ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption eines Event-basierten Sensor-Frameworks zur Datenerhebung auf mobilen Endgeräten

Diplomarbeit an der Universität Ulm

Vorgelegt von:
Artur Jabs
artur.jabs@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Dr. Vera Künzle

Betreuer:
Johannes Schobel

2015

Fassung 31. Juli 2015

Satz: PDF-LaTeX 2ε

Kurzfassung

Heutzutage existieren eine Vielzahl von unterschiedlichen Sensoren, die beispielsweise über mobile Endgeräte, wie Smartphones oder Tablets, angesprochen werden können. Sie ermöglichen so neue Anwendungen, zum Beispiel im Gesundheits-, Fitness- oder Medizin-Bereich. So können beispielsweise die Endverbraucher selbst ihre eigenen Vitaldaten über einen Zeitverlauf überwachen, oder die Heizung im Haus fernsteuern.

Bei der Implementierung solcher sensorunterstützten Anwendungen werden Entwickler meist mit der Anbindung und der Ansteuerung der einzelnen Sensoren konfrontiert. Dies wird dadurch erschwert, dass es oft keine einheitliche Schnittstelle zur Anbindung dieser Sensoren gibt. Dadurch erhöht sich die Entwicklungsdauer und die Komplexität einer sensorunterstützten Anwendung deutlich.

Um diesen Problemen zu begegnen wurde in dieser Arbeit ein Sensor-Framework entwickelt, welches die Anbindung und Ansteuerung der Sensoren unterstützt. Dieses bietet dazu sowohl für den Anwendungsentwickler als auch für den Sensortreiberentwickler einheitliche Schnittstellen. Im Rahmen dieser Arbeit wird die Konzeption des Sensor-Frameworks geschildert und einige Implementierungsdetails des Sensor-Frameworks erläutert.

Inhaltsverzeichnis

1

Einführung

Das Voranschreiten in der Entwicklung von mobilen Endgeräten in den letzten Jahren hat zu einer stetigen Leistungssteigerung dieser Geräte geführt. Auf Grund dessen konnten in den letzten Jahren immer mehr Sensoren in mobile Endgeräte integriert werden (beispielsweise GPS- und Accelerometer-Sensor). Zudem sind auch verschiedene Betriebssysteme für die mobilen Endgeräte entwickelt worden. Darunter gilt Android als das am weitesten verbreitete Betriebssystem für mobile Endgeräte [Sta15].

Die Entwicklung dieses Technologiezweigs wird im kommerziellen Bereich gut genutzt, in dem verschiedenste Anwendungen implementiert werden, darunter auch Fitness- und Gesundheits-Anwendungen. Diese sind im Zusammenhang mit der Selbstvermessung aufgekommen, bei der die Endverbraucher selbst ihre Vitaldaten überwachen [ges14]. Dazu wurden auch verschiedene neue externe Sensoren entwickelt, beispielsweise Smart Body Analyzer von Withings [Wit15] und Flex von Fitbit [Fit15]. Dabei versuchen Withings und Fitbit, sich in diesem Bereich zu etablieren, indem sie eigene Sensoren, beispielsweise zur Schrittzählung oder zum Puls messen, anbieten. Meist bieten diese Unternehmen zusätzlich eigene Anwendungen an, um mit den von ihnen entwickelten Sensoren zu kommunizieren und zu interagieren. Andere Sensoren können jedoch oft nicht durch diese Anwendung angesprochen werden.

Weiterhin wurden auch verschiedene Anwendungen im medizinischen Bereich entwickelt, um die Erstellung einer Diagnose verbessern zu können. Dazu wurden verschiedene tragbare Sensoren entwickelt, durch die vor Ort bestimmte Vitaldaten gemessen werden können. Um die Aufnahme der Daten und die Analyse zu erleichtern erfordert es eine Anbindung von Sensoren. Eine Anwendung, die diesen Prozess unterstützt, ist beispielsweise VitaBit, eine Plattform zur Unterstützung der ambulanten Pflege. Dabei

liegt ein besonderes Augenmerk, auf dem Mobile Client, welcher die Kommunikation mit verschiedenen Sensoren unterstützt und aufbaut [WIB11].

Weiter bieten mobile Endgeräte heute eine Vielzahl an internen Sensoren an. So kann beispielsweise durch das Mikrofon der Geräuschpegel in der Umgebung gemessen, oder über GPS die aktuelle Position bestimmt werden. Dies wird in der LärmApp eingesetzt, welche in Kooperation mit Hals-Nasen-Ohren Ärzten entwickelt wurde [HNO]. Weiterhin bietet ein mobiles Endgerät verschiedene Anbindungsmöglichkeiten für externe Sensoren an. Dabei sind externe Sensoren Zusatzgeräte, die über ein Kabel oder eine kabellose Verbindung, wie beispielsweise Bluetooth, mit einem mobilen Endgerät verbunden werden können.

Bei der Entwicklung von sensorunterstützten Anwendungen, stoßen die Anwendungsentwickler meist auf zeitraubende und langandauernde Hindernisse, die die Entwicklung der eigentlichen Logik der Anwendung oder des Sensors nicht betreffen. Dazu zählt beispielsweise die Anbindung des Sensors. Werden verschiedene Sensoren über denselben Kommunikationskanal angebunden, wird meist ähnlicher Code genutzt. Das führt zu redundantem Code, der unnötig den eigentlichen Quellcode vergrößert und schnell zu Inkonsistenzen führen kann.

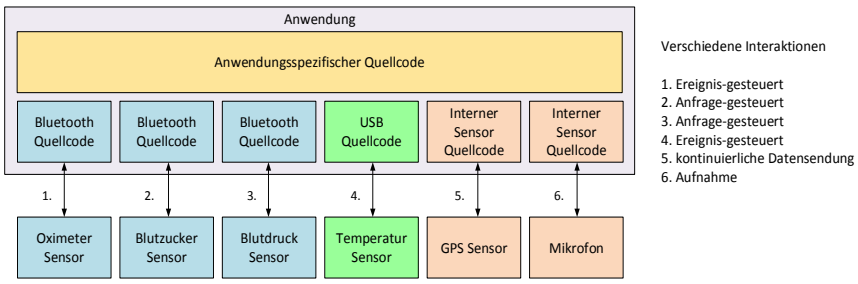

Abbildung 1.1: Schematische Darstellung der Komplexität bei sensorunterstützten Anwendungen

In Abbildung 1.1 wird der bisher geschilderte Sachverhalt und die daraus resultierende Komplexität schematisch dargestellt. Werden allerdings verschiedene Sensoren über unterschiedliche Kommunikationsarten angebunden, wird die Entwicklung der Anwendung

um einiges komplexer, da für jeden einzelnen Sensor entsprechende Kommunikationskanäle betrachtet und bereitgestellt werden müssen. Zwar bietet die Bluetooth-Special Interest Group bereits einige Bluetooth-Profile an, diese betrachten jedoch nur die Bluetooth Anbindung und sind meist gerätespezifisch anzuordnen. Zusätzlich zu den Kommunikationsarten müssen die Sensoren auch anhand ihrer Interaktionsmuster untersucht werden, da diese auch meist unterschiedlich sind.

Daraus folgt, dass eine Möglichkeit entwickelt werden muss, durch die die Realisierung von sensorunterstützten Anwendungen erleichtert und die Anbindung und Interaktion mit dem eigentlichen Sensor übernimmt. Dazu wird in dieser Arbeit ein Sensor-Framework entwickelt, das eben solche Funktionen bereitstellt.

1.1 Zielsetzung

Im Rahmen dieser Arbeit wird ein ereignisgesteuertes Sensor-Framework entwickelt, das einfach in bereits bestehende und zukünftige mobile Anwendungen integrierbar sein soll. Darüber hinaus soll es die Entwicklung solcher Anwendungen erleichtern. Das Sensor-Framework bildet eine Art *Zwischenschicht* zwischen der eigentlichen Anwendung und den zu verwendenden Sensoren (siehe Abbildung 1.2). Weiterhin soll eine Möglichkeit gefunden werden, das Sensor-Framework möglichst modular zu entwickeln, damit die einzelnen Bestandteile erweiterbar oder einfach austauschbar sind. Außerdem soll es möglich sein, dass sowohl innerhalb als auch außerhalb des Sensor-Frameworks über Ereignisse kommuniziert werden kann.

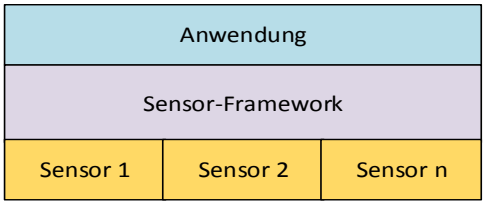

Abbildung 1.2: Grobe Darstellung der Einordnung des Sensor-Frameworks

Weiterhin sollen verschiedene Schnittstellen zur Interaktion mit möglichst vielen Sensoren entwickelt werden. Außerdem soll das Sensor-Framework unterschiedliche Sensoren anbinden können. Dazu muss eine Schnittstelle entwickelt werden, durch die die verschiedenen Kommunikationsarten von der eigentlichen Anwendungs- und Sensorlogik gekapselt werden. Dabei muss darauf geachtet werden, dass das Sensor-Framework selbst keine Treiber enthält, sondern lediglich die Schnittstellen zur Anbindung und Interaktion der Sensoren anbietet. Darauf aufbauend soll eine Funktionalität entwickelt werden, um neue Sensortreiber zur Anwendung hinzuzufügen und im Sensor-Framework zu integrieren.

Zudem soll ein Mechanismus zur Klassifizierung der Sensoren entwickelt werden. So soll es möglich sein, nach bestimmten Sensoren zu filtern und alle aktuell verfügbaren Sensoren aufzulisten. Anschließend soll ein Ansatz gefunden werden, durch den die Sensoren zur Laufzeit der Anwendung, konfiguriert werden können. Abschließend soll ein Mechanismus erarbeitet werden, durch das eine Visualisierung von Sensor Daten ermöglicht wird.

1.2 Aufbau der Arbeit

Der weitere Aufbau der Arbeit ist wie folgt: In Kapitel 2 werden verschiedene Grundlagen, die für das weitere Verständnis der Arbeit wichtig sind, erläutert. Danach werden in Kapitel 3 verwandte Arbeiten diskutiert, die verschiedene Ansätze verfolgen und dem hier zu entwickelnden Sensor-Framework gegenüber gestellt werden. Daraufhin werden in Kapitel 4 die Anforderungen für das zu entwickelnde Sensor-Framework zusammengefasst. Aufbauend auf den Anforderungen wird in Kapitel 5 das Sensor-Framework aus konzeptioneller Sicht betrachtet und die Architektur beschrieben. Danach werden in Kapitel 6 einige Implementierungsdetails des entwickelten Sensor-Frameworks besprochen. In Kapitel 7 wird auf die Implementierung eines Sensortreibers und die mögliche Integration in eine Anwendung eingegangen. Abschließend wird in Kapitel 8 eine Zusammenfassung über die Ergebnisse, sowie ein Ausblick über mögliche Erweiterungen diskutiert.

2

Grundlagen

In diesem Kapitel werden einige Grundlagen, die für das weitere Verständnis dieser Arbeit relevant sind, genauer erläutert. Dazu wird zunächst auf die Event-Driven Architecture eingegangen. Danach wird die Funktionsweise von Bluetooth und USB besprochen, die zur Anbindung von externen Sensoren genutzt werden können. Im Anschluss wird ein grober Überblick über verschiedene externe und interne Sensoren gegeben. Im Verlauf der Arbeit wird von Anwendungsentwicklern und Sensortreiberentwicklern gesprochen. Dabei ist der Anwendungsentwickler der Implementierer einer sensorunterstützten Anwendung und der Sensortreiberentwickler ist zuständig für die Implementierung der Sensorlogik.

2.1 Event-Driven Architecture

Die Event-Driven Architecture (EDA) ist ein Konzept, welches im Grunde auf die Ereignisverarbeitung ausgerichtet ist. Die Aufgaben, die eine Ereignisverarbeitung mit sich bringt sind die Erzeugung, Weiterleitung und Bearbeitung eines Events [Pat03]. Ein *Event* kann dabei als eine Statusänderung betrachtet werden. Das kann beispielsweise eine Temperaturveränderung oder ein Empfang einer Nachricht sein.

Dabei ist die Ereignisverarbeitung ein *One Way Trip*. Darunter versteht man, dass ein auftretendes Event sofort weitergeleitet wird um auf ein weiteres Event reagieren zu können. Dieses Konzept ermöglicht eine schnelle Reaktion auf Statusänderungen und kann prinzipiell auftretende Events in Echtzeit bearbeiten [Hug09]. Eine EDA wird meist in Geschäftsprozessen eingesetzt um auf bestimmte *Events* sofort reagieren zu können.

Dabei kann beispielsweise durch ein *Event* angezeigt werden ob der Lagerbestand eines Produktionsguts fast ausgeschöpft ist.

2.1.1 Definition einer EDA

Wie schon erwähnt, dient eine Event-Driven Architecture der Ereignisverteilung, Ereignisverarbeitung und Ereignisüberwachung. Dazu hat eine EDA im einfachsten Fall einen *Event Producer*, einen *Event Consumer*, einen *Event Processor* und einen *Event Channel* (oft auch Message Backbone genannt). Die wichtigen Begriffe werden nachfolgend definiert.

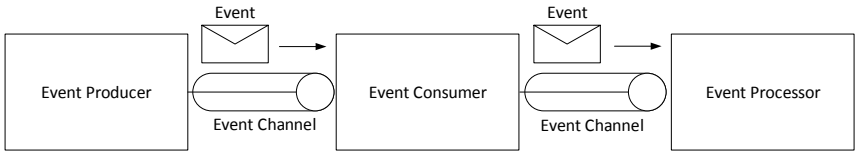

Abbildung 2.1: Bestandteile einer EDA

Event: Events sind ein wesentlicher Bestandteil in einer EDA. Ein Event kann dabei als Nachricht angesehen werden, die eine Reaktion erfordert. In Abbildung 2.1 ist das Event eine Nachricht, die erzeugt wurde.

Event Producer: Der Event Producer, der in der Abbildung 2.1 ganz links dargestellt wird, erzeugt das Ereignis, welches verarbeitet oder behandelt werden soll.

Event Consumer: Ein Event Consumer, erhält als erstes das Event, aus der Abbildung 2.1 ersichtlich. Dazu hört dieser auf ein bestimmtes Event und reagiert darauf. In den meisten Fällen übergibt der Event Consumer das erhaltene Event an einen Event Processor.

Event Processor: Um ein Event zu verarbeiten benötigt es einen Event Processor (oder auch Event Handler genannt). Dieser erhält das Event von dem Event Consumer durch den Event Channel, wie in Abbildung 2.1 ersichtlich. Der Event Processor

betrachtet das Event und stellt fest, was damit geschehen soll. Oft sind Event Consumer und Event Processor identisch [Hug09, Bre06].

Event Channel: Ein Event Channel ist wie in Abbildung 2.1 dargestellt die Verknüpfung zwischen den einzelnen Komponenten einer EDA. Er dient zur Übertragung des Events an den Event Consumer bzw. Event Processor [Hug09, Bre06].

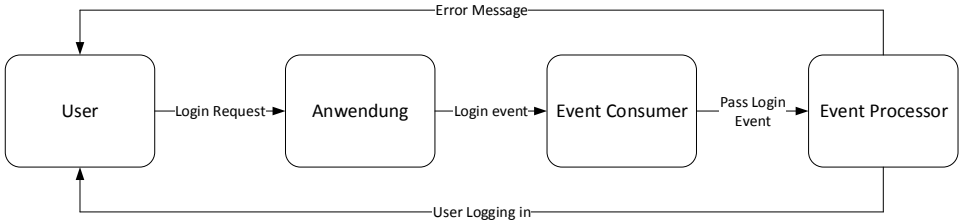

Abbildung 2.2: Einfaches Beispiel zur Ereignisverarbeitung

Eine Ereignisverarbeitung wird nun anhand Abb. 2.2 dargestellt. Dabei tätigt der Benutzer ein Login Request, dazu gibt er seine Login Daten ein und sendet die mittels Klick auf den Login Button ab. Die Anwendung erzeugt daraufhin das Login Event und übergibt es an den Event Consumer. Der Event Consumer reicht dieses Event weiter an den Event Processor. Der Event Processor überprüft daraufhin die Anmeldedaten des Benutzers und startet den Login Vorgang. Falls die Daten nicht korrekt waren, wird dem Benutzer durch eine Error Message dies mitgeteilt.

2.1.2 Publish-Subscribe Pattern

Um aufzuzeigen wie eine Event-Driven Architecture durch Publish-Subscribe erweitert werden kann, wird nun an dieser Stelle das Publish-Subscribe Pattern erläutert. Bei diesem Pattern handelt es sich um ein Designkonzept welches auf Nachrichten beruht. Der Publisher erzeugt Nachrichten von einem bestimmten Typ und sendet diese an den Mediator. Dieser leitet diese wiederum an alle Subscriber weiter, die sich dafür registriert

haben. Der Subscriber andererseits registriert sich für eine Art von Nachrichten und wartet dann auf die Zustellung dieser Nachrichten über den Mediator [Pat03].

Das Publish-Subscribe Pattern ermöglicht es den einzelnen Parteien völlig unabhängig voneinander zu arbeiten. Somit ist es möglich, Services zu entwickeln die wiederverwertbar und austauschbar sind. Die Kommunikation zwischen den einzelnen Komponenten wird durch Abbildung 2.3 dargestellt.

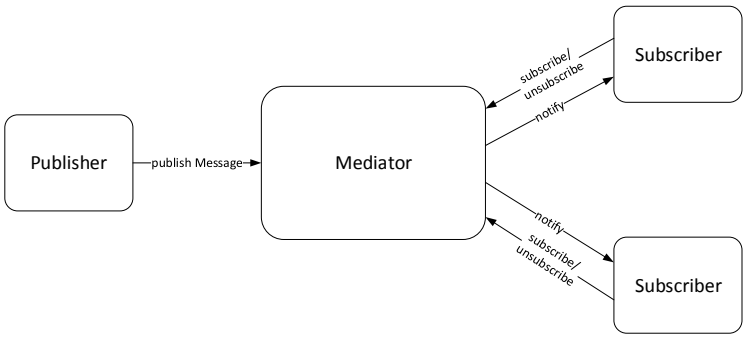

Abbildung 2.3: Einfaches Publish-Subscribe System [Pat03]

2.2 Bluetooth

Bluetooth gilt als eine der gängigen Verbindungsarten für mobile Endgeräte, um mit externen Geräten (wie beispielsweise Sensoren) zu kommunizieren. Dazu wird in diesem Kapitel der Bluetooth-Protokollstack näher betrachtet und auf die einzelnen Bestandteile näher eingegangen, die für diese Arbeit relevant sind. Die Verwaltung der Bluetooth-Standards ist die Aufgabe der Bluetooth Special Interest Group, die 1998 von Ericson, IBM, Intel, Nokia und Toshiba gegründet wurde [Blu15a].

Bluetooth ermöglicht eine kabellose Kommunikation zwischen verschiedenen Geräten auf kurzer Distanz. Diese Technologie wird nicht nur im betrieblichen und medizinischen Bereich sondern auch im privaten Bereich genutzt. Heutzutage enthält jedes gängige mobile Endgerät diese Technologie. Aufgrund von geringem Stromverbrauch und günstigen Hardwarekomponenten [Sau13].

2.2.1 Bluetooth-Protokollstack

Da heutzutage sehr viele verschiedene Hersteller an Bluetooth-fähigen Geräten arbeiten, ist es wichtig ein einheitliches Konzept zu definieren, welches eine Kommunikation zwischen den verschiedenen Endgeräten ermöglicht. Aufgrund dessen wurde ein Bluetooth-Standard definiert. Zum jetzigen Zeitpunkt existieren unterschiedliche Versionen, die auch mit Einschränkungen abwärtskompatibel sind. So kann zum Beispiel ein Gerät mit Bluetooth 2.1 auch mit einem Gerät, das Bluetooth 3.0 unterstützt, kommunizieren [Sau13].

Der grundlegende Aufbau des Bluetooth-Protokollstacks ist in Abbildung 2.4 rechts dargestellt. Dabei ist zu erkennen, dass die Schichten Physical Radio, Baseband, Link Manager Logical Link Control Adaptation Protocol (L2CAP), Audio und Control auf die Physische und Data Link Schicht des ISO Models abgebildet werden können. Weiterhin gehören Physical Radio, Baseband, Link Manager und L2CAP zu den *Bluetooth Core System Protocols*. Diese werden bei jeder Kommunikation über Bluetooth zwischen den Geräten genutzt [Blu15b].

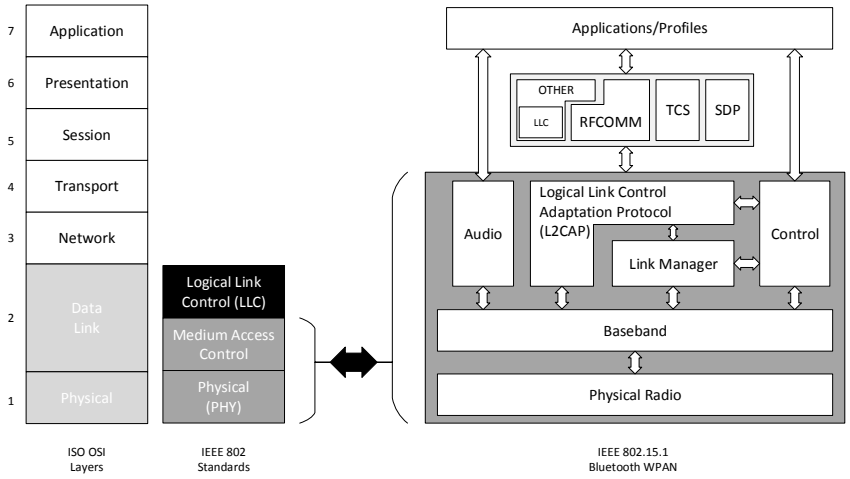

Abbildung 2.4: ISO-Schichtenmodel Referenz zum Bluetooth-Protokollstack [IEE02]

Des Weiteren beinhaltet der Bluetooth-Protokollstack nicht nur Bluetooth eigene Protokolle, wie zum Beispiel Link Manager Protocol oder L2CAP, sondern auch ande-

re unabhängig vom Bluetooth nutzbare Protokolle. Diese sind in Abbildung 2.4 als *Other* gekennzeichnet. Ein Beispiel dafür wäre das Object Exchange Protocol (OBEX). OBEX ist ein Transportprotokoll, welches Datenobjekte definiert, diese können zwischen zwei Geräten ausgetauscht werden [Sau13]. Beim Entwurf des gesamten Bluetooth-Protokollstacks wurde darauf geachtet, dass jede Schicht wiederverwendbar ist um die eingangs erwähnte Abwärtskompatibilität zu sichern [IEE02].

Die Physical Radio Schicht des Protokollstacks ist verantwortlich für die Übertragung der Bits zwischen zwei Bluetooth fähigen Geräten. Die Schicht übernimmt die Transformation der eintreffenden Funkwellen zum Bitstrom und umgekehrt. Darüber liegend ist die Baseband Schicht, die ebenfalls Teil der vom ISO Model spezifizierten Physischen Schicht ist. Die Baseband Schicht übernimmt die Aufgaben eines Data Link Protokolls. Dazu gehört das Framing von Datenpaketen. Zur Paketdatenübertragung verwendet das Baseband ein Asynchronous Connectionless Paket (ACL). Das besteht aus 68 bis 72 Bits Access Code, einem 18 Bit Header und 0 bis 2744 Bits an eigentlichen Nutzdaten [Sau13].

An das Baseband knüpft zum einen der Link Manager, der hat die Aufgabe eine Verknüpfung herzustellen und diese zu überwachen. Die Signale werden durch den Link Manager interpretiert und gefiltert bevor diese an die höhere Schicht weitergereicht werden [IEE02].

Neben dem Link Manager liegt das Logical Link Control and Adaptation Protocol auf dieses wird im nächsten Abschnitt eingegangen. Darauf aufbauend liegt das Radio Frequency Communication Protocol (RFCOMM) und auf selber Ebene das Service Discovery Protocol. Das Service Discovery Protocol, das auch zu den *Bluetooth Core System Protocols* gehört, wird in Kapitel 2.2.4 im Rahmen der Universally Unique Identifier (UUID) betrachtet.

2.2.2 Logical Link Control and Adaptation Protocol

Logical Link Control and Adaptation Protocol (L2CAP) befindet sich über der Baseband Schicht. Die L2CAP Schicht unterstützt die darüber liegenden Transportprotokolle, indem

es das passende Transportprotokoll zur Datenübertragung anspricht. Dazu fügt das L2CAP Protokoll Zusatzinformationen in den ACL Paket Header ein. Dadurch ist es möglich das Transportprotokoll, welches genutzt werden soll, eindeutig zu identifizieren. Falls das zu sendende Paket, größer als ein ACL Paket ist (maximal 2834 Bits), kann die L2CAP Schicht dieses Paket automatisch unterteilen. Nach der Übertragung werden diese dann auf dem anderen Gerät in richtiger Reihenfolge wieder zusammengesetzt. Um herauszufinden an welche Position welches Paket gehört wird die nötige Information dazu im Header des ACL Pakets eingefügt.

Darüber hinaus stellt die L2CAP Schicht logische Kanäle zur Verfügung, die L2CAP Channels genannt werden. Diese haben zur eindeutigen Identifizierung einen Channel Identifier (CID). Dies ist auch notwendig, da es mehrere L2CAP Channels geben kann, die alle in den meisten Fällen am RFCOMM Protokoll angedockt sind [Sau13, Blu15c].

2.2.3 Radio Frequency Communication Protocol

Die Radio Frequency Communication Protocol Schicht baut auf der L2CAP Schicht auf. Die RFCOMM Schicht ist ein Transportprotokoll, durch das eine Serielle Schnittstelle, auf Basis des RS-232 Standards, emuliert wird. Dadurch können ältere Anwendungen oder Hardwarekomponenten, die auf serieller Datenübertragung beruhen, direkt durch die RFCOMM Protokoll Schicht an die L2CAP Schicht angedockt werden, ohne die Daten transformieren zu müssen. Das RFCOMM Protokoll unterstützt bis zu 60 gleichzeitige Verbindungen zwischen zwei Geräten. Die Anzahl der benutzbaren Verbindungen von einem Bluetooth-Gerät hängt von der Implementierung des Geräts ab. Des Weiteren gibt es verschiedene Datenflusskontrollmechanismen im RFCOMM Protokoll, die die eintreffenden Daten kontrollieren [Blu15f].

2.2.4 Universally Unique Identifier

Der Universally Unique Identifier (UUID) wird beim Auffinden von Diensten benötigt. Im Bluetooth-Protokollstack wird dazu das Service Discovery Protocol (SDP) genutzt. UUID ist ein standardisiertes 128-bit Format für String IDs, die benutzt werden um

Services eines Bluetooth Gerätes zu identifizieren. Es gibt dazu verschiedene, schon vordefinierte UUIDs. Das Service Discovery Protocol ist ein einfaches Protokoll mit geringen Anforderungen an die darunter liegende Transportschicht. Es benutzt ein einfaches Request/Response Model in dem jede Transaktion aus einer Request Protocol Data Unit (PDU) und einer Response PDU besteht. Das SDP dient zur Abfrage von den Informationen von allen möglichen Services, die das Bluetooth Device anbietet [Blu15e, Sau13].

2.2.5 Pairing

Um das Pairing (Verbinden) zwischen zwei Bluetooth-fähigen Geräten zu bewerkstelligen, muss eines der beiden Geräte den Discovery Prozess initialisieren. Dabei wird dann die nähere Umgebung des Gerätes gescannt um andere, sich in diesem Bereich befindende und sichtbare Geräte zu identifizieren. Danach kann das Pairing initialisiert werden. Dazu gibt es die folgenden Möglichkeiten: Bei der ersten wird durch den Benutzer bei beiden Geräten eine PIN eingegeben über diese das Pairing vollzogen wird. Bei der zweiten tauschen die Geräte untereinander die PINs aus, unter der Voraussetzung, dass der Benutzer auch ein Pairing mit dem gefundenen Gerät vollziehen möchte [Sau13].

2.3 Universal Serial Bus

In diesem Kapitel wird nun der Universal Serial Bus (USB) erläutert, der neben Bluetooth eine weitere gängige Möglichkeit ist um verschiedene Geräte miteinander zu verbinden. Seit USB im Jahre 1996 eingeführt wurde, hat sich dieser Standard immer weiter verbreitet und gehört heute zu dem am weitesten verbreiteten Standards. Die USB-Standards sind abwärtskompatibel, da sie auf derselben Hardware arbeiten. Um zum Beispiel die Neuerung in Version 3.0 zu unterstützen sind neue Leitungen eingebunden worden. Es wurde aber darauf geachtet, eine Abwärtskompatibilität zu gewährleisten, beispielsweise indem die alten Leitungen beibehalten werden.

2.3.1 USB Connection

Wenn ein USB Gerät angeschlossen wird, wird ein Prozess (Enumeration Process) gestartet. Dadurch kann der Host mit dem Gerät kommunizieren und die nötigen Informationen zum Installieren des passenden Treibers erfragen. Die Enumeration wird durch ein Reset Signal an das Gerät gestartet. Nach dem Reset liest der Host die Geräte Informationen aus und weist diesem Gerät eine eindeutige 7-bit Adresse zu. Falls das Gerät durch den Host unterstützt wird, wird der benötigte Treiber für das Gerät geladen und installiert. Um dies zu prüfen wird vom Gerät eine VendorID erfragt, die zur eindeutigen Identifizierung dessen dient. Falls diese bekannt ist, kann das Betriebssystem diesen Treiber stellen. Wenn diese jedoch nicht bekannt ist, muss das Gerät dazu den benötigten Treiber selber mitliefern. Danach wird der Status des Gerätes als konfiguriert eingestuft [Jan15].

2.3.2 USB Datenübertragung

Jede Datenübertragung zwischen den einzelnen Komponenten besteht aus verschiedenen Transaktionen, die wiederum aus Paketen besteht. Der Host hat während der Übertragung die Aufgabe den Datenfluss zu steuern. Dabei sind die Device Endpoints und Pipes relevant, diese werden nun erläutert.

Device Endpoint: Alle Datenübertragungen werden zwischen Host und einem Device Endpoint abgewickelt. Ein Endpoint ist ein Puffer, der die zu übertragenden Daten, speichert. Der Host selber hat zwar auch einen Puffer für bereits gesendete Daten, ist jedoch der, der die Kommunikation zu einem Device Endpoint initiiert. In der Spezifikation wird ein Device Endpoint als eine eindeutig adressierbare Einheit eines USB Geräts bezeichnet, also die Quelle oder der Empfänger von Informationen in einer Kommunikation zwischen dem Host und einem Gerät. Somit ist ein Endpoint entweder ein Eingang oder Ausgang von Daten [Jan15].

Pipes: Damit eine Datenübertragung überhaupt beginnen kann muss erst eine Pipe zwischen dem Host und dem Gerät bereitgestellt werden. Dabei ist eine Pipe kein physisches Medium sondern eine virtuelle Verbindung zwischen dem Host und

dem Device Endpoint. Eine Pipe wird sofort nach der Verbindungsaufnahme mit dem Gerät erstellt. Sobald dieses Gerät wieder entkoppelt wird, werden alle nicht mehr benötigten Pipes automatisch geschlossen.

Datenübertragungsarten: USB unterstützt vier verschiedene Datenübertragungsarten. Die erste Übertragungsart ist Control Transfer. Dies ist die einzige Übertragungsart, die Funktionen besitzt, die in der USB Spezifikation definiert wurden. Durch sie kann der Host zum Beispiel die für die Anbindung eines Gerätes nötigen Informationen abfragen. Eine weitere Übertragungsart ist Bulk, um Daten sehr schnell übertragen zu können. Wird der Bus allerdings schon benutzt, werden Bulk-Übertragungen hinten angestellt. Interrupts sind eine weitere Übertragungsart. Diese wird meist durch Low-Speed Geräte (zum Beispiel Mäuse oder Tastaturen) genutzt, die sonst nicht ihre Übertragung bewerkstelligen könnten. Die letzte Übertragungsart ist Isochronous. Diese garantiert eine Datenübertragung in einer bestimmten Zeit, bietet jedoch keine Fehlerbehandlung an.

Wenn eine Anwendung des Hosts mit dem Gerät kommunizieren möchte, dann startet es eine Übertragung. In der Spezifikation wird eine Übertragung als ein Prozess bezeichnet, in dem ein Kommunikationsrequest erstellt und übertragen wird. Dieser Request wird vom Betriebssystem an den zuständigen Gerätetreiber weitergeleitet, welcher diesen zu weiteren System-Level Treibern und schließlich zum Host Controller weiterleitet. Der Host Controller startet dann die Übertragung auf dem Bus [Jan15].

2.3.3 USB Standards

Mit der Weiterentwicklung von USB kamen immer weitere Standards auf den Markt. Durch jede neue Standarddefinition wurde die Datenübertragungsgeschwindigkeit erhöht. Dabei wurde aber dennoch auf eine Abwärtskompatibilität zu den älteren Standards geachtet.

USB 1.0: USB 1.0 wurde im Jahre 1996 eingeführt. Dadurch wurden Daten mit bis zu 12 Mb/s übertragen, und dieser Standard konnte bis zu 127 Geräte unterstützen.

USB 1.1: Durch die Einführung von USB 1.1 im Jahre 1998 wurden hauptsächlich Fehler im Standard behoben. Dazu wurde die Möglichkeit eingeführt, zwei verschiedene Speed Modi zu verwenden.

USB 2.0: Durch die Einführung von USB 2.0 im April 2000 wurde die Datenübertragungsrate auf bis zu 480 Mbit/s erhöht.

USB 3.0: USB 3.0 wurde im August 2008 eingeführt und brachte eine weitere Steigerung der Datenübertragungsrate auf 4.8 Gbit/s.

USB 3.1: USB 3.1 (auch als SuperSpeed+ USB bezeichnet) erhöht die Datenübertragungsrate gegenüber dem 3.0 Standard auf 10 Gbit/s. Es bietet eine verbesserte Datencodierung für eine effizientere Übertragung. Zudem wurde der Stromverbrauch des angeschlossenen Geräts verringert, indem es kein Polling mehr benutzt und somit weniger Strom benötigt [USB].

2.4 Sensoren

In diesem Kapitel wird nun ein Überblick über verschiedene Sensoren geliefert. Ein Sensor ist eine mechanische oder elektronische Komponente, die dazu genutzt wird um elektronische oder optische Signale zu erfassen und auf diese zu reagieren. Dabei wandelt der Sensor eine physikalische Größe in eine messbare Einheit um. Eine physikalische Größe kann dabei zum Beispiel die Temperatur, der Blutdruck, der Sauerstoffgehalt oder die Geschwindigkeit sein. Dazu wird auch betrachtet, wo ein Sensor eingesetzt werden kann und in welche Kategorien die Sensoren unterteilt werden können [Snu14].

2.4.1 Sensor Kategorien

Es gibt heutzutage sehr viele verschiedene Sensoren, die nach ihrem Einsatz- und Verwendungszweck unterteilt werden können. Zudem unterscheidet man zwischen passiven und aktiven Sensoren im Hinblick auf die benötigte Hilfsenergie bei der Umwandlung der Eingangsgröße in eine messbare Einheit [Nor08].

Aktive Sensoren wandeln die Eingangsmessgröße direkt in elektrische Impulse um. Zum Beispiel wird dabei die thermische Messgröße in elektrische Impulse umgewandelt. Dabei erzeugen die aktiven Sensoren eine Spannung, die zur weiteren Verarbeitung genutzt werden kann. Dadurch braucht ein Aktiver Sensor keine Hilfsenergie und er muss keine Signalumformung betreiben [Nor08]. Ein Beispiel für einen aktiven Sensor wäre der Lichtsensor.

Passive Sensoren müssen extern mit Energie versorgt werden, damit sie eine Eingangsgröße in einen elektrischen Impuls umwandeln können. Die verbauten Bestandteile des Sensors reagieren dabei auf die Änderung einer nicht elektrischen Größe. Ein Beispiel dazu wäre ein Magnetfeldsensor. Es gibt verschiedene Möglichkeiten um passive Sensoren mit Energie zu versorgen. Darunter gibt es Verfahren, die eine direkte Zufuhr von elektrischer Energie oder eine Energiegewinnung aus verschiedenen Quellen unterstützen. Solche Verfahren werden auch als *Energy Harvesting* bezeichnet [Nor08].

Darüber hinaus können die Sensoren auch nach ihrem Typ unterschieden werden. Darunter kann man verstehen, was für eine Art von Sensor es ist, und auf welche physikalische Messgröße er reagiert. In Tabelle 2.1 werden einige Sensortypen und Beispiele dazu aufgeführt.

Sensor Typ	**Beispiel**
Ablaufsensor	Windgeschwindigkeitsmesser
Akustischer Sensor	Mikrofon
Bewegungssensor	Beschleunigungsmesser, Kilometerzähler
Chemischer Sensor	Wasserstoffsensor, Sauerstoffsensor, Rauchmelder
Drucksensor	Barometer, Piezometer
Navigationssensor	Höhenmesser, Gyroskop, Magnetischer Kompass
Optische Sensoren	Kamera, Infrarotsensor, Lichtsensor
Temperatursensor	Thermometer, Bolometer
Umgebungssensor	Aktinometer, Wetterradar

Tabelle 2.1: Sensor Typ mit Beispiel [Snu14]

Sensoren können aber auch nach dem Kommunikationskanal unterschieden werden. Dazu gehören beispielsweise Bluetooth (siehe Kapitel 2.2) oder USB (siehe Kapitel 2.3). Darüber hinaus können Sensoren auch über WiFi angebunden werden. Heutzutage unterstützen alle mobilen Endgeräte diesen Standard [Bri97]. Als eine weitere Anbindungsmöglichkeit für externe Sensoren kann Near Field Communication (NFC) benutzt werden. Eine Verbindung über das NFC ist allerdings nur in begrenztem Umfeld möglich. Diese Technologie wird beispielsweise für das bargeldlose Zahlen benutzt [Ved13]. Eine weitere kabellose Verbindungsart im Nahbereich ist WiBree, diese wurde von Nokia entwickelt und kann im Heimbereich eingesetzt werden [Joh08]. Darüber hinaus gibt es noch ZigBee, diese wird meist für Adhoc-Netzwerke genutzt [Pao07]. ANT+ wäre eine weitere kabellose Verbindungsart, diese wird von einigen gängigen Smartphones genutzt und arbeitet im lizenzfreien 2,4-GHz-Frequenzband [Dyn14].

Ein weiteres Unterscheidungsmerkmal zwischen den Sensoren liegt in der eigentlichen Interaktion. Dabei kann zum Beispiel eine Datenübertragung eines Sensors nur durch ein initiales Startsignal gestartet werden und daraufhin sendet dieser kontinuierlich Daten bis ein Stoppsignal geschickt wird. Bei einer weiteren Interaktion könnte ein Datentransfer erst getätigt werden, wenn eine Datenanfrage gesendet wurde. In einem weiteren Interaktionsablauf, kann durch ein Startsignal die Aufnahme eines Videos gestartet werden und einige Zeit später durch ein Stoppsignal wieder gestoppt werden. Dieses kann noch erweitert werden durch Pausieren und Fortsetzen der Aufnahme.

2.4.2 Anwendungsgebiete

Heutzutage werden immer mehr Sensoren sowohl im Industrie-, Medizin-, oder Privat-Bereich eingesetzt. In der Industrie kann mit Hilfe der Sensoren beispielsweise der Produktionsprozess für Güter automatisch überwacht werden. Weiterhin werden Sensoren in Automobilien benutzt. Im Automobil werden Sensoren beispielsweise als Einparkhilfen benutzt. Dabei werden meist Abstandsmesser oder Kamerasensoren genutzt. Durch die Abstandsmesser werden dabei akustische Signale geliefert, die dem Fahrer signalisieren wie weit er noch von dem hinter ihm stehenden Hindernis entfernt ist. Im Medizin-Bereich werden Sensoren zur kontinuierlichen Überwachung von Vitalfunktionen

der Patienten genutzt. Dazu gehört zum Beispiel der Blutdrucksensor, der Blutzuckermesser, Sauerstoff-Sensor und CO2-Sensor. Dadurch können sowohl die Ärzte als auch die Privatpersonen selbst ihre gesundheitlichen Werte überwachen und so schneller auf etwaige Ereignisse zu reagieren.

Zusätzlich zu den externen Sensoren können auch die internen Sensoren des mobilen Endgeräts unterstützt werden. Dabei liegt jedoch das Hauptaugenmerk auf den externen Sensoren, da sie keine einheitliche Schnittstelle zur Kommunikation zwischen Anwendung und Sensor bieten. Die internen Sensoren werden meist durch die API des jeweiligen Betriebssystems unterstützt. Mittlerweile hat jedes mobile Endgerät bis zu 20 Sensoren die durch das Betriebssystem angesprochen werden können.

Im folgenden Kapitel werden verwandte Arbeiten vorgestellt. Dabei werden Kernfunktionen von unterschiedlichen Sensor-Frameworks herausgearbeitet und dargestellt.

3

Verwandte Arbeiten

Dieses Kapitel befasst sich mit verwandten Arbeiten, die sich in dem Kontext diese Arbeit bewegen. Dabei wird, wenn möglich, auf die Architekturen der Sensor-Frameworks eingegangen. Zudem werden der Funktionsumfang, sowie die zur Verfügung stehenden Schnittstellen beschrieben.

3.1 Open Data Kit

Bei der Entwicklung des Open Data Kit (ODK) Sensor-Frameworks war das Hauptziel die Entwicklung einer Anwendung, die Sensoren verwendet, zu vereinfachen. Weiterhin ist ODK ein *Open Source* Projekt, das für Android Anwendungen verwendet wird. Dazu dienten die Abstraktion der Kommunikationskanäle sowie die Unterteilung von anwendungsspezifischem und sensorspezifischem Code. Bei der Definition der Schnittstellen wurde darauf geachtet, dass identische Funktionalität zusammengefasst wurde damit möglichst wenig Aufwand zur Einbindung des Sensor-Frameworks entsteht [Roh12].

3.1.1 Architektur

Das Open Data Kit bietet Schnittstellen sowohl für den Anwendungsentwickler als auch für den Sensortreiberentwickler. Die Schnittstellen für den Anwendungsentwickler sind dabei das *Service Interface* und der *Content Provider*. Darunter liegt der *Sensor Manager*, welcher die Verbindungen zu den einzelnen Sensortreibern beinhaltet. Unter dem Sensor Manager liegen *Channel Manager*, die die Sensoren mit dem Framework verbinden.

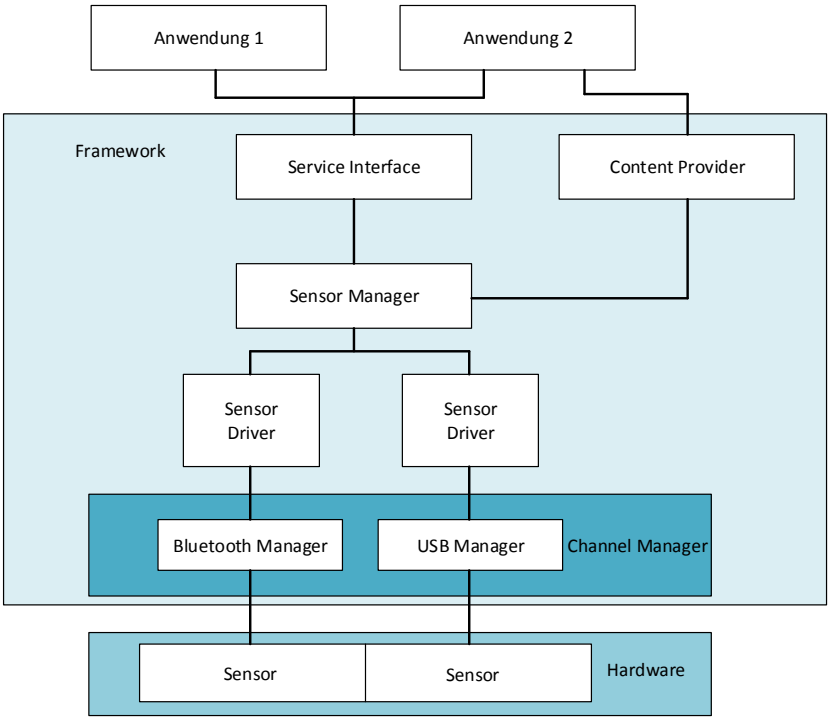

Abbildung 3.1: Schematische Darstellung Open Data Kit Architektur [Roh12]

Werden Daten durch einen Sensor an eine Anwendung geschickt, müssen diese zunächst über den *Channel Manager* geschickt werden. Dieser dient zum Aufbau des Kommunikationskanals und der Datenübertragung vom Sensor zum Framework. Um den Aufbau und die Verwaltung der Kommunikationskanäle zu vereinfachen, abstrahieren die Channel Manager von den eigentlichen Details, die zur Kommunikation verwendet werden (beispielsweise der Bluetooth-Protokollstack zur Anbindung eines Bluetooth Gerätes). Durch die Channel Manager werden dann die Daten an den Sensortreiber (*Sensor Driver*) weitergeleitet. Diese dienen dabei als Schnittstelle für die Sensortreiberentwickler um die eigentliche Logik für die Sensoren zu beschreiben. Dabei kann ein Sensortreiber direkt in das Sensor-Framework mit integriert werden oder durch eine externe sensorspezifische Anwendung in das Sensor-Framework eingebunden werden. Der Sensortreiber interpretiert und wandelt die Daten in ein für die Anwendung lesbares Format um. Diese Daten werden dann an den *Sensor Manager* weitergeleitet,

der diese daraufhin über das *Service Interface* an die Anwendung schickt. Sind schon lokal gespeicherte Daten vorhanden, die zusätzlich verwendet werden sollen, können auch diese über *Content Provider* an die Anwendung weitergeleitet werden. Dieser Sachverhalt ist in Abbildung 3.1 dargestellt [Roh12].

Diese Architektur wurde in ODK 2.0 um eine Möglichkeit der Visualisierung und Aufbereitung der Daten erweitert. Somit ist es nun den einzelnen Anwendern möglich, ihre Daten sofort nach der Datenerhebung lokal auf dem mobilen Endgerät zu betrachten. Dazu stellt ODK auf den mobilen Endgeräten zur Datenaufbereitung und Visualisierung Tabellen zur Verfügung. Diese unterstützen Anwender bei der Zusammenfassung, Bereinigung und Analyse der Daten. Dabei hat der Benutzer die Möglichkeit auch weitere Tabellen anzulegen. Die Tabellen können untereinander verknüpft werden um beispielsweise verschiedene Daten in einem gemeinsamen Kontext zu betrachten. Zu der Architektur wurde auch die Nutzung eines Cloud-Services hinzugefügt, der vom Anwender genutzt werden kann. Um diese Funktionalität nutzen zu können, muss jedoch geeignete Hardware bereitgestellt werden. Diese Cloud Lösung bietet dem Anwender noch zusätzlich eine Transformation der Daten in Standard Formate an [Way13].

3.2 Mobile Sensor Data Engine

Mobile Sensor Data Engine (MOSDEN) ist ein Sensor-Framework, das die Entwicklung von *Crowd Sensing* Anwendungen mit Sensorunterstützung vereinfachen soll. Eine Crowd Sensing Anwendung kann genutzt werden, um Daten unter Einbeziehung von einer großflächig verteilten Menschenmenge zu sammeln und zu verarbeiten. Um dies zu erreichen, unterstützt MOSDEN eine Abstraktion von Datensammlung, Verarbeitung und Speicherung gegenüber der eigentlichen Anwendungslogik. Zusätzlich unterstützt MOSDEN auch die Ansteuerung von sowohl internen als auch externen Sensoren und es ermöglicht eine anwendungsspezifische Implementierung von Datenverarbeitungs- und Datenanalysealgorithmen [Pre13].

3.2.1 Architektur

Die MOSDEN Architektur folgt einer komponentenbasierten Architektur. Dies erlaubt eine einfache Erweiterung des Sensor-Frameworks ohne den Kern des Frameworks verändern zu müssen.

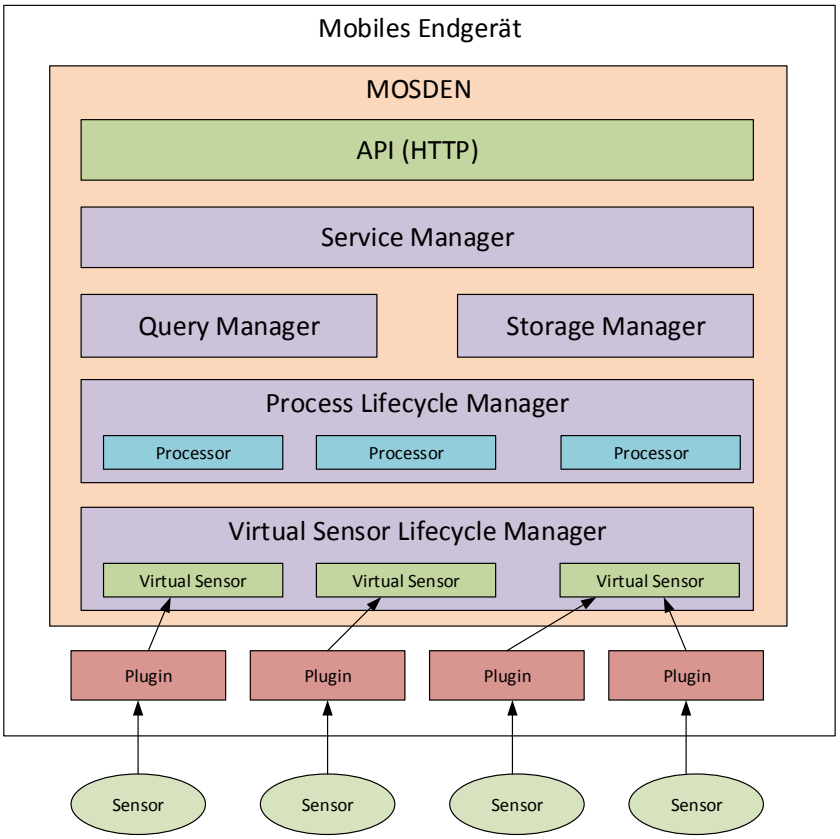

Abbildung 3.2: MOSDEN Architektur [Pre13]

In Abbildung 3.2 wird die Architektur von MOSDEN dargestellt. Durch sogenannte *Plug-ins* werden die Sensoren an das Framework angebunden. Durch diese ist es möglich, neue Sensoren hinzuzufügen, ohne an dem eigentlichen Framework-Code selbst etwas ändern zu müssen. Dazu wurde ein Plugin Deskriptor entwickelt um es einem Anwendungsentwickler zu ermöglichen, einen eigenen Sensortreiber zu entwickeln. MOSDEN kann anschließend zur Laufzeit neu hinzugefügte Sensortreiber dynamisch nachladen.

Der *Virtual Sensor Lifecycle Manager* übernimmt die Instanziierung, Erneuerung und Entfernung der einzelnen Virtual Sensor. Dieser wiederum dient zur Abstraktion der ihm zugeordneten Datenquelle. Der darüber liegende *Processor Lifecycle Manager* übernimmt die Verwaltung der *Processor* Einheiten. Dabei dient eine *Processor* Einheit als Schnittstelle für die vom Anwendungsentwickler implementierten Datenverarbeitungsalgorithmen. Dadurch können die Daten zusammengefasst und analysiert werden [Pre13].

Durch den *Storage Manager* werden die nun gesammelten Daten lokal auf dem mobilen Endgerät gespeichert. Der *Query Manager* dient zur Verarbeitung von Anfragen, die beispielsweise durch eine Anwendung oder den Anwender selbst gestellt werden. Darüber liegt der *Service Manager*, der die Registrierung einer Anwendung oder einer weiteren MOSDEN Instanz, auf die gesammelten Daten verwaltet. Dazu vermerkt der *Service Manager* die Registrierungsanfrage auf die gesammelten Daten und wenn möglich wird ein Datentransfer zur beispielsweise weiteren MOSDEN Instanz gestartet. Das *Application Programmable Interface (API)* beschreibt Schnittstellen um durch eine Anwendung mit dem Framework zu interagieren. Die API Anfragen werden über HTTP an MOSDEN verschickt [Pre13].

3.3 Medical Sensor Data Collection Framework

Bei der Entwicklung dieses Frameworks wurde mit betrachtet, dass die Ärzte und medizinischen Helfer leichter Daten erheben können und diese direkt an das Electronic Medical Record (EMR) System eines Krankenhauses weiterleiten können. Falls gerade keine Internet-Verbindung verfügbar ist, muss es dennoch möglich sein weitere Datenerhebungen durchzuführen. Aufgrund dessen ermöglicht das Framework eine lokale Datenspeicherung auf dem mobilen Endgerät. Die Weiterleitung der Daten übernimmt das Framework automatisch sobald eine Internet-Verbindung verfügbar ist [Rak15].

3.3.1 Architektur

Um die genannten Aufgaben zu erfüllen, kann das Framework mit den sensorspezifischen Anwendungen über eine vordefinierte Schnittstelle kommunizieren. Nach der Datenerhebung kann das Framework über eine weitere Schnittstelle die Daten an das EMR-System weiterleiten. Zurzeit unterstützt dieses Framework EMR-Systeme, die auf REST Webservices basieren [Roy00].

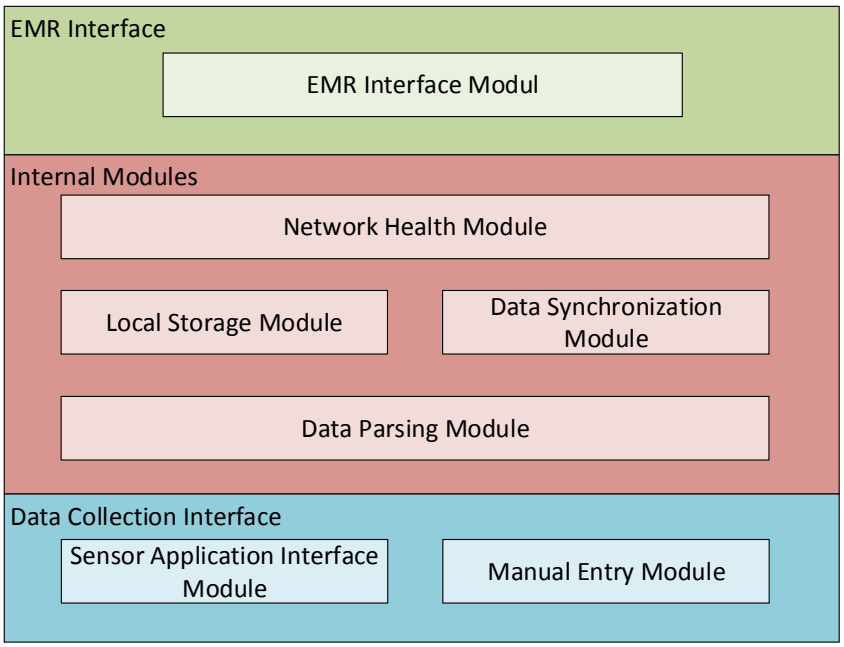

Abbildung 3.3: EMR Framework Architektur [Rak15]

Das Framework selbst ist in mehrere Module unterteilt, von denen jedes eine spezifische Aufgabe erfüllt. In Abbildung 3.3 wird die Architektur des Frameworks dargestellt. Dabei stellt das *Data Collection Interface* die Grundlage zur Erhebung von Patientendaten dar. Das *Sensor Application Interface* ist für die Anbindung von sensorspezifischen Anwendungen und die Sammlung der Sensordaten zuständig. Dazu muss sich die Anwendung bei dem Sensor Application Interface durch ein Broadcast registrieren. Dabei versendet die Anwendung eine asynchrone Nachricht, die signalisiert, dass etwas ausgeführt werden soll. Ebenfalls in diesem Modul liegt das *Manual Entry Module*, durch

den medizinische Helfer die Personenbezogenen Daten aufnehmen. Danach werden die erhobenen Daten im XML Format abgespeichert. Die Daten die durch den Sensor oder durch manuelle Eingabe erhoben werden, werden an das *Data Parsing Modul* weitergeleitet [Rak15].

Das *Data Parsing Modul* verarbeitet die Daten, die im XML Format vorliegen und formatiert diese daraufhin in ein Key-Value Paar, welches durch das *Local Storage Module* in einer NoSQL basierten Datenbank abgespeichert wird. Als Datenbank wurde dafür die CouchBase gewählt, da sie eine Datenspeicherung zu einem Cloud System ermöglicht und eine Synchronisation mit den Daten, die bereits auf dem Cloud System existieren vollziehen kann. Das *Data Synchronization Module*, welches neben dem Local Storage Modul liegt, erlaubt sowohl die Synchronisierung mit einem EMR System, als auch mit dem Cloud System welches durch die CouchBase erreichbar ist. Im Anschluss liegt nun das *Network Health Module* welches die Netzwerkverbindung prüft und wenn das Netzwerk verfügbar ist sendet dieses die Daten über das *EMR Interface Module* an das EMR-System. Falls das Netzwerk jedoch nicht zum Zeitpunkt der Übermittlung verfügbar ist, speichert das Netzwerk Health Module die Daten in einer lokalen Queue. Ganz oben in der Architektur ist nun die Schnittstelle zum EMR-System, das ist das *EMR Interface Module*. Das Modul unterstützt nur EMR-Systeme die auf REST Webservices basieren [Rak15].

3.4 HealthKit

HealthKit ist ein von Apple entwickeltes Framework, das den Entwicklern von Fitness-Anwendungen ermöglicht Daten zwischen verschiedenen Anwendungen auszutauschen. Das Framework kann auch mit verschiedenen zu Healthkit kompatiblen Gesundheits- und Fitnessgeräten kommunizieren und die Daten in der Health Datenbank speichern. Auf diese wird durch den Health Store zugegriffen, der als Schnittstelle agiert. Zusätzlich zu dem HealthKit wurde von Apple auch eine Health Anwendung entwickelt, die als Benutzerschnittstelle zum HealthKit dient. Die Health Anwendung stellt Visualisierungs- und Datenbearbeitungsmöglichkeit für den Anwender zur Verfügung [App15].

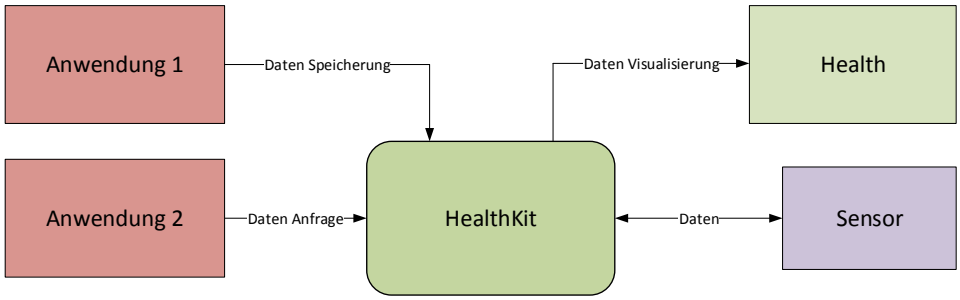

Abbildung 3.4: Health Kit Nutzung

In Abbildung 3.4 wird eine mögliche Nutzung vom HealthKit veranschaulicht. Dabei werden die Daten von der Anwendung 1 über das HealthKit gespeichert. Dazu wird Health Store als Schnittstelle zur internen Datenbank von HealthKit genutzt. Diese Daten können dann von Anwendung 2 über das HealthKit abgerufen werden. Da die Gesundheitsdaten einer Person sehr sensible Daten sind, ermöglicht HealthKit dem Anwender selbst zu entscheiden welche Daten von den jeweiligen Anwendungen verwendet und gespeichert werden dürfen. Falls eine Anwendung keine Zugriffsrechte auf bestimmte Daten hat (beispielsweise die gesammelten Blutglukosedaten), sind aus der Anwendungssicht auch keine Daten vorhanden [App15].

Die moderne Gesundheits- und Fitnesswelt liefert verschiedene Facetten, was die Sammlung, Analyse und Visualisierung gesammelter Daten betrifft. Diese Aspekte müssten, wenn sie verwendet werden sollten, bei der Entwicklung einer Fitness- oder Gesundheitsanwendung berücksichtigt werden. Diese Aufgaben können durch das HealthKit unterstützt werden. Somit ist es möglich einen besseren Überblick über den Gesundheitszustand eines Anwenders zu bekommen. Um den besagten Datenaustausch zwischen den Anwendungen zu ermöglichen, müssen die sensorspezifischen Daten in ein von HealthKit vorgegebenen Datentyp umgewandelt werden. Dadurch ist es auch möglich, dass verschiedene Anwendungen Daten miteinander teilen, ohne diese umständlich transformieren zu müssen. Nachteil dabei ist jedoch, dass die Entwickler keine eigenen Datentypen definieren können [App15].

3.5 CRNTC+

Das CRNTC+ Sensor Framework basiert auf der Context Recognition Network Toolbox (CRNT), die eine komponentenbasierte Architektur zur Analyse von Daten ist. Das CRNTC+ erweitert dabei die CRNT Architektur um Ein-/ Ausgabe Komponenten zur Anbindung von Services und internen und externen Sensoren. Das Hauptziel bei der Entwicklung des Sensor-Frameworks war es, ein erweiterbares und leicht konfigurierbares Framework zu entwickeln, um schneller Anwendungen zur mobilen Datenerhebung entwickeln zu können [Gab13].

3.5.1 Architektur

Die Architektur ist aufgeteilt in die Bereiche *Smartphone-specific Layer* und *Generic Data Processing Layer*. Durch diese Unterteilung können die plattformspezifischen und CRNT spezifischen Komponenten unabhängig voneinander erweitert werden. CRNTC+ beinhaltet nicht nur die Plattformspezifischen Komponenten sondern auch die CRNT Komponenten, die zur Signalbehandlung und Analyse benötigt werden. Des Weiteren sind verschiedene CRNT *Writer* wie beispielsweise der LogWriter direkt benutzbar.

Die Hauptbestandteile in der Architektur sind *Reader* und *Writer*. Die Reader stellen hierbei die Schnittstellen zu den Sensoren dar und können vom Smartphone benutzt werden. Durch die Reader wird ein Kommunikationskanal zu den einzelnen Sensoren aufgebaut. Dabei kann beispielsweise der Reader ein BluetoothReader sein durch den Bluetooth Sensoren angebunden werden und die Datensammlung gestartet werden kann. Da die Reader von verschiedenen Plattform-APIs abhängen, wurden diese in dem *Smartphone-specific Layer* verankert. Ein weiterer Bestandteil sind die *Writer*, die für die Codierung der Daten sowie die weitere Analyse und Visualisierung zuständig sind. Dazu können die Writer in beiden Bereichen vorhanden sein. Dabei kann beispielsweise ein Writer vom *Generic Data Processing Layer* der LogWriter, der zur Speicherung auf dem mobilen Endgerät genutzt wird, oder TCPWriter, der zur Speicherung in der Cloud genutzt wird, sein. Ein möglicher Writer vom *Smartphone-specific Layer* wäre beispielsweise ein Visualisierer, der die Daten in einer Grafik darstellt. Hinter der zentra-

len Komponente *Data Processing Components* sind die zur Analyse, Aufbereitung und Filterung der Daten genutzten Komponenten zusammengefasst [Gab13]. In Abbildung 3.5 wird die beschriebene Architektur dargestellt.

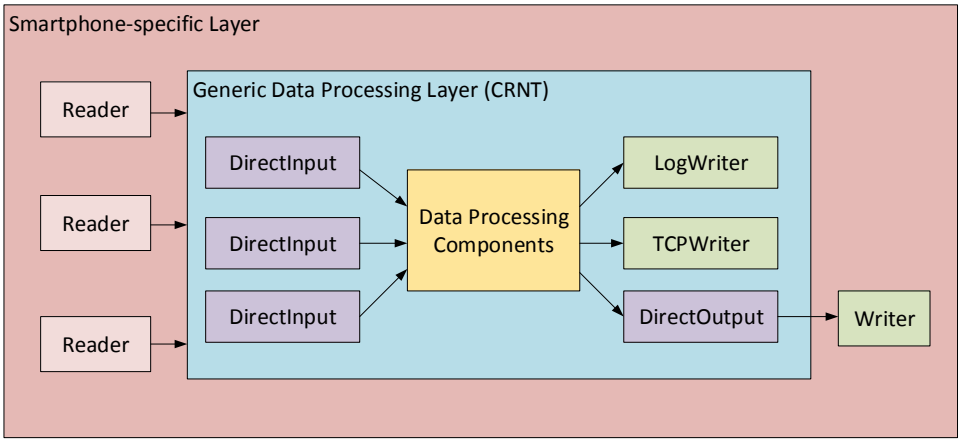

Abbildung 3.5: CRNTC+ Architektur frei nach [Gab13]

In diesem Framework wird die Konfiguration der einzelnen Komponenten durch ein JSON-Objekt beschrieben. Dazu werden zu jeder Komponente die durch den Entwickler definierten Parameter und die Verbindung zu einer anderen Komponente gespeichert. CRNTC+ bietet zur Verbindung der beiden Bereiche *DirectInput* und *DirectOutput* Komponenten. Diese Komponenten werden benötigt als Standard Schnittstellen zwischen den beiden Schichten um zu garantieren, dass trotz der flexiblen Konfiguration ein Datenaustausch möglich bleibt [Gab13].

3.6 Vergleich

In diesem Abschnitt werden die hier beschriebenen Sensor-Frameworks mit dem in dieser Arbeit entwickelten Sensor-Framework verglichen. Dazu werden zunächst in der folgenden Tabelle die einzelnen Sensor-Frameworks anhand ihrer Funktionen miteinander verglichen. Dabei steht MSDC für das Medical Sensor Data Collection Framework.

	ODK	MOSDEN	MSDC	HealthKit	CRNTC+
Unterstützung der Interaktion	○	○	○	○	○
Abstraktion der Verbindungsart	●	○	○	○	●
Hinzufügen neuer Sensortreiber	●	●	○	○	●
Persistente Speicherung	●	●	●	●	●
Visualisierung der Daten	●	○	○	●	●
Analyse der Daten	●	●	●	●	●
Sensor Konfiguration	○	○	○	○	○
Logging	○	○	○	○	○

Tabelle 3.1: Vergleich der beschriebenen Sensor-Frameworks

Bei dem in Tabelle 3.1 dargestellten Vergleich, fällt auf, dass die beschriebenen Sensor-Frameworks keine der Interaktionsabläufe, die möglich wären, unterstützt. Dies wird im hier zu entwickelnden Sensor-Framework berücksichtigt, in dem Interaction Patterns unterstützt werden (siehe Kapitel 5.2.2). Weiterhin wird das Hinzufügen von neuen Sensortreibern in ODK nur durch das Einbeziehen in das Sensor-Framework oder durch extra implementierte Anwendungen unterstützt. In MOSDEN hingegen ist es möglich, einen neuen Sensortreiber zur Laufzeit hinzuzufügen. Dies wird auch im zu entwickelnden Sensor-Framework berücksichtigt (siehe Kapitel 6.2.4). Weiterhin wird durch das hier zu entwickelnde Sensor-Framework eine Konfiguration des Sensors ermöglicht (siehe Kapitel 6.3.3). Darüber hinaus bietet das im Rahmen dieser Arbeit

zu entwickelnde Sensor-Framework ein Logging-Mechanismus, das alle Ereignisse, die während einer Kommunikation auftreten, mit protokolliert (siehe Kapitel 6.2.2).

Durch die Darstellung, der hier beschriebenen Sensor-Frameworks, ist nun ersichtlich welche Anforderungen ein Sensor-Framework umfassen sollte. Im folgenden Kapitel werden nun die Anforderungen für das Sensor-Framework, das im Zuge dieser Arbeit entwickelt wird, beschrieben. Dazu werden die Anforderungen in funktionale und nicht-funktionale Anforderungen unterteilt und jeweils kurz beschrieben was im Kontext des Sensor-Frameworks darunter zu verstehen ist.

4

Anforderungen

Dieses Kapitel umschreibt die Anforderungen, die das im Zuge dieser Arbeit zu entwickelnde Sensor-Framework erfüllen muss. Dazu wird auf verschiedene funktionale und nicht-funktionale Anforderungen eingegangen.

4.1 Funktionale Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen beschrieben. Diese umschreiben die Funktionalität des zu entwickelnden Sensor-Frameworks.

AF1 (Plug-and-Play): Eine Kernanforderung an das Sensor-Framework ist das *Plug-And-Play* Konzept. Dabei soll der Anwendungsentwickler die Möglichkeit haben, neue Sensortreiber zum Sensor-Framework hinzufügen zu können. Dazu sollte der Anwendungsentwickler jedoch nicht das Sensor-Framework bearbeiten müssen. Somit sollte beim Hinzufügen eines Sensortreibers kein detailliertes Wissen über das Sensor-Framework notwendig sein. Des Weiteren sollte der Sensortreiberentwickler nur die Sensorlogik implementieren müssen und auf gemeinsam nutzbare Funktionalität, wie beispielsweise die Bluetooth Anbindung, zurückgreifen können. Dazu sollte das Sensor-Framework verschiedene Schnittstellen sowohl für den Anwendungsentwickler als auch für den Sensortreiberentwickler anbieten.

AF2 (Unterstützung verschiedener Interaktionsabläufe): Es gibt verschiedene Interaktionsarten mit den Sensoren. Dabei kann beispielsweise eine Interaktion durch ein initiales Startsignal gestartet werden, nach Eingang dieses Signals startet der Sensor einen kontinuierlichen Datentransfer bis der Sensor ein Stoppsignal

erhält (beispielsweise Puls-Sensor). Andere Sensoren könnten immer nur bei einer Anfrage einmalig Daten verschicken (beispielsweise Foto-Kamera). Diese und weitere Interaktionsabläufe sollte das Sensor-Framework unterstützen. Dazu muss das Sensor-Framework verschiedene Schnittstellen für die Sensoren anbieten, um mit ihnen interagieren zu können.

AF3 (Klassifizierung der Sensoren): Im Industrie-, Medizin- und Privat-Bereich werden immer mehr Sensoren eingesetzt. Diese können, in verschiedene Kategorien unterteilt werden (Vergleich Kapitel 2.4.1). Dabei können beispielsweise die Sensoren nach ihrer Anbindung unterschieden werden. Um diese Klassifizierung der Sensoren zu unterstützen, sollte das Sensor-Framework eine Möglichkeit anbieten, die Sensoren nach Meta-Informationen zu gruppieren (beispielsweise nach Kommunikationsart oder Datentyp). Das Sensor-Framework muss dazu eine Schnittstelle anbieten, um solche Anfragen tätigen zu können.

AF4 (Serialisierung/Deserialisierung): Damit die durch den Sensor gesammelten Daten exportiert (und später wieder importiert) werden können, sollte das Sensor-Framework eine Schnittstelle zur Serialisierung (und Deserialisierung) der Daten anbieten. Dabei können verschiedene Export Formate unterstützt werden (beispielsweise XML und JSON). Für den Sensortreiberentwickler soll es möglich sein, diese zu benutzen, ohne den Sensor-Framework-Code zu kennen. Zusätzlich sollte auch die Möglichkeit bestehen, eine eigene Serialisierung und Deserialisierung für Sensoren bereitzustellen.

AF5 (Abstraktion der konkreten Schnittstelle): Um das Sensor-Framework einbinden zu können, oder eine Anbindung zu einem Sensor zu initialisieren, sollten konkrete Schnittstellen definiert werden, die von dem eigentlichen Vorgang abstrahieren. Dabei kann hinter einer Schnittstelle zum Sensor das Protokoll zum Verbindungsauf- oder Verbindungsabbau stehen. Weiter kann durch eine Schnittstelle eine gemeinsam genutzte Funktionalität gekapselt werden. Diese Funktionalität kann dann durch den Sensortreiberentwickler oder durch den Anwendungsentwickler genutzt werden. Dadurch ist es möglich, dass durch eine Schnittstelle

beispielsweise ein Bluetooth-fähiger oder ein interner Sensor angesprochen werden kann.

AF6 (Unterschiedliche Kommandos für Sensoren): Einige Sensoren bieten spezifische Kommandos an, die durch das Sensor-Framework übermittelt werden müssen. Das kann beispielsweise das Kommando für ein Startsignal, oder einen anderen Konfigurationsmodus sein.

AF7 (Logging): Bei einer Interaktion zwischen dem Sensor-Framework und den verbundenen Sensoren können verschiedene Ereignisse auftreten. Diese sollten durch ein Mechanismus des Sensor-Frameworks protokolliert werden. Dazu soll das Sensor-Framework einen Logging Mechanismus anbieten, der durch den Anwendungsentwickler frei konfigurierbar ist. Dabei soll es möglich sein, Ereignisse auf verschiedenen Schichten des Sensor-Frameworks zu dokumentieren.

AF8 (Visualisierung): Einige Sensoren liefern Daten in einer leicht visualisierbaren Form (beispielsweise Puls-Sensor). Es soll möglich sein, diese Daten durch verschiedene Diagramme anzuzeigen. Diese Diagramme sollten auch beim Eintreffen weiterer Daten sich automatisch aktualisieren können. Darüber hinaus soll es möglich sein, die Daten in verschiedenen Diagramme gleichzeitig zu veranschaulichen und die Ausprägungen zu verdeutlichen. Dazu sollte das Sensor-Framework eine Schnittstelle anbieten durch die die Diagramme konfiguriert und abgerufen werden können.

4.2 Nicht-Funktionale Anforderungen

Dieser Abschnitt befasst sich mit den nicht-funktionalen Anforderungen, diese beschreiben die Eigenschaften des Sensor-Frameworks.

NFA1 (Zuverlässigkeit): Gegenüber Fehlern, die bei einer Interaktion mit einem Sensor auftreten können, sollte eine Fehlertoleranz gegeben sein. Dabei sollte eine Anwendung die das Sensor-Framework nutzt, nicht durch einen Verbindungsfehler abstürzen. Das Sensor-Framework sollte dafür eine Fehlerbehandlung anbieten

und wenn nötig einen erneuten Versuch zur Anbindung von externen Geräten ermöglichen.

NFA2 (Benutzbarkeit): Die Entwicklung einer mobilen Anwendung, die mit Sensoren kommunizieren soll, sollte durch das Sensor-Framework erleichtert werden. Dazu sollten die Schnittstellen, die zur Einbindung des Sensor-Frameworks und zum Hinzufügen der Sensortreiber genutzt werden, leicht verständlich und gut dokumentiert sein.

NFA3 (Erweiterbarkeit): Das Sensor-Framework soll mit möglichst vielen Sensoren interagieren können. Dazu muss es einfach möglich sein, weitere Sensoren einzubinden. Ebenso muss es auch möglich sein, das Sensor-Framework um andere Interaktionsabläufe zu erweitern. Um das Sensor-Framework um andere Verbindungsarten zu erweitern, muss es auch möglich sein, weitere Sensor-Manager einzubinden. Dabei dient der Sensor-Manager zur Verwaltung von einer Verbindungsart und den darüber angebundenen Sensoren.

NFA4 (Durchgängige Dokumentation): Die Schnittstellen für den Anwendungsentwickler und für den Sensortreiberentwickler sollten umfassend dokumentiert werden, damit der jeweilige Entwickler keine fundierten Kenntnisse über das Sensor-Framework benötigt, dennoch aber die Funktionalitäten nutzen kann. Des Weiteren soll die interne Funktionalität hinsichtlich der Wartung und der Erweiterbarkeit des Sensor-Frameworks genau dokumentiert werden.

NFA5 (Lokale Nutzung): Zur Nutzung des Sensor-Frameworks sollte keine Internet-Verbindung notwendig sein. Dabei sollte die gesamte Funktionalität, die das Sensor-Framework anbietet, lokal auf dem mobilen Endgerät ablaufen. Somit sollte das Sensor-Framework auch bei Anwendungen nutzbar sein, die in einem Umfeld genutzt werden, in dem keine Netzwerkanbindung möglich oder nur schwer zu bewerkstelligen ist.

Das folgende Kapitel befasst sich mit der Konzeption des Sensor-Frameworks. Dazu wird zunächst ein grober Überblick über die gesamte Architektur des Sensor-Frameworks diskutiert. Danach werden die einzelnen Schichten des Sensor-Frameworks und deren Aufgaben im Detail besprochen.

5

Sensor-Framework

Dieses Kapitel befasst sich mit der konzeptionellen Sicht auf das zu entwickelnde Sensor-Framework, das in die bestehende Anwendung, die in [Joh13] entwickelt wurde, eingesetzt werden kann. Dazu wird zunächst die grundlegende Architektur des Sensor-Frameworks veranschaulicht, und dabei ein grober Überblick über die einzelnen Schichten gegeben. Danach werden die einzelnen Schichten genauer betrachtet.

5.1 Architektur

In diesem Kapitel wird auf die Architektur des Sensor-Frameworks eingegangen und erklärt, wie die einzelnen Schichten miteinander kommunizieren. Dazu wird auch geschildert, wie die Sensoren in einzelne Kategorien unterteilt werden. Die Architektur des Sensor-Frameworks wird in Abbildung 5.1 dargestellt. Darin ist ganz oben der *Sensor-Framework-Manager*, dieser ist über *Anwendung-Event Bus* mit der Anwendung verbunden.

Der Sensor-Framework-Manager ist zuständig für die Anfragen, die von der Anwendung über den Anwendung-Event Bus geschickt werden. Dieser reicht diese Anfragen über den *Sensor-Framework-Event Bus* an den entsprechenden *Sensor-Manager* (beispielsweise USB Manager) weiter. Der Sensor-Manager nimmt den Request entgegen und verarbeitet diesen, das kann beispielsweise die Initialisierung eines Sensors, oder das Aufnehmen von neuen Sensortreibern sein.

Werden hingegen über eine Anfrage Daten von einem Sensor benötigt, wird dieser von dem entsprechenden Sensor-Manager an den zugehörigen *Sensortreiber* über den

Sensor-Manager-Event Bus weitergeleitet, dabei hat jeder Sensor-Manager einen Event Bus (beispielsweise USB Bus). Der Sensortreiber führt daraufhin die benötigte Funktion des *Sensors* aus. Auf die Details dieser Schichten des Sensor-Frameworks wird in den folgenden Kapiteln näher eingegangen.

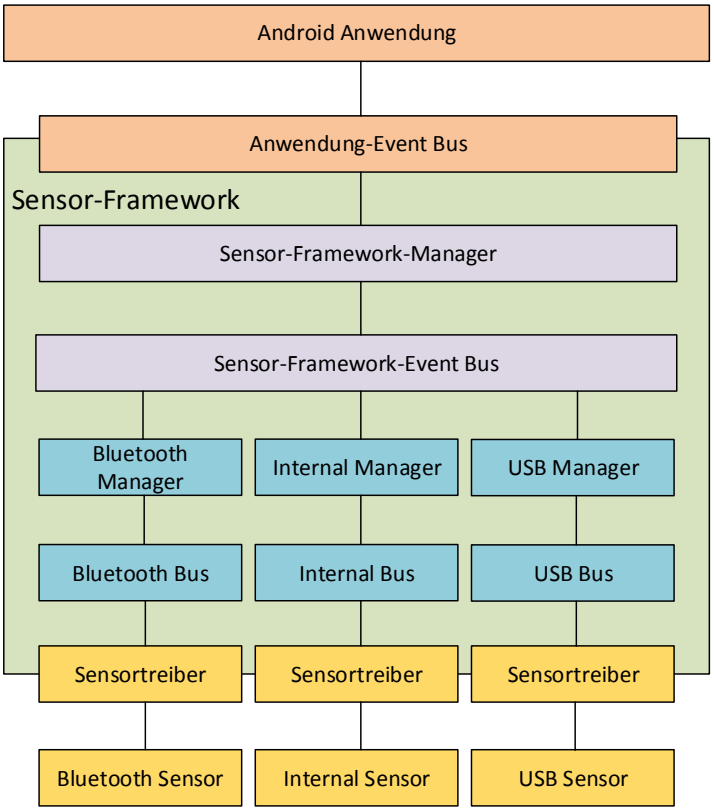

Abbildung 5.1: Schematische Darstellung der Sensor-Framework Architektur

5.1.1 Event Bus

Zur Kommunikation zwischen den Schichten wird ein Event Bus eingesetzt. Dazu hat jede Schicht einen eigenen EventBus, damit beispielsweise die Anwendung nicht ein Ereignis erhalten kann, das für die Sensor-Manager bestimmt ist. Dadurch ist auch gewährleistet, dass die Anwendung nicht mitbekommt, was im Hintergrund nach

einer Anfrage geschieht. Da die Kommunikation zwischen den Schichten komplett auf Ereignissen basiert, erleichtert der Event Bus die Übertragung und die Verarbeitung der Ereignisse, da dafür notwendige Event-Listener und Event-Handler nicht implementiert werden müssen. Um eine Ereignisverarbeitung zu gewährleisten, dient der Event Bus als Mediator, der die Ereignisse auf die verschiedenen Teilnehmer verteilt. Die Teilnehmer selber dienen dabei sowohl als Event-Producer als auch als Event-Handler [Gre15b].

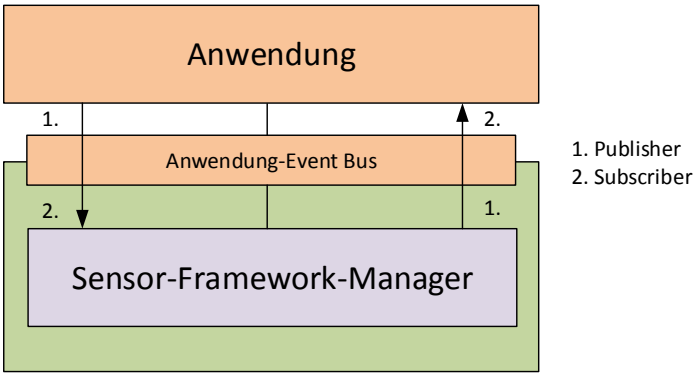

Abbildung 5.2: Ausschnitt aus Sensor-Framework Architektur

Der Event Bus basiert auf dem *Publish-Subscribe Pattern*, welches bereits in Kapitel 2.1.2 näher beschrieben wurde. In Abbildung 5.2 wird durch einen Ausschnitt aus der Sensor-Framework-Architektur dargestellt, wie eine Kommunikation durch einen Event Bus bewerkstelligt wird. Bei einer Kommunikation kann ein Publisher die Anwendung sein, die ein Ereignis sendet, das durch den Event Bus dann an den Sensor-Framework-Manager weitergeleitet wird.

Aus Sicht des Event Busses kann man die Interaktionen mit den Service Interaction Patterns *One-To-Many Request* und *One-From-Many Response* dargestellt werden [AB05]. Dabei dient der Event Bus als Broadcaster der einen Request an mehrere Parteien weiterleitet und daraufhin auch die Responses von den einzelnen Teilnehmern erhält und verarbeitet bzw. weiterleitet.

Die Vorteile, die die Verwendung eines Event Busses mit sich bringt, sind einerseits dass die einzelnen Teilnehmer der Kommunikation keine Details voneinander wissen müssen. Die Teilnehmer können sich direkt bei einer Instanz des Event Busses registrieren und über Reflection erhält der Event Bus die Information welche Ereignisse die einzelnen Teilnehmer erwarten. Dazu sucht der Event Bus bei den Teilnehmern nach *onEvent(Object event)* Methoden, durch die, die Teilnehmer als Event-Handler gesehen werden.

Ein weiterer Vorteil ist, dass dabei verschiedene Aktivitäten, Hintergrund Services oder Threads als Teilnehmer fungieren können. Da Ereignisse nicht immer sofort verarbeitet werden, muss eine Art Zwischenspeicher gewährleistet werden. Dazu dient ein weitere Vorteil, dass sind die *Sticky-Events*, die bleiben dabei solange auf dem Bus, bis diese verarbeitet werden können. Wie schon eingangs erwähnt wurde, erleichtert die Nutzung des Event Busses die Implementierung in der Hinsicht, dass man keine Listener und Handler selber implementieren muss.

5.1.2 Klassifizierung

Als eine Anforderung an das Sensor-Framework wurde die Anfrage nach Sensoren über eine Klassifizierung genannt (siehe AF3). Somit kann ein Anwender beispielsweise eine Anfrage nach Datentyp oder nach einer Gruppe von Sensoren stellen. Darüber hinaus ist es auch möglich durch die Klassifizierung Sensoren zu erfragen, welche über Bluetooth verbunden sind. Im Bluetooth-Standard wurden bereits von der Bluetooth Special Interest Group (SIG) Klassen definiert. Diese sollen die Identifizierung von Geräten, die über Bluetooth verbunden sind, erleichtern.

Laut [Pat13] konnte bei einem Test mit dem dort verwendeten Pulsoximeter Sensor, das Gerät nicht identifiziert werden. Des Weiteren wurde auch in [Pat13] auf die in Bluetooth spezifizierten Profile eingegangen, die zur Anbindung von spezifischen Sensoren dienen [Blu15d]. Aufgrund dessen sind diese Eigenschaften von Bluetooth nicht ausreichend um die Anforderung an das Sensor-Framework zu erfüllen.

Bei USB werden zur Klassifizierung sogenannte *Device Classes* verwendet. Diese wurden in [Pat13] jedoch nicht betrachtet. Sie dienen zur hardwareseitigen Identifizierung des USB-Gerätes, und können genutzt werden um beispielsweise zu erfragen, ob ein Gerät durch den Sensortreiber unterstützt wird. Jedoch wäre die Anfrage über eine Device Class schon sehr hardwarenah, daher ist es für das Sensor-Framework nicht relevant [USB99].

Die Standard API von Android bietet lediglich zur Klassifizierung der internen Sensoren einige vordefinierte Konstanten [Goo15c]. Diese genügen jedoch nicht der Anforderung *AF1 (Plug-And-Play)*, die das Hinzufügen von neuen Sensortreibern definiert. Bei einer Anbindung eines weiteren Sensors müssen neue Meta-Informationen mit einbezogen werden.

Um ein erweiterbares Sensor-Framework zu unterstützen, muss eine Klassifizierung spezifiziert werden, die eben diese Anforderung *AF1 (Plug-And-Play)* unterstützt. Dadurch soll es möglich sein, neue Sensor Meta-Daten hinzufügen zu können, anhand derer die Sensoren auffindbar sind. Zudem sollen so klare Aussagen über die Sensoren gemacht werden können. Dies kann durch allgemeine Definition von Gruppen gemacht werden. Dabei kann beispielsweise als Datentyp *Numerical* spezifiziert werden, dieser kann anschließend wieder verfeinert werden, indem der Sensortreiberentwickler zusätzlich noch die Klasse des Datentyps angibt (zum Beispiel: Numerical, Puls.class).

Dadurch ist es möglich im Sensor-Framework vordefinierte Konstanten anzubieten um die Sensoren in Gruppen zu unterteilen. Diese Konstante kann dann durch den Sensortreiberentwickler genutzt werden um den Sensor in eine Kategorie einzuordnen. Andererseits kann der Anwendungsentwickler über diese Kategorie auf eine Gruppe gleichartiger Sensoren zurückgreifen. Diese Anfrage kann auch verfeinert werden indem ein vom Sensortreiberentwickler spezifizierter Sensortyp genutzt wird um die Anfrage zu filtern.

5.1.3 Ablauf einer Datenanfrage

In diesem Abschnitt wird die durch Abbildung 5.3 dargestellte Anfrageverarbeitung beschrieben. Eine Datenanfrage wird hier beispielsweise auf Grundlage des *Single-Datarequest Pattern* (siehe Kapitel 5.2.2) mit einem Kamera-Sensor beschrieben. Die Kamera-Anwendung leitet dabei die Interaktion mit dem Sensor-Framework durch ein *StartDataRequest* ein. Dieser wird über den Anwendung-Event Bus an den Sensor-Framework-Manager gesendet. Der Sensor-Framework-Manager überprüft anschließend, welcher Sensor-Manager für die Beantwortung dieser Anfrage verantwortlich ist.

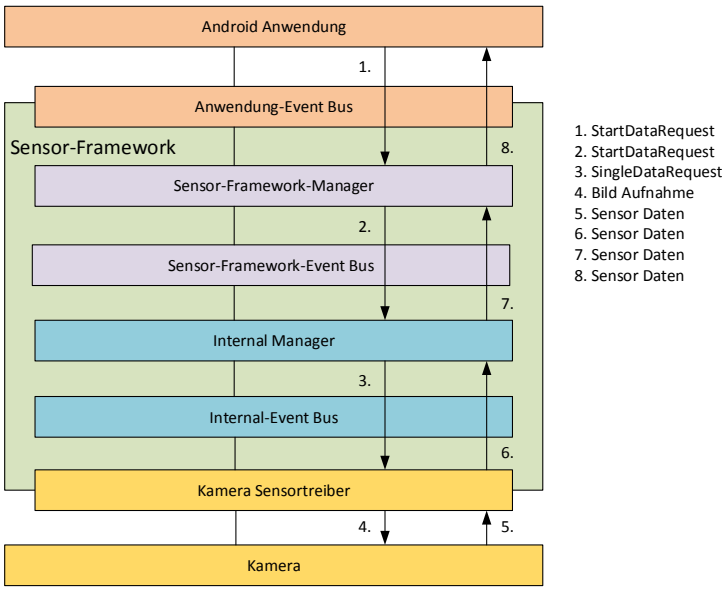

Abbildung 5.3: Ablauf einer Datenanfrage

In diesem Fall wird die Anfrage an den *Internal-Manager* über den Sensor-Framework-Event Bus weitergeleitet. Dieser verarbeitet die Anfrage, indem er die dazu passende Aktion ausführt. Dabei sendet er einen *SingleDataRequest* über den Internal-Event Bus an den Kamera Sensortreiber. Der Kamera Sensortreiber startet daraufhin die Aufnahme eines Bildes.

Nach der Aufnahme des Bildes, werden die *Sensor Daten* durch den Kamera Sensortreiber über den Internal-Event Bus an den Internal-Manager weitergeleitet. Der Internal-Manager sendet die Daten über den Sensor-Framework-Event Bus an den Sensor-Framework-Manager weiter. Der Sensor-Framework-Manager speichert die Daten und leitet diese über den Anwendung-Event Bus an die Anwendung weiter. Diese kann die Daten nun weiterverarbeiten.

5.2 Sensortreiber

In diesem Kapitel wird die Schnittstelle zu den Sensoren betrachtet. Als Schnittstelle zu den Sensoren dienen die Sensortreiber, durch die das Sensor-Framework auf die Funktionalität der dahinter liegenden Sensoren zugreifen kann. Dazu werden zunächst die einzelnen Funktionalitäten der Sensortreiber besprochen. Danach werden die Interaction Patterns beschrieben, durch die ein Interaktionsablauf mit einem Sensor vollzogen werden kann.

5.2.1 Funktionalität

Die unterste Schicht des Sensor-Frameworks bilden die Sensortreiber, die die Schnittstelle zu den Sensoren beschreiben. Die Sensortreiber sind über einen speziellen Sensor-Manager-Event Bus (beispielsweise Bluetooth Bus) mit dem ihnen zugeordneten Sensor-Manager (beispielsweise Bluetooth Manager) verbunden.

In Abbildung 5.4 werden die Aufgaben der Sensortreiber aufgeführt. Die daraus resultierenden Funktionalitäten des Sensortreibers sind hiermit, die direkte Kommunikation mit dem Sensor, bereitstellen der Informationen eines Sensors, bereitstellen einer Konfigurationsdatei und die Serialisierung bzw. Deserialisierung der Daten.

Bei der Kommunikation mit einem Sensor, wird der eigentliche Verbindungsaufbau durch den Sensortreiber durchgeführt. Dabei kann der Sensortreiberentwickler auf bereits vorhandene Funktionalität des Sensor-Frameworks oder der Plattform zurückgreifen. Das kann beispielsweise bei der Anbindung von Bluetooth Sensoren die Nutzung der Android

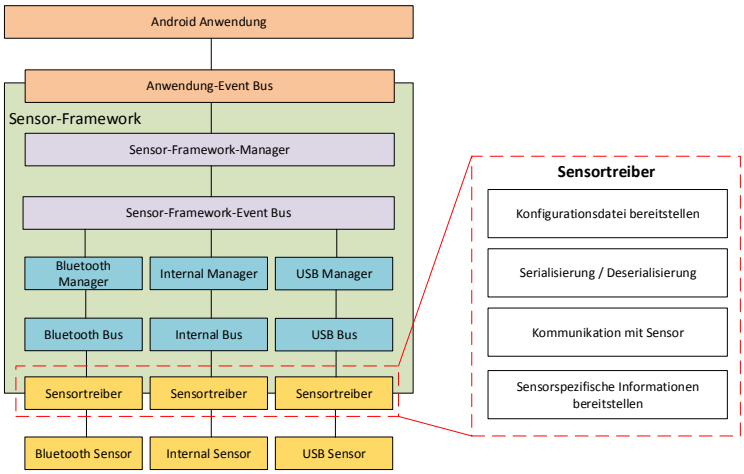

Abbildung 5.4: Aufgaben der Sensortreiber

Standard API sein. Dabei wurde die Android API genau betrachtet und festgestellt, dass keine einheitliche Schnittstelle für sowohl interne als auch externe Sensoren es gibt. Viele von den internen Sensoren, wie beispielsweise der Bewegungssensor, werden dabei durch einen Mechanismus, der in der Android API beschrieben wird, angesprochen und er kann diese dann auch auflisten [Goo15c]. Die weiteren internen Sensoren, wie beispielsweise der Kamera Sensor, können direkt durch die jeweilige Schnittstelle, die in der Android API beschrieben wird angesprochen werden. Zur Interaktion mit dem Sensor nutzt der Sensortreiber die Interaction Patterns, durch die der Sensortreiberentwickler einen Sensortreiber für dieses Sensor-Framework implementieren kann.

Um eine Konfiguration des Sensors zu ermöglichen, muss der Sensortreiber eine Konfigurationsdatei bereitstellen, die durch den Sensor-Manager eingelesen werden kann (siehe Kapitel 6.3.3). Weiterhin muss der Sensortreiber durch ein Meta Daten Objekt verschiedene Informationen des Sensors bereitstellen. Dazu gehört beispielsweise der Sensorname, die Verbindungsart, der Sensortyp und weitere. Diese werden als Meta-Daten eines Sensors bezeichnet. Sie können durch den zuständigen Sensor-Manager abgefragt werden und können als Grundlage für die Klassifizierung dienen.

Damit es möglich ist die Sensor Daten zu ex- und importieren, bietet der Sensortreiber eine Schnittstelle an, um eine Serialisierung bzw. Deserialisierung zu implementieren.

Damit der Sensortreiberentwickler sich nicht selber darum kümmern muss, kann er auf Funktionalität des Sensor-Frameworks zurückgreifen. Die Serialisierung bzw. Deserialisierung der Daten kann zur Laufzeit durch den Anwender ausgelöst werden um die Daten in der Anwendung weiter zu verarbeiten.

5.2.2 Interaction Patterns

Um Interaktionsabläufe mit einem Sensor zu unterstützen, werden Interaction Patterns definiert. Dazu wurden die Interaction Patterns, die in [Pat13] beschrieben wurden, näher betrachtet und auf ein ereignisgesteuertes Sensor-Framework portiert. Dabei kam hervor, dass das Event Pattern, welches für Ereignis gesteuerte Sensoren benötigt wird, leicht mit einbezogen werden kann. Da das Sensor-Framework auf Grundlage eines Event Busses arbeitet, reicht dafür eine Registrierung des Sensortreibers durch den zuständigen Sensor-Manager am Event Bus. Der Sensor kann dann die gesammelten Daten solange senden, bis der zuständige Sensortreiber vom ihm übergeordneten Sensor-Manager deregistriert wird. Der Ablauf der Daten Sendung verläuft dann ähnlich wie bei *Multiple-Dataresponse Pattern*. Dabei muss dann statt dem Start- und Stop-Request eine Registrierung und Deregistrierung des Sensortreibers erfolgen.

Des Weiteren werden in [Pat13] das *Asynchronous Request Pattern* und *Synchronous Request Pattern* beschrieben. Dabei werden im Asynchronous Request Pattern nach Eingang einer Datenanfrage und einer gewissen Bearbeitungszeit einmalig Daten gesendet. Beim Synchronous Pattern werden hingegen direkt nach der Datenanfrage die Daten gesendet. Die beiden Interaction Pattern können durch das Single-Datarequest Pattern abgedeckt werden.

Multiple-Dataresponse Pattern

Dabei wird ein Start-Ereignis gesendet, welches einen kontinuierlichen Datentransfer des Sensors auslöst. Der Datentransfer wird dabei solange getätigt bis ein Stopp-Ereignis gesendet wird. Dieser Sachverhalt wird in Abbildung 5.5 dargestellt. Dieses Interaction

Pattern ist notwendig, da beispielsweise der implementierte *Location Sensor* nach dem Start in bestimmten Zeitabständen neue Daten mit der aktuellen Position sendet.

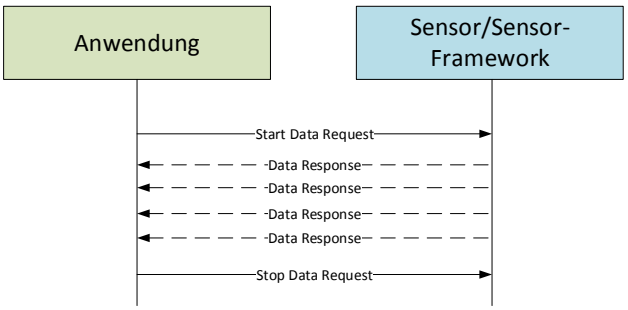

Abbildung 5.5: Multiple-Dataresponse Pattern

Single-Datarequest Pattern

Wie bereits erwähnt wurde, kann das Single-Datarequest Pattern für das Asynchronous Request Pattern und für das Synchronous Request Pattern genutzt werden. Dabei wird ein Datentransfer getätigt wenn eine Datenanfrage gesendet wurde. In Abbildung 5.6 wird dieser Sachverhalt aufgezeigt. Dieses wird als Beispiel beim aktuell implementierten *Kamera Sensor* genutzt, der auf Anfrage eine Bildaufnahme auslöst und anschließend einen Datentransfer initiiert.

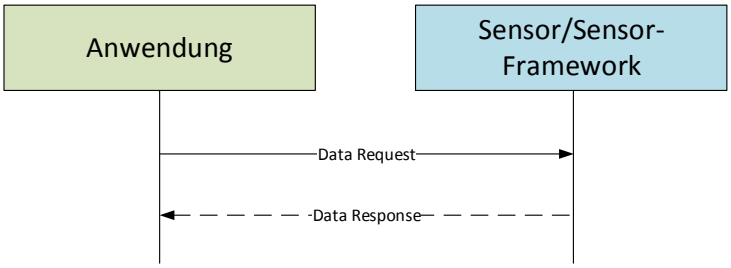

Abbildung 5.6: Single-Datarequest Pattern

Recording Pattern

Das Recording Pattern, welches in [Pat13] beschrieben wurde, war nur darauf ausgelegt, dass eine einzelne Aufnahme getätigt werden kann. Dieses Pattern wird um die Möglichkeit Aufnahme zu pausieren und fortzusetzen erweitert. Der beschriebene Sachverhalt wird in Abbildung 5.7 dargestellt.

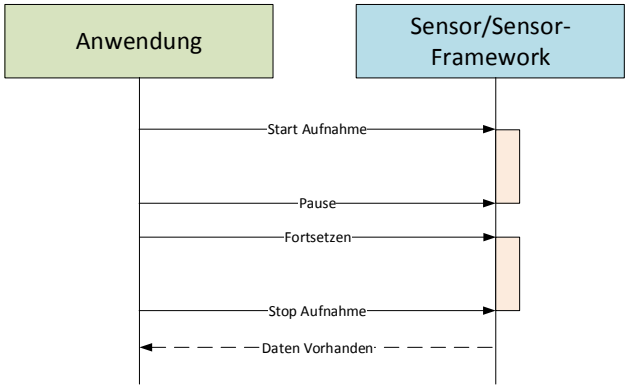

Abbildung 5.7: Recording Pattern

5.3 Sensor-Manager

In diesem Kapitel werden die Sensor-Manager beschrieben. Dabei ist ein Sensor-Manager für eine Verbindungsart zuständig. Dazu werden die Aufgaben der Sensor-Manager erläutert und welche Funktionalität sie dafür nutzen.

5.3.1 Funktionalität

Der Sensor-Manager verarbeitet alle Anfragen, die über den Sensor-Framework-Event Bus an ihn weitergeleitet werden. Eine Anfrage kann beispielsweise ein Datenrequest an einen spezifischen Sensor oder die Initialisierung der Serialisierung bzw. Deserialisierung der Daten sein.

Bei der Initialisierung eines Sensors muss der Sensor-Manager den passenden Sensortreiber holen und ihn durch das Interaction Pattern, welches durch den Sensortreiber spezifiziert wird, initialisieren. Der gestartete Sensor wird daraufhin in die Liste der verbundenen Sensoren aufgenommen. Dadurch ist es möglich eine Anfrage nach allen aktuell mit dem Sensor-Framework verbundenen Sensoren zu beantworten. Die hier beschriebenen Funktionen des Sensor-Managers werden in Abbildung 5.8 aufgeführt.

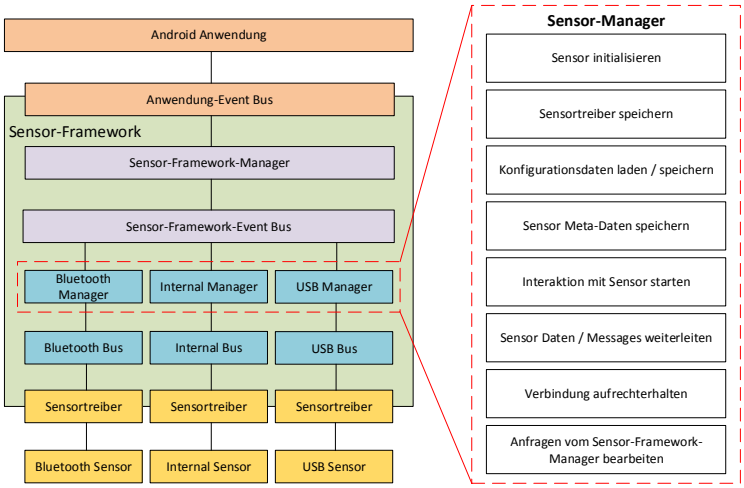

Abbildung 5.8: Funktionen eines Sensor-Managers

Die Weiterleitung der Daten und der Nachrichten vom Sensor an das Sensor-Framework übernehmen die einzelnen Sensor-Manager. Dadurch kann beispielsweise ein Fehler, der bei der Verbindung oder bei einem Datenrequest aufkommt an den Sensor-Framework-Manager propagiert werden. Des Weiteren sind die Sensor-Manager auch für die Konfiguration der Sensoren zuständig. Dazu benötigt der Sensor-Manager eine Konfigurationsdatei, die vom Sensortreiber mitgeliefert wird. Zum Einlesen der Konfigurationsdatei wird der Settings-Manager genutzt. Dieser ist ein Bestandteil der Sensor-Manager, und wird durch den Sensor-Manager initialisiert. Der Settings-Manager übernimmt die Aufgabe des Einlesens der Konfigurationsdaten und teilt die eingelesenen Daten in aktuelle Einstellungen und weitere mögliche Einstellungen für einen Sensor. Die

Konfigurationsdaten können vom Sensor-Manager bei der Initialisierung des Sensors genutzt werden, oder direkt zur Laufzeit geändert werden.

Beim initialisieren eines Sensors werden die aktuell gesetzten Einstellungen an den zuständigen Sensortreiber übergeben. Dazu werden entweder die Einstellungen vom Anwender oder die Einstellungen des Sensortreiberentwicklers genutzt. Der Settings-Manger speichert zudem die aktuellen Einstellungen des Sensors, damit beim nächsten Start der Anwendung die letzten Einstellungen wiederhergestellt werden können. Dabei werden die Funktionen des Settings-Managers durch den Sensor-Manager gestartet.

Eine weitere Funktion der Sensor-Manager ist die Verwaltung der aktuellen Verbindungen zu den Sensoren. Dazu speichert der Sensor-Manager eine Liste der gestarteten Sensoren um die Kommunikation schneller aufzugreifen.

5.4 Sensor-Framework-Manager

Dieser Abschnitt befasst sich mit dem Sensor-Framework-Manager, dieser bildet die oberste Schicht des Sensor-Frameworks. Der Sensor-Framework-Manager ist über den Anwendung-Event Bus mit einer Android Anwendung verbunden. Zur Erläuterung des Sensor-Framework-Managers, werden zunächst die Funktionen des Sensor-Framework-Managers aufgezeigt. Dabei wird auch der *LogWriter* und der *Visualisierungsmanager* beschrieben, die in dem Sensor-Framework-Manager integriert sind.

5.4.1 Funktionalität

Der Sensor-Framework-Manager ist für die Verarbeitung der Anfragen einer Anwendung, die das Sensor-Framework verwendet, zuständig. Eine solche Anfrage kann beispielsweise das Laden aktueller Sensor Daten oder deren Visualisierung sein. Zudem bietet der Sensor-Framework-Manager einen Zugriff auf den LogWriter an, der sämtliche Ereignisse protokolliert und Einblick im Fehlerfall gewährt.

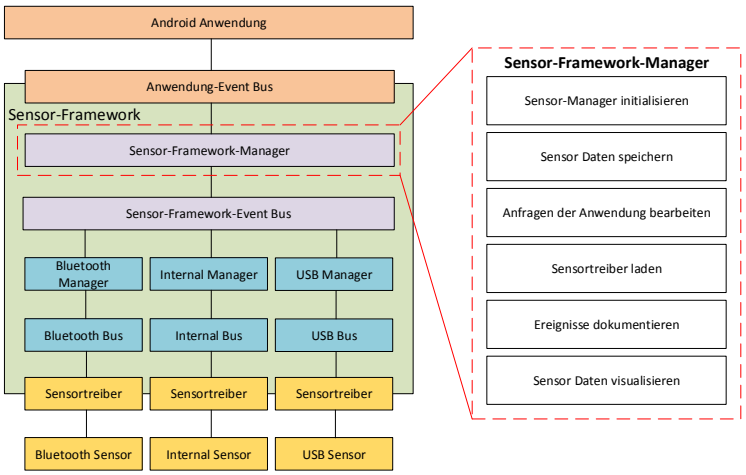

Abbildung 5.9: Funktionen des Sensor-Framework-Managers

In Abbildung 5.9 werden alle Funktionen des Sensor-Framework-Managers aufgeführt. Diese werden im Folgenden genauer erläutert. Durch den LogWriter ist *AF7 (Logging)* gewährleistet. Der LogWriter ist von dem Anwendungsentwickler frei konfigurierbar. Da die interne Kommunikation komplett auf Ereignissen beruht und über Event Busse abläuft, muss der LogWriter nur die Busse abhören.

Um *AF1 (Plug-And-Play)* zu unterstützen, können dem Sensor-Framework-Manager über die Anwendung zur Laufzeit Sensortreiber übergeben werden. Diese werden durch den Sensor-Framework-Manager an die dafür zuständigen Sensor-Manager weitergeleitet, diese werden gegebenenfalls vom Sensor-Framework-Manager initialisiert. Bei der Initialisierung der Sensor-Manager werden diese auch gleichzeitig bei dem Sensor-Framework-Event Bus registriert.

Beim Laden der Sensortreiber speichert der Sensor-Framework-Manager benötigte Informationen für sich ab. Diese werden genutzt um bei einer Interaktion mit einem Sensor zu erfahren, an welchen Sensor-Manager die Anfrage delegiert werden muss. Des Weiteren werden alle Daten oder Nachrichten, die von einem Sensor-Manager weitergeleitet wurden, durch den Sensor-Framework-Manager über den Anwendung-Event Bus an die Anwendung weitergeleitet.

Der Sensor-Framework-Manager unterstützt eine Visualisierung der Sensor Daten. Dazu können durch den Anwendungsentwickler verschiedene Diagramme konfiguriert werden, diese werden über den Anwendung-Event Bus an den Sensor-Framework-Manager weitergeleitet. Der Startet daraufhin für jedes Diagramm, das angefordert wurde, einen Visualization-Manager. Der Visualization-Manager liegt wie der LogWriter im Kontext des Sensor-Framework-Managers.

Das Sensor-Framework wurde bisher aus konzeptioneller Sicht betrachtet, in der auf die Implementierungsdetails weitestgehend nicht eingegangen wurde. Im Folgenden Kapitel werden verschiedene Details der Implementierung vom Sensor-Framework diskutiert. Dazu wird zunächst der Sensor-Framework-Manager genauer betrachtet. Danach werden die Sensor-Manager im Detail erläutert. Dabei werden die Klassen der einzelnen Teilnehmer betrachtet und wie die hier genannten Funktionen unterstützt werden.

6

Sensor-Framework Implementierung

In diesem Kapitel wird das Sensor-Framework aus Implementierungssicht betrachtet. Dazu wird der Ablauf zum Hinzufügen eines Sensortreibers vorgestellt. Anhand dieses Ablaufs wird auf die einzelnen Schichten eingegangen und diskutiert, welche Aufgabe sie dabei erfüllen. Bei den einzelnen Schichten wird zusätzlich auf weitere Funktionen eingegangen, die bereits in Kapitel 5 erwähnt wurden.

6.1 Kommunikation im und mit dem Sensor-Framework

In diesem Abschnitt wird die Kommunikation zwischen den einzelnen Schichten des Sensor-Frameworks und zu der Anwendung beschrieben. Dazu wird zunächst die Funktionsweise des Event Busses beschrieben, der als Grundlage für die Kommunikation genutzt wird. Danach wird der Ablauf zum Laden eines Sensortreibers grob beschrieben.

6.1.1 Funktionsweise

Dieser Abschnitt befasst sich mit dem Event Bus, welcher für die Kommunikation zwischen den einzelnen Schichten zuständig ist. Dabei wird als Event Bus der *Greenrobot Event Bus* [Gre15b] genutzt. Dieser hat aufgrund von höherer Performance den Vorzug vor dem *Square's Otto Event Bus* erhalten [Gre15a]. Damit eine Kommunikation möglich ist, müssen sich die Teilnehmer zunächst am Event Bus registrieren. Dabei wird über den Befehl `<EventBus>.register(Subscriber)` (beispiels- weise kann ein `<EventBus>` eine Instanz des Sensor-Framework-Event Busses sein) ein Kommunikationsteilnehmer zum Event Bus hinzugefügt.

Dabei wird der Teilnehmer in eine Liste von Subscribern hinzugefügt. Bei der Registrierung durchsucht der Event Bus, den Teilnehmer nach `onEvent(event)` Methoden. Falls der mögliche Teilnehmer keine solche `onEvent(event)` Methode beinhaltet, wird durch den Event Bus eine `EventBus Exception` ausgelöst.

Der Event Bus ermittelt durch Reflection die Event-Klassen auf die sich der Teilnehmer registriert hat, die vom Event Bus in eine Liste für den Teilnehmer gespeichert werden. Bei einer Kommunikation durch einen Event Bus sendet ein Teilnehmer durch `<EventBus>.postSticky(event)` ein Ereignis an den Event Bus. Dieser prüft ob Teilnehmer, die dieses Ereignis verarbeiten können, vorhanden sind. Daraufhin wird das Ereignis an den Teilnehmer übergeben. Dazu wird die `onEvent(event)` Methode des Teilnehmers aufgerufen und der Teilnehmer verarbeitet daraufhin das Ereignis.

6.1.2 Laden eines Sensortreibers

Um diesen Ablauf der Kommunikation zu verdeutlichen, wird das exemplarisch durch das Laden eines neuen Sensortreibers vorgestellt. Der Ablauf wird durch die Android Anwendung eingeleitet indem sie den Sensor-Framework-Manager startet. Danach sendet sie über den Anwendung-Event Bus ein *SensorListing*, das ein oder mehrere implementierte Sensortreiber enthält, an den Sensor-Framework-Manager. Der Sensor-Framework-Manager startet den benötigten Sensor-Manager und übergibt den Sensortreiber über den Sensor-Framework-Event Bus an den zuständigen Sensor-Manager.

Dieser speichert den Sensortreiber bei sich in der Sensortreiberliste. Nachdem der Sensor-Manager den Sensortreiber gespeichert hat, sendet er eine Nachricht mit dem Erfolg vom Hinzufügen des Sensortreibers. Diese Nachricht wird durch den Sensor-Framework-Event Bus an den Sensor-Framework-Manager übergeben. Der Sensor-Framework-Manager leitet diese Nachricht über den Anwendung-Event Bus an die Android Anwendung weiter.

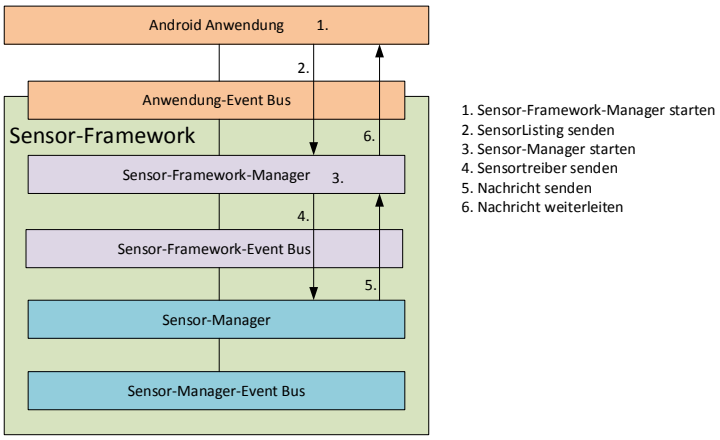

Abbildung 6.1: Ablauf Laden eines Sensortreibers

Der Ablauf wird in Abbildung 6.1 dargestellt. Anhand des Ablaufs wird nun im Folgenden auf die Implementierungsdetails der einzelnen Schichten eingegangen.

6.2 Sensor-Framework-Manager

Dieser Abschnitt befasst sich mit den Implementierungsdetails vom Sensor-Framework-Manager. Dazu wird zunächst erläutert was der Sensor-Framework-Manager beim Hinzufügen eines neuen Sensortreibers im Detail verarbeiten muss. Danach wird beschrieben wie der LogWriter arbeitet. Daraufhin wird beschrieben, wie ein Visualization-Manager von dem Sensor-Framework-Manager gestartet wird und was dazu benötigt wird.

6.2.1 Laden der Sensortreiber

Der Sensor-Framework-Manager erhält über den Anwendung-EventBus von der Android Anwendung ein `LoadDriverEvent`. Dieses liefert ein `SensorListing` Objekt zurück, welches den Sensortreiber enthält. Dabei kann im SensorListing Objekt auch mehr als ein Sensortreiber enthalten sein. Das SensorListing Objekt wird in dem Sensortreiber Package miteinbezogen.

Das Ereignis wird in der `onEvent(LoadDriverEvent)` Methode vom Sensor-Framework-Manager verarbeitet. Dies wird in Listing 6.1 aufgezeigt. Dabei entpackt der Sensor-Framework-Manager erstmal das SensorListing und prüft ob eine Sensortreiberliste übergeben wurde. Falls eine Liste übergeben wird, wird jeder Sensortreiber über `addDriverToList(driver)` zu der Sensortreiberliste vom Sensor-Framework-Manager hinzugefügt.

```
1  public void onEvent(LoadDriverEvent event){
2          // check if listing contains more then one driver
3          if(event.getSensorListing().getDriverList().size() > 0 ){
4          /*add drivers to list*/}
5          else if(!(event.getSensorListing().getDriver() == null)){
6          addDriverToList(driver);}
7          else{ /*send error message*/}
8  }
9  public void addDriverToList(Sensor driver){
10 if(!this.sensorList.containsKey(driver.getSensorname())){
11         if(driver.isDeviceDetected()){
12                 this.sensorList.put(driver.getSensorname(), driver);
13                 this.loadDriver(driver);
14         }else{/*send error message*/}
15 }
```

Listing 6.1: Sensor-Framework-Manager *LoadDriverEvent* Verarbeitung

Wird durch das SensorListing eine leere Sensortreiberliste übergeben, wird anschließend geprüft ob ein einzelner Sensortreiber enthalten ist. Falls dies zutrifft, wird dieser durch `addDriverToList(driver)` zu der Liste der Sensortreiber hinzugefügt. Durch `addDriverToList(driver)` wird `loadDriver(driver)` aufgerufen, darin wird geprüft welcher Sensor-Manager für den Sensor zuständig ist, dieser Sensor-Manager wird dann gestartet. Danach wird durch den Sensor-Framework-Event Bus der Sensortreiber an diesen übergeben.

Um den Sensortreiber über den Event Bus zu übergeben, ruft der Sensor-Framework-Manager bei sich `sfEventBus.postSticky(LoadClassEvent)` auf. Dadurch sendet der Sensor-Framework-Manager das Ereignis an den Sensor-Framework Event Bus,

dieser übergibt daraufhin das Ereignis an den Sensor-Manager, der sich dafür registriert hat. Dadurch wird die Anforderung *AF1 (Plug-And-Play)* unterstützt.

6.2.2 Logging

Beim Start der Android Anwendung kann der *LogWriter* konfiguriert werden. Die Konfiguration des LogWriters wird durch ein `InitLogWriterEvent` über den Anwendung-Event Bus an den Sensor-Framework-Manager übergeben. Nach Erhalt des Ereignisses initialisiert der Sensor-Framework-Manager den LogWriter mit der gegebenen Konfiguration.

```
1 public void onEvent(InitLogWriterEvent){
2         logWriter = new SFLogWriter(outputPath);
3         logWriter.setEventBus(aEBus, sfEBus, blEBus, intEBus);
4         logWriter.setLogLevel(logLevel);
5         logWriter.initLogfile();
6 }
```

Listing 6.2: LogWriter Initialisation

Dies wird in Listing 6.2 beschrieben. Darin wird eine Instanz vom LogWriter erstellt, diese erhält die Event Bus Instanzen und registriert sich anhand des Loglevels an den gegebenen Event Bussen. Ein Log-level kann beispielsweise der `LogLevel.MANAGER` sein. Bei diesem Log-level registriert sich der LogWriter an dem Anwendung-Event Bus, am Sensor-Framework-Event Bus und dem Sensor-Manager-Event Bus. Dabei liest der LogWriter alle Ereignisse mit, zu denen er die Berechtigung hat.

Aufgrund dessen, dass der Event Bus das auftretende Ereignis an alle dafür registrierten Teilnehmer übergibt, wird durch das Mitlesen des LogWriters die Kommunikation nicht unterbrochen. Der LogWriter schreibt Informationen über die Ereignisse, die er mitliest in eine Log-Datei. Die Log-Datei wird in dem Pfad, welcher über das `InitLogWriterEvent` übergeben wurde, hinterlegt. Damit die Log-Datei erstellt werden kann, muss der Anwendung durch die *WRITE_EXTERNAL_STORAGE* Berechtigung der Zugriff auf den externen Speicher (beispielsweise SD Card) erteilt werden. Damit diese Berechtigung

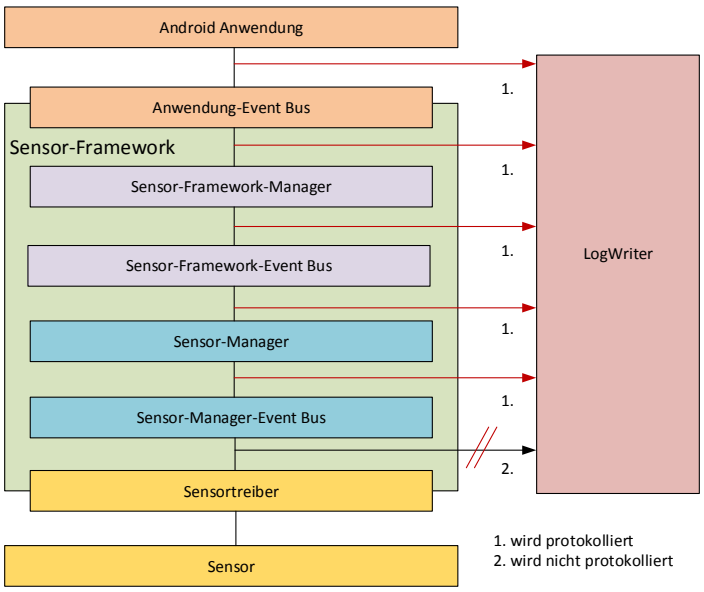

Abbildung 6.2: LogWriter mit LogLevel Manager

erteilt werden kann, muss ein Eintrag dazu in der *Manifest* Datei der Anwendung gemacht werden. Bei der Installation der Anwendung muss dann der Anwender dieser Berechtigung zustimmen. Der externe Speicher wurde gewählt, da der Anwender die benötigten Rechte zum Zugriff auf die Dateien besitzten muss und dies beim privaten Speicher der Anwendung nicht der Fall ist.

Bei dem beschriebenen Log-Level, wie in der Abbildung 6.2 dargestellt werden die Ereignisse des Sensortreibers nicht mit protokolliert, da durch den gewählten Log-Level der LogWriter keine Berechtigung dazu hat. Der Log-Level kann auch zur Laufzeit neu gesetzt werden, wenn diese Funktion vom Sensor-Framework bei der Entwicklung der Android Anwendung mit einbezogen wird. Die geschriebene Log-Datei kann bei der Entwicklung einer Android Anwendung als Hilfestellung genutzt werden um auftretende Fehler nachzuvollziehen.

6.2.3 Visualization

Der Visualization-Manager wird durch den Sensor-Framework-Manager initialisiert. Die Initialisierung wird durch ein `VisualizationRequestEvent` gestartet, das der Sensor-Framework-Manager von der Anwendung erhält. Dieses Ereignis liefert Informationen zur Konfiguration für das angeforderte Diagramm und welcher Sensor die Daten für die Visualisierung liefern soll. Bei der Verarbeitung des Ereignisses erstellt der Sensor-Framework-Manager eine Instanz des Visualization-Managers. Dieser erhält ein `VisualizationConfig` Objekt, durch das das Layout eines Diagramms frei gestaltet und beeinflusst werden kann. Dabei kann beispielsweise eingestellt werden, wie die einzelnen Datenpunkte darzustellen sind. Dazu können Eigenschaften, wie beispielsweise die Größe und Farbe der Punkte gewählt werden.

Danach erhält der Visualization-Manager die gesammelten Daten des Sensors durch `setDataToVisualize(..)`. Daraufhin werden die Daten von dem Visualization-Manager für die Visualisierung aufbereitet. Dabei werden die Daten in ein Format übersetzt, welches von der darunter liegenden Bibliothek verstanden wird. Beim Auslesen der Daten wird die Reflection Bibliothek von Android genutzt, die es ermöglicht auf Parameter und Methoden von externen Objekten zuzugreifen. Dadurch kann der Visualization-Manager alle Informationen, welche ein Daten-Objekt enthält, verarbeiten.

```
1 visualManager = new VisualizationManager(event.getVisualizationConfig());
2 visualManager.setDataToVisualize(sensorDataMap.get(event.getSensorname()));
3 createdViewList.add(visualizationManager.initChartView());
4 createdVisualizationManagers.add(visualizationManager);
5 /*..*/
6 sensorToVisualizeMap.put(sensorname,createdVisualizationManagers);
```

Listing 6.3: Initialisierung vom Visualization-Manager

Weiter ist es auch möglich, mehrere Diagramme gleichzeitig anzufordern (beispielsweise Puls und Sauerstoffsättigung). Dabei wird für jedes angeforderte Diagramm ein Visualization-Manager initialisiert und in eine Liste, der zu diesem Sensor erstellten Visualization-Manager, hinzugefügt. Somit ist es dem Anwendungsentwickler auch frei

gestellt, die Daten aus verschiedenen Gesichtspunkten darzustellen. Dazu bietet der Visualization-Manager bisher, die Daten in einem Linien-, Punkt- und Balkendiagramm darzustellen an. Welches Diagramm zur Darstellung der Daten genutzt wird, wird durch das entsprechende Konfigurationsobjekt festgelegt.

Als Grundlage für die Diagramme wird eine externe Bibliothek genutzt. Dazu wurden *MPAndroidChart* [Phi15], *AchartEngine* [The13] und *AndroidPlot* [And15] verglichen. Dabei hat unter anderem die MPAndroidChart aus den folgenden Gründen den Vorzug gegenüber den anderen Bibliotheken erhalten.

MPAndroidChart bietet einheitliche Datenformate für die einzelnen Datenpunkte von den meist genutzten Diagrammen. Dabei hat beispielsweise das Linien-Diagramm das gleiche Datenformat für die Datenpunkte wie ein Punkt-Diagramm. Es wurde auch beachtet, dass die benutzte Bibliothek eine durchgängige Dokumentation anbietet und weiterentwickelt wird. Bei AchartEngine war dies durch eine Javadoc gegeben. Die Nutzung der Bibliothek wurde jedoch nur anhand einer Demoanwendung beschrieben. Eine Weiterenticklung von AchartEngine war leider nicht ersichtlich.

AndroidPlot bietet eine gute Dokumentation anhand von Beispielen, aber leider keine ausführliche Erläuterung, welche weitere Diagramm Eigenschaften noch einstellbar sind. Dagegen bietet MPAndroidChart eine vollständige Dokumentation an. Dabei wird jede Funktionalität der Bibliothek erläutert und was für Eigenschaften zusätzlich noch eingestellt werden können, die das Layout oder das Aussehen der grafischen Details (beispielsweise Größe und Form der Punkte) betrifft.

Dabei ist es auch möglich bei MPAndroidChart auf Default Einstellungen zurückzugreifen, dies ist leider sowohl bei AchartEngine als auch bei AndroidPlot nicht möglich. Dies erleichtert die Konfiguration der Diagramme bei MPAndroidChart. Was zusätzliche Funktionen anbelangt, bietet MPAndroidChart eine eingebaute Zoomfunktion um die Datenpunkte genauer zu betrachten und auch eine Funktion um das Diagramm dynamisch zu aktualisieren, während beispielsweise ein Sensor kontinuierlich Daten liefert. Bei AndroidPlot und AchartEngine muss dafür ein eigener Listener implementiert werden.

Des Weiteren wurde auch eine Dialog-Klasse implementiert, die als Detailansicht des ausgewählten Datenpunktes genutzt werden kann. Diese wird durch einen Listener

aufgerufen, wenn ein angezeigter Datenpunkt ausgewählt wird. Damit ein anderer Dialog eingebunden werden kann, wird eine abstrakte Klasse angeboten, die neue Dialog Klassen erweitern müssen. So kann beispielsweise der Entwickler eines Sensortreibers auch die Darstellung in den Diagrammen beeinflussen.

6.2.4 Dynamisches Nachladen der Sensortreiber

Das Sensor-Framework bietet die Möglichkeit neue Sensortreiber dynamisch zur Laufzeit nachzuladen. Dazu muss der Sensortreiber in einem JAR-Archiv verfügbar sein und in einem Pfad auf dem mobilen Endgerät abgespeichert werden. Um das dynamische Laden zu ermöglichen, wurde zunächst die Android spezifische Reflection-API betrachtet [Goo15b]. Diese bietet jedoch nicht die Möglichkeit um eine Klasse dynamisch aus einem gegebenen Archiv nachladen zu können.

Danach wurde die Standard-API von Android genauer betrachtet, dabei wurde der Android spezifische *DexClassLoader* entdeckt. Der DexClassLoader kann genutzt werden um Klassen, die noch nicht zur Anwendung gehören nachzuladen. Dazu kann er Klassen aus einem JAR- oder APK-Archiv auslesen. Dazu müssen die Klassen in dem Archiv in einer `classes.dex` Datei gegeben sein. Dieses Archiv kann auf dem externen Speicher hinterlegt werden.

Der DexClassLoader muss zur Ausführung den Pfad zum Archiv, einen Pfad zu einem privaten Verzeichnis der Anwendung und eine Instanz des ClassLoaders von der Anwendung, die den DexClassLoader ausführt, erhalten. Das private Verzeichnis wird vom DexClassLoader genutzt um den Inhalt des Archivs zu entpacken und die `classes.dex` Datei zu laden. Aus dieser Datei, kann daraufhin der DexClassLoader die Sensortreiberklassen auslesen und in die Anwendung mit einbinden [Goo15a].

Um ein Archiv, das eine `classes.dex` Datei enthält, zu erstellen. Kann durch die Android IDE ein Android Projekt erstellt werden, in dem der Sensortreiber entwickelt wird. Dieses Projekt sollte nicht als Library gekennzeichnet werden, da es somit nicht kompilierbar wird. Durch das Kompilieren des Android Projektes kann ein solches Archiv erstellt werden, welches nach der Kompilierung im `bin` Verzeichnis des Projektes

abgelegt wird. Eine weitere Möglichkeit um ein solches Archiv zu erstellen, ist die Nutzung eines Befehlszeilentools `dx`, das in dem Android Software Development Kit (SDK) enthalten ist. Dazu muss dem Befehlszeilentool ein Archiv als Parameter übergeben werden, welches Standard Java-Byte-Code enthält. Dadurch wird dann ein JAR- oder APK-Archiv erstellt, das eine `classes.dex` Datei beinhaltet [Tho11].

6.3 Sensor-Manager

Dieses Kapitel befasst sich mit der Implementierung der Sensor-Manager. Dabei wird im Detail auf die Aufgabe eines Sensor-Managers beim Hinzufügen eines Sensortreibers eingegangen. Daraufhin wird beschrieben was der Sensor-Manager beim Anbinden eines Sensors zum Sensor-Framework genau zu bewerkstelligen hat.

6.3.1 Hinzufügen eines Sensortreibers

Um die Anforderung *AF1 (Plug-And-Play)* zu unterstützen, kann durch den zuständigen Sensor-Manager ein Sensortreiber geladen und zum Sensor-Framework hinzugefügt werden. Dazu erhält der Sensor-Manager ein spezifisches `LoadClassEvent` von dem Sensor-Framework-Manager. Dies kann wie in Listing 6.4 ersichtlich beispielsweise ein `InternalLoadClassEvent` sein. Damit nicht jeder Sensor-Manager das `LoadClassEvent` erhält und somit bei sich auch nicht überprüfen muss, ob er für den Sensor zuständig ist, wurde jedem Sensor-Manager ein spezifisches `LoadClassEvent` bereitgestellt.

```
1 public void onEvent(InternalLoadClassEvent event){
2         Sensor sensorDriver = (Sensor)event.getClassToLoad();
3         this.loadedDrivers.put(sensorDriver.getSensorname(), sensorDriver);
4         this.loadSettingsOfSensor(sensorDriver);
5 }
```

Listing 6.4: Hinzufügen eines Sensortreibers

Dadurch muss der Sensor-Manager sich ausschließlich um das Hinzufügen des übergebenen Sensortreibers in die Liste der Sensortreiber kümmern. Durch den Aufruf von `loadSettingsOfSensor(sensorDriver)` wird daraufhin die Konfigurationsdatei des aktuellen Sensortreibers geladen. Damit es zu keinem Konflikt bei der Ansteuerung eines Sensors kommt, ist pro Sensor nur ein Sensortreiber in der Liste. Dadurch ist es möglich eine Anbindung eines Sensors der das Recording Pattern unterstützt wie die Anbindung eines Sensors, der das Multiple-Dataresponse Pattern unterstützt, diese wurden in Kapitel 5.2.2 beschrieben, zu behandeln.

6.3.2 Anbinden eines Sensors

Der Vorgang zum Anbinden eines Sensors wird durch das `CommandEvent` gestartet, dieses Ereignis wird durch die Anwendung an den Sensor-Framework-Manager gesendet. Dieser reicht die Connect-Anweisung an den zuständigen Sensor-Manager weiter. Im Falle eines Internal-Sensors ist das der Internal-Manager. Der Internal-Manager erhält vom Sensor-Framework-Manager über das `InternalConnectSensorEvent` den Sensornamen des Sensors der angebunden werden soll.

Der Sensor-Manager prüft daraufhin ob ein Sensortreiber für den Sensor zur Verfügung steht, in dem er die Liste der geladenen Sensortreiber prüft. Falls ein Sensortreiber verfügbar ist, wird dieser an die `connectSensor(..)` Methode übergeben, diese prüft welches Interaction-Pattern durch den Sensortreiber unterstützt wird und startet ihn durch die zugehörige abstrakte Klasse des Interaction Patterns.

```
1 internalSingleDataRequest = (InternalSingleDataRequestPattern)sensor;
2 internalSingleDataRequest.setEventBus(intEventBus);
3 internalSingleDataRequest.start();
4 initSettingsOfSensor(internalSingleDataRequest);
5 //check if startet sensor is connected to Driver
6 waitForDriverResponse(sensor.getSensorname());
7 startedDrivers.put(sensor.getSensorname(), internalSingleDataRequest);
```

Listing 6.5: Anbindung eines Sensors mit SingleDataRequestPattern

In Listing 6.5 wird dieser Sachverhalt anhand des *Single-Datarequest Pattern* für einen internal Sensor dargestellt. Der Sensortreiber erhält eine `intEventBus` Instanz, durch diesen kommuniziert der Internal-Manager mit dem angebundenen internal Sensor. Da jeder angebundene Sensortreiber als Thread ausgeführt wird, wird dieser durch den Aufruf wie in diesem Fall `internalSingleDataRequest.start()` initialisiert.

Das Multi-Threading ist in diesem Fall notwendig, da sonst bei einer Kommunikation mit dem Sensor, der UI-Thread zu sehr blockiert wird. Die Folge daraus wäre, das die Benutzeroberfläche einer Anwendung nicht mehr auf Eingaben von dem Anwender reagiert. Wenn dies längere Zeit anhält, kann es auch zu einem Absturz der Anwendung führen, was vermeidet werden sollte. Daher wird jeder Sensortreiber und Sensor-Manager in einem separaten Thread ausgeführt, die zur Kommunikation wie bereits erwähnt, ebenfalls einen Event Bus nutzen.

Danach wird dem so gestarteten Sensortreiber die gewählte Einstellung übergeben, nach dieser dann der Sensor durch den Sensortreiber konfiguriert wird. Nach einer erfolgreichen Anbindung des Sensors, wird durch den Sensortreiber dem Sensor-Manager ein `ConnectionSuccessEvent` übermittelt. Auf dieses Ereignis wartet der Sensor-Manager in `waitForDriverResponse(..)`, falls dieses Ereignis nicht innerhalb einer gewissen Zeitperiode auftritt, wird eine Error-Nachricht an den Sensor-Framework-Manager und durch den Sensor-Framework-Manager an die Android Anwendung weitergeleitet.

6.3.3 Konfiguration

Beim Hinzufügen eines neuen Sensortreibers hat der Sensor-Manager zusätzlich die Aufgabe, die Einstellungen des Sensors zu laden und anzupassen. Um eine Konfigurationsdatei des Sensors auszulesen, nutzen die Manager den `SettingsLoader`, der für das Auffinden und auslesen dieser zuständig ist. Dazu versucht er die Konfigurationsdatei in dem `Settings`-Verzeichnis des Sensortreiber-Packages zu finden. Falls keine Konfigurationsdatei gefunden wird, geht man davon aus, dass der Sensor nicht konfigurierbar ist.

```
1 public void loadSettingsOfSensor(Sensor sensor){
2         settingsCon = sensorSettingsLoader.getSettingsOfDriver(outputPath);
3         if(settingsCon != null){
4                 this.sensorSettings.put(sensor.getSensorname(), settingsCon);
5         }else{/*no settings found Message*/}
6 }
```

Listing 6.6: Laden der Sensoreinstellungen

Wird eine Konfigurationsdatei gefunden, so wird der Inhalt der Datei durch Gson ausgelesen. Gson kann genutzt werden um eine JSON-Datei auszulesen und deren Inhalt auf ein gegebenes Java-Object zu mappen [Goo]. Weiter ist es auch möglich ein gegebenes Java-Object in eine JSON-Datei zu schreiben. Dabei wird der Inhalt der Datei in einer HashMap als Key-Value Paar gespeichert.

Die HashMap beinhaltet sowohl Default-Einstellungen als auch weitere mögliche Einstellungen des Sensors und die konkreten Werte der einzelnen Einstellungspunkte. Die so geladenen Einstellungen werden dann in einem `SettingsContainer` untergebracht, dieser dient als Zwischenspeicher für die Einstellungen des Sensors. Der `SettingsContainer` wird daraufhin zusammen mit dem Namen des Sensors in einer Liste gespeichert. Beim Anbinden eines Sensors werden initial die Default-Einstellungen durch `initSensorSettings(sensor)` des Sensors ausgelesen.

```
1 public void onEvent(InternalEvents.InternalSettingsRequestEvent event){
2         if(this.sensorSettings.get(event.getSensorname()) != null){
3                 sfEventBus.postSticky(SettingsResponseEvent(/*Settings*/));
4         }else{/*no settings available message*/}
5 }
```

Listing 6.7: Anfrage der Sensoreinstellung

Dabei werden durch den `SettingsLoader` die eigentlichen Werte der einzelnen Einstellungen ausgelesen. Diese werden in eine Liste als Key-Value Paar gespeichert und dann dem Sensortreiber übergeben. Vor der Anbindung des Sensors, können die Einstellungen durch den Anwender eingesehen und gegebenenfalls angepasst werden.

Die Abfrage der aktuellen Einstellungen erfolgt durch ein `SettingsRequestEvent`, dieses wird durch die Anwendung an den Sensor-Framework-Manager gesendet, der das Ereignis an den zuständigen Sensor-Manager weiterleitet.

Der Sensor-Manager prüft nach Erhalt des Ereignisses ob Einstellungsmöglichkeiten für den Sensor verfügbar sind. Falls Einstellungen für den Sensor verfügbar sind, dann wird ein `SettingsResponseEvent` vorbereitet und durch `sfEventBus.postSticky(..)` an den Sensor-Framework-Event Bus übergeben. Dies wird in Listing 6.7 dargestellt. Danach erhält der Sensor-Framework-Manager das Ereignis und leitet dies über den Anwendungs-Event Bus an die Anwendung weiter. Wie die Anwendung dieses Ereignis verarbeiten kann, wird in Kapitel 7 näher erläutert. Die durch den Anwender veränderten Einstellungen werden daraufhin durch ein `SetSettingsEvent` durch die darunter liegenden Schichten an den zuständigen Sensor-Manager weitergeleitet. Dieser überschreibt daraufhin die alten Default-Einstellungen im SettingsContainer vom Sensor. Beim Anbinden des Sensors werden diese Einstellungen dann übergeben.

Um die gewählten Einstellungen für den Sensor bei einem Neustart oder nach dem Pausieren der Anwendung wiederherzustellen, werden die Daten in einer `config.json` Datei persistiert. Diese wird auf dem externen Speicher in einem dafür vorgesehen Verzeichnis abgelegt. Dieses Verzeichnis trägt den selben Namen wie die Anwendung und enthält ein Unterverzeichnis für den entsprechenden Sensor. Damit die Sicherung der Einstellungen möglich ist benötigt das Sensor-Framework die *WRITE_EXTERNAL_STORAGE* Berechtigung.

6.3.4 Unterstützung der Interaction Patterns

Damit eine Interaktion mit dem Sensor reibungslos verläuft, unterstützen die Sensor-Manager die Ansteuerung der Sensoren durch die entsprechenden Ereignisse. Dies wird am Beispiel des Recording Patterns (vergleich Kapitel 5.2.2) dargestellt. Dabei wird die Aufnahme durch ein `CommandEvent` mit dem Parameter `RECORD_START` durch die Anwendung gestartet. Der Sensor-Framework-Manager sendet daraufhin ein `RecordRequestEvent` beispielsweise `InternalRecordRequest`.

```
1 public void onEvent(InternalRecordRequest event){
2 switch(event.getCommand()){
3  case RECORD_START:/*post StartRecordingEvent*/break;
4  case RECORD_PAUSE:/*post PauseRecordingEvent*/break;
5  case RECORD_RESUME:/*post ResumeRecordingEvent*/break;
6  case RECORD_STOP:/*post StopRecordingEvent*/break;
7 default:/*..*/break;}
8 }
```

Listing 6.8: Auszug aus Internal-Manager zum Recording Pattern

Das Ereignis wird durch die in Listing 6.8 aufgezeigte Methode verarbeitet. Dabei holt sich der Sensor-Manager (hier der Internal-Manager), das durch das Ereignis übergebene Sensor-Framework-Kommando und prüft welches von den Ereignissen gesendet werden soll. In dem hier beschriebenen Fall, wurde das *RECORD_START* Kommando gesendet. Somit muss der Internal-Manager ein `StartRecordingEvent` an den Sensortreiber schicken. Durch dieses Ereignis erhält der Sensortreiber zur Verifikation, dass dieses Ereignis auch ihn betrifft, den Namen des Sensors für den er zuständig ist. Zusätzlich wird auch ein sensorspezifisches Kommando mitgeliefert, welches den Sensor eine bestimmte Funktion ausführen lässt.

Nach dem `RECORD_STOP`, welches ebenfalls gleich abläuft wie das eben diskutierte *RECORD_START* Kommando, kann der Sensortreiber die Aufnahmedaten an den Sensor-Manager senden. Diese werden daraufhin durch die restlichen Schichten bis zu der Anwendung propagiert. Falls die Daten direkt durch den Sensortreiber lokal gespeichert werden, kann der Sensortreiber eine Nachricht senden, dass die Aufnahme erfolgreich beendet und wo die Daten abgelegt wurden.

6.3.5 implementierte Sensor-Manager

Bisher wurde in dem Sensor-Framework der Internal-Manager und der Bluetooth-Manager implementiert. Das Sensor-Framework ist allerdings für weitere Sensor-Manager ausgelegt. Bei den aktuell verfügbaren Sensor-Managern ist für die Benutzung des

Sensor-Frameworks etwas zu beachten. Um eine Bluetooth Anbindung zu bewerkstelligen, muss die Android Anwendung verschiedene Berechtigungen besitzen. Dazu zählt `Permission:Bluetooth`, diese muss im `Manifest` der Android Anwendung vorhanden sein. Dadurch wird die Anwendung berechtigt, auf die Bluetooth-Funktion des mobilen Endgerätes zuzugreifen. Weiter wird beim Start der Anwendung, wenn eine Bluetooth Berechtigung gegeben ist, ein Dialog geöffnet. Dadurch kann der Anwender sein mobiles Endgerät für andere in der Nähe befindliche Geräte sichtbar machen. Bei diesem Request wird auch gleichzeitig der Anwender aufgefordert, wenn das Bluetooth noch nicht aktiviert wurde, dieses zu aktivieren. Nach den Dialogen kann dann der Pairing-Prozess gestartet werden. Dazu muss jedoch auch das andere Gerät auffindbar sein, so dass die Geräte untereinander den Pairing-Prozess starten können, sobald sie sich gegenseitig entdecken. Dabei muss dann der Anwender lediglich abgleichen ob die PIN-Nummern, die die Geräte ausgetauscht haben, auch übereinstimmen und dem Pairing dann zustimmen.

6.4 Sensortreiber

Dieser Abschnitt befasst sich mit der Implementierung der untersten Schicht im Framework, den Sensortreibern. Dabei wird aufgezeigt, wie dieser den Ablauf in einer Interaktion unterstützt und wo die sensorspezifischen Code-Fragmente eingebunden werden können. Danach wird auf die gemeinsam nutzbare Funktionalität für die Serialisierung bzw. Deserialisierung eingegangen.

6.4.1 Unterstützung der Interaction Pattern

Die Unterstützung der Interaction Pattern aus Sicht eines Sensortreibers wird anhand der Kamera (CameraSensorDriver) erläutert. Dabei erbt der Kamerasensortreiber von dem `InternalSingleDataRequestPattern`. Das InternalSingleDataRequestPattern ist wie in Abbildung 6.3 ersichtlich eine abstrakte Klasse, die das `SingleDataRequestPattern` Interface implementiert. Durch die in InternalSingleDataRequestPattern implementierte `onEvent(..)` Methode erhält der Sensortreiber

ein Ereignis durch das eine Datenanfrage an den Sensor gestartet wird. Darin wird die `singleDataRequest(int)` Methode aufgerufen.

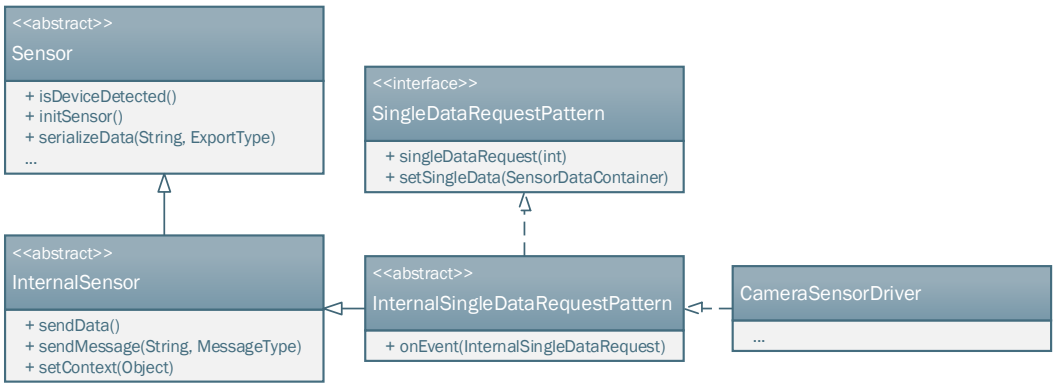

Abbildung 6.3: Vereinfachtes Klassendiagramm für einen Kamera Sensor

In der `singleDataRequest(int)` Methode kann der Sensortreiberentwickler den sensorspezifischen Code, der zur Ausführung einer Datenanfrage an den Sensor benötigt wird, einbinden. Das `InternalSingleDataRequestPattern` erbt hingegen vom *InternalSensor*, der die `sendData()` Methode für die internen Sensoren implementiert. Durch diese kann der Sensortreiber die gesammelten Daten über den Event Bus an den Internal-Manager senden. Um beispielsweise eine Error-Nachricht zu übermitteln, bietet der InternalSensor die `sendMessage(..)` Methode an. Diese erhält als Parameter die Nachricht und den MessageType. Als MessageType kann dabei `ERROR`, `WARNING` oder `INFO` angegeben werden. Verfolgt man die Vererbungshierarchie von dem InternalSensor aus weiter kommt man zum `Sensor`. Diese Klasse ist die Oberklasse aller Sensoren und bietet verschiedene Methoden an (beispielsweise die `isDeviceDetected()` Methode). Näheres dazu folgt in Kapitel 7. Des Weiteren werden in der abstrakten Sensor-Klasse die Methoden `subscribe()` (bzw. `unsubscribe()`) implementiert. Durch diese wird der Sensortreiber an der ihm übergebenen Event Bus Instanz registriert (bzw. deregistriert).

6.4.2 Serialisierung und Deserialisierung

Als Grundlage für die Serialisierung (bzw. Deserialisierung) bietet das Sensor-Framework eine gemeinsam nutzbare Funktionalität an. Diese bietet die Möglichkeit die Daten in XML-Format zu serialisieren (bzw. deserialisieren). Um dies zu nutzen kann auf die implementierte `serializeToXML(String)` Methode zurückgegriffen werden. Diese nutzt intern den implementierten Serializer. Der Serializer übernimmt die Serialisierung (bzw. Deserialisierung) der Daten vom Sensor. Die serialisierten Daten werden in einem Verzeichnis auf dem externen Speicher hinterlegt. Um die Daten zu serialisieren muss der Serializer den Namen der resultierenden Datei und den Pfad bekommen.

Um die Serialisierung (bzw. Deserialisierung) zu starten, wird von der Anwendung ein `SerializeDeserializeEvent` gesendet. Dieser wird dann durch den Sensor-Framework-Manager an den zuständigen Sensor-Manager weitergeleitet. Der Sensor-Manager prüft nach Erhalt des Ereignisses, ob ein Verzeichnis für die Serialisierung bereits vorhanden ist, falls es nicht der Fall ist, wird ein Verzeichnis durch den Sensor-Manager angelegt. Dabei wird der Anwendungsname als Verzeichnisname verwendet. Darin wird dann ein Unterverzeichnis für den entsprechenden Sensor angelegt. Um diese Verzeichnisse anzulegen und dann auch durch den Sensortreiber die Datei mit den Sensordaten abzuspeichern wird hier auch die Berechtigung zum Zugriff auf den externen Speicher benötigt.

Um die Serialisierung (bzw. Deserialisierung) zu bewerkstelligen wurde zunächst getestet ob JAXB dazu genutzt werden kann. Es wurde festgestellt, dass verschiedene für JAXB benötigte Bibliotheken in Android nicht verfügbar sind. Um diese dennoch mit einzubeziehen, müssen die Bibliotheken, die eigentlich in einer Core-Bibliothek von Java liegen, neu verpackt werden. Dazu müssen alle Packages die benötigt werden umbenannt und zur Android Anwendung hinzugefügt werden. Dabei ist jedoch außeracht gelassen, dass die Bibliothek stetig wachsen kann und somit bei jeder Änderung immer wieder neu die Bibliothek verpackt werden muss.

Dieser Ansatz wurde jedoch verworfen, da die gefundene Lösung, die hier beschrieben wird, zuletzt vor 3 Jahren erneuert wurde [Plu12]. Es wurde daraufhin nach einem möglichen Ersatz für die JAXB Bibliothek gesucht. Im Zuge dessen wurde ein SAX-Parser

entdeckt. Als SAX-Parser wird dabei die `Simple-XML` Bibliothek verwendet [Nia15]. Dabei werden zur definition der einzelnen Elemente eines *Dokument Object Model* (DOM) in der Bibliothek definierte Annotationen genutzt. Weiterhin wurde ein XML-Parser als alternative implementiert, da für die Nutzung von *Simple-XML* eine weitere eigenständige Bibliothek sowohl auf Seiten des Sensortreiberentwicklers als auch des Anwendungsentwicklers mit einbezogen werden muss. Dazu wird als Grundlage das *Dokument Object Model* genutzt, ein sprach- und plattformunabhängiges Objekt-Framework zur Manipulation von beispielsweise XML-Dokumenten [Gav01].

Dazu wird noch die Android spezifische Reflection-Bibliothek genutzt. Dabei werden einmalig für jedes Datenobjekt welches serialisiert werden soll die Methoden eines Datenobjektes ausgelesen. Diese werden dann genutzt um die einzelnen Informationen der Daten zu extrahieren. Danach werden diese als String in einem Textelement eingebettet. Dabei werden die get-Methoden des Datenobjektes genutzt um diese Informationen auszulesen. Im Gegensatz dazu werden bei der Deserialisierung die set-Methoden benutzt um die ausgelesenen Daten wieder in ein Datenobjekt zu speichern.

Im Folgenden Kapitel wird auf die Anwendungsintegration vom Sensor-Framework eingegangen. Dazu wird zunächst grob geschildert wie die Paket-Struktur vom Sensor-Framework aussieht und was in die Anwendung mit einbezogen werden muss. Danach wird auf die Implementierung eines Sensortreibers eingegangen. Daraufhin werden die Schnittstellen zur Anwendung beschrieben und wie dadurch das Sensor-Framework eingebunden werden kann.

7

Anwendungsintegration

Dieses Kapitel befasst sich mit der Einbindung des entwickelten Sensor-Frameworks in eine Android Anwendung. Dazu wird zunächst ein grober Überblick über die Paket-Struktur des Sensor-Frameworks gegeben. Dabei wird aufgezeigt was alles in die Anwendung eingebunden werden muss um das Sensor-Framework nutzen zu können. Danach wird auf die Implementierung des Sensortreibers eingegangen, diese wird anhand eines Beispiels erläutert.

7.1 Paket-Struktur

In diesem Abschnitt wird auf die grobe Paket-Struktur des Sensor-Frameworks eingegangen. Dazu wird in Abbildung 7.1 schematisch dargestellt was in dem Sensor-Framework Paket enthalten ist und was zusätzlich mit dem Sensor-Framework Paket in die Anwendung mit einbezogen werden muss um die jeweilige Funktion zu nutzen.

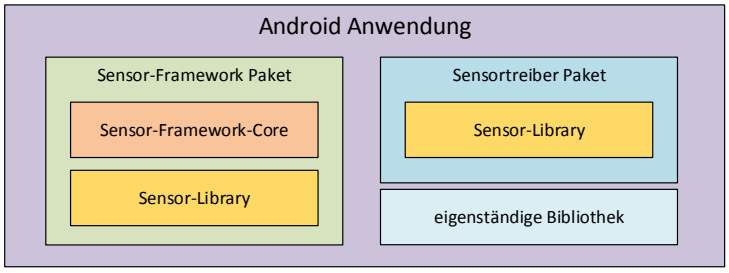

Abbildung 7.1: Vereinfachte Darstellung der gesamten Paket-Struktur

Das Sensor-Framework Paket enthält den Sensor-Framework-Core und die Sensor-Library. Dabei beinhaltet das Sensor-Framework-Core die Hauptbestandteile des entwickelten Sensor-Frameworks. Zu denen gehört der Sensor-Framework-Manager, über den alle Befehle von der Android-Anwendung verarbeitet werden. Dazu kommen der LogWriter und der Visualization-Manager, um den Visualization-Manager nutzen zu können muss noch die benötigte eigenständige Bibliothek (siehe Kapitel 6.2.3) mit eingebunden werden. Diese wird in Abbildung 7.1 als eigenständige Bibliothek dargestellt. Die weiteren Bestandteile sind die einzelnen Sensor-Manager, das sind beispielsweise der Bluetooth-Manager und der Internal-Manager. Für die Kommunikation zwischen den einzelnen Schichten, dient der Event Bus, dieser ist auch ein Bestandteil vom Sensor-Framework-Core. Die Bestandteile wurden bereits in Kapitel 6 diskutiert.

Wie bereits erwähnt ist die Sensor-Library ein Bestandteil vom Sensor-Framework Paket. Diese enthält die abstrakten Klassen, durch die auf die Funktionen des implementierten Sensortreibers zugegriffen werden kann. Die Sensor-Library muss auch, wie in der Abbildung 7.1 ersichtlich durch das Sensortreiber Paket referenziert werden. Neben dem Sensor-Framework Paket sollte die Android Anwendung auch das Sensortreiber Paket enthalten. Das Sensortreiber Paket beinhaltet den implementierten Sensortreiber, wobei ein Sensortreiber Paket auch mehrere Treiber enthalten kann. Dabei können auch mehrere gleichartige Sensoren zusammengefasst werden, beispielsweise die Sensortreiber für die beiden Kameras (auf der Vorder- und Rückseite) auf dem mobilen Endgerät.

Um diese Treiber nun zum Sensor-Framework hinzufügen zu können, enthält das Sensortreiber Paket ein `SensorListing`, welches eine Liste der Sensortreiber enthält. Wobei das `SensorListing` auch in einem Sensortreiber Paket enthalten sein muss wenn ein einzelner Sensortreiber implementiert wurde. Weiterhin enthält das Sensortreiber Paket auch eine Sensor-Daten Klasse, durch die die gesendeten Daten repräsentiert werden können und aus dem gesendeten Daten-Objekt extrahiert werden können.

7.2 Implementierung der Sensortreiber

In diesem Abschnitt wird anhand eines Beispiels erläutert, wie ein Sensortreiber implementiert wird. Dazu wird noch aufgezeigt was der Sensortreiberentwickler alles bereitstellen muss damit das möglich wird. In Abbildung 7.2 wird der `CameraSensorDriver` mit den Oberklassen dargestellt. Aus dieser ist ersichtlich welche Methoden der Sensortreiberentwickler implementieren muss. Dazu kommt das um einen Sensortreiber zum Sensor-Framework hinzuzufügen, muss ein `SensorListing` implementiert werden, dieses erbt vom gegebenen SensorListing.

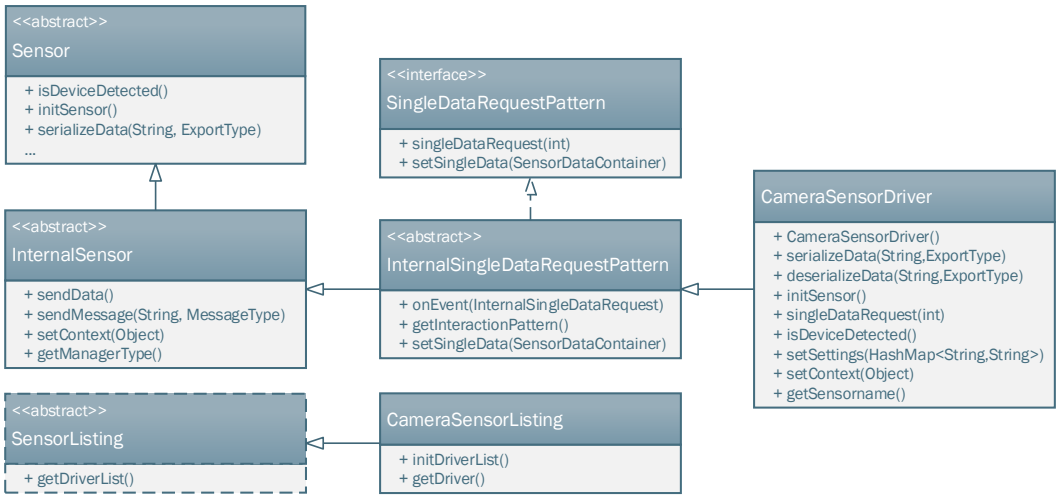

Abbildung 7.2: Vererbungshierarchie eines Kamerasensortreibers mit SensorListing

Darin soll die `initDriverList()` Methode implementiert werden, dabei wird eine `ArrayList<Sensor>` initialisiert mit den im Paket enthaltenen Sensortreibern. Falls die `initDriverList()` Methode nicht implementiert wurde, dann gibt `getDriverList()` eine leere Liste zurück. In diesem Fall wird dann auf `getDriver()` zurückgegriffen. Durch diese Methode sollte der Sensortreiberentwickler den Sensortreiber oder `null` zurückgeben. Wie schon erwähnt ist aus der Abbildung ersichtlich dass der Sensortreiberentwickler mehrere Methoden implementieren muss, diese stammen aus

den Oberklassen oder auch der Sensor-Klasse. Manche von den Methoden werden durch eine der Oberklassen implementiert, beispielsweise die `sendData()` Methode, aus der abstrakten Klasse `InternalSensor`. Es kann beispielsweise auch in der `serializeData(String,ExportType)` auf die `serializeToXML(String)` Methode zurückgegriffen werden, falls auf eine eigene Serialisierung verzichtet wird. Dazu kann auch wenn die gegebene Serialisierung genutzt wird auch die verfügbare Deserialisierung benutzt werden, auf diese kann durch `deserializeFromXML(String)` zurückgegriffen werden.

Darüber hinaus ist es auch möglich auf die Implementierung von `setSettings(...)` zu verzichten, falls der Sensortreiber keine Einstellungsmöglichkeiten anbietet. Ansonsten muss dabei nur eine `HashMap<String,String>` gesetzt werden um daraus die aktuellen Einstellungen zu konfigurieren. Daraus folgt, dass nicht viele Methoden für den Sensortreiber implementiert werden müssen. Beispielsweise wird im Falle eines Bluetooth-Sensors durch die abstrakte Klasse `Bluetooth-Sensor`, die eine Oberklasse von beispielsweise `BluetoothRecordPattern` ist, eine Anbindung für den Sensor angeboten. Weiterhin ist in diesem Fall durch die Oberklasse `InternalSensor` die Methode `getManagerType()` implementiert. Durch die `getInteractionPattern()` Methode wird definiert durch welches Interaction-Pattern mit dem Sensor kommuniziert wird, diese Methode wird in der entsprechenden Oberklasse implementiert, die durch den zu implementierenden Sensortreiber erweitert wird.

Daraus folgt, dass neben zusätzlichen Methoden für die Funktionalität der Sensortreiber nur wenige weitere Methoden implementiert werden müssen. Dabei werden in `initSensor()` die zur Anbindung des Sensors notwendigen Funktionen aufgerufen und die möglichen Einstellungen, die durch `setSettings(...)` übergeben werden, eingebunden. Durch die Methode `isDeviceDetected()` wird geprüft ob der Sensor zu dem der Sensortreiber gehört auch verfügbar ist. Dazu muss der Sensortreiberentwickler angeben wie es geprüft werden kann. Im Beispiel des Kamera Sensors wurde geprüft ob auf dem mobilen Endgerät auf dem dieser ausgeführt wurde auch eine Kamera verfügbar ist. Dadurch ist es dann auch möglich die Instanz für den Zugriff auf den benötigten Sensor zu sichern. Die eigentliche Funktionalität des Sensortreibers kann in der Methode `singleDataRequest(int)` implementiert werden. Weiterhin muss

die `getSensorname()` Methode implementiert werden, dadurch kann der zuständige Sensor-Manager und der Sensor-Framework-Manager den Sensornamen abfragen. Dieser kann als Grundlage genutzt werden um mit exakt dem Sensor durch die einzelnen Schichten zu kommunizieren.

Falls es für den Sensor einige Einstellungsmöglichkeiten gibt, muss eine `config.json` Datei zur Verfügung gestellt werden. Darin wird in Key-Value Paaren die einzelnen Optionen und die realen Werte der Einstellungen beschrieben. Diese muss im Sensortreiber Paket in einem Settings-Verzeichnis enthalten sein. Der Sensortreiberentwickler muss darüber hinaus auch eine Klasse für die Sensordaten anbieten, durch die die Informationen der Daten referenziert werden. Diese wird später auch für die Serialisierung bzw. Deserialisierung der Daten genutzt. Wenn die Sensordaten durch ein Diagramm veranschaulicht werden können, muss der Sensortreiberentwickler einige Informationen anbieten, aufgrund derer die Datenpunkte in einem Diagramm dargestellt werden können. Dazu müssen diese Informationen als Konstanten (Enum) gegeben sein. Dabei müssen die Konstanten als Werte die Bezeichner der get-Methoden enthalten. Um die Daten an den Internal-Manager zu senden, muss der Sensortreiberentwickler diese mit `setSingleData(SensorDataContainer)` setzen danach können diese durch `sendData()` an den Internal-Manager gesendet werden.

7.3 Schnittstellen zur Anwendung

An dieser Stelle werden die Schnittstellen zur Anwendung beschrieben. Dazu wird aufgezeigt, wie das Sensor-Framework in eine eigene Anwendung eingebungen werden kann. Des Weiteren wird das `SensorActivity` Interface erläutert, welches vom Anwendungsentwickler implementiert werden muss. Darüber hinaus wird auf die `ActivityEvents` Klasse eingegangen, die alle Ereignisse definiert, die zur Kommunikation mit dem Sensor-Framework benötigt werden.

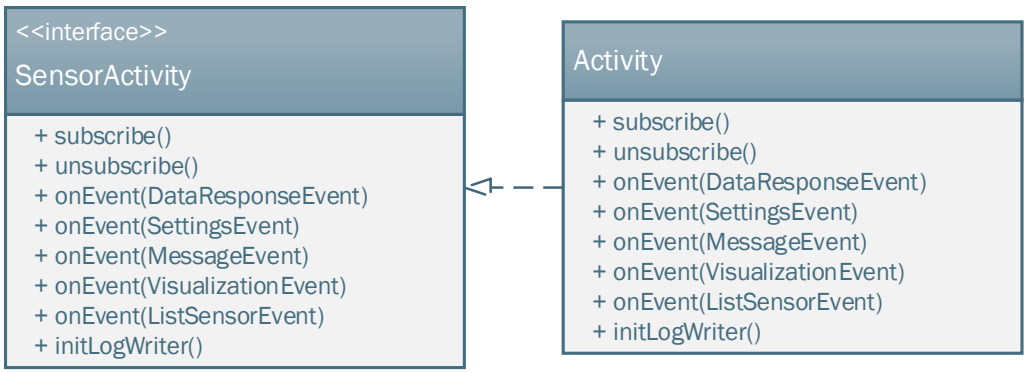

Abbildung 7.3: Interface SensorActivity

In der Abbildung 7.3 wird das Interface mit den Methoden dargestellt, die in der Android Anwendung implementiert werden müssen um die Ereignisse, die von dem Sensor-Framework an die Anwendung gesendet werden, korrekt zu verarbeiten. Damit die Anwendung mit dem Sensor-Framework über den Anwendung-Event Bus kommunizieren kann (siehe Kapitel 5.1), muss zunächst der Sensor-Framework-Manager initialisiert werden. Der Sensor-Framework-Manager benötigt eine Instanz der `Activity` in deren Kontext das Sensor-Framework ausgeführt wird. Daraufhin kann durch den Zugriff auf `getActivityEventBus()` eine Instanz des Anwendung-Event Busses geholt werden. Damit nun die Anwendung sich beim Anwendung-Event Bus registrieren kann muss die `subscribe()` Methode implementiert werden. Darin muss lediglich der Befehl zur Registrierung der Anwendung implementiert werden (siehe Kapitel 6.1.1). Nach der Initialisierung des Event Busses muss lediglich die `subscribe()` Methode aufgerufen werden, damit eine Registrierung vollzogen werden kann. Wenn die Anwendung beendet wird, kann sich die Anwendung durch `unsubscribe()` vom Anwendung-Event Bus trennen. Bevor sich die Anwendung jedoch vom Anwendung-Event Bus trennt, muss sie dafür sorgen, das sich die anderen Teilnehmer ebenfalls trennen. Dazu muss die Anwendung lediglich das `UnsubscribeEvent` über den Anwendung-Event Bus an den Sensor-Framework-Manager schicken.

Um den LogWriter nutzten zu können muss die `initLogWriter()` Methode implementiert werden. Durch diese muss ein `InitLogWriterEvent` initialisiert und gesendet werden, dieses erwartet einen Pfad in dem die Log-Datei abgelegt werden kann und den Log-Level (siehe Kapitel 6.2.2). Danach können die neuen Sensortreiber hinzugefügt werden. Dazu muss zunächst das `SensorListing` des Sensortreibers initialisiert werden und dadurch `setDriverList()` aufgerufen werden, dieses wird in Listing 7.1 dargestellt. Daraufhin kann das SensorListing durch das `LoadDriverEvent` an den Sensor-Framework-Manager gesendet werden (siehe Kapitel 6.2.1).

```
1 LocationSensorListing listing = new LocationSensorListing();
2 listing.setDriverList();
3 activityEventBus.postSticky(actEvents.new LoadDriverEvent(listing));
```

Listing 7.1: Hinzufügen eines Sensortreibers

Durch das `SettingsEvent`, das aus der Abbildung 7.3 ersichtlich auch durch eine zu implementierende Methode verarbeitet werden kann, kann die aktuelle Einstellung des gewählten Sensors erhalten werden. Um dieses zu erhalten muss initial ein `SettingsRequest` mit dem Sensornamen gesendet werden. Die Antwort enthält zwei HashMaps, mit den `default`-Einstellungen und weiteren Einstellungen des Sensors. Diese können beispielsweise durch eine ListView angezeigt werden. Um daraufhin die neuen Einstellungen an das Sensor-Framework zu übermitteln, kann das `SettingsChangeEvent` genutzt werden. Dieses erwartet als Parameter eine `HashMap<String,String>`, die die neuen Einstellungen beinhaltet und den Sensornamen um diese daraufhin dem richtigen Sensor zuordnen zu können. Damit eine Anbindung zu einem Sensor gestartet wird, kann ein `CommandEvent` mit dem Sensornamen und dem Start Kommando an den Sensor-Framework-Manager gesendet werden. Durch das `CommandEvent` können noch weitere Kommandos an den Sensor-Framework-Manager gesendet werden, das kann beispielsweise das Setzen eines neuen Log-Levels sein oder der Start einer Aufnahme. Somit kann durch das `CommandEvent` auch eine Interaktion mit einem Sensor gestartet werden. Damit die Anwendung die Daten, die durch einen Sensor gesammelt werden erhalten kann, wird die `onEvent(DataResponseEvent)` Methode angeboten. Darin muss der

Anwendungsentwickler lediglich die Daten verarbeiten. Dazu kann er auf die Daten durch die `getData()` Methode vom `DataResponseEvent` zugreifen. Dadurch wird eine Instanz der zuletzt gesammelten Daten eines Sensors übergeben. Wenn die Daten ein Format enthalten, welches durch Diagramme visualisiert werden kann, kann das `VisualizeRequestEvent` genutzt werden. Dieses erwartet als Parameter den Sensornamen, dessen Daten visualisiert werden sollen, dazu noch ein oder mehrere `VisualizationConfigs`. Durch diese werden die Layouts der Diagramme eingestellt (siehe Kapitel 6.2.3). In Listing 7.2 wird dargestellt, wie ein Diagramm eingestellt werden kann. Dazu muss zunächst ein `VisualizationConfig` Objekt initialisiert werden, in diesem Fall ist es `ScatterChartConfig`, dieses erwartet eine `Activity` Instanz und eine mögliche Dialog-Klasse. Falls jedoch keine Dialog-Klasse angegeben wird, wird der mitgelieferte `ChartValueDialog` genutzt. In diesem Beispiel wird der Wert für die X-Achse und die Y-Achse eingestellt. Danach wird noch die Farbe der dargestellten Punkte konfiguriert. Die Layout-Einstellungen des gewählten Diagramms, die nicht gesetzt wurden, erhalten `default` Werte.

```
1 ScatterChartConfig config = new ScatterChartConfig(activity,Dialog.class);
2 config.setXValue(LocationSensor.XYValues.LATITUDE.getValue());
3 config.setYValue(LocationSensor.XYValues.LONGITUDE.getValue());
4 config.setPointColor(Colors.GREEN);
5
6 actEB.postSticky(new VisualizeRequest(sensorname,config));
```

Listing 7.2: Einstellung eines Diagramms und Übergabe des Ereignisses

Zu diesen Einstellungen können auch weitere Einstellungen für die Darstellung und das Layout gesetzt werden. Dazu gehört beispielsweise die Größe der Punkte oder Form der Punkte. Als Resultat für die im Beispiel dargestellte Anfrage kann daraufhin über `onEvent(VisualizeResponse)` ein View Objekt erwartet werden, dieses kann in einer anderen View oder in einem Fragment eingesetzt werden. Darüber hinaus können alle Nachrichten, die an die Anwendung durch den Sensor-Framework-Manager gesendet werden, durch `onEvent(MessageEvent)` verarbeitet werden. Darin muss der Anwendungsentwickler implementieren, wie die jeweiligen Nachrichten verarbeitet

werden können. Das `MessageEvent` enthält eine Nachricht und den `MessageType`, anhand dessen entschieden werden kann, wie die Nachricht verarbeitet werden soll. Wie bereits erwähnt können durch das `CommandEvent` noch weitere Kommandos gesendet werden, dazu zählt auch die Anfrage zu allen Sensoren, die im Moment verfügbar sind. Die Antwort dazu kann durch das `ListSensorEvent` erwartet werden. Darin ist eine `HashMap<String,Boolean>` enthalten. Dadurch kann erfragt werden zu welchen Sensoren im Moment die Treiber verfügbar sind und welche davon schon mit dem Sensor-Framework verbunden sind. Des Weiteren kann durch die Anwendung auch gesteuert werden, ob die bisher gesammelten Daten serialisiert oder schon serialisierte Daten deserialisiert werden sollen. Dazu muss das `Serialize_DeserializeEvent` gesendet werden. Dieses erhält als Parameter ob eine Serialisierung (bzw. Deserialisierung) durchgeführt und welches Format (Beispielsweise XML oder JSON) dazu genutzt werden soll.

Im folgenden Kapitel wird eine Zusammenfassung der Ergebnisse von der Entwicklung des Sensor-Frameworks gegeben. Dazu wird erläutert, um welche Bestandteile das Sensor-Framework noch erweitert werden könnte und ein Ausblick auf weitere Fragestellungen gegeben.

8

Zusammenfassung und Ausblick

In diesem Kapitel werden verschiedene Erkenntnisse, die durch die Entwicklung eines Sensor-Frameworks für die Android-Plattform gewonnen wurden, zusammengefasst. Abschließend wird auf mögliche Erweiterungen und Anwendungen dieses Sensor-Frameworks eingegangen.

8.1 Zusammenfassung

Um eine mobile Anwendung durch weitere Daten anzureichern, können verschiedene Sensoren genutzt und angebunden werden. Bei der Entwicklung einer solchen Anwendung muss jedoch meist ein hoher Aufwand für die Ansteuerung dieser Sensoren betrieben werden. Dabei ist die Einarbeitung in die verschiedenen Verbindungsarten und in verschiedene Details der Sensoren sehr zeitaufwendig. Bei externen Sensoren ist zudem ein hoher Aufwand zu betreiben, da in diesen Fällen auf verschiedene Datenformate, Interaktionsabläufe und Verbindungsarten eingegangen werden muss. Bei der Betrachtung von verschiedenen internen Sensoren wurde unter Berücksichtigung der Android-API erkannt, dass keine Möglichkeit zur einheitlichen Ansteuerung dieser Sensoren angeboten wird. Die betrachteten Sensoren werden meist durch unterschiedlich aufgebaute Schnittstellen der Android-API angesteuert.

Im Zuge dieser Arbeit wurde ein Sensor-Framework entwickelt, welches die Implementierung von Sensorunterstützten Anwendungen erleichtern soll. Dabei kann das Sensor-Framework wie eine Bibliothek in eine bereits existierende Android-Anwendung eingebunden werden und ermöglicht den Zugriff, sowie die Anbindung verschiedener

Sensoren über eine einheitliche Schnittstelle. Um dies zu ermöglichen wurden verschiedene Konzepte bei der Entwicklung mit einbezogen. Als Grundlage für das Sensor-Framework wurde eine *ereignisgesteuerte Architektur* genutzt. Diese stellt durch das Publish-Subscribe Pattern eine erweiterbare Architektur bereit, die eine Modularisierung der einzelnen Bestandteile ermöglicht. Dabei wurde auch bei der Implementierung darauf geachtet, dass die einzelnen Bestandteile des Frameworks austauschbar sind.

Um verschiedene Interaktionsabläufe mit Sensoren zu unterstützen wurde das Konzept der *Interaction Patterns* eingeführt. Dazu wurden verschiedene abstrakte Klassen entwickelt, die genutzt werden können um neue Sensortreiber zu implementieren und mit diesen zu interagieren.

Weiterhin wurde ein *Plug-And-Play* Konzept entwickelt, durch das das Hinzufügen neuer Sensortreiber zum Sensor-Framework ermöglicht wird. Dabei wurde darauf geachtet, dass das Sensor-Framework frei von Sensorlogik ist und lediglich die Anbindung und Interaktion mit einem Sensor unterstützt. Dadurch ist es möglich, nur die benötigten Sensortreiber zu installieren und zu laden, ohne dabei das Sensor-Framework neu kompilieren zu müssen.

Außerdem wurde eine Möglichkeit zur *Visualisierung der Daten*, die ein Sensor liefert, bereitgestellt. Dabei wurden verschiedene externe Bibliotheken miteinander verglichen und eine Schnittstelle entwickelt, um die gewählte Bibliothek austauschen zu können.

Weiterhin wurde ein Ansatz entwickelt, der es ermöglicht, alle Ereignisse, die im Laufe einer Kommunikation aufkommen, auf unterschiedlichen Ebenen des Frameworks zu *protokollieren*. Dabei wurde ebenfalls auf die genutzte Event Bus Bibliothek, die das Konzept darstellt, zurückgegriffen. Des Weiteren wurde auch aufgezeigt, wie der Anwendungsentwickler das Sensor-Framework und die Sensortreiber in eine Anwendung einfach integrieren kann.

8.2 Ausblick

Das im Rahmen dieser Arbeit entwickelte Sensor-Framework kann bei der Implementierung von zukünftigen Android-Anwendungen zur mobilen Datenerhebung genutzt

werden. Es existieren jedoch noch verschiedene Ansatzpunkte um das in dieser Arbeit beschriebene Sensor-Framework zu erweitern.

Ein Ansatzpunkt ist die Erweiterung des Sensor-Frameworks um weitere Verbindungsarten. Dabei ist die naheliegendste Erweiterung eine Anbindung über USB, da jedes gängige mobile Endgerät einen solchen Anschluss besitzt. Zudem wird diese Anbindungsart neben dem Bluetooth in der Android API genau beschrieben. Dazu müsste das Sensor-Framework um einen USB-Manager erweitert und die dazu benötigten Ereignisse (zur Kommunikation mit dem USB-Manager bzw. dem USB-Sensor) implementiert werden. Weiter wäre es auch denkbar die Wifi-Anbindung mit zu integrieren, da dieser Standard eine wesentlich höhere Datenübertragungsrate als Bluetooth gewährleistet und ebenfalls von allen gängigen mobilen Endgeräten unterstützt wird. Zudem wird eine Anbindung zwischen verschiedenen Geräten über Wifi durch die Android-API unterstützt.

Eine weitere Überlegung wäre, CSV als ein weiteres Format für die Serialisierung bzw. Deserialisierung der gesammelten Daten zu verwenden. Da dieses Format durch verschiedene Programme (beispielsweise Excel), darunter auch frei verfügbare Programme, unterstützt wird. Dadurch wäre es Dritten (beispielsweise Ärzten) möglich die Daten durch ein zusätzliches Programm auszulesen und zu analysieren ohne ein mobiles Endgerät zur Verfügung haben zu müssen. Außerdem sollte das Sensor-Framework um eine mögliche Klassifizierung erweitert werden. Dazu wurde im Rahmen dieser Arbeit auf eine Klassifizierungsart eingegangen und diese beschrieben, jedoch nicht vollständig implementiert.

Weiter kann ein Versuch gestartet werden um das hier vorgestellte Konzept auf weitere mobile Betriebssysteme, wie beispielsweise Apple iOS oder Windows-Phone, zu portieren. Dabei kann getestet werden ob dieses Konzept auf den anderen Plattformen so direkt umgesetzt werden kann.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AB05] Artur H.M. ter Hofstede Alister Barros, Marlon Dumas. Service Interaction Patterns. In Fabio Casati Francisco Curbera Wil M.P. van der Aalst, Boualem Benatallah, editor, *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 302–318. Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-28238-9. DOI: 10.1007/11538394_20.

[And15] AndroidPlot - API for creating dynamic and static Charts, 2015. URL: http://androidplot.com/ (zuletzt eingesehen am 09.07.2015).

[App15] Apple Inc. The Healthkit Framework, 2015. URL: https://developer.apple.com/library/ios/documentation/HealthKit/Reference/HealthKit_Framework/ (zuletzt eingesehen am 24.06.2015).

[Blu15a] Bluetooth Special Interest Group Inc. 15 Years of Bluetooth Technology, 15th Anniversary, 2015. URL: http://www.bluetooth.com/Pages/15-Years-of-Bluetooth-Technology.aspx (zuletzt eingesehen am 14.06.2015).

[Blu15b] Bluetooth Special Interest Group Inc. Core System Architecture - Core System Protocols and Signaling, 2015. URL: https://developer.bluetooth.org/TechnologyOverview/Pages/Core.aspx (zuletzt eingesehen am 02.07.2015).

[Blu15c] Bluetooth Special Interest Group Inc. Logical Link Control and Adaptation (L2CAP) Architecture, 2015. URL: https://developer.bluetooth.org/TechnologyOverview/Pages/L2CAP.aspx (zuletzt eingesehen am 14.06.2015).

[Blu15d] Bluetooth Special Interest Group Inc. Profiles Overview - Adopted Bluetooth Profiles, Services and Protocols, 2015. URL: https://developer.bluetooth.org/TechnologyOverview/Pages/Profiles.aspx (zuletzt eingesehen am 02.07.2015).

[Blu15e] Bluetooth Special Interest Group Inc. Service Discovery Protocol - Universally Unique Identifier, 2015. URL: https://www.bluetooth.org/en-us/specification/assigned-numbers/service-discovery (zuletzt eingesehen am 14.06.2015).

[Blu15f] Bluetooth Special Interest Group Inc. Technology Overview RFCOMM with TS 07.10, 2015. URL: https://developer.bluetooth.org/TechnologyOverview/Pages/RFCOMM.aspx (zuletzt eingesehen am 14.06.2015).

[Bre06] Brenda M. Michelson. Event-Driven Architecture Overview - Event-Driven SOA Is Just Part of the EDA Story. *Patricia Seybold Group*, 2, Februar 2006. DOI: 10.1571/bda2-2-06c.

[Bri97] Brian P Crow, Indra Widjaja, Jeong Geun Kim and Prescott T Sakai. IEEE 802.11 Wireless Local Area Networks. *Communications Magazine, IEEE*, 35(9):116–126, September 1997. DOI: 10.1109/35.620533.

[Dyn14] Dynastream Innovations Inc. ANT Message Protocol and Usage, 2014. Rev. 5.1.

[Fit15] Fitbit, Inc. FitBit, c2015. URL: http://www.fitbit.com/de (zuletzt eingesehen am 22.07.2015).

[Gab13] Gabriele Spina, Frank Roberts, Jens Weppner, Paul Lukowicz, Oliver Amft. CRNTC+: A smartphone-based sensor processing framework for prototyping personal healthcare applications. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2013 7th International Conference on*, pages 252–255, Mai 2013.

[Gav01] Gavin Nicol, Lauren Wood, Mike Champion and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification. *W3C Working Draft*, 13:1–146, 2001.

[ges14] Gesundheits-Apps: Mobile Health von HealthKit bis Hardware, 2014. URL: https://www.axa.de/das-plus-von-axa/gesund-und-reisen/selftracking-gesundheits-app (zuletzt eingesehen am 30.06.2015).

[Goo] Google, Inc. google-gson - A Java library to convert JSON to Java objects. URL: https://github.com/google/gson (zuletzt eingesehen am 26.06.2015).

[Goo15a] Google Inc. Dalvik System DexClassLoader - Loads classes from JAR and APK Files to execute, 2015. URL: http://developer.android.com/reference/dalvik/system/DexClassLoader.html (zuletzt eingesehen am 09.07.2015).

[Goo15b] Google Inc. Reflect API, 2015. URL: http://developer.android.com/reference/java/lang/reflect/package-summary.html (zuletzt eingesehen am 09.07.2015).

[Goo15c] Google Inc. Sensors Overview, 2015. URL: http://developer.android.com/guide/topics/sensors/sensors_overview.html (zuletzt eingesehen am 02.07.2015).

[Gre15a] Greenrobot. Event Bus Comparison - Comparison with Square's Otto, 2015. URL: https://github.com/greenrobot/EventBus/blob/master/COMPARISON.md (zuletzt eingesehen am 28.05.2015).

[Gre15b] Greenrobot. Event Bus for Android, 2015. URL: https://github.com/greenrobot/EventBus (zuletzt eingesehen am 28.05.2015).

[HNO] HNO-Ärzte im Netz. Kostenlose LärmApp zeigt akute Lärmbelastung an. URL: http://www.hno-aerzte-im-netz.de/news/neue-laermapp.html (zuletzt eingesehen am 30.06.2015).

[Hug09] Hugh Taylor, Angela Yochem, Les Phillips, Frank Martinez. *Event-driven architecture: how SOA enables the real-time enterprise*. Addison-Wesley, Upper Saddle River, N.J., c2009. ISBN: 978-0-321-59138-8.

[IEE02] IEEE Computer Society. IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Specific Requirements - Part 15: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs). *IEEE Std 802.15.1-2002*, pages 1–473, Juni 2002. DOI: 10.1109/IEEESTD.2002.93621.

[Jan15] Jan Axelson. *USB Complete: The Developers Guide*. Lakeview Research, März 2015. ISBN: 978-1-931448-28-4.

[Joh08] John Kooker. Bluetooth, Zigbee, and Wibree: A Comparison of WPAN Technologies. *CSE 237A*, November 2008.

[Joh13] Johannes Schobel and Marc Schickler and Rüdiger Pryss and Hans Nienhaus and 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*, pages 509–518, May 2013.

[Nia15] Niall Gallagher, Slashdot Media. Simple XML Serialization, c2015. URL: http://simple.sourceforge.net/home.php (zuletzt eingesehen am 22.07.2015).

[Nor08] Norbert Schwesinger, Carolin Dehne, Frederic Adler. *Lehrbuch Mikrosystemtechnik*. Oldenbourg Wissenschaftsverlag, München, Dezember 2008. ISBN: 978-348657929-1.

[Pao07] Paolo Baronti, Prashnat Pillai, Vince W.C. Chook, Stefano Chessa, Alberto Gotta, Y. Fun Hu. Wireless sensor networks: A survey on the state of the art and the 802.15. 4 and ZigBee standards. *Computer Communications*, 30(7):1655–1695, 2007.

[Pat03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[Pat13] Patrick Zeller. Konzeption und Realisierung eines Sensor-Frameworks für mobile Anwendungen und Integration von Sensorinformationen am Beispiel einer mobilen Fragebogen-Applikation. Master's thesis, University of Ulm, November 2013. URL: http://dbis.eprints.uni-ulm.de/1008/.

[Phi15] Philipp Jahoda. MPAndroidChart - A powerful Android Chart View Library, 2015. URL: https://github.com/PhilJay/MPAndroidChart (zuletzt eingesehen am 09.07.2015).

[Plu12] Plutext Pty Ltd. JAXB can be made to run on Android, 2012. URL: http://www.docx4java.org/blog/2012/05/jaxb-can-be-made-to-run-on-android/ (zuletzt eingesehen am 11.07.2015).

[Pre13] Prem Prakash Jayaraman, Charith Perera, Dimitrios Georgakopoulos, Arkady Zaslavsky. Efficient opportunistic sensing using mobile collaborative platform MOSDEN. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 77–86, October 2013.

[Rak15] Rakshit Wadhwa, Pushpendra Singh, Meenu Singh, Saurabh Kumar. An EMR-enabled medical sensor data collection framework. In *Communication Systems and Networks (COMSNETS), 2015 7th International Conference on*, pages 1–6, January 2015.

[Roh12] Rohit Chaudhri, Waylon Brunette, Mayank Goel, Rita Sodt, Jaylen VanOrden, Michael Falcone, 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*, page 9. ACM, 2012.

[Roy00] Roy Thomas Fieldings. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[Sau13] Martin Sauter. *Grundkurs Mobile Kommunikationssysteme - UMTS, HSPA und LTE, GSM, GPRS, Wireless LAN und Bluetooth*. Springer Fachmedien Wiesbaden, 2013. ISBN: 978-3-658-01460-5. DOI: 10.1007/978-3-658-01461-2.

[Snu14] Snuti Kumari, Garima Rathi, Priyanka Attri, Manee Kumar. Types of Sensors and Their Applications. In *International Journal of Engineering Research and Development*, pages 72–85, April 2014.

[Sta15] Statista. Marktanteile der führenden mobilen Betriebssysteme an der Internetnutzung mit Mobilgeräten weltweit von Januar 2009 bis Mai 2015 , 2015. URL: http://de.statista.com/statistik/daten/studie/

184335/umfrage/marktanteil-der-mobilen-betriebssysteme-weltweit-seit-2009/ (zuletzt eingesehen am 30.06.2015).

[The13] The 4ViewSoft Company. AchartEngine - A charting Software Library for Android, c2009-2013. URL: http://www.achartengine.org/ (zuletzt eingesehen am 09.07.2015).

[Tho11] Thomas Schmitz. Entwicklung einer mobilen Software zum Steuern und Überwachen von Wohnungstüren auf Basis von Android im Umfeld von Ambient Assisted Living. *Praxisprojekt, Fachhochschule Düsseldorf*, 2011.

[USB] USB Implementers Forum Inc. SuperSpeed USB 10Gbps (USB 3.1 Gen 2) from the USB-IF. URL: http://www.usb.org/developers/ssusb/ (zuletzt eingesehen am 16.06.2015).

[USB99] USB Implementers Forum Inc. *Universal Serial Bus Class Definitions for Communication Devices*, Januar 1999. Version 1.1.

[Ved13] Vedat Coskun, Busra Ozdenizci, Kerem Ok. A Survey on Near Field Communication (NFC) Technology. *Wireless personal communications*, 71(3):2259–2294, 2013. ISSN: 0929-6212. DOI: 10.1007/s11277-012-0935-5.

[Way13] Waylon Brunette, Mitchell Sundt, Nicole Dell, Rohit Chaudhri, Nathan Breit and Gaetano Borriello. Open Data Kit 2.0: Expanding and Refining Information Services for Developing Regions. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, pages 10:1–10:6, New York, NY, USA, 2013. ACM. ISBN: 978-1-4503-1421-3, DOI: 10.1145/2444776.2444790.

[WIB11] WIBU-SYSTEMS AG. VitaBIT Pflegeservice von morgen bereits heute im Einsatz, c2007-2011. URL: http://www.vitabit.org/ (zuletzt eingesehen am 30.06.2015).

[Wit15] Withings SA. Withings inspire Health, c2009-2015. URL: http://www2.withings.com/eu/de/ (zuletzt eingesehen am 22.07.2015).

Name: Artur Jabs Matrikelnummer: 617893

Erklärung

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

Ulm, den .....

Artur Jabs