ebene bzw. aus Benutzersicht. Nicht gezeigt ist hier die Implementierungsperspektive, in der die Applikation in *Activities*, *Fragments*, und *Dialogs* aufgeteilt ist. Anhand dieser Aufteilung der Anwendung aus Benutzersicht, werden folgend die jeweils über einen Bildschirm zugänglichen Funk-

tionen erklärt. Hierbei wird der Einfachheit halber von den beiden Benutzertypen *Arzt* und *Schwangere* gesprochen und bezüglich des Typs Arzt nicht zwischen Gynäkologe, Hebamme und den jeweils anderen geschlechtsspezifischen Formen unterschieden. Ein Arzt ist derjenige Benutzer, der Zugriff auf den internen Bereich hat und sich dort die ausgefüllten Fragebögen und Auswertungen ansehen kann. Die Schwangere füllt jeweils einen Fragebogen aus.

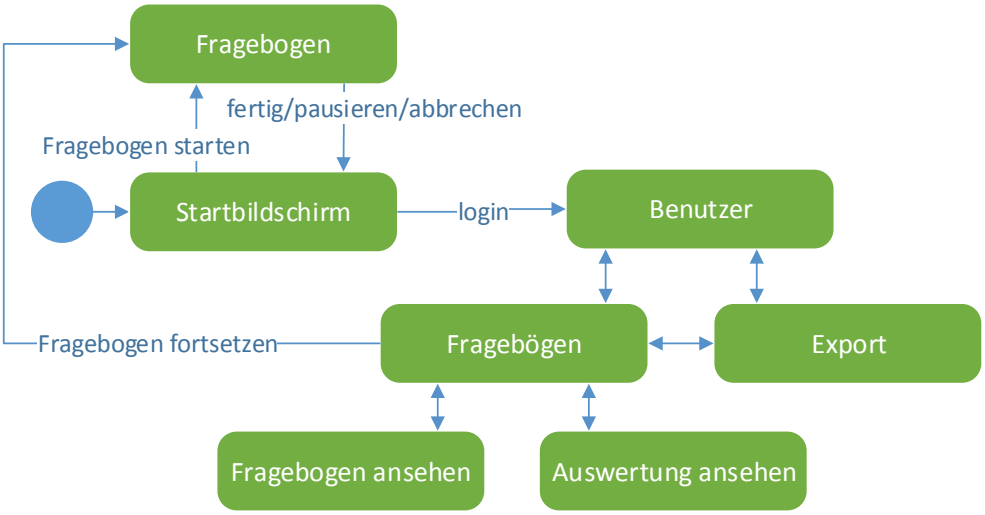


Abbildung 4.1: Überblick über die logische Struktur der Navigation durch die Applikation.

Startbildschirm: Vom Startbildschirm aus kann ein Fragebogen gestartet oder der interne, passwortgeschützte Bereich aufgerufen werden. In beiden Fällen ist ein Login nötig, da ein Fragebogen einem Account zugeordnet ist und mit einem auf sichere, standardisierte Weise[2] aus dem Passwort abgeleiteten kryptografischen Schlüssel verschlüsselt wird. Ist noch kein Benutzer vorhanden, beispielsweise beim ersten Start der Applikation, kann dieser zunächst angelegt werden.

Fragebogen: Zunächst sind hier einige vorbereitende Fragen vom Arzt auszufüllen, wobei Felder wie Datum, Zeit und persönliche Daten des Accounts, wie beispielsweise der Name des Arztes und die Adresse der Praxis, bereits vorausgefüllt, trotzdem jedoch änderbar sind. Nach einer Outro-Seite für den Arzt sowie einer Intro-Seite mit Instruktionen für die Schwangere beginnt dann der eigentliche

[2] siehe Abschnitt 4.3.1 für weitere Details

Fragebogen. Die Antworten werden hier über Textfelder oder eigens erstellte *UI-Komponenten (User Interface)* eingegeben. In Abbildung 4.2 werden zwei Fragen gezeigt, in dem die eigens erstellten *Slider* mit neutralem Bereich[3] eingesetzt werden. Wird eine Teilfrage nicht ausgefüllt, wird hierrauf vor dem Abschluss dieser Frage hingewiesen und gegebenenfalls der sichtbare Bildschirmausschnitt auf die noch unbeantwortete Teilfrage verschoben. Die Antwortdaten werden nach jeder Frage in verschlüsselter Form persistent in einer XML-Datei gespeichert. Außerdem besteht jederzeit die Möglichkeit den Fragebogen zu pausieren oder zu beenden.

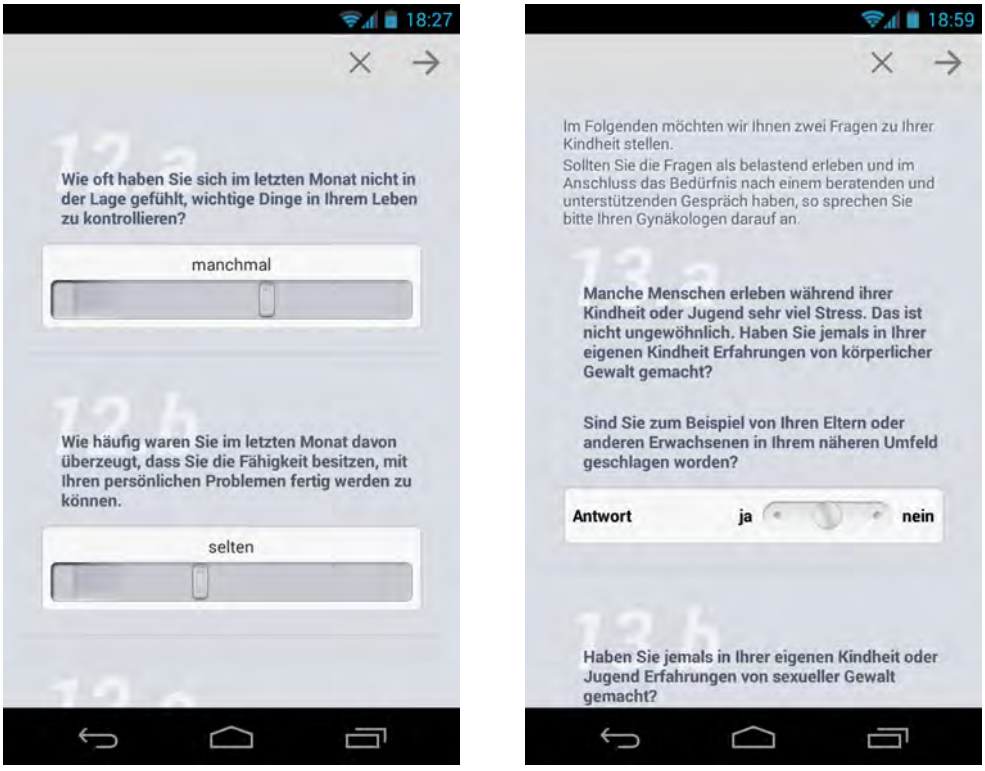
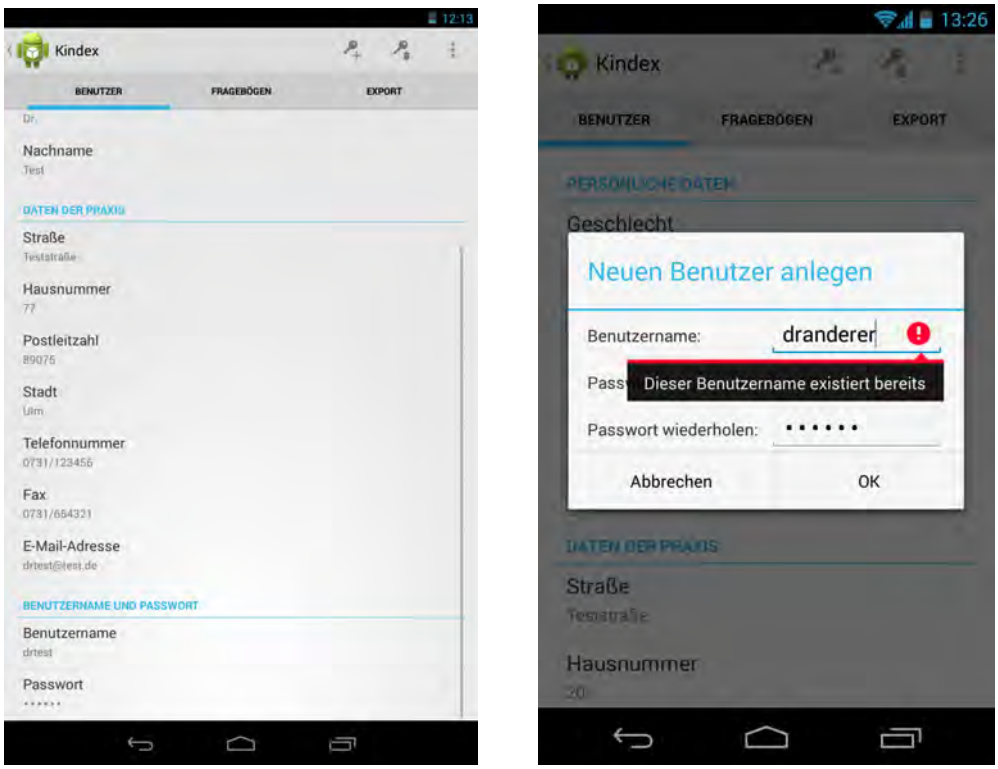


Abbildung 4.2: Der Fragebogen auf dem Nexus 4 (Diagonale 4,7").

Benutzer: Dieser Bildschirm wird in Abbildung 4.3 gezeigt. Er ist Teil des internen Bereichs, zwischen dessen Bildschirmen durch horizontales Wischen hin- und hergewechselt werden kann. Hier können über Dialoge Benutzerdaten vom Arzt

[3] siehe Abschnitt 4.3.2 für weitere Details zu den erstellten UI-Komponenten

eingegeben bzw. geändert werden. Diese werden zusammen mit jedem Fragebogen abgespeichert und können außerdem teilweise zur Personalisierung der Fragebogendarstellung eingesetzt werden. Über die *Action Bar* am oberen Bildschirmrand können neue Benutzer hinzugefügt oder Benutzer gelöscht werden.

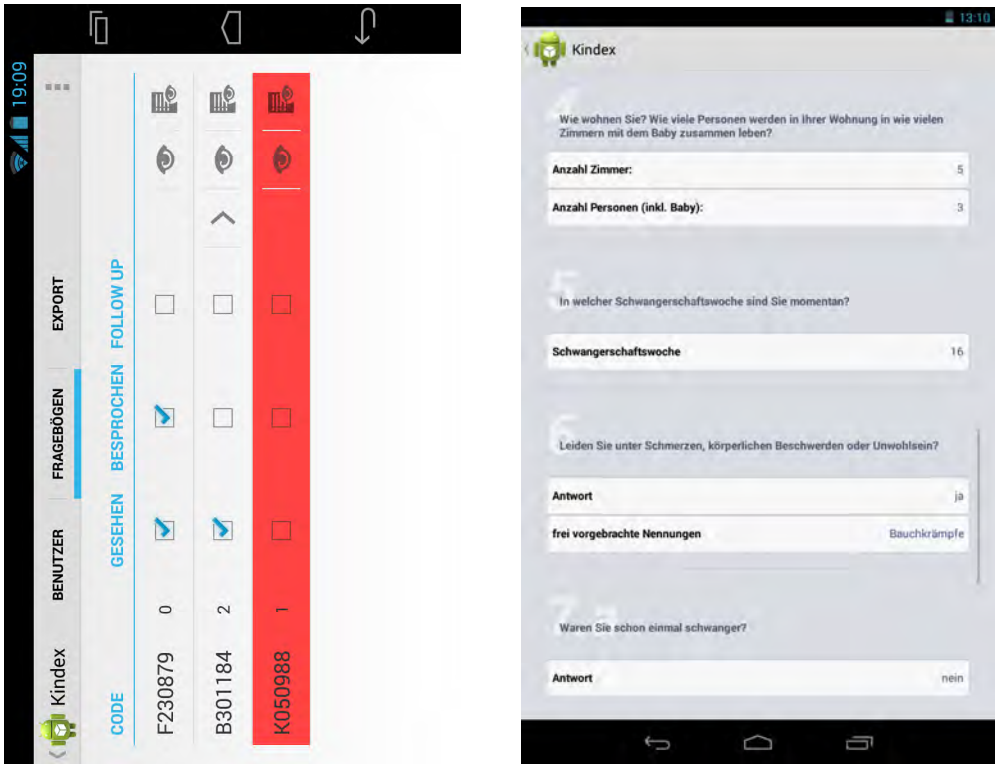


(a) Nexus 7 (Diagonale 7") (b) Nexus 4 (Diagonale 4,7")

Abbildung 4.3: Benutzerdaten im internen Bereich. Die Auswahl eines Datums öffnet einen jeweils angepassten Änderungsdialog. Über die *Action Bar* sind Funktionen zur Benutzerverwaltung zugänglich. Durch Wischen kann zwischen den Tabs gewechselt werden.

Fragebögen: Abbildung 4.4a zeigt diesen Bildschirm. Hier werden alle gespeicherten Fragebögen in Listenform dargestellt. Die Anzahl der Risikofaktoren wird zu jedem Fragebogen angezeigt und es kann über Checkboxen ausgewählt werden, ob der Fragebogen beispielsweise bereits angesehen oder besprochen wurde. Durch Auswahl des Listeneintrags oder über das entsprechende Icon kann ein Fragebogen angesehen werden. Außerdem ist die Auswertung zugänglich und

falls der Fragebogen nicht bereits beendet oder abgebrochen wurde, kann er über die Liste der Fragebögen fortgesetzt werden. Am Beispiel dieses Bildschirms ist gut zu erkennen, dass die Android-Designrichtlinien [And13a] für Navigation, Icons, Abstände und Farben in der gesamten Applikation berücksichtigt wurden.

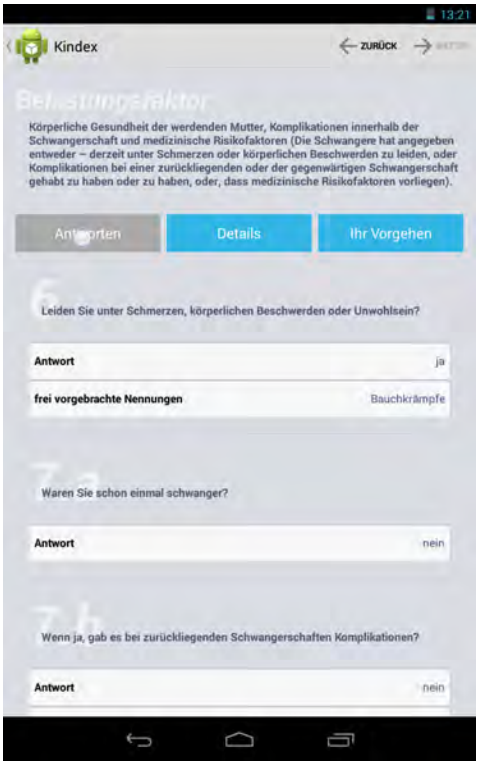


(a) Nexus 4 (b) Nexus 7

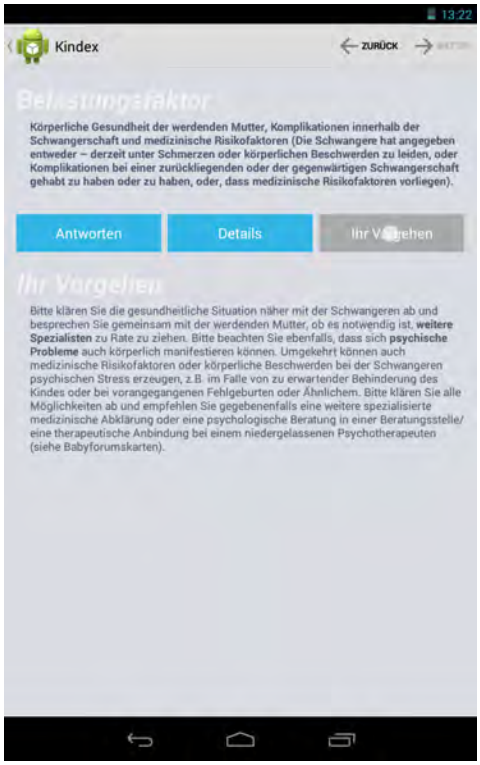
Abbildung 4.4: Fragebögenübersicht und Ansehen eines ausgefüllten Fragebogens. In Abbildung a ist von oben nach unten ein abgeschlossener, ein pausierter und ein abgebrochener Fragebogen zu sehen.

Fragebogen ansehen: In diesem Bildschirm kann ein komplett oder teilweise ausgefüllter Fragebogen angesehen werden (siehe Abbildung 4.4b). Alle Antworten werden hier einheitlich in Textform dargestellt, auch wenn diese beispielsweise über Slider eingegeben wurden. Das Design entspricht dabei dem, das auch beim Ausfüllen des Fragebogens eingesetzt wird.

Auswertung ansehen: Abbildung 4.5 zeigt die Auswertung eines Fragebogens, die von einem Arzt über den internen Bereich erreicht werden kann. Durch seitliches Wischen kann zwischen den einzelnen, durch die Auswertung des Fragebogens erkannten Risikofaktoren hin- und hergewechselt werden. Zu jedem erkannten Risikofaktor sind hierbei weitere Details abrufbar, sowie Hinweise, wie der verantwortliche Arzt im Rahmen eines Beratungsgesprächs auf den jeweiligen Risikofaktor eingehen und reagieren sollte. Außerdem kann genau die Teilmenge der Fragen und Antworten angezeigt werden, die für die Auswertung bezüglich des gerade betrachteten Risikofaktors relevant ist.



(a) Für den betrachteten Risikofaktor ausschlaggebende Antworten.



(b) Vorschlag für das Vorgehen des Arztes bezüglich des betrachteten Risikofaktors.

Abbildung 4.5: Auswertung eines teilweise oder komplett ausgefüllten Fragebogens. Die hellen Kreise stellen Berührungen des Benutzers dar und sind in der Anwendung nicht sichtbar.

Export: Diese Funktion ist derzeit noch nicht implementiert. Hier soll der Export der Fragebögen über einen *Webservice* möglich sein. Die Ergebnisse der Fragebögen liegen jedoch bereits jeweils in einer XML-Struktur vor, sodass bereits einige Vorarbeiten für solch eine Exportfunktion abgeschlossen sind.

4.3 Implementierungsdetails

Während die Anwendung bisher hauptsächlich aus Benutzersicht dargestellt wurde, werden im Folgenden einige wichtige Implementierungsaspekte diskutiert.

Bei der Implementierung der Fragebogenanwendung wurde das Potential ausgenutzt, dass die seit Android 3.0 *(Honeycomb)* verfügbaren *Fragments* für die flexible Oberflächengestaltung bieten. Beispielsweise wird für das Ansehen eines ausgefüllten Fragebogens und für die Anzeige einer Teilmenge der Fragen im Rahmen der Auswertung das selbe `Fragment` verwendet und jeweils entsprechend durch Parameter konfiguriert. Fragments kommen außerdem immer dann zum Einsatz, wenn zwischen verschiedenen Bildschirmen durch seitliches Wischen hin- und hernavigiert werden kann.

Für die persistente Datenspeicherung wird sowohl die von der Plattform bereitgestellte *SQLite*-Datenbank als auch die Speicherung in XML-Form verwendet. Letztere wird für die Antwortdaten verwendet. Hier existiert pro ausgefülltem Fragebogen eine XML-Datei, welche ausschließlich verschlüsselt in den persistenten Speicher geschrieben wird.

Für die Implementierung des eigentlichen Fragebogens boten sich mehrere Möglichkeiten. Die Oberfläche der einzelnen Fragen kann komplett in XML definiert werden, wobei einzelne Bausteine, beispielsweise eine Art Antwort-Box, mithilfe von `<include />` kombiniert werden. Dies ist jedoch nur möglich, wenn die so eingefügten Bausteine nicht wiederum dynamisch weitere Bausteine kombinieren müssen. Es ist also hierdurch nur eine dynamische Hierarchieebene möglich. Aus diesem und weiteren, hier nicht näher dargestellten Gründen, wird die Oberfläche des Fragebogens im Code durch die Kombination von in XML definierten Bausteinen zusammengesetzt. Beispielsweise gibt je es eine Methode um eine Antwort-Box mit der benötigten Eingabekomponente

hinzuzufügen, die auch bereits einen entsprechenden *Listener* hinzufügt, über den die gegebene Antwort gespeichert wird.

4.3.1 Verschlüsselung

Im Sinne des Datenschutzes sind in Bezug auf Android mehrere Szenarien zu beachten, in denen Unbefugte Zugriff auf die privaten Daten einer Applikation erhalten können. Im Falle der in dieser Arbeit entwickelten Fragebogenanwendung ist dies speziell für die gespeicherten Antwortdaten relevant. Ist auf dem entsprechenden Gerät *Root* aktiviert, können weitere auf dem Gerät installierte Anwendungen, welche die notwendigen Rechte besitzen, auf diese Daten zugreifen. Wird das Gerät gestohlen und die Festplatte mithilfe von externer Hardware ausgelesen, kann auf die Daten unter Umgehung der Schutzmechanismen des Betriebssystems zugegriffen werden. Zudem ist auch eine fehlerhafte Implementierung dieser Schutzmechanismen möglich, wodurch weitere auf dem Gerät installierte Anwendungen die notwendigen Zugriffsrechte erhalten könnten.

Da es sich bei den per Fragebogen erhobenen Antwortdaten um Angaben über die Gesundheit und damit um besondere Arten personenbezogener Daten nach § 3 Absatz 9 Bundesdatenschutzgesetz handelt, sind geeignete Maßnahmen wie „die Verwendung von dem Stand der Technik entsprechenden Verschlüsselungsverfahren“ zum Schutz dieser Daten zu treffen [Bds10]. Um die Daten zu verschlüsseln werden kryptografische Schlüssel benötigt, die entweder auf dem Gerät gespeichert, oder bei Bedarf jedes mal erneut der Anwendung zugeführt werden müssen. Android bietet keine zentrale, geschützte Schlüsselverwaltung an, die sich für die Speicherung symmetrischer Schlüssel einsetzen lässt [Ele12]. Ein Schlüssel ist aufgrund seiner Länge und des zufälligen Aufbaus jedoch auch zu komplex um direkt durch einen Benutzer eingegeben werden zu können. Abhilfe bietet hier *PKCS#5 (Public Key Cryptography Standard)* [Kal00]. Dieser Standard erlaubt es kryptografische Schlüssel auf standardisierte, sichere Art aus einem vom Benutzer eingegebenen Passwort abzuleiten. Das Verfahren schützt durch ein zufälliges *Salt* vor Wörterbuchangriffen und durch *Key-Stretching*, der absichtlichen Wahl einer rechenintensiven Funktion zur Schlüsselerzeugung, vor Angriffen nach der Brute-Force-Methode.

Listing 4.1 zeigt, wie dies konkret auf Codeebene umgesetzt wird. Zunächst werden *Salt* und *Initialisierungsvektor* zufällig erzeugt. Diese müssen nicht geheim gehalten werden und können unverschlüsselt gesichert werden, sollten aber nicht mehrfach verwendet werden. Mithilfe dieser beiden Parameter wird anschließend mit dem in *PKCS#5* empfohlenen Algorithmus *PBKDF2*, der im Prinzip aus der mehrfachen Anwendung einer Hashfunktion besteht, der eigentliche Schlüssel erzeugt. Mit diesem wird der Klartext mit *AES-256 (Advanced Encryption Standard)* verschlüsselt und zusammen mit *Salt* und *Initialisierungsvektor* abgespeichert.

```
1  public static String encrypt(String plaintxt, String pw) {
2    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
3    // generate salt and iv
4    byte[] iv = new byte[cipher.getBlockSize()];
5    SecureRandom random = new SecureRandom();
6    random.nextBytes(iv);
7    byte[] salt = new byte[SALT_LENGTH];
8    random.nextBytes(salt);
9    IvParameterSpec ivParams = new IvParameterSpec(iv);
10
11   // generate key with PBKDF2 key derivation function
12   KeySpec keySpec = new PBEKeySpec(pw.toCharArray(), salt,
        ITERATIONS, KEY_LENGTH);
13   SecretKeyFactory keyFac = SecretKeyFactory.getInstance("
        PBKDF2WithHmacSHA1");
14   byte[] result = keyFac.generateSecret(keySpec).getEncoded();
15   SecretKey key = new SecretKeySpec(result, "AES");
16
17   // encrypt
18   cipher.init(Cipher.ENCRYPT_MODE, key, ivParams);
19   byte[] ciphrtxt = cipher.doFinal(plaintxt.getBytes("UTF-8"));
20   return toBase64(iv) + SEPARATOR + toBase64(salt) + SEPARATOR
        + toBase64(ciphrtxt);
21 }
```

Listing 4.1: Verschlüsselung mithilfe eines Passworts nach PKCS#5

In der in dieser Arbeit entwickelten Fragebogenanwendung werden die von den Schwangeren eingegebenen Antwortdaten mithilfe eines Schlüssels, der aus dem Benutzerpasswort des Arztes abgeleiteten wird, verschlüsselt. Es wird für jeden Fragebogen ein neues *Salt* und damit auch ein neuer Schlüssel verwendet. Für die Anzahl der Iterationen und weitere Parameter werden die im Standard vorgeschlagenen Werte verwendet. Durch das *Key-Stretching* und den damit verbundenen Rechenaufwand bei der Schlüsselerzeugung muss beachtet werden, dass der selbe Schlüssel nicht unnötig

mehrfach berechnet sondern im flüchtigen Speicher gesichert wird, da, abhängig von den gewählten Parametern und der Leistung des Geräts, eine gewisse Zeit für die Berechnung eines Schlüssels benötigt wird[4].

Weiterhin ist zu beachten, dass das Benutzerpasswort eines Arztes nicht im Klartext abgespeichert werden darf. Daher wird zu jedem Passwort ein *Salted Hash* gespeichert mit dem keine Rückschlüsse auf das eigentliche Passwort möglich sind, mithilfe dessen jedoch erkannt werden kann, ob das eingegebene Passwort korrekt ist. Durch das *Salt* und den Einsatz von *Key-Stretching* in *PBKDF2* werden mögliche Attacken auch hier erschwert [Def13]. Listing 4.2 zeigt, wie der *Salted Hash* erzeugt wird. Die Funktion `keyDerivationPBKDF2(byte[], String)` wurde bereits in Listing 4.1 dargestellt. Zur Vereinfachung wurde der Code dieser Methode dort jedoch in `encrypt(...)` eingefügt.

```
1 public static byte[] getSaltedHash(String pw, byte[] salt) {
2   return keyDerivationPBKDF2(salt, pw);
3 }
```

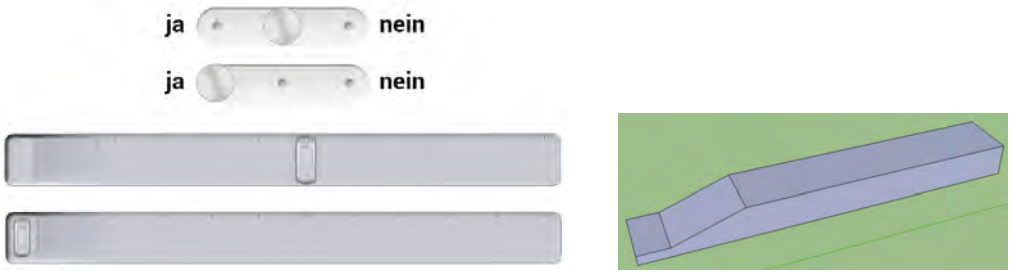
Listing 4.2: Aus einem Benutzerpasswort wird ein *Salted Hash* erzeugt.

4.3.2 Entwickelte UI-Komponenten

Bei der Erstellung der Fragebogen-Anwendung musste darauf geachtet werden, dass das Ergebnis einer Befragung nicht durch ungeeignete Eingabekomponenten verfälscht wird. Beim Start einer Frage darf beispielsweise nicht bereits ein bestimmter Wert vorselektiert sein. Sind für eine Frage bereits Eingaben erfolgt, muss es außerdem möglich sein, die gegebene Antwort wieder zu löschen. Wird in einer Frage ein *Slider* eingesetzt, muss es daher eine neutrale Stellung geben, welche den Anfangszustand der Eingabekomponente darstellt in die sie auch wieder zurückversetzt werden kann. Eingabekomponenten, die diesen Anforderungen entsprechen, wurden bereits im Rahmen einer weiteren studentischen Arbeit konzipiert, jedoch nicht implementiert. Abbildung 4.6a zeigt die Umsetzung dieser Komponenten in der vorliegenden Arbeit. Der Slider, der dort im unteren Bereich dargestellt wird, ist in Abbildung 4.6b in einer 3D-Ansicht zu

[4] Mit Schlüssellänge 256 Bit, Salt-Länge 8-Bit und 1000 Iterationen benötigt das Nexus 4 ca. 120ms um einen Schlüssel zu erzeugen.

sehen. Am linken Ende befindet sich der neutrale Bereich am Fuß einer Rampe. Diese Rampe wird durch die Eingabekomponente dadurch simuliert, dass sich der Anfasser nur langsam aus dem neutralen Bereich bewegen lässt, sich während dieser Bewegung vergrößert und falls er auf der Rampe losgelassen wird, langsam wieder herunterrutscht.



(a) Verschiedene *Slider* mit neutraler Stellung. (b) *3D-Darstellung* des *Sliders* aus a.

Abbildung 4.6: Entwickelte UI-Komponenten.

Auf Codeebene wurde der in 4.6a unten dargestellte Slider durch Erweiterung der Klasse `View` realisiert, der Basisklasse für alle UI-Komponenten in Android. So umgesetzt, kann der Slider auf die selbe Art wie die bereits in der Android-API vorhandenen Komponenten in die Benutzeroberfläche eingebettet werden. Dabei kann der Slider zudem durch weitere Parameter wie Maximalwert und Minimalwert konfiguriert werden. Daher lassen sich die in dieser Arbeit implementierten UI-Komponenten bei Bedarf sehr einfach auch in zukünftig zu entwickelnden Android-Anwendungen verwenden. Zu diesem Zweck wurden zwei Demo-Anwendungen implementiert, welche die Verwendung dieser Komponenten bei der Entwicklung einer mobilen Anwendung zeigen.

4.4 Integration des Sensor-Frameworks in mobile Anwendungen

Soll in einer mobilen Anwendung, wie zum Beispiel der soeben gezeigten Fragebogenanwendung, auf externe Sensoren zugegriffen werden, lässt sich das in dieser Arbeit entwickelte Sensor-Framework als *Library Project* oder *Java Archive* einbinden. Daraufhin kann das Framework wie bereits in Listing 3.2 beispielhaft dargestellt verwendet

werden. Die benötigten Treiber werden im Rahmen der Initialisierung übergeben. Im Stil des *Singleton-Patterns* kann dann eine Referenz auf die Klasse `SensorFramework` erhalten werden. Danach werden über die in Listing 3.6 gezeigte Schnittstelle die von der Anwendung benötigten Sensoren anhand ihres Typs abgefragt. Beispielsweise fragt die Anwendung nach allen Sensoren, die einen Puls messen können. Anschließend wird über die durch das jeweilige *Interaction Pattern* des Sensors unterstützten Methoden mit dem Sensor interagiert.

4.4.1 Integration in die Fragebogenanwendung

In die in diesem Kapitel gezeigte Fragebogenanwendung könnte das Sensor-Framework auf Ebene der Benutzeroberfläche als eine Art zusätzliche Leiste eingebunden werden, die entweder nur während der vom Arzt ausgefüllten vorbereitenden Fragen oder während des gesamten Fragebogenablaufs sichtbar ist. In dieser werden die verfügbaren Sensoren entsprechend ihrer verschiedenen Klassifikationen mit jeweils passenden Symbolen angezeigt. Für die passende Visualisierung kann beispielsweise die Verbindungsart, der Sensortyp, der Name des Sensors, welche Daten der Sensor messen kann und das unterstützte Interaction Pattern dynamisch abgefragt werden. Das Interaction Pattern wirkt sich dabei auf das Verhalten und das Aussehen des jeweiligen Buttons zum Aktivieren des Sensors aus. Ein Blutzuckersensor, der über Bluetooth kommuniziert, kann zum Beispiel mit einem passenden Sensorsymbol, einem Bluetooth-Symbol, dem Namen und einem Button, der eine einmalige Messung signalisiert (z.B. durch das *Asynchronous Request Pattern* realisiert), angezeigt werden. Während der Fragebogen ausgefüllt wird, können so verschiedene Daten entsprechend der ausgewählten Sensoren aufgezeichnet werden. Das Sensor-Framework stellt eine Funktion zur Serialisierung der Messdaten bereit. Diese unterstützt aktuell eine XML-Serialisierung und kann daher benutzt werden, um die gemessenen Daten direkt in die bereits vorhandene XML-Datei des Fragebogens einzufügen.

4.4.2 Integration in weitere mobile Anwendungen

In einem weiteren studentischen Projekt wird das in dieser Arbeit entwickelte Sensor-Framework bereits verwendet. Es handelt sich dabei um eine Art elektronisches Patiententagebuch in Form einer Android-Anwendung. Ein Patient kann mit dieser Anwendung selbstständig seinen Blutzuckerspiegel messen und wird dabei durch ein prozessunterstütztes Assistenzsystem durch die dazu notwendigen Schritte geleitet[5]. Mithilfe des Sensor-Frameworks aus dieser Arbeit wird hierbei über Bluetooth ein externer Blutzuckersensor nach dem *Asynchronous Request Pattern* angesprochen.

Das Projekt ist noch nicht abgeschlossen, es gibt jedoch bereits erste positive Rückmeldungen bezüglich der Integration des Sensor-Frameworks. Das bedeutet, die Kommunikation mit dem Sensor und der Abruf der gemessenen Daten über das Sensor-Framework funktioniert und ist für den Anwendungsentwickler leicht zu integrieren und zu verwenden.

[5] vgl. hierzu die Ähnlichkeit zu Anwendungsfall **UC4** in Kapitel 1

5

Zusammenfassung und Ausblick

Abschließend werden die Ergebnisse dieser Arbeit zusammengefasst und einige über den Rahmen dieser Arbeit hinausgehende Thematiken in Form eines Ausblicks präsentiert.

5.1 Zusammenfassung

Es gibt viele mögliche Einsatzbereiche von mobilen Anwendungen mit Sensorunterstützung. Der Fokus dieser Arbeit richtet sich dabei hauptsächlich auf Anwendungen im medizinischen sowie psychologischen Bereich, da in diesen Bereichen bereits einige Arbeiten am Institut für Datenbanken und Informationssysteme der Universität Ulm durchgeführt wurden und voraussichtlich noch durchgeführt werden. Da jedoch keine einheitliche Schnittstelle zum Zugriff auf interne und externe Sensoren existiert, ist die

Entwicklung sensorunterstützter Anwendungen komplex, aufwändig und benötigt eine hohe Einarbeitungszeit in die Kommunikationsprotokolle der Sensoren. Insbesondere in Bezug auf externe Sensoren sieht sich ein Anwendungsentwickler mit unterschiedlichen Verbindungsarten, Datenformaten und Interaktionsmustern konfrontiert. Werden nur die internen Sensoren betrachtet, bietet die in dieser Arbeit betrachtete Android-Plattform jedoch auch hier keine einheitliche Schnittstelle an, da interne Sensoren wie Kamera, Mikrofon oder Infrarot-Sensor nicht zu den Sensoren gezählt werden und über separate Schnittstellen angesprochen werden müssen.

Um die Entwicklung von mobilen, sensorunterstützten Anwendungen in zukünftigen Projekten zu erleichtern, wurde in dieser Arbeit ein Sensor-Framework entwickelt. Dieses kann auf einfache Weise als Bibliothek in zukünftigen Android-Anwendungen integriert werden und ermöglicht diesen den Zugriff auf eine Vielzahl von verschiedenen internen und externen Sensoren über eine abstrakte, einheitliche Schnittstelle. Um dies zu realisieren, wurden in dieser Arbeit verschiedene Konzepte entworfen.

Das Prinzip der *Interaction Patterns* stellt ein erweiterbares Konzept dar, mit dem sich verschiedene Sensoren nach dem Interaktionsablauf kategorisieren lassen um sie über eine passende Schnittstelle verfügbar machen zu können. Ein *Typensystem* ermöglicht es einer Anwendung auf Sensoren über bestimmte Klassifizierungen zuzugreifen und so alle verfügbaren Sensoren abzurufen, die beispielsweise einem bestimmten Typ entsprechen oder Daten eines bestimmten Typs messen können. Außerdem wurde eine *Plug-and-Play-Architektur* entwickelt, die es möglich macht zusätzliche Sensoren durch Hinzufügen von Treibern an das Sensor-Framework anzubinden. Das Sensor-Framework vereinfacht dabei die Treiberentwicklung durch das Bereitstellen gemeinsam benötigter Funktionalität, sodass in einem Sensortreiber nur die jeweils sensorspezifische Logik neu geschrieben werden muss.

Als ein Beispiel einer mobilen Anwendung aus dem medizinisch-psychologischen Bereich, die sich für die zusätzliche Integration von Sensordaten eignet, wurde der Fragebogen *KINDEX Mum Screen*, ausgehend von einer früheren iOS-Umsetzung, auf Basis der Android-Plattform neu entwickelt. In diesem Zuge wurde das Design und die Navigation innerhalb der Applikation überarbeitet und neue Anforderungen wie beispielsweise die

verschlüsselte Datenspeicherung oder das Pausieren und spätere Fortsetzen eines Fragebogens einbezogen.

Das Sensor-Framework wird aktuell bereits in einem weiteren studentischen Projekt verwendet. Die Kommunikation mit dem dort verwendeten externen Blutzuckersensor über das Framework funktioniert bereits, sodass gezeigt werden konnte, dass das entwickelte Sensor-Framework über einen reinen Prototyp hinausgeht.

5.2 Ausblick

Das in dieser Arbeit entwickelte Sensor-Framework kann in zukünftigen Arbeiten eingesetzt werden, in denen Sensoren in eine mobile Anwendung integriert werden sollen. Es bieten sich jedoch auch einige mögliche Anknüpfungspunkte für zukünftige Arbeiten, sowie Erweiterungsmöglichkeiten am vorgestellten Framework selbst. Durch den modularen Aufbau bezüglich der Treiber, können weitere Sensoren mithilfe weiterer Treiber durch das Framework angesprochen werden. Sollen auch zusätzliche Verbindungsarten unterstützt werden, müssen hierfür jeweils neue Sensor-Manager implementiert werden. Unter den in Abschnitt 2.1.2 aufgeführten Kommunikationskanälen wären hier ein USB-Manager, aufgrund der bei mobilen Endgeräten verbreiteten Schnittstelle, oder ein ANT-Manager, wegen dem hierfür angebotenen Android-SDK, möglicherweise die naheliegendsten ersten Erweiterungsansätze. Durch das Hinzufügen weiterer Treiber und Sensor-Manager könnte überdies festgestellt werden, inwieweit das Sensor-Framework sowie die entwickelte abstrakte Schnittstelle zum Sensor-Framework generisch genug sind um sehr viele unterschiedliche Sensoren unterstützen zu können.

Um aus dem Typensystem des Sensor-Frameworks größeren Nutzen ziehen zu können, sollten weitere vordefinierte Sensortypen und Datentypen hinzugefügt werden. Denn dieses Typensystem und die von den Sensoren dynamisch abfragbaren Informationen stellen die Basis für eine mögliche generische Visualisierungskomponente der Sensoren und der von ihnen gelieferten Daten dar. Ähnlich wie bereits in Abschnitt 4.4.1 im Kontext der Fragebogenanwendung angedeutet könnte solch eine Visualisierungskomponente als Teil des Sensor-Frameworks in verschiedenen Anwendungen verwendet werden

und dort sowohl die verfügbaren Sensoren, als auch die Messdaten darstellen. Die verfügbaren Sensoren werden beispielsweise in einer Leiste angezeigt und sind dort aktivierbar, wobei die Darstellung der Sensor-Buttons von den vom Sensor abfragbaren Informationen, etwa zum Typ, dem Sensornamen und ähnlichen Daten, abhängt. Die von den Sensoren gelieferten Messdaten könnten durch entsprechende Diagramme visualisiert werden, wobei die über die Datentypen abfragbaren Informationen zur passenden Darstellung herangezogen werden können.

Weitere Ansatzpunkte bietet die Fragestellung, welche neuen Möglichkeiten sich durch Bluetooth 4.0 *(Bluetooth Low Energy)* ergeben. Dieser Standard wird erstmals durch Android Version 4.3 unterstützt, welche erst während dieser Arbeit veröffentlicht wurde. Speziell für Sensoren aus dem medizinischen Bereich gibt es hier ein eigenes Profil, das die Kommunikation mit entsprechenden Bluetooth-Sensoren vereinheitlichen kann.

Auch Android 4.4 eröffnet neue Möglichkeiten, die für diese Arbeit aufgrund der erst kürzlichen Veröffentlichung nicht mehr genutzt werden konnten. Mit dem neuen *Immersive Mode* ist die Ausführung und gleichzeitige Interaktion mit Anwendungen im Vollbildschirm-Modus möglich. Dieser ist für Fragebogenanwendungen nützlich, da durch Ausblenden der System-UI verhindert wird, dass ein Benutzer versehentlich die Anwendung verlässt oder Menüs des Betriebssystems öffnet.

Abbildungsverzeichnis

Literatur

[BDH05] Alistair Barros, Marlon Dumas und Arthur ter Hofstede. “Service Interaction Patterns”. In: *Business Process Management*. Hrsg. von Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati und Francisco Curbera. Bd. 3649. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 302–318. ISBN: 978-3-540-28238-9. DOI: 10.1007/11538394_20.

[Bds10] Bundesbeauftragter für den Datenschutz und die Informationsfreiheit, Hrsg. *Bundesdatenschutzgesetz (BDSG)*. Aktualisierte, nicht amtliche Fassung. 11. Juni 2010. URL: http://www.baden-wuerttemberg.datenschutz.de/gesetzeverordnungen/bundesdatenschutzgesetz/.

[Bee+08] Virginia Beecher, Deanna Bradshaw, Rima Dave, Mark Kennedy und Alex Prazma. *Oracle Fusion Middleware Developer’s Guide for Oracle SOA Suite*. Version 11.1.1. Oracle. Apr. 2008.

[Bru+12] Waylon Brunette, Rita Sodt, Rohit Chaudhri, Mayank Goel, Michael Falcone, Jaylen Van Orden und Gaetano Borriello. “Open data kit sensors: a sensor integration framework for android at the application-level”. In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. (Low Wood Bay, Lake District, UK). MobiSys ’12. New York, NY, USA: ACM, 2012, S. 351–364. ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307669.

[Cha+12] Rohit Chaudhri, Waylon Brunette, Mayank Goel, Rita Sodt, Jaylen VanOrden, Michael Falcone und Gaetano Borriello. “Open Data Kit Sensors: Mobile Data Collection with Wired and Wireless Sensors”. In: *Proceedings of the*

2nd ACM Symposium on Computing for Development. (Atlanta, Georgia). ACM DEV '12. New York, NY, USA: ACM, 2012, 9:1–9:10. ISBN: 978-1-4503-1262-2. DOI: 10.1145/2160601.2160614.

[Cho11a] Choicemmed. *Fingertip Pulse Oximeter Instruction Manual*. Version 1.0318T. MD300C318T. Beijing Choice Electronic Technology Co.,Ltd. 20. Mai 2011.

[Cho11b] Choicemmed. *Pulse Oximeter MD300C318T Bluetooth Communication Protocol*. Version 1.0.3. 5. Sep. 2011.

[Ele12] Nikolay Elenkov. *Using Password-based Encryption on Android*. 27. Apr. 2012. URL: http://nelenkov.blogspot.de/2012/04/using-password-based-encryption-on.html (besucht am 10. 11. 2013).

[HP13] Brian Hardy und Bill Phillips. *Android Programming. The Big Nerd Ranch Guide*. First edition. Addison-Wesley Professional, März 2013. ISBN: 978-0321804334.

[Kal00] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification*. Request for Comments 2898. Version 2.0. RSA Laboratories. IETF, Sep. 2000. URL: http://www.ietf.org/rfc/rfc2898.txt.

[Lan12] David Langer. "MEDo: Mobile Technik und Prozessmanagement zur Optimierung des Aufgabenmanagements im Kontext der klinischen Visite". Diplomarbeit. Universität Ulm, Apr. 2012. URL: http://dbis.eprints.uni-ulm.de/806/.

[Lie12] Martin Liebrecht. "Technische Konzeption und Realisierung einer mobilen Anwendung für den Konstanzer-Index zur Erhebung von psychosozialen Belastungen während der Schwangerschaft". Diplomarbeit. Universität Ulm, Sep. 2012. URL: http://dbis.eprints.uni-ulm.de/851/.

[LRZ10] Felix Xiaozhu Lin, Ahmad Rahmati und Lin Zhong. "Dandelion: a framework for transparently programming phone-centered wireless body sensor applications for health". In: *Wireless Health 2010*. (San Diego, California). WH '10. New York, NY, USA: ACM, 2010, S. 74–83. ISBN: 978-1-60558-989-3. DOI: 10.1145/1921081.1921091.

[Mai12] Fabian Maier. “Entwicklung eines mobilen und Service getriebenen Workflow-Clients zur Unterstützung von evaluierten Studien der klinischen Psychologie am Beispiel der AristaFlow BPM Suite und Android”. Bachelorarbeit. Universität Ulm, Nov. 2012. URL: http://dbis.eprints.uni-ulm.de/880/.

[Nie13] Hans Nienhaus. “Development of a Sensor Framework for the Determination of Vital Signs by the Example of a Fitness Application”. Bachelorarbeit. Universität Ulm, Jan. 2013. URL: http://dbis.eprints.uni-ulm.de/901/.

[Pry+12] Rüdiger Pryss, David Langer, Manfred Reichert und Alena Hallerbach. “Mobile Task Management for Medical Ward Rounds - The MEDo Approach”. In: *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*. LNBIP 132. Springer, Sep. 2012, S. 43–54. URL: http://dbis.eprints.uni-ulm.de/839/.

[Sch+13] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Hans Nienhaus und Manfred Reichert. “Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned”. In: *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*. Mai 2013, S. 509–518. URL: http://dbis.eprints.uni-ulm.de/918/.

[Sch13] Arnim Schindler. “Technische Konzeption und Realisierung des MACE-Tests mittels mobiler Technologie”. Bachelorarbeit. Universität Ulm, Jan. 2013. URL: http://dbis.eprints.uni-ulm.de/893/.

[And13a] Android Open Source Project. *Design - Android Developers*. 2013. URL: http://developer.android.com/design (besucht am 12. 11. 2013).

[And13b] Android Open Source Project. *Sensors Overview*. 2013. URL: http://developer.android.com/guide/topics/sensors/sensors_overview.html (besucht am 12. 11. 2013).

[Blu12] Bluetooth Special Interest Group. *Serial Port Profile*. Specification. Version 12. Draft. 24. Juli 2012. URL: https://www.bluetooth.org/

docman/handlers/DownloadDoc.ashx?doc_id=260866&vId=290097 (besucht am 09. 11. 2013).

[Blu13a] Bluetooth Special Interest Group. *Assigned Numbers*. Specification. 2013. URL: https://www.bluetooth.org/en-us/specification/assigned-numbers (besucht am 12. 11. 2013).

[Blu13b] Bluetooth Special Interest Group. *Bluetooth Basics. A Look at the Basics of Bluetooth Wireless Technology*. 2013. URL: http://www.bluetooth.com/Pages/Basics.aspx (besucht am 09. 11. 2013).

[Blu13c] Bluetooth Special Interest Group. *Tech. Overview - Serial Port Profile*. 2013. URL: http://developer.bluetooth.org/TechnologyOverview/Pages/SPP.aspx (besucht am 09. 11. 2013).

[Def13] Defuse Security. *Salted Password Hashing - Doing it Right*. 2013. URL: https://crackstation.net/hashing-security.htm (besucht am 10. 11. 2013).

[Fit13] Fitbit Inc. *fitbit*. 2013. URL: http://www.fitbit.com (besucht am 09. 11. 2013).

[Goo13] Google Inc. *Google I/O 2013*. 2013. URL: https://developers.google.com/events/io (besucht am 09. 11. 2013).

[Int94] International Telecommunication Union. *Open Systems Interconnection - Model and Notation. ITU-T X.200*. Standard. Version 4. Juli 1994. URL: http://handle.itu.int/11.1002/1000/2820.

[Nik13] Nike Inc. *Nike+*. 2013. URL: http://nikeplus.nike.com/plus/ (besucht am 09. 11. 2013).

[Ope13a] Open Data Kit. *Open Data Kit. magnifying human resources through technology*. 2013. URL: http://opendatakit.org (besucht am 11. 11. 2013).

[Ope13b] Open Data Kit. *Open Data Kit Sensors Framework*. 2013. URL: http://opendatakit.org/use/sensors/ (besucht am 09. 11. 2013).

[Pal13] Palo Pacific Technology Pty Ltd. *Bluetooth Tutorial - Profiles*. 2013. URL: http://www.palowireless.com/infotooth/tutorial/profiles.asp (besucht am 09. 11. 2013).

[Ste13] Stephen Roberts. *ListDemo. Simple list implementation in the style of google building blocks*. 29. März 2013. URL: https://github.com/steprobe/ListDemo (besucht am 07. 11. 2013).

[USB11] USB Implementers Forum, Inc. *USB Class Codes*. 7. Dez. 2011. URL: http://www.usb.org/developers/defined_class (besucht am 10. 11. 2013).

[ron13] ronmamo. *reflections. java runtime metadata analysis*. Version 0.9.9-RC1. 2013. URL: http://code.google.com/p/reflections/ (besucht am 12. 11. 2013).

[run13] runtastic GmbH. *runtastic. makes sports funtastic*. 2013. URL: http://www.runtastic.com (besucht am 09. 11. 2013).

Name: Patrick Zeller Matrikelnummer: 644203

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Patrick Zeller