Nidra

An Android application designed to record sleep with Flow sensors

Jagat Deep Singh

Thesis submitted for the degree of
Master in Network and system administration
30 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

Nidra

An Android application designed to record sleep with Flow sensors

Jagat Deep Singh

Nidra

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Leo a diam sollicitudin tempor. Elit ullamcorper dignissim cras tincidunt lobortis feugiat vivamus. Consectetur a erat nam at lectus urna duis. Ullamcorper sit amet risus nullam eget felis eget nunc. Sollicitudin nibh sit amet commodo. Volutpat maecenas volutpat blandit aliquam etiam erat. Sed viverra ipsum nunc aliquet bibendum enim facilisis gravida neque. Metus dictum at tempor commodo. Nunc vel risus commodo viverra maecenas accumsan lacus. Vitae justo eget magna fermentum iaculis eu non diam. Habitant morbi tristique senectus et. Sed enim ut sem viverra aliquet. Lectus mauris ultrices eros in cursus.

Turpis massa tincidunt dui ut ornare lectus. Elit sed vulputate mi sit amet mauris commodo quis imperdiet. Etiam non quam lacus suspendisse faucibus interdum posuere lorem ipsum. Morbi non arcu risus quis. Quis viverra nibh cras pulvinar mattis nunc sed. Tellus cras adipiscing enim eu turpis egestas. Nec tincidunt praesent semper feugiat nibh sed. Ipsum dolor sit amet consectetur. Duis convallis convallis tellus id interdum. Nulla aliquet enim tortor at.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Leo a diam sollicitudin tempor. Elit ullamcorper dignissim cras tincidunt lobortis feugiat vivamus. Consectetur a erat nam at lectus urna duis. Ullamcorper sit amet risus nullam eget felis eget nunc. Sollicitudin nibh sit amet commodo. Volutpat maecenas volutpat blandit aliquam etiam erat. Sed viverra ipsum nunc aliquet bibendum enim facilisis gravida neque. Metus dictum at tempor commodo. Nunc vel risus commodo viverra maecenas accumsan lacus. Vitae justo eget magna fermentum iaculis eu non diam. Habitant morbi tristique senectus et. Sed enim ut sem viverra aliquet. Lectus mauris ultrices eros in cursus.

Turpis massa tincidunt dui ut ornare lectus. Elit sed vulputate mi sit amet mauris commodo quis imperdiet. Etiam non quam lacus suspendisse faucibus interdum posuere lorem ipsum. Morbi non arcu risus quis. Quis viverra nibh cras pulvinar mattis nunc sed. Tellus cras adipiscing enim eu turpis egestas. Nec tincidunt praesent semper feugiat nibh sed. Ipsum dolor sit amet consectetur. Duis convallis convallis tellus id interdum. Nulla aliquet enim tortor at..

Contents

List of Listings vi

List of Tables vii

List of Figures ix

I Introduction and Background 1

1 Introduction 3
1.1 Background and Motivation 3
1.2 Problem Statement 3
1.3 4
1.4 Research Methods 5
1.5 Contributions 5
1.6 Thesis Outline 5

2 Background 7
2.1 Project Architecture 7
2.2 MVVM 7
2.3 Sleep Apnea 7
2.4 Flow Wrapper 7
2.5 Bluetooth LE 7
2.6 Android OS 7
2.7 Daniel 7
2.8 Viet 7
2.9 Svein 7

3 Related Work 9
3.1 Extensible Data Acquisition Tool 9
3.2 Extensible Data Stream Dispatching Tool 11
3.3 Database Model for Storing OSA Data 14

II Design and Implementation 21

4 Analysis and High-Level Design 23
4.1 Requirement Analysis 24

4.1.1 Stakeholders . . . 24
4.1.2 Resource Efficiency . . . 24
4.1.3 Security and Privacy . . . 24
4.2 High-Level Design . . . 25
4.2.1 Task Analysis . . . 25
4.3 Seperation of Concerns . . . 27
4.3.1 Recording . . . 27
4.3.2 Sharing . . . 30
4.3.3 Modules . . . 32
4.3.4 Analytics . . . 33
4.3.5 Storage . . . 34
4.3.6 Presentation . . . 36
4.4 Data Structure . . . 38
4.4.1 Data Formats . . . 38
4.4.2 Data Tables . . . 40
4.4.3 Data Packets . . . 43

5 Implementation **45**
5.1 Architecture (MVVM) . . . 45
5.2 IPC? . . . 45
5.3 Permissions & Android Manifest . . . 45
5.4 Implementation of Concerns . . . 45
5.4.1 Recording . . . 45
5.4.2 Sharing . . . 45
5.4.3 Modules . . . 45
5.4.4 Analytics . . . 50
5.4.5 Storage . . . 50
5.4.6 Presentation . . . 50

III Evaluation and Conclusion **53**

6 Experiments **55**

7 Future Work **57**

8 Conclusion **59**

Appendix **60**

A Power Data **63**

Listings

4.1 My Caption 39
4.2 My Caption 39
4.3 My Caption 43
5.1 My Caption 48

List of Tables

4.1 Example entry in record table 41
4.2 Example entry in record table 41
4.3 Example entry in record table 42
4.4 Example entry in record table 42

List of Figures

1.1 Structure of the project, separating functionality into three independent layers [3] 4

3.1 Sharing the collected data between multiple applications [11] 11
3.2 Sharing the collected data between multiple applications [3] 13
3.3 Binary, recursive and n-ary relationships [18] 17
3.4 Logical model of the OSA database - Person and Clinic [18] . 18
3.5 Logical model of the OSA database - Source and Recording [18] 19

4.1 Recording 25
4.2 Recording 28
4.3 Sharing 31
4.4 Modules 32
4.5 Analytics 34
4.6 Storage 35
4.7 Presentation 37
4.8 Modules 40

5.1 Implementation of recording functionality (A) 46
5.2 Implementation of recording functionality (B) 46
5.3 Implementation of recording functionality (C) 47
5.4 Implementation of sharing functionality (A) 47
5.5 Implementation of sharing functionality (B) 48
5.6 Implementation of module functionality(A) 49
5.7 Implementation of module functionality(B) 49
5.8 Implementation of module functionality(C) 50
5.9 Implementation of analytics functionality 51
5.10 Entity Relationship Diagram 51

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Background and Motivation

The CESAR project aim to use low cost sensor kit to prototype applications using physiological signals related to heart rate, brain activity, oxygen level in blood to monitor sleep and breathing related illnesses, like Obstructive Sleep Apnea (OSA). Side effects of OSA do not only cause sleepiness during day time (which might affect daily chores), but also serious illnesses like diabetes and cardiac dysfunctions. Statistically speaking, it is estimated that about 25% of the adult population in Norway has OSA, but only 10% of them are diagnosed. A major problem with diagnosing OSA is polysomnography in *sleep laboratories* [17]. This is both really expensive and inefficient due to lacking capacity to perform sufficient tests with patients. Hence, the CESAR project aims to contribute to this situation with a low-cost Android and BiTalino based system to tackle these problems in a minimal invasive approach.

The project has been developed by various people over the years, and the system has been divided into three parts (illustrated in Figure 1.1). The data acquisition part, the data streams dispatching part, and the application part. The first two parts are already implemented (summarized in the section below), thus, the last part is what we will be focusing throughout this thesis.

1.2 Problem Statement

As indicated in the background and motivation section, we decided to look into there are opportunity of extending the system even further. The market has new and affordable sensors that can aid with the data acquisition, which we seamlessly can integrate with the extensible data acquisition tool. In the end, we can hopefully strengthen the detection of

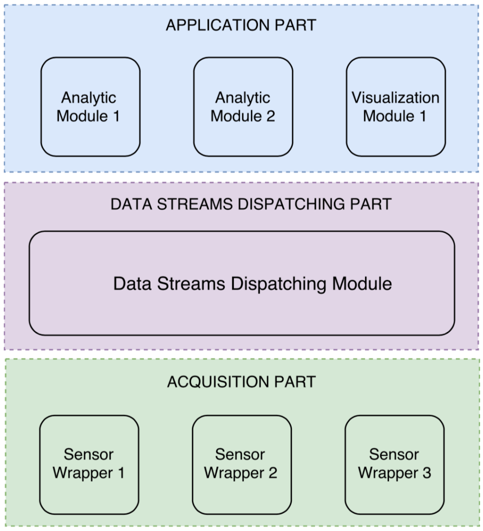

Figure 1.1: Structure of the project, separating functionality into three independent layers [3]

abnormal sleeping patterns, and decreasing the risk of the symptoms they may endure.

In this thesis, we will continue to build on the project by implementing the following:

1. *Integrate new a sensor wrapper*
   The following section Flow Sensor Kit introduces to a new sensor type to be used in the CESAR project. Thus, creating a new wrapper to integrate it in the system is necessary. By utilizing the library that comes with the sensor, we can integrate that with the interface and protocol of the CESAR project.

2. *Visualization of the activities on an Android device*
   Designing, architecting, and modulating an Android application that implements the new sensor data in an adequately layout.

3. *Detecting anomalies with the help of Machine Learning*
   Classifying the data so we can detect abnormal sleeping patterns, and hopefully training an model that can detect the sleeping patterns, locally on the device, without any external supervisor (human intervention) analyzing the data.

1.3

Limitations

1.4 Research Methods

1.5 Contributions

1.6 Thesis Outline

Chapter 2

Background

2.1 Project Architecture

2.2 MVVM

2.3 Sleep Apnea

2.4 Flow Wrapper

2.5 Bluetooth LE

2.6 Android OS

2.7 Daniel

2.8 Viet

2.9 Svein

Chapter 3

Related Work

This chapter survyes previous work related to development of the CESAR project.

3.1 Extensible Data Acquisition Tool

In the thesis "Extensible data acquisition tool for Android" by Svein Petter Gjøby [11], we are proposed a *data acquisition system* for Android to make application development comprehensible. The thesis proposes a system that hides the low-level sensor specific details into two components, *providers* and *sensor wrappers*. The provider is responsible for the functionality that is common for all data sources (e.g., starting and stopping the data acquisition), whilst the sensor wrapper is responsible for the data source specific functionality (e.g., communicating with the data source).

The thesis solves the difficulties around creating an extensible data acquisition tool, connecting new and existing sensors, and finding a common interface. The problem statement of the thesis address the following concerns regarding sensors:

- *Common abstraction/interface for the interchanged data*
  Sensor platform manufacturers have their own low-level protocol to support the functionality of their product. Typically, the manufacturers provide an software development kits (SDKs) to hide the low-level protocols so third-party development can be easier, however, both the low-level protocol and the SDKs are not standardized. Thus, for each sensor there might be exposure of different commands and methods.

- *Various Link Layer technologies*
  Each sensor might use different Link Layer technology (i.e., Ethernet, USB, Bluetooth, WiFi, ANT+ and ZigBee), which means establishing a connection between a device and a sensor might differ. For instance,

Bluetooth devices need to be paired, whilst devices on the WiFi can address each other without any pairing.

- *Reusability of sensor code*
  Applications that implement support for the low-level protocol of a sensor type can not be shared between different applications. Thus, introducing duplicate work and code if multiple application wish to use the same sensor type. A framework that isolates the sensor that applications can use, might make it easier for application to utilize the collected data. In addition, isolating the sensors into modules improves the robustness and quality of the implementation.

In the thesis, the goal is to develop an extensible system, which enables applications to collect data from various external and built-in sensors through one common interface. The solution around an extensible system is to have the core of the application unchangeable when adding support for a new data source, regardless of the Link Layer technology and communication protocol used by the data source. Making all the data sources behave as the same, is a naive solution to the problem. However, separating the software into two different components, a *provider* component and a *sensor wrapper* component, enables the reuse of functionality that is common amongst the data sources.

The sensor wrapper application is tailored to suit the Link Layer technology and data exchange protocol of one particular data source. Additionally, responsible for connectivity and communication with the data source. The provider application is responsible for managing the sensor wrappers - starting and stopping the data acquisition - and processing the data received from the sensor wrapper application. Thus, everything that is independents of the data source, should be a part of the provider application. With this type of solution, we gain the possibility to reuse the sensor wrapper application for different provider applications. However, there are some overheads with this solution. Mostly, the interprocess communication that might be costly and increase the complexity of the code. Nonetheless, the flexibility and extensibility gained by the separating the functionalists out weights the cost.

When a connection is established with the provider application, a package of metrics and data type (all the data does not change during the acquisition as metadata) is sent describing the context of the data collected. The metadata is necessary because different sensors might sample data in different environments, and some applications are depending on recognizing the environment of the data acquisition. Therefore, it is critical to know what data values are measured. Consequently, exposing sensors and data channels through one common interface requires a field of metadata which can be used to: (1) *distinguish* sensor wrapper and data channel the data originated from; (2) determine the *capabilities* of the sensors (i.e., EEG, ECG, LUX); (3) determine the *unit* the data is represented in (i.e., for temperature, Celsius or Fahrenheit); (4) describing the data channel (i.e., placement of the sensor); and (5) a time stamp of when the

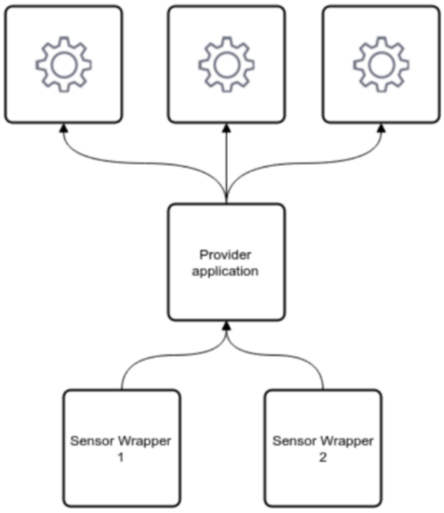

Figure 3.1: Sharing the collected data between multiple applications [11]

data was sampled.

To summarize, the task of a *sensor wrapper* is to establish a connection to and collect data from exactly one specified data source, and to send the collected data to the *provider application* that is listening for it. A data source (e.g., BiTalion) can have support for multiple sensor attachments (defined as data channel in the thesis), although, only on sensor wrapper is necessary for each data source and their data channels. Each sensor wrapper is tailored to adapt to the data source's Link Layer technology and the communication protocol of a respective data source. Upon activation by a provider application, the data is collected by the sensor wrapper, and pushed to the provider application in a JSON-format. An illustration of the structure is visualized in Figure 3.1.

3.2 Extensible Data Stream Dispatching Tool

The extensible data acquisition tool developed by Gjøby leaves some space for improvements. Such improvements are discussed in the thesis "Extensible data streams dispatching tool for Android" by Daniel Bugajski [3]. Bugajski analyses the potential improvements of the data acquisition tools, which can be extracted into:

- *Lack of reusability* - only the components that have started the collection can receive the data, and no other components can access the collected data.
- *Lack of sharing* - components that perform specific analysis on the collected data in real-time, have no way of share the results of the analysis such that other components can use them.

- *Lack of tuning* - it is not allowed to change the frequency of collection after the start. Thus, the user has to stop the collection and manually change the frequency of the sensor and then restart the collection.

- *Lack of customization* - the set of channels can not be changed during a collection, and the collector receives data from all channels even if it needs only one of them. Thus, the data packet size and resource usage become larger than necessary.

In the thesis, the modularity of the architecture is improved by extracting the functional requirement of the , and determining the responsibility of each element by. First, finding a model of all available data channels should be implemented. Then, developing a mechanism for cloning a data packet to allow reusing of data across modules. Finally, letting the modules have support for choosing channels they want to receive data from and publish their own data to. In the model, these components are distinguished as:

1. *sensor-capability model* - is a representation of all distinct data types and contains all information about the channel. A sensor board usually reads and sends different type of data to a mobile device. Thus, this module is used to control every available data type, such that they can be access from the application part at anytime.

2. *demultiplexer* (DMUX) - is a data cloner, that receives data packets from one input (i.e., from one channel), and duplicates the data a number of times - based on number of subscribers.

3. *publish-subscribe* mechanism - is an interface responsible for providing possibilties of becoming a subscriber or a publisher, in addition to be able to terminate these statuses. Additionally, every module from the application will be able to see all capabilities represented by this component, enabling the option to choose a frequency the data should be collected with.

The combination of how these modules cooperate and communicate with each other, affects the modularity and performance of the architecture. In the thesis, there are various proposed solutions. The naive solution, were to fit all of the elements into each respective sensor wrapper, thus, prioritizing the performance and low resource usage, but making it impossible to distinguish data type from two sensor wrapper with same data type. This is optimal for the cases where collection only occurs on one sensor board. An improved solution, includes to place the demux between the application part and the wrapper layer and inserting the remaining elements in the respective application part. This solution resolves the overload of sensor wrappers performing other task besides collecting data, thus, the wrappers are untouched and they send the data to the demux. However, there are several issues with this solution, e.g., due to various obstacles such as: (1) every application module has to configure its own sensor-capability model; (2) filter requested data packets from all the channels; (3) and deal with the collection speed on its own.

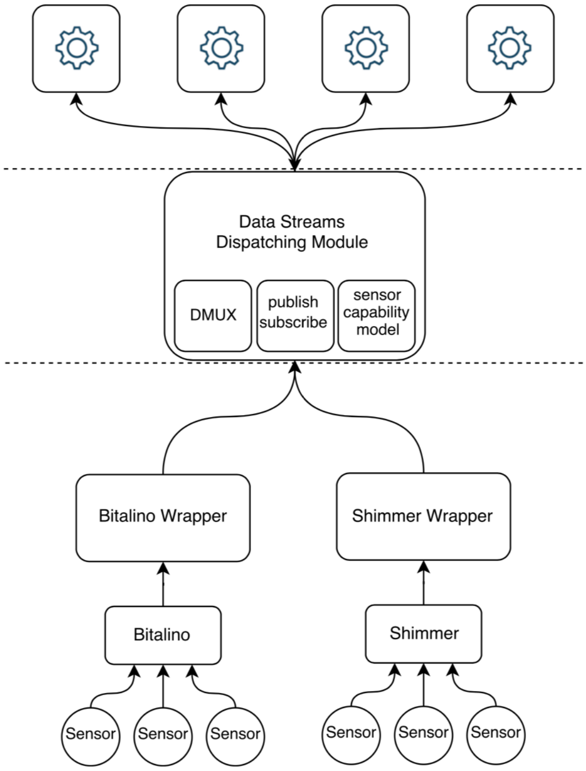

Figure 3.2: Sharing the collected data between multiple applications [3]

Addressing these issues leads us to the final architecture, which is presented in Figure 3.2, and meets the demands identified in the requirements. In this solution, all elements are placed between the application part and the wrapper layer, forming the data stream dispatching module. The sensor wrapper connects directly to the data streams dispatching module, the module discovers all installed wrappers and populates the sensor-capability model with the data types from all installed sensors. By this, all applications can access a shared sensor-capability mode. A publish-subscribe mechanism enables application modules to subscribe to any capabilities with a preferred sampling rate. Correspondingly, an application module can publish data to other applications through the same interface. The demultiplexing element creates for each subscriber a copy of the data packet.

These three elements together establish *the data stream dispatching module*. The final architecture has a couple of advantages, such as it is very extensible due to its maintainability. For instance, all communication with other layers occurs through one interface, this way, new instances can be added at any time, without the need of modifying large parts of the system. The system is also very effective, by the means of packets are immediately sent to the application on request (without any buffers), and packets are only sent to the application requesting them, resulting in less resource and power usage, and more battery.

3.3 Database Model for Storing OSA Data

A database for storing Obstructive Sleep Apnea related data is discussed in the thesis "An open database model for storing Obstructive Sleep Apnea data" by Viet Thi Tran [18]. The thesis presents the design and implementation of a relational *database model* for **storing** OSA signals and bio-physiological signals, simplify the **analysis** of the signals, and supporting acquisition from **future data sources**.

In terms of storing data in a database system, the context of what the database is used for, and what it must contain, are usually crucial to identify the appliance of the database. For storing OSA related data, the database system must satisfying the requirements of the sensor sources, and requirements of the users of the system. The users in the system are *patients, physicians, and researchers*. A few characteristics of the actions the users can take are:

1. *Patients* – the users of the group are able to execute a simple function (i.e. inserting, deleting and queries). Such function might be finding records, storing sample from CESAR tool, and import/exporting certain recordings. Mostly, their action are to import and to export.

2. *Physicians* – are able to apply functions that modify existing data, manually training data for a recording, retrieve all recording of patients, etc... In other words, they have knowledge of OSA health data and may tweak values based on their expertise in the field.

3. *Researchers* – often evaluates tasks on the system to find the best solution. Thus, they would most likely want to evaluate the quality of the source that is used for collecting the signals. Some actions they can take are to evaluate quality of sources and channels, perform raw query to find a cost and performance beneficial query, and applying possible mining algorithms.

The benefits of using a relational database to store the OSA related signals are: *data analysis* can be performed on client's device (e.g. mobile devices) by utilizing SQL and its supporting algorithms; *remote services* to fetch parts of the client's data by using remote querying; and *privacy of patients* is not violated as the data remains on their devices and they can decide which data they would like to share.

A logic data model describes the abstract structure of the database system without considering physical structure, i.e., how the database is implemented. The features of a logical data model includes: *tables (entities)* and all relations between them; *columns (attributes)* for the entities; *primary key* for all of the entities; and *foregin key* for identifying relations between different entities.

Before analyzing the proposed data model of the system, we identify the entities that are part of the system:

- **Source** – is the entity storing the data obtained from CESAR acquisition tool, in addition to EDF files. Mainly, the entity stores the bio-signals data from sensor sources (such as BiTalino).
- **Recording** – is a session of sample recorded on the *users* device.
- **Person** – is both the *Patient* and the *Physician* (however, I could not find any listing of *Researchers* in the thesis), because these two entities share common attributes.
- **Clinic** – provides advanced diagnostic or treatment services for specific diseases. The bio-physiological signals are usually stored in formats tailored specific to a clinic. Thus, the clinics have their own way of formatting and manipulating the bio-physical signals to their standards, and the data may be varying between clinics.

The actual data model, we can define the relationship between the entities based on the identified requirements. In Figure 3.3, we have a logical data model for the entities in the system, based on the following relations:

a) Each Source has many Recordings, but one Recording belongs to only one Source.

b) Each Source can be used by many different Persons, and each Person can use many different Sources.

c) Each Source can be used by many different Clinics, and each Clinic can use many difference Sources.

d) Person has many Recordings, but one Recording belongs to only one Person.

e) Person collects many Recording, but one Recording is collected by only one Person.

f) Each Recording is produced by a Source, for a Person at a certain Clinic at a certain time.

g) Each Person works/belongs to many Clinics, and each Clinic employs/has many Persons.

h) Each Person (Physician) observe many Persons (Patient), and many Person (Patient) are observed by a Persons (Physician).

Normalization is a technique in relational database used for integrity, maintainability and performance of database. The purpose is to reduce the redundancy of the data stored in a database system, so the database can become more reliable and efficient. The table of data can be classified on: (1NF) first normal form; (2NF) second normal form; (3NF) third normal form; (BCNF) Boyce-Codd normal form; and (4NF) fourth normal form. Where a higher degree of a normal form is preferable.

In the thesis, the entities (source, recording, person, and clinic) of the system have their normal form evaluated by determining the functional dependencies and the multivalued dependencies based on the attributes of the entity. The normal form can then be derived by applying set of algorithms and rules (i.e., determining the normal form of the FDs/MVDs and decomposing it until the tables is on BCNF/4NF). By doing this, we ensure the data to be loss less on joins, in addition to reducing the redundant storage. The end result of the following steps, splits some of the entities into smaller groups of entities with common attributes:

The final result of the logical data model is presented in Figure 3.4 and 3.5. Theoretically, the model can be implemented and deployed on a database management system, which then can be used to store Obstructive Sleep Apnea data.

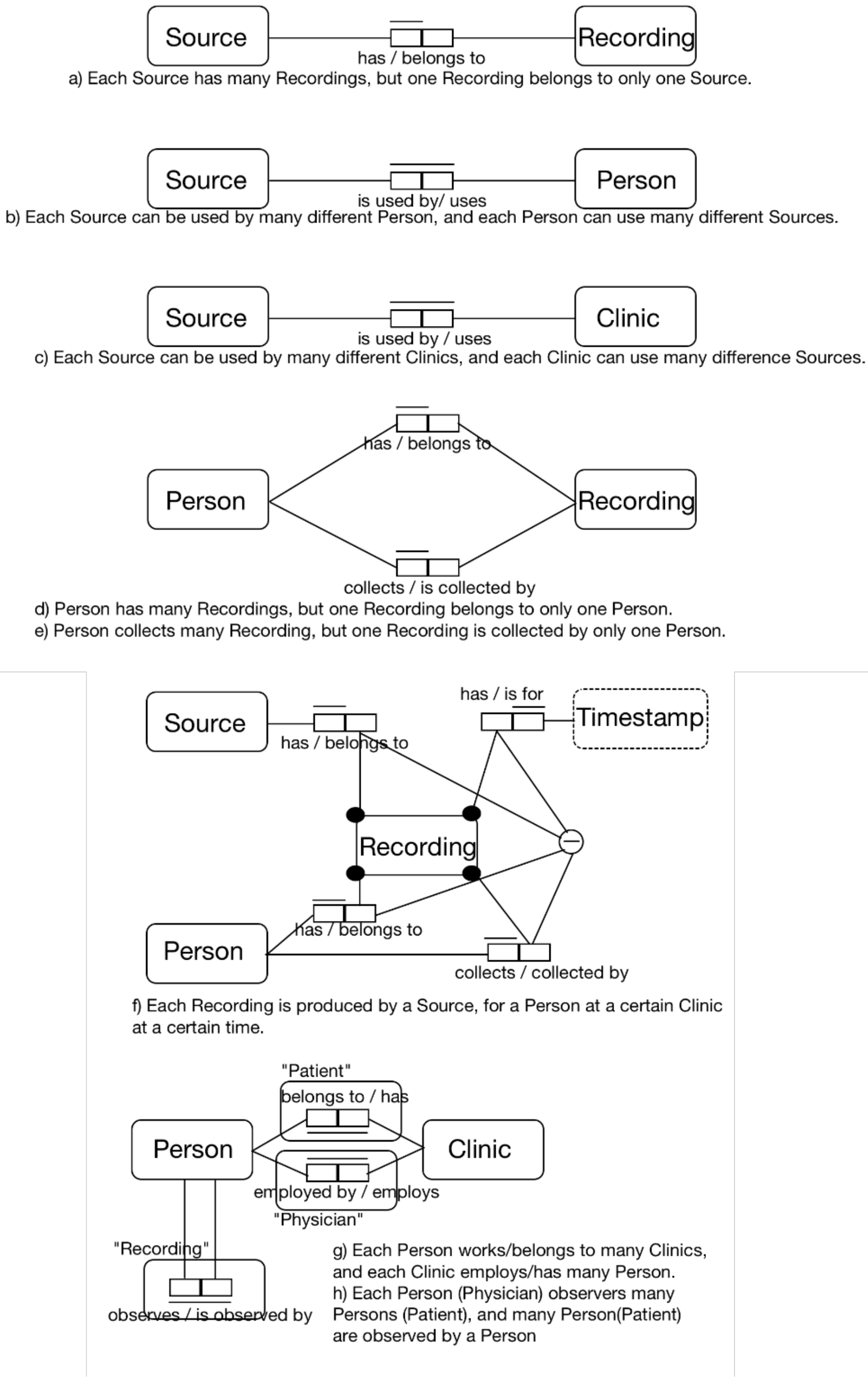

Figure 3.3: Binary, recursive and n-ary relationships [18]

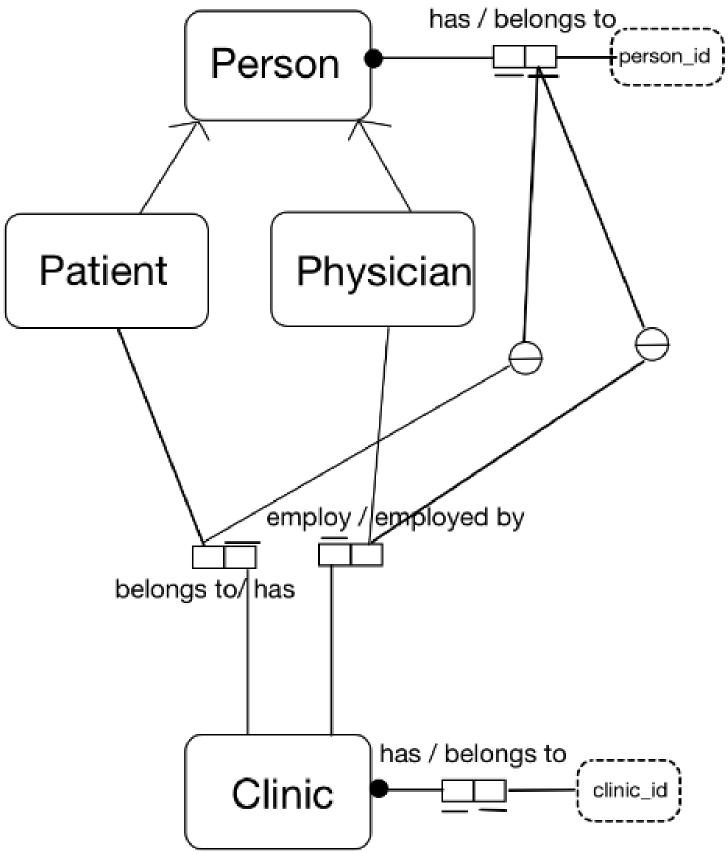

Figure 3.4: Logical model of the OSA database - Person and Clinic [18]

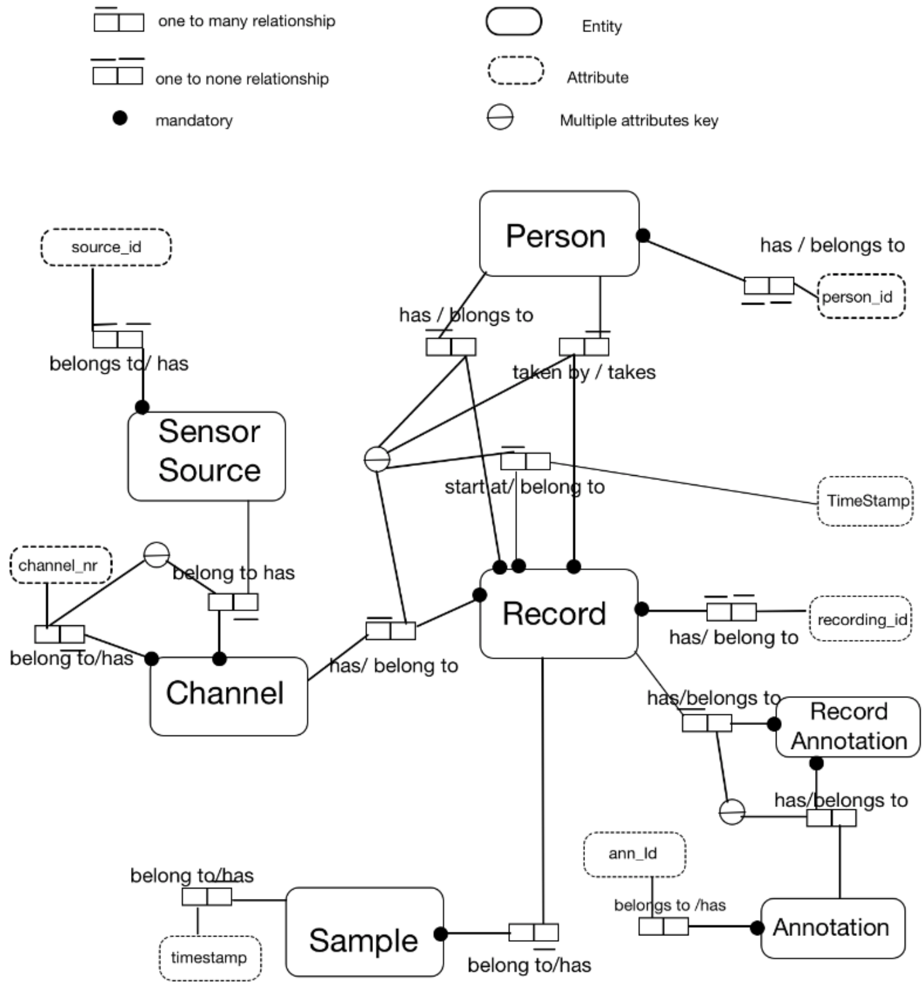

Figure 3.5: Logical model of the OSA database - Source and Recording [18]

Part II

Design and Implementation

Chapter 4

Analysis and High-Level Design

It is the goal of this thesis to enable detection of sleep-related illnesses with the aid of an Android device and low-cost sensors, and to further analyze and evaluate sleep- and breath-related patterns. We developed an application, called *Nidra*, which attempts to collect, analyze and share data collected from external sensors, all on a mobile device. Also, Nidra acts as a platform for modules to enrich the data, thus extending the functionality of the application.

The motivation behind this application is to provide an interface for patients to potentially run a self-diagnostic from home, and to aid researchers and doctors with analysis of sleep- and breathing-related illnesses (e.g., Obstructive Sleep Apnea). An overview of the Nidra application pipeline can be found in Figure x, beginning with data acquired from a sensor, and ending with the data in the Nidra application. As for now, Nidra consists of three main functionalities, each related to the requirements defined in Section [Problem Statement].

1. The application should provide an interface for the patient to 1) record physiological signals (e.g., during sleep); 2) present the results; and 3) share the results.

2. The application should provide an interface for the developers to create modules to enrich the data from records or extend the functionality of the application.

3. The application should ensure a seamless and continuous data stream, uninterrupted from sensor disconnections and human disruptions.

This chapter will give a detailed look at the design of Nidra, including the tasks which constitute the structure of the application, the seperate concerns in relation to the tasks, and the structure of the data in the application.

4.1 Requirement Analysis

4.1.1 Stakeholders

McGrath and Whitty [15] describe the term stakeholder as those persons or organizations that have, or claim an interest in the project. They distinguish stakeholders into four categories: 1) *contributing (primary) stakeholders* participate in developing and sustaining the project; 2) *observer (secondary) stakeholders* affect or influence the project; 3) *end-users (tertiary stakeholder)* interact and uses the output of the application; and 4) *invested stakeholders* has control of the project. In Nidra, there are three stakeholders who affect the application, and each can be categorized respectfully:

- `Patients`: Are identified as an end-user; they interact with the application.
- `Researchers/Doctors`: Are identified as an observer stakeholder; they might not use the application itself; however, they might use the data obtained from the patients' recordings for further analysis. Also, request functionality in the application.
- `Developers`: Are identified as a contributor stakeholder; they maintain the application from bugs or extend the functionally of the application. Additionally, they can contribute to developing modules that extend the functionality of the application.

4.1.2 Resource Efficiency

The application is designed for the use on a mobile device; modern mobile devices are empowered with multi-core processors, a sufficient amount of ROM, and a variety of sensors. However, the battery capacity is restrictive and based on usage. The device may only last for one day before a charge, due to the size of the battery capacity [13]. The average battery capacity of a mobile device is approximately 2000 mAh on budget devices and around 3000 mAh on high-end devices [8]. The application should be able to run at least 7 hours without any power supply. Also, the device should be capable of handling various sensor connections simultaneously. Therefore, the application should be designed to be resource efficient, by utilizing least amount of battery resources during a recording. Also, ensure sufficent amount of power on the device before starting a recording session.

4.1.3 Security and Privacy

The proposed use of the application is to monitor the sleeping patterns of a patient. The application manages and stores personal- and health-related data about the patient. As a consequence, the application should incorporate the CIA triad, which stresses data confidentiality, integrity, and

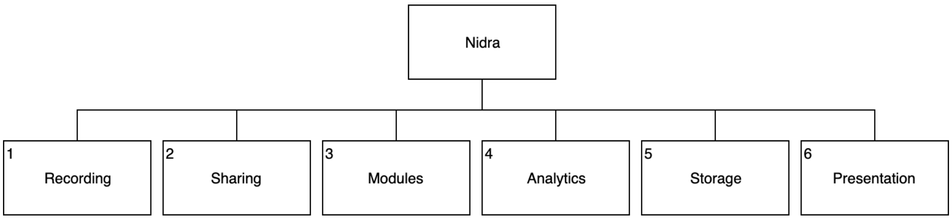

Figure 4.1: Recording

availability [19]. Any unauthorized access to the data, data leaks, and confidentiality should be appropriately managed on the device. Sharing the data across application or with researchers/doctors should be granted with the consent from the patient. Besides, a mobile device can be connected to the Internet, which makes it vulnerable to attacks. Also, other installed application on the device can manipulate the access to the data. Therefore, revising the security policy defined by Android [2] should be incentivized.

4.2 High-Level Design

4.2.1 Task Analysis

Task analysis is a methodology to facilitate the design of complex systems. Hierarchical task analysis (HTA) is an underlying technique that analyzes and decomposes complex tasks such as planning, diagnosis, and decision making into specific subtasks [5]. In Figure 4.1, an illustration of the the tasks of the application is presented. This Subsection will introduce these tasks, which are an integral part to the development of the application.

Recording

A *recording* is a process of collecting and storing physiological signals from sensors over an extended period (e.g., overnight). To enable a recording, we need to establish connections to available sensors, collect samples from the sensors, and storing the samples on the device. A *sensor* is a device that transforms analog signals from the real world into digital signals. The digital signals are transmittable over Link Layer technologies (e.g., Bluetooth), and the communication between a sensor and device occurs over an application programming interface (API). A *sample* is a single sensor reading containing data and metadata, such as time and the physiological data. During a recording session, ensuring for a consistent and uninterrupted data stream from the sensors is vital to obtaining persistent and meaningful data. Once a recording session has terminated, a *record* with metadata about the recording session is stored, alongside the samples.

Sharing

Sharing is a mechanism to export and import records across applications. *Exporting* consists of bundling one or more records with correlated samples into a transmittable format, and transferring the bundled records over a media (e.g., mail). *Importing*, on the other hand, consists of locating the bundled records on the device, parsing the content and storing it on the device. [skrive mer?]

Module

A *module* is an independent application that can be installed and launched in Nidra (hereafter: application), to provide extended functionality and data enrichment. A module does not necessarily interact with the application; however, it utilizes the data (e.g., records). For example, a module could be using the records to feed a machine learning algorithm to predict obstructive sleep apnea. Installing a module is achieved by locating the module-application on the device, and storing the reference in the application. Due to limitations in Android, the module-application cannot be executed within the application. Therefore, the module-application is a standalone Android application; furthermore, the development of the module-application is independent from the application.

Analytics

Analytics is the visualization and interpretation of patterns in the records. The application facilitates the recording of physiological signals, which enables the detection and analysis of sleep-related illnesses. There are various analytical methods, ranging from graphs to advanced machine learning algorithms. Incorporating a simple time series plot can indirectly aid in the analysis. For instance, plotting a time series graph where the physiological signals are on the Y-axis and the time on X-axis, provides a graphical representation of the data that can be further analyzed within the application.

Storage

Storage is the objective of achieving persistent data; data remain available after application termination. To enable storage, we use a database for a collection of related data that is easily accessed, managed, and updated. The database should be able to store records, samples, modules, and biometrical data related to user (i.e., gender, age, height, and weight). Structuring a database that is reliable, efficient, and secure is a crucial part of achieving persistent storage. Android provides several options to enable storage on the device (e.g., internal storage and database).

Presentation

Presentation is the concept of exhibiting the functionality of the application to the user. A user interface (UI) is the part of the system that facilitates interaction between the user and the system. In Nidra, determining the layout and view of the application, color palette, interactions, and feedback on actions is part of the development of a user interface.

4.3 Seperation of Concerns

Separation of concern is a paradigm that classifies an application into concerns at a conceptual and implementational level. It is beneficial for reducing complexity, improving understandability, and increasing reusability [12]. The concerns in this thesis are the individual tasks defined in task analysis. Each concern is conceptualized with a graph of components, the functionality of each component when combined constitutes a structure. The structure of each concern is derived based on research and development. In this Section, we will analyze and decompose the tasks defined in task analysis into subtasks (hereafter: components), where each component is functionality

4.3.1 Recording

The structure of a recording is restrictive in terms of arranging the components due to the design of CESAR. There are numerous ways of presenting the recording view; however, a recording structure is limited to the components of starting a recording, establishing sensor connection, monitoring of samples, and finalize sensors and recording. Additional components can be incorporated to aid a recording without causing disruption. For instance, the connectivity state component (4.3.1.1) provides extended functionality to the recording structure. In Figure 4.2, the illustration of a recording structure with the components and their dependencies are shown:

1.1 `Sensor Discover`: Find all eligible sensors that can enable a recording.

1.1.1 `Select Sensors`: From the sensor discovery, we can choose preferable sensors sources.

1.1.2 `All Sensors`: more Straightforward, we sample from all of the available sensors.

1.2 `Sensor Initialization`: Once we have a list of sensors sources, we need to establish and initialize a connection with the sensors. Occasionally a sensor might use some time to connect, or unforeseen occurrence is hindering the initialization of the sensor. Therefore, halting the sensor initalization or actively checking for sensor initalization is important.

1.3 `Sensor Connectivity Setup`: Establish a connection between the application and the sensor source through an API or IPC connection. All data exchange will occur over the established interface.

1.4 `Connection State`: Based on sensors establishments we can proceed to either start or stop a recording.

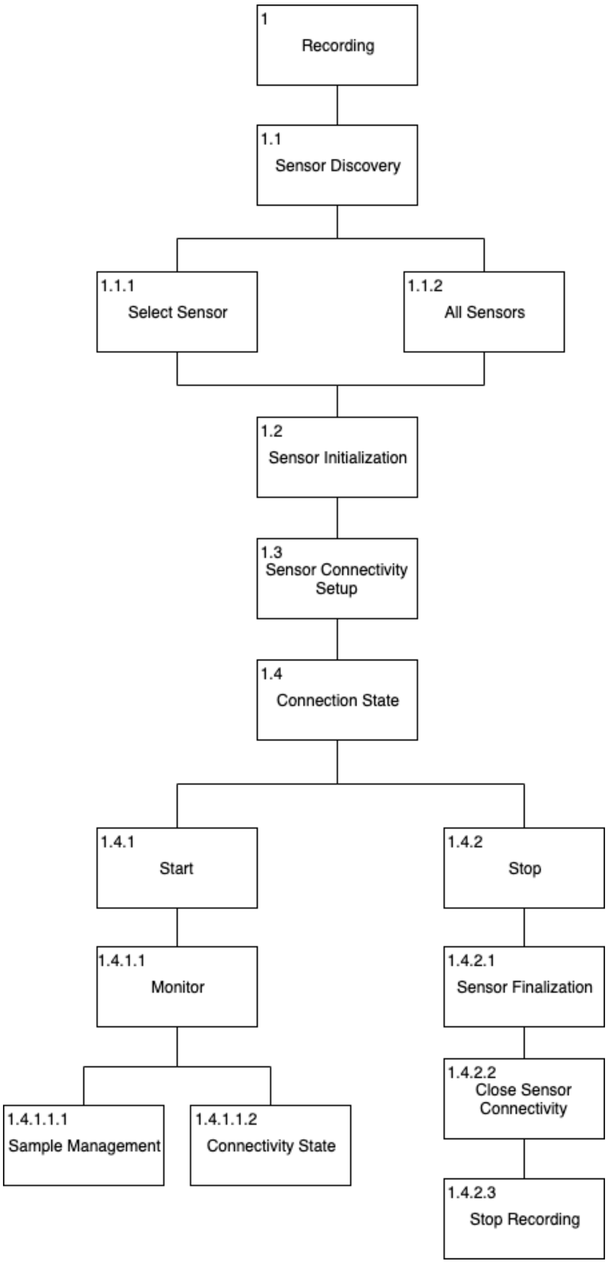

Figure 4.2: Recording

1.4.1 `Start`: By starting, we notify the sensors to begin collecting data, and the application should display that a recording has begun accordingly. Also, start a timer to display time elapsed on the current recording.

1.4.1.1 `Monitor`: Is a mechanism to handle the connectivty state and the incoming samples. It is actively listening to the interface for new data from the sensors, and appropriately distributing the data to the sample management component.

1.4.1.1.1 `Sample Management`: Handles a single sample from a sensor by parsing the content according to the payload of the sensor (each sensor might have different payload structure), such that it is according to our data structure.

1.4.1.1.2 `Connectivity State`: Activly checking the state of sensor connectivty (read more below)

1.4.2 `Stop`: Stop the recording timer and proceed to display results.

1.4.2.1 `Sensor Finalization`: Notify the sensor to stop sampling data, and close establishment.

1.4.2.2 `Close Sensor Connectivity`: Close the interface establishment between the application and the sensors.

1.4.2.3 `Stop Recording`: Once the sensors has closed its connections, add additional information to the recording (e.g., title, description, rating). In the end, the recording has concluded and it is stored on the mobile device.

4.3.1.1 Connectivity State Component

Connectivity state is a component that monitors for unexpected sensor disconnections or disruptions. Unexpected behavior can occur due to anomalies in the sensor, or the sensor being out of reach from the device for a brief moment. A naive solution would be to ignore the connectivity state component, and assuming the sensors are connected to the device indefinitely. However, upon disconnections or disruptions, the recording would be missing samples which will result in a lacking record. This component solves the issue of missing samples by actively reconnecting the sensor based on a time interval, resulting in more accurate record with fewer gaps between samples. The following design questions for this component are 1) should the connectivity state component, which implements a time interval that tries to reconnect with the sensor, be implemented in the sensor wrapper, or should it be in the proposed recording structure?; and 2) should the interval between sample arrival be a fixed time or a dynamical time?

1. To achieve a mechanism of reconnecting to the sensor on unexpected disconnects or disruptions, establishing a time interval that monitors

for sample arrivals within a time frame (e.g., every 10 seconds) is required. Incorporating the time interval in the sensor wrapper reduces the complexity of Nidra. However, it introduces extra complexity to the sensor wrappers. A sensor wrapper has to distinguish actual disconnects from unexpected disconnects. Although, by extending the functionality of sensor wrapper by implementing a state that indicates whether a recording is undergoing or stopped solves the problem. All future sensor wrappers would then have to implement the proposed solution, resulting in a complicated and time-consuming sensor wrappers development. While implementing the proposed solution in the sensor wrappers is possible, extending the recording structure with the logic in Nidra would be more meaningful and time-saving. In our design, we will be implementing the connectivity state in the recording structure.

2. A time interval triggers an event every specified time frame. If an event is triggered, a sample has not arrived, meaning the sensor either has been disconnected or disrupted. A time frame can be in a fixed size (e.g., every 10 seconds) or a dynamical size (e.g., start with 10 seconds, then incrementally increase the frame by X seconds). Implementing a fixed time frame increases the stress on put on the sensor, whereas a dynamical time might miss samples if the time frame is significant. Depending on how critical the recording is, a suitable solution for the time frame should be configurable. Also, limiting the number of attempts made to reconnect should be considered, due to actively reconnecting to a sensor that is dead or completely out of reach can cause stress on the device. Thus, stopping the recording once a limited number of attempts has been reached. In our design, we implemented a dynamical time and limited the number of attempts to 10.

4.3.2 Sharing

Sharing is separable into two concerns: export and import. The scope of exporting in Nidra is to select desired records, format and bundle the records into a transmittable file, and distributing the bundle over a media (e.g., mail). The scope of importing is to locate the file on the device, parse the content based on the format, and store it on the device. In Figure 4.3, the structure of sharing is presented with components and their dependencies:

2.1 `Import`: Is a mechanism that locates a file, parse the data, and stores it on the device.

2.1.1 `Locate File`: To enable this, the user has to download the file on the device. Then, locate the file on the device by using an interface to browse downloaded files. An interface can be developed; however, using the Android document picker (ref) is more straightforward solution.

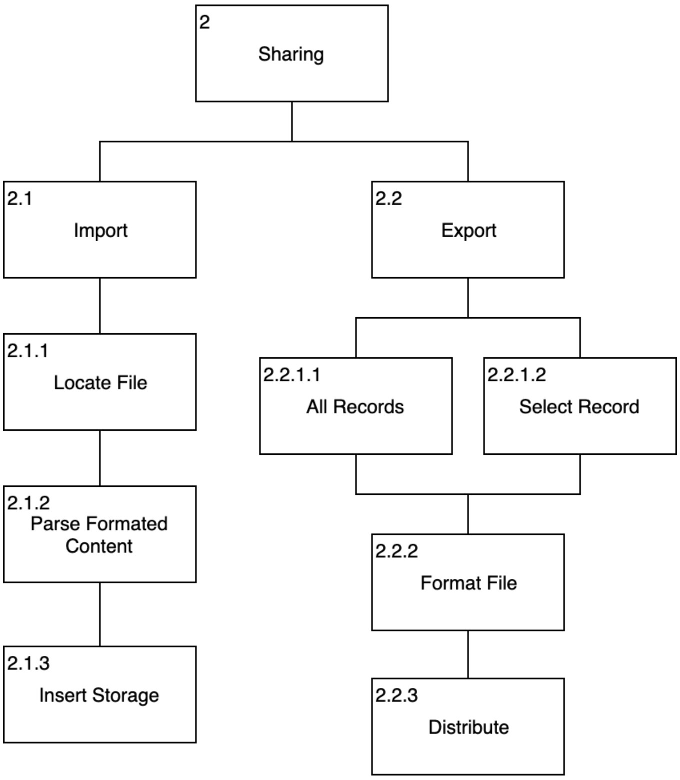

Figure 4.3: Sharing

2.1.2 `Parse Formated Content`: Parse the content of the file accordingly to the data format (ref).

2.1.3 `Insert Storage`: Retrieve the necessary data from the parsed file, to store on the device without overriding existing data.

2.2 `Export`: Is a mechanism that selects all or a specific record, format the record into a formatted filed (ref), and exports the file across device.

2.2.1.1 `All Records`: Export all of the records on the device.

2.2.1.2 `Select Record`: Pick one specific record to export.

2.2.2 `Format File`: When a preferred format for the records is selected, bundling the data into a formatted file (ref) for transmittal can be done. It is essential to identify the name of the file uniquely to prevent duplicates and overrides of data. For instance, identifying the name of the file with the device identification appended with the time of exporting.

2.2.3 `Distribute`: Send the file across application (read more below).

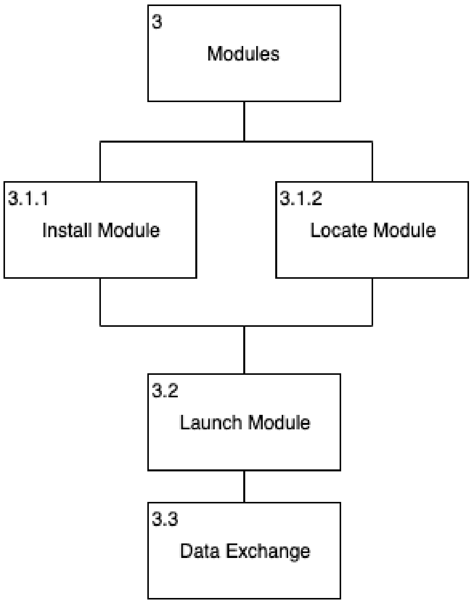

Figure 4.4: Modules

4.3.2.1 Distribute Component

The distribute component uses the formatted file and transfers it across applications. There are two distinctive methods to perform this task which is efficient and practical: 1) implement an interface with recipients to share data with, by establish a web-server with logic to handle users and sharing data with the desired recipient. Also, implementing an interface to retrieve the file within the application; and 2) using the interface provided by Android to share files.

While the first option might be favorable in terms of practicality, this solution introduces additional concerns (e.g., the privacy matters of storing user data on a server) which is out of the scope for this project. For this reason, using the interface provided by Android is a reasonable solution. The user of the application can utilize the Android interface for sharing files over installed applications; however, e-mail is a flexible media to transfer the file, and the user can specify the recipients accordingly.

4.3.3 Modules

Modules are independent applications that provide extended functionality and data enrichment to Nidra. The components for locating and launching a module is limited to Android design; however, the component for data exchange between a module and Nidra can be designed variously. In Figure 4.4, the structure of modules is presented with components and their dependencies:

3.1.1 `Install Modules`: Is the process of locating the application on the device, and storing the reference of the application package name in the storage.

3.1.2 `Locate Module`: Retrieve the list of stored modules, and display the installed modules to the user.

3.2 `Launch Module`: Get the application location stored in the module, and launch the application with the use of Android Intent.

3.3 `Data Exchange`: Enrich the module with data from the application (read more below)

4.3.3.1 Data Exchange Component

The data exchange component facilitates the transportation of data between Nidra and a module. As of now, the data is records and corresponding samples, which is formatted (Section Data) accordingly. The two distinct methods to exchange data between a module and Nidra are 1) formatting all of the data and bundling it into the launch of the module, and 2) establishing a communication link for bi-directional requests between Nidra and the module.

Android provides an interface to attach extra data on activity launch. The first solution is, therefore, convenient and efficient; all of the data is formatted and bundled into the launch. However, once Nidra has launched the module-application, there are no ways of transmitting new data besides relaunching the module-application. For this reason, the second option allows for continuous data flow by establishing a communication link with IPC between the applications. The data exchange between Nidra and modules can then be bidirectional; the module can request desired data any time, and Nidra can collect reports and results generated by the module.

One could argue that new records are not obtained while managing and using a module. However, there might be future modules that do a real-time analysis of a recording, but that will require an interface for continuous data flow. For the simplicity of our design, we will be going with the first option of bundling all of the data and sending it on launch.

4.3.4 Analytics

Analytics uses techniques and methods to gain valuable knowledge from data. Nidra provides a simple illustration of the data in a time-series plot; however, other techniques can be in incorporated. In essence, the facilitation of modules in the application enables the development opportunities for advanced analytics of the data. In Figure 4.5, the structure of analytics is presented with components and their dependencies:

4.1.1 `Select Record`: locate the file on the device.

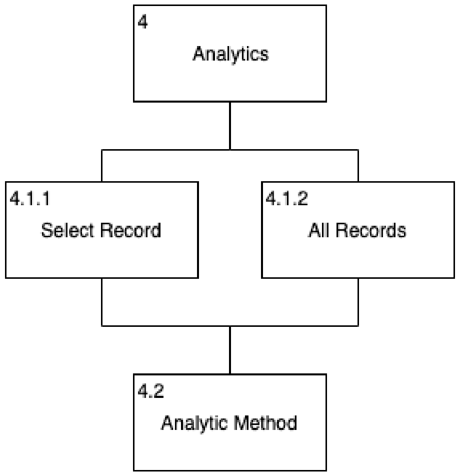

Figure 4.5: Analytics

4.1.2 `All Records`: parse the content of the file

4.2 `Analytic Method`:

4.3.4.1 Analytic Method Component

The analytic method component uses the records on the device for representation or analysis. Graphical and non-graphical are two techniques for representing data. Graphical techniques visualize the data in a graph that enables analysis in various ways. A few graphical techniques are diagrams, charts, and time series. Non-graphical techniques, better known as statistical data tables, represent the data in tabular format. This provides a measurement of two or more values at a time [10]. More advanced techniques to analyze the data, are to use machine learning. Machine learning is concerned with developing data-driven algorithms, which can learn from observations without explicit instructions. For example, using recurrent neural networks (e.g., RNN, LSTM) or regression models (e.g., ARIMA), can be used to predict the sleeping patterns [9].

In Nidra, a time series graph is used to represent the data of a record. The time series graph represents the respiration data on the Y-axis and the time on the X-axis. Essentially, the facilitation of modules in the application is designed to enable advanced techniques to predict, analyze, and interpret the data acquisition. Therefore, in Nidra, the analytic methods are limited; however, the modules enable developers to construct any method they desire.

4.3.5 Storage

Storage is the objective of achieving persistent data; data that is available after application termination. The data is characterized into four data

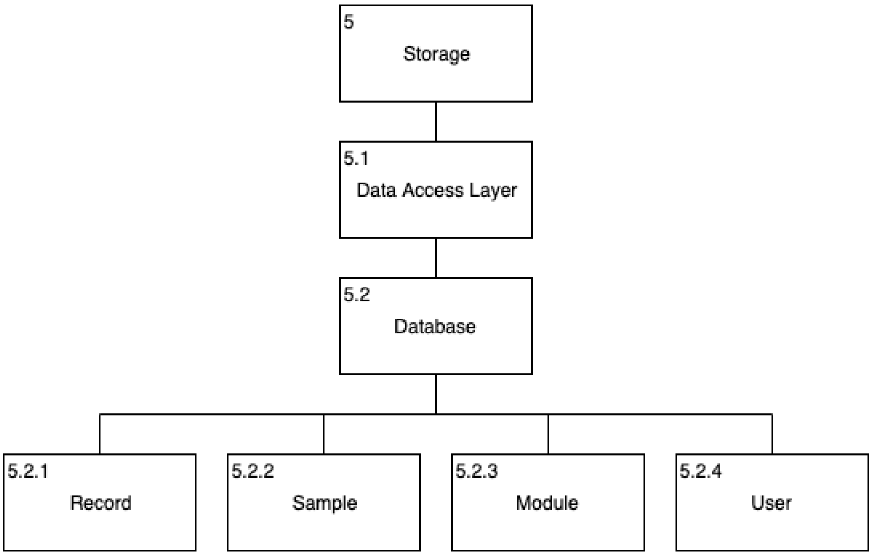

Figure 4.6: Storage

entities (i.e., record, sample, module, and user) that contains individual properties. The components in the storage structure are constructed to be extensible and scalable in terms of future data, restructure of data, and removal of data. In Figure 4.6, the structure of storage is presented with components and their dependencies:

5.1 `Data Access Layer`: Also known as an Data Access Object (DAO), that provides an abstract interface to a database. It exposes spesific operations (such as insertion of a record) without revealing the database logic. The advantage of this interface is to have a single entry point for each database operation, and to easily extend and modify the operation for future data.

5.2 `Database`: Is the storage of all the data (read below).

5.2.1 `Record`: Is a table in the database, which contains fields appropriately to the record structure. A record contains meta-data about a recording (e.g., name, recording time, user). An example of a recording record is illustrated in (ref).

5.2.2 `Sample`: Is a table in the database, which contains fields appropriately to a single sample from a sensor. A sample contains data received from a sensor during a recording. An example of a sample record is illustrated in (ref).

5.2.3 `Module`: Is a table in the database, which contains fields appropriately to the sample structure. A sample contains the name of the module and a reference to the application package. An example of a module record is illustrated in (ref).

5.2.4 `User`: Is a table in the database, which contains fields appropriately to the user structure. A user contains person information about a user (e.g., name, weight, height). An example of a user record is illustrated in (ref).

4.3.5.1 Database Component

Android provides several options to store data on the device; depending on space requirement, type of data that needs to be stored, and whether the data should be private or accessible to other applications. Two suitable options for storage are 1) internal file storage - storing files to the internal storage private to the application; and 2) database - Android provides full support for SQLite databases, and the database access is private to the application [7]. Based on the options, some following design questions are 1) should the data be stored in a flat file database on the internal file storage, or should it be stored in an SQLite database?

Flat files database encode a database model (e.g., table) as a collection of records with no structured relationship, into a plain text or binary file. For instance, each line of text holds on a record of data, and the fields are separable by delimiters (e.g., comma or tabs). Another possibility is to encode the data in a preferable data format (ref formats). Flat file databases are easy to use and suited for small scale use; however, they provide no type of security, there is redundancy in the data, and integrity problems [14]. Locating a record is made possible by loading the file, and systematically iterating until the desired record is found. Similarly, updating a record and deleting a record. Consequently, the design of flat file databases is for simple and limited storage.

SQLite is a relational database management system, which is embedded and supported in Android. Relational database management system (RDBMS) provides data storage in fields and record, represented as columns (fields) and rows (records), in a table. The advantage is the ability to index records, relations between data stored in tables, and support querying of complex data with a query language (e.g., SQL). Also, RDBMS provides data integrity between transactions, improved security, and backup and recovery controls [14].

While a flat file database is applicable to store small and unchangeable data, it is not suitable for scalable and invasive data change. In Nidra, the samples acquisition makes it unreasonable to use a flat file database. Therefore, SQLite is a preferable solution to storing samples. Also, establishing a relationship between a record and a sample is made possible [rewrite].

4.3.6 Presentation

Presentation facilitates the user interface of the application, in terms of visualizing the functionality of the application to the user. The

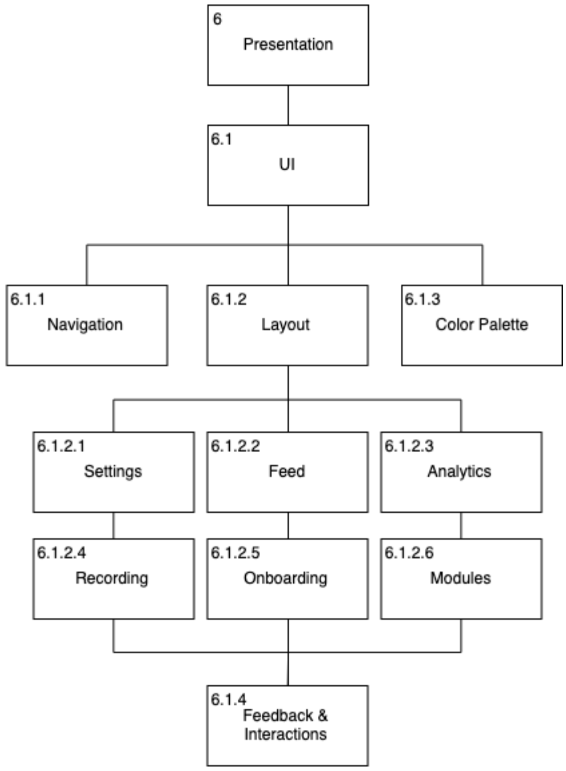

Figure 4.7: Presentation

user interface derives from the functionality (concerns discussed) in the application, and research on the topic. In Figure 4.7, the structure of storage is presented with components and their dependencies:

6.1 `UI`: The user interface where interaction between users and the application occurs.

6.1.1 `Layout`: Are the screen with the content of the current screen. The layout incorporates the color palette and some views have the navigation displayed.

6.1.2 `Navigation`: The navigation is a menu with options to change the layout.

6.1.3 `Color Palette`: A color selection that persist throughout the application (read more below).

6.1.1.1 `Settings`: Is s screen with user details, permissions and credits, with options to modify permission and user details.

6.1.1.2 `Feed`: Is a list of all records for the user, displayed with details specific to the record to make it distinguishable and easily recognizable.

6.1.1.3 `Analytics`: A interactive time-series graph for a single record.

6.1.1.4 `Recording`: The process of establishing a recording session, in addition to showing the results after a recording session has ended.

6.1.1.5 `Onboarding`: The initial screen displayed to the user, where the user can supply the application with their biometrical data.

6.1.1.6 `Modules`: A list of all installed modules, also an option to add more modules.

6.1.4 `Feedback & Interactions`: Each layout has different feedback and interaction, which should be handled appropriately.

4.3.6.1 Color Palette Component

Color palette is a component that decides the color scheme in the application. In the proposal of a color system in the design guidelines by Google [16], it is essential to pick colors that reflect the style of the application accordingly to: 1) primary colors - the most frequently displayed color in the application; 2) secondary colors - provides an accent and distinguish color in the application; and 3) surface, background, error, typography and iconography colors - colors that reflect the primary and secondary color choices.

Moreover, choosing colors that meet the purpose of the application is critical. Nidra is most likely to be used during the evening and the morning. According to Google [6], a dark color theme reduces luminance emitted by the device screen, which reduces eye strain, while still meeting the minimum color contrast ratios, and conserving battery power. From this, we will be choosing a dark color theme.

4.4 Data Structure

4.4.1 Data Formats

The data format is a part of the process of serialization, which enables data storage in a file, transmittal over the Internet, and reconstruction in a different environment. Serialization is the process of converting the state of an object into a stream of bytes, which later can be deserialized by rebuilding the stream of bytes to the original object. There are several data serialization formats; however, JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) are the two most common data serialization formats. In this Section, we will discuss these formats. In the end, we will compare them and choose the format that meets the criteria of being compact, human-readable, and universal.

4.4.1.1 JSON

JSON or JavaScript Object Notation is a light-weight and human-readable format that is commonly used for interchanging data on the web. The format is a text-based solution where the data structure is built on two structures: a collection of name-value pairs (known as objects) and ordered list of values (known as arrays). The JSON format is language-independent and the data structure universally recognized [1, 20]. However, it is limited to a few predefined data types (i.e., string, number, boolean, object, array, and null), and extending the data type has to be done with the preliminary types.

```
{
    "user": {
        "firstname": "Ola"
        "lastname": "Nordmann"
    }
}
```

Listing 4.1: My Caption

4.4.1.2 XML

XML or eXtensible Markup Language is a simple and flexible format derived from Standard Generalized Markup Language (SGML), developed by the XML Working Group under the World Wide Web Consortium (W3C). An XML document consists of markups called tags, which are containers that describe and organize the enclosed data. The tag starts with < and ends with >; the content is placed between an opening tag and a closing tag (see listing). [4, 20] XML provides mechanisms to define custom data types, using existing data types as a starting point, making it extensible for future data.

```
<user>
    <firstname>Ola</firstname>
    <lastname>Nordmann</lastname>
</user>
```

Listing 4.2: My Caption

4.4.1.3 Comparing

We will compare JSON and XML features and performance with the study conducted by Saurabh and D'Souza [20]. There are apparent differences in the two data formats which affect the overall readability, extensibility, bandwidth performance, and ease of mapping. XML documents are easy to read, while JSON is obscure due to the parenthesis delimiters. XML allows for extended data types, while JSON is limited to a few data types.

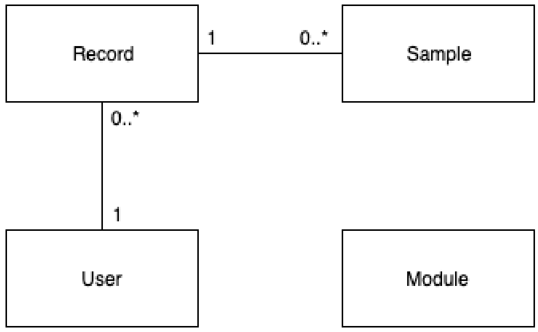

Figure 4.8: Modules

XML takes more bandwidth due to the metadata overhead, while JSON data is compact and use less amount of bandwidth.

Moreover, a few benchmarks were conducted to measure memory footprint and parsing runtime when serializing and deserializing JSON and XML data. From the conclusion, in terms of memory footprint and parsing runtime, JSON performances better than XML but at the cost of readability and flexibility. While these format structures are applicable for transmitting data, choosing a format that is compact, human-readable, and a standard format that is extensible and scalable for future data is essential. In our design, we will be using the JSON format for transmission of the data.

4.4.2 Data Tables

Data tables are a collection of structured and related data in a tabular format in a database. Subsection about Storage introduces four data tables in the application. In Figure 4.8, the relation between these tables are shown. Record and sample are separated in order to reduce data redundancy and improve data integrity; however, samples have a reference to a record so they can be associated with each other. A user can have many records; however, a record can only have one user. The module table has no relation to the other tables in the database. This Subsection will demonstrate the properties of each table in Nidra, with an illustration of a record inside each table.

4.4.2.1 Record

A record is a table in the database that stores metadata related to the recording session. In Table 4.1, an illustration of the structure of a record is shown, with an entry of dummy data. In Nidra, the fields in the table for a record describe the data that is stored, which is separated into:

- `ID`: Unique identification of a record, also a primary key for the entry.
- `Name`: A name of the record to easily recognize the recording.

id	name	description	monitorTime	rating	user	createdAt	updatedAt
1	Record #1		5963088	2.5	1	1554406256000	1554406256000

Table 4.1: Example entry in record table

id	recordId	explicitTS	implicitTS	sample
1	1		5963088	Time=0ms, deltaT=100, data=1906,1891,1884,1881,1876,1718,1690

Table 4.2: Example entry in record table

- `Description`: A summary over the recording session, used to describe briefly how the recording session felt (e.g., any abnormalities during the sleep).
- `MonitorTime`: The recording session duration in milliseconds.
- `Rating`: Giving a rating on how the sleeping session felt, in a range between 0-5.
- `User`: Identification of the user, also a foreign key to the user table.
- `CreatedAt`: Date of creation of the recording in milliseconds (since January 1, 1970, 00:00:00 GMT).
- `UpdatedAt`: Date of update of the recording in milliseconds (since January 1, 1970, 00:00:00 GMT).

4.4.2.2 Sample

A sample is a single sensor reading containing data and metadata related to the recording session. Samples are stored separated from a record; however, they are linked with a field in the table. In Table 4.1, an illustration of the structure of a sample is shown, with an entry of dummy data. In Nidra, the fields in the table for a sample describe the data that is stored accordingly to the data provided by Flow SweetZpot (ref), which are separated into:

- `ID`: Unique identification of a sample, also a primary key for the entry.
- `RecordID`: An identification to its correlated record, also a foreign key.
- `ExplicitTS`: Timestamp of sample arrival based on the time in the sensor.
- `ImplicitTS`: Timestamp of sample arrival based on the time on the device.
- `Sample`: Sensor reading contains metadata and data according to Flow Sweetzpot. The sensor aggregates seven samples in a single sensor reading.

id	name	packageName
1	OSA Predicter	com.package.osa_predicter

Table 4.3: Example entry in record table

id	name	age	gender	height	weight	createdAt
1	Ola Nordmann	50	Male	180	60	–

Table 4.4: Example entry in record table

4.4.2.3 Module

A module is a table in the database that stores all modules installed by the user in the application. In Table 4.3, an illustration of the structure of a module is shown, with an entry of dummy data. In Nidra, the fields in the table contain the name of the module and the reference to the module and can be summarized as:

- `ID`: Unique identification of a module, also a primary key for the entry.
- `Name`: The name of the module-application.
- `PackageName`: The package name of the module-application, such that it can be launched from Nidra.

4.4.2.4 User

Nidra considers the user of the application as the patient, hence storing biometrical data is part of the application to enrich the record data. In Table 4.4, an illustration of the structure of a user is shown, with an entry of dummy data. In Nidra, the fields in the table for a user contains biometrical data related to the user describe the data which are:

- `ID`: Unique identification of a module, also a primary key for the entry.
- `Name`: The name of the module-application.
- `Age`: Age of the user.
- `Weight`: Weight of the user in kilograms.
- `Height`: Height of the user in centimeters.
- `CreatedAt`: Date of creation of the user in milliseconds (since January 1, 1970, 00:00:00 GMT).

4.4.3 Data Packets

Data packets are parcels of data that Nidra receives from external applications (e.g., sensor wrappers) or send to other application (e.g., sharing). From the design choice In the Section above, the format of all the desired data should be according to the JSON format. In this Section.

4.4.3.1 Sharing

In Section (ref), a proposal to the structure of exporting and importing data is discussed. Two of the components (`Parse Formatted Content` and `Format File`) uses JSON to either encode or decode the data. Listing 4.3 illustrate the content of the encoded data from our application to gain a broader understanding of how the data exchange in sharing operates. The attractive attributes from the encoding are the record (ref) and the samples. A record is an object that contains meta-data with name, number of samples, recording time, creation date, and user information. Samples are an array of objects that contains data, timestamp, and identification to correlated record.

```
{
    "record":{
        "id": 1,
        "name": "Record 1",
        "rating": 2.5,
        "description": "",
        "nrSamples": 6107,
        "monitorTime": 5963088,
        "createdAt": "Apr 4, 2019 9:30:56 PM",
        "updatedAt": "Apr 4, 2019 9:30:56 PM",
        "user": {
            "age": 50,
            "createdAt": "---",
            "gender": "Male",
            "height": 180,
            "name": "Ola Nordmann",
            "weight": 60
        }
    },
    "samples": [
        {
            "explicitTS":"Apr 4, 2019 5:51:26 PM",
            "implicitTS":"Apr 4, 2019 7:51:26 PM",
            "recordId":1,
            "sample":"Time=0ms, deltaT=100, data
                =1906,1891,1884,1881,1876,1718,1690"
```

```
        },
        ...
    ]
}
```

Listing 4.3: My Caption

4.4.3.2 Sensor Data

Sensor data is data acquisition through the Data Dispatching (section background). The data format discussed in (section background).

Chapter 5

Implementation

5.1 Architecture (MVVM)

5.2 IPC?

5.3 Permissions & Android Manifest

5.4 Implementation of Concerns

5.4.1 Recording

5.4.2 Sharing

5.4.3 Modules

Modules have three actions: A) retrieve/display modules; B) install modules; and C) launch module.

In Figure 5.6, the method calls and the interactions between the components are shown. Based on the actions:

A.1 Test

A.2 Test

A.3 Test

A.4 Test

B.1 Test

B.2 Test

B.3 Test

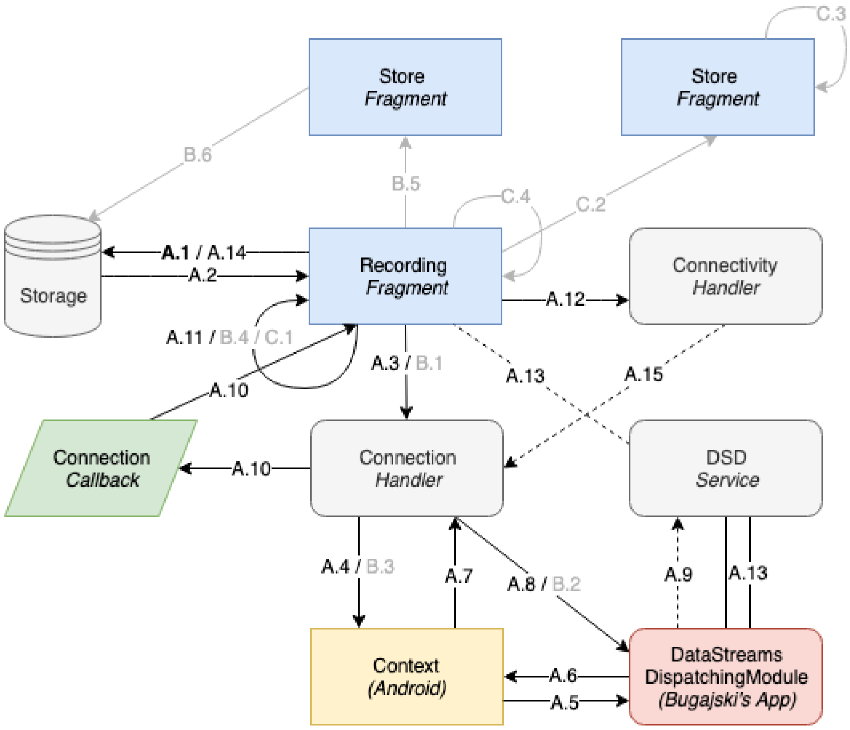

Figure 5.1: Implementation of recording functionality (A)

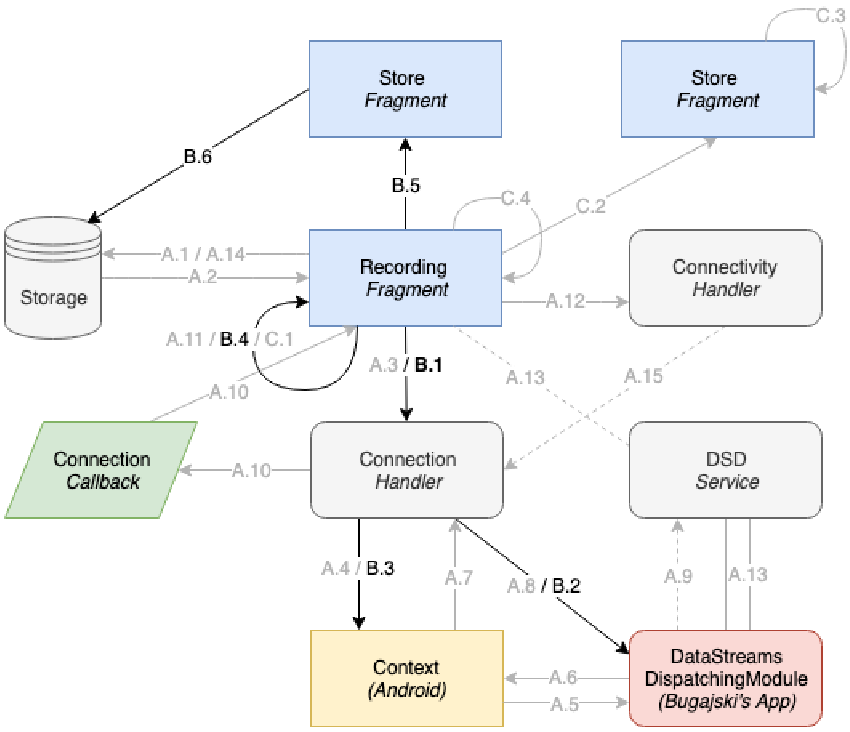

Figure 5.2: Implementation of recording functionality (B)

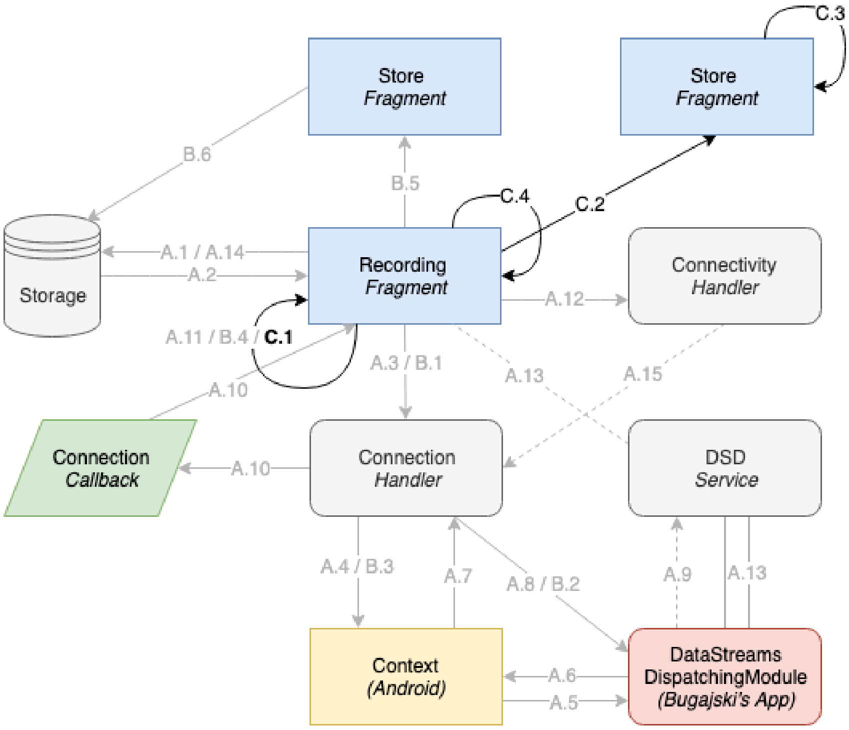

Figure 5.3: Implementation of recording functionality (C)

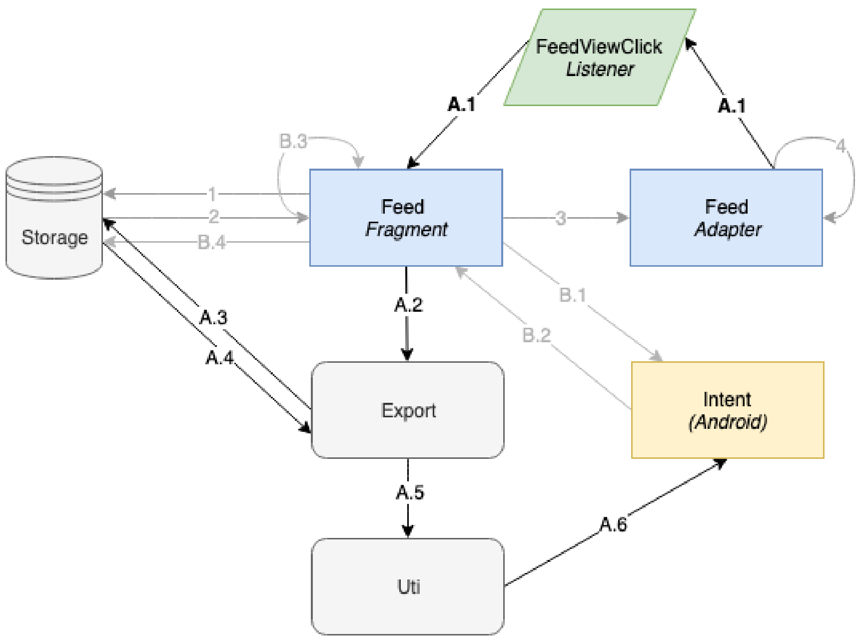

Figure 5.4: Implementation of sharing functionality (A)

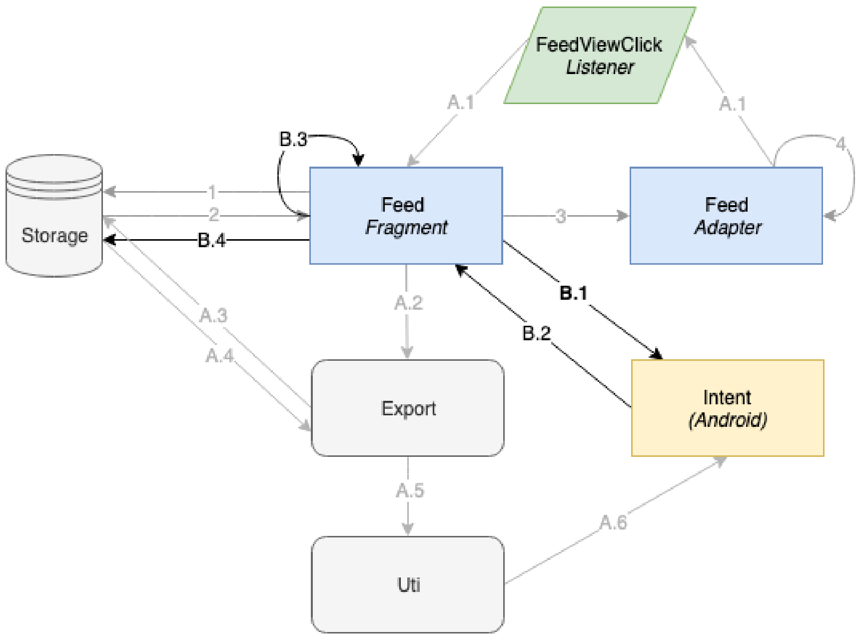

Figure 5.5: Implementation of sharing functionality (B)

B.4 Test

B.5 Test

B.6 Test

C.1 Test

C.2

5.4.3.1 Data Exchange Implementation

```
public void onLaunchModuleClick(String packageName) {
    Intent moduleApplication = context.getPackageManager().
        getLaunchIntentForPackage(packageName);

    if (moduleApplication == null) return;

    String data = formatAllRecordsToJSON();

    Bundle bundle = new Bundle();
    bundle.putString("data", data);

    moduleApplication.putExtras(bundle);

    startActivity(moduleApplication);
}
```

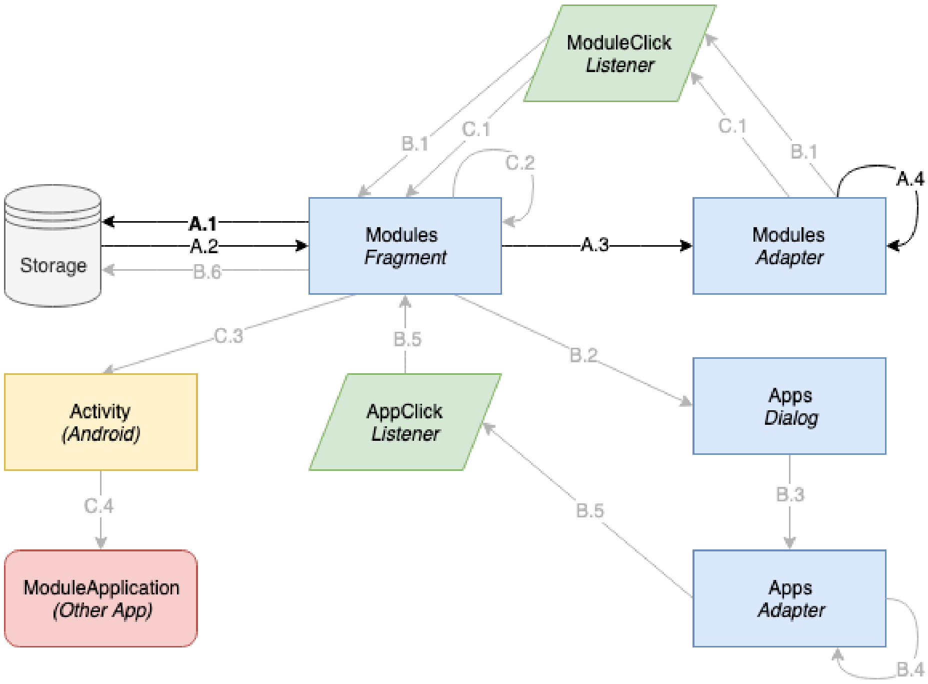

Figure 5.6: Implementation of module functionality(A)

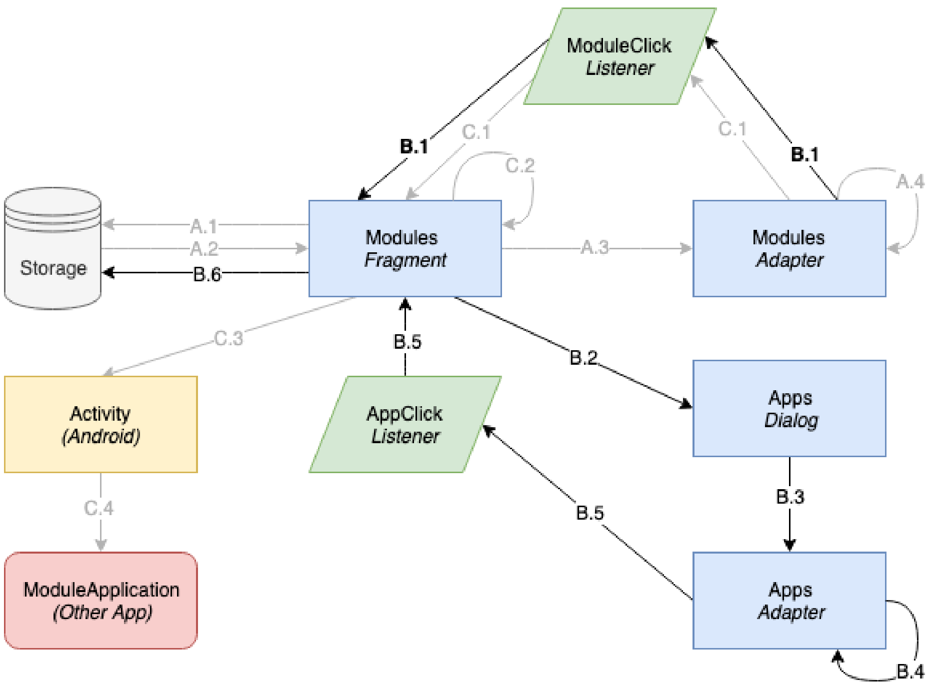

Figure 5.7: Implementation of module functionality(B)

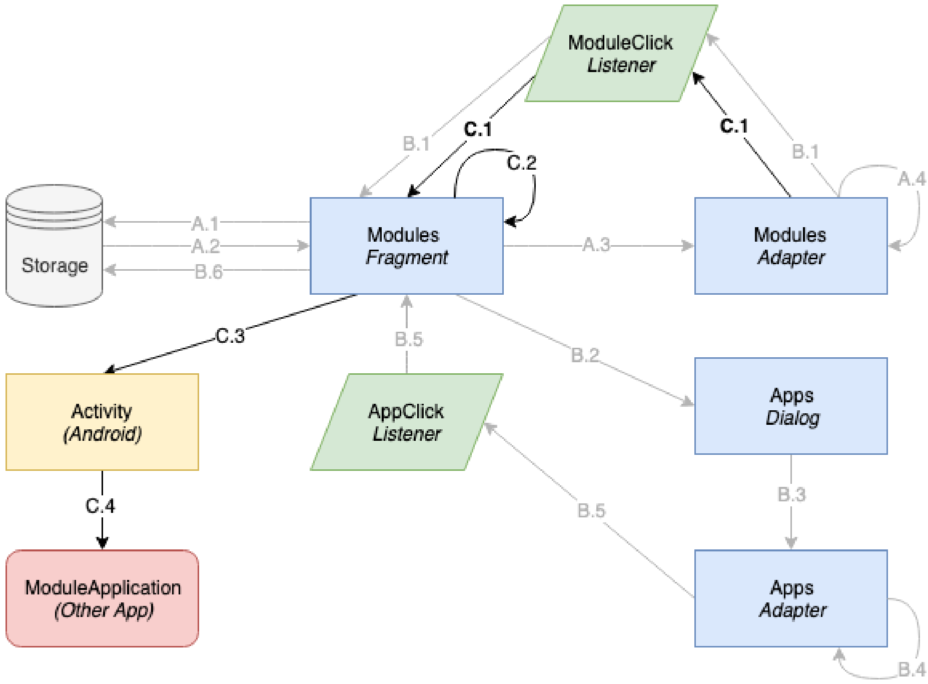

Figure 5.8: Implementation of module functionality(C)

Listing 5.1: My Caption

5.4.4 Analytics

5.4.5 Storage

5.4.6 Presentation

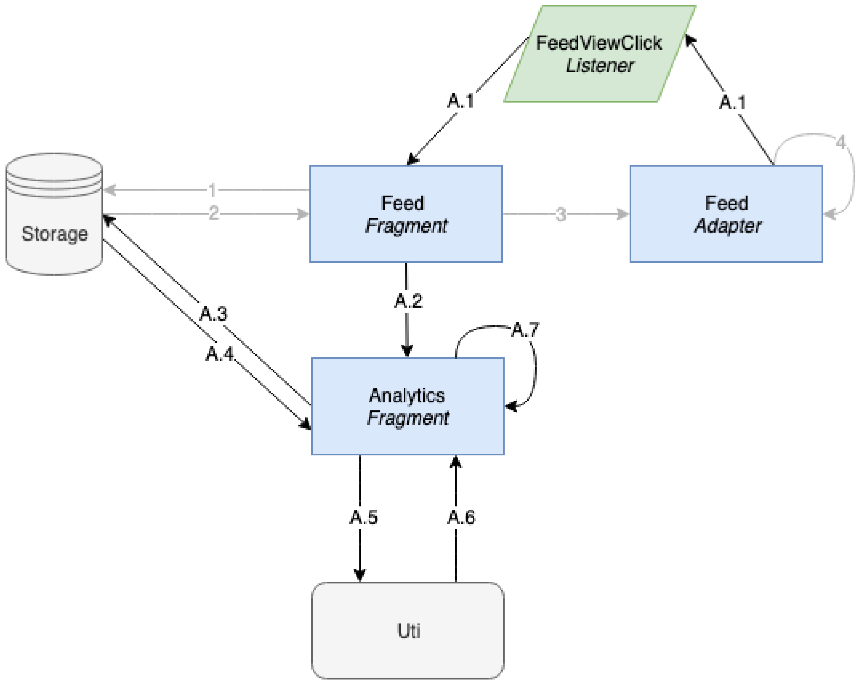

Figure 5.9: Implementation of analytics functionality

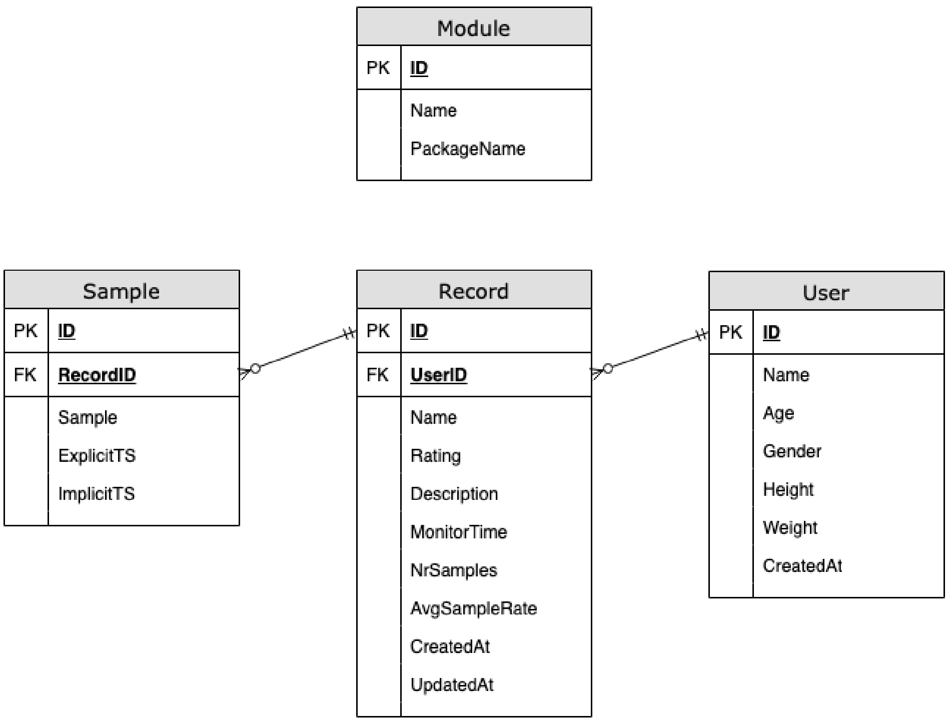

Figure 5.10: Entity Relationship Diagram

Part III

Evaluation and Conclusion

Chapter 6

Experiments

Chapter 7

Future Work

Chapter 8

Conclusion

Appendix

Appendix A

Power Data

Bibliography

[1] Ecma International 2017. *The JSON Data Interchange Syntax*. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 01/06/2019).

[2] Android. *Secure an Android Device*. URL: https://source.android.com/security (visited on 30/05/2019).

[3] Daniel Bugajski. 'Extensible data streams dispatching tool for Android'. In: (201).

[4] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL: https://www.w3.org/TR/REC-xml/ (visited on 01/06/2019).

[5] Abe Crystal and Beth Ellington. 'Task analysis and human-computer interaction: approaches, techniques, and levels of analysis'. In: *AMCIS*. 2004, pp. 2–3. URL: https://pdfs.semanticscholar.org/fbd2/b61c998cb3ad8427759a45370a02d9338c31.pdf (visited on 02/05/2019).

[6] *Dark theme*. Material Design. URL: https://material.io/design/color/dark-theme.html#usage (visited on 04/06/2019).

[7] *Data and file storage overview*. Android Developers. URL: https://developer.android.com/guide/topics/data/data-storage (visited on 05/06/2019).

[8] *Fact check: Is smartphone battery capacity growing or staying the same?* Android Authority. URL: https://www.androidauthority.com/smartphone-battery-capacity-887305/ (visited on 30/05/2019).

[9] R. Fu, Z. Zhang and L. Li. 'Using LSTM and GRU neural network methods for traffic flow prediction'. In: *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*. Nov. 2016, pp. 324–328. DOI: 10.1109/YAC.2016.7804912. (Visited on 07/06/2019).

[10] H Gray Funkhouser. 'Historical development of the graphical representation of statistical data'. In: *Osiris* 3 (1937), pp. 269–404. URL: https://www.journals.uchicago.edu/doi/pdfplus/10.1086/368480 (visited on 07/06/2019).

[11] Svein Petter Gjøby. 'Extensible data acquisition tool for Android'. In: (2016).

[12] Walter L. Hursch and Cristina Videira Lopes. 'Separation of Concerns'. In: (Mar. 1995), pp. 3, 16. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=031E83D200FD8770C255B48EA0C2E1C2?doi=10.1.1.29.5223&rep=rep1&type=pdf (visited on 24/05/2019).

[13] Huoran Li, Xuanzhe Liu and Qiaozhu Mei. 'Predicting Smartphone Battery Life based on Comprehensive and Real-time Usage Data'. In: (Jan. 2018), p. 2. URL: https://www.researchgate.net/publication/322498404_Predicting_Smartphone_Battery_Life_based_on_Comprehensive_and_Real-time_Usage_Data (visited on 30/05/2019).

[14] Innocent Mapanga and Prudence Kadebu. 'Database Management Systems: A NoSQL Analysis'. In: *International Journal of Modern Communication Technologies And Research (IJMCTR)* Volume-1 (Sept. 2013), pp. 12–18. URL: https://www.researchgate.net/publication/258328266_Database_Management_Systems_A_NoSQL_Analysis (visited on 15/05/2019).

[15] Stephen McGrath and Jonathan Whitty. 'Stakeholder defined'. In: *International Journal of Managing Projects in Business* 10 (July 2017), pp. 4, 13, 14. DOI: 10.1108/IJMPB-12-2016-0097. (Visited on 01/05/2019).

[16] *The color system*. Material Design. URL: https://material.io/design/color/the-color-system.html# (visited on 04/06/2019).

[17] Stein Kristiansen Thomas Peter Plagemann Vera Hermine Goebel. *Android based eHealth applications with BiTalino sensors*. URL: http://www.mn.uio.no/ifi/stud based-ehealth-applications-with-bitalino-s.html (visited on 05/05/2018). 2015. URL: http://www.mn.uio.no/ifi/studier/masteroppgaver/dmms/android-based-ehealth-applications-with-bitalino-s.html.

[18] Viet Thi Tran. 'An open database model for storing Obstructie Sleep Apnea data'. In: (2017).

[19] M. U.Farooq et al. 'A Critical Analysis on the Security Concerns of Internet of Things (IoT)'. In: *International Journal of Computer Applications* 111.7 (18th Feb. 2015), pp. 1–6. ISSN: 09758887. DOI: 10.5120/19547-1280. URL: http://research.ijcaonline.org/volume111/number7/pxc3901280.pdf (visited on 30/05/2019).

[20] Saurabh Zunke and Veronica D'Souza. 'JSON vs XML: A Comparative Performance Analysis of Data Exchange Formats'. In: 3.4 (2014), pp. 2–4. URL: http://ijcsn.org/IJCSN-2014/3-4%20JSON-vs-XML-A-Comparative-Performance-Analysis-of-Data-Exchange-Formats.pdf (visited on 30/05/2019).