SAARLAND UNIVERSITY

Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's thesis


# AndroidSmart

## An Android framework for sensor data acquisition via Bluetooth

Stephan Just
Bachelor's Program in Media Informatics
December 2015

**Advisor:**
Frederic Kerber, German Research Center for Artificial Intelligence,
Saarbrücken, Germany

**Supervisor:**
Prof. Dr. Antonio Krüger, German Research Center for Artificial Intelligence,
Saarbrücken, Germany

**Reviewers:**
Prof. Dr. Antonio Krüger, German Research Center for Artificial Intelligence,
Saarbrücken, Germany
Dr.-Ing. Tim Schwartz, German Research Center for Artificial Intelligence,
Saarbrücken, Germany

**Submitted**
21th December, 2015

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Campus - Building E1.1
66123 Saarbrücken
Germany

**Statement in Lieu of an Oath:**

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, 21th December, 2015

**Declaration of Consent:**

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 21th December, 2015

## Abstract

With the rise of wearables, such as smart watches and fitness trackers, mobile developers face the issue of adding support for a basically endless amount of different devices. While it is already tedious to support a huge amount of devices for one application, this work is done by nearly every developer of any application that wants to support said wearables, resulting in a lot of rewritten code.

This thesis suggests a modular Android framework that supports multiple very popular devices with the possibility to be extended to support further wearables. Developers using the framework do not have to bother about any software development kits or application programming interfaces provided by the manufacturers. A public interface allows to interact with each device in the same way, therefore using the framework results in support for any device currently supported by the framework.

# List of Figures

# Contents

# Chapter 1
# Introduction

This chapter provides an introduction into the wearable market, gives statistics about mobile applications, explains the problem that arises from the combination of both topics and proposes a solution to the issue.

## 1.1 Wearables

Wearables are all kind of electronic devices that can be worn with clothing or on the body. From a technical point of view every mobile computer, such as smartphones or tablets, could be called a wearable. The industry however uses the term "wearable" for small, light and intelligent devices that integrate effortless into our everyday life. Most of these devices connect with our smartphones to display the information they gathered as these small devices very often do not have a display.

In 1972 the Hamilton Watch Company created the first digital watch called the Pulsar [16]. Using LEDs (light emitting diodes) to power its display, it consumed that much energy that constantly having the display turned on would have led to a very short battery life. Therefore the Pulsar had a button to enable the display for a short period of time.

Starting with the Pulsar, many companies entered the market of wearable electronic devices, mainly combining computers and watches such as the Casio AT-550 in 1984, a calculator watch with a touch screen [4]. In the same year Seiko introduced the RC-1000 wrist terminal and one year later the RC-20 wrist computer. Both devices were able to interface with desktop computers, such as the Apple II, the Commodore 64 or the IBM PC, allowing both devices to interact with each other.

While the first wearables were mostly electronic watches with added computational capabilities, they are nowadays all kinds of different devices such as fitness trackers, chest straps, footwear and much more. Especially in the last few years manufacturers created a lot of different wearables equipped with sensors to be carried by a user to monitor his movement, his behavior and the world around him. The devices are packed with sensors intended to interact with the user and provide a unique customized experience. For example, data from GPS modules are used to localize search results as in tourist attraction recommendation apps [20]. Accelerometers are used to count steps [19] and provide basic fitness tracking for users. Although devices with sensors existed for many years, the rise of smartphones and their possibility to interact with basically everything opened a huge market for intelligent wearable devices.

The database of Vandrico Inc currently lists 347 different wearables[1]. In 2014 about 17.6 million of these wearable devices have been sold. The GfK (Gesellschaft für Konsumforschung[2]) estimated that in 2015 there will be a total of 51 million wearables sold [12].

## 1.2 Mobile Applications for Smartphones

The two biggest app stores for mobile phones, Google Play and the Apple App Store, contained around 1.8 million *applications* (apps) in 2013 [1, 7] and 2.6 million apps in 2014 [13]. This number has grown to 3.1 million apps in July 2015 [2].

The fitness and health care sector has created a lot of new wearables and fitness apps. Kailas et al. [10] claimed that there were over 7,000 health related mobile apps in 2010. According to the U.S. Food and Drug Administration this number has grown to over 31,000 [8] in 2013. In 2014 this number has increased to over 100,000 [17]. Compared to the 37.5% growth of all apps in the app stores from 2013 to 2014, the 222% growth of health care and fitness apps during the same period is an enormous amount.

## 1.3 The Problem

Many of these apps, especially fitness or health care apps, require real time user data that have to be acquired from either a phone or a wearable device. While certain data such as movement can be tracked by using phone sensors, many additional information such as heart rate can mostly only be acquired by using further wearables as in the form of a chest strap or wrist band. An example for this is HealthGear [14], a real-time wearable system that monitors a person's physiological state by using an array of non-invasive Bluetooth sensors.

[1] `http://vandrico.com/wearables`, Retrieved August 14, 2015
[2] Society for Consumer Research

In certain situations it is also not possible or impractical to carry a smartphone, such as fast paced sports, sports that include physical play, swimming or sleeping whereas a wrist-worn device can easily be worn in these situations. Lastly certain kinds of movement are also not trackable by phones in a normal situation, such as hand or head movements as the phone is normally in a pocket or handbag.

While some wearables have a display and function on their own, most wearables are useless without a host device running a special software to communicate with it. However, developers have a hard time supporting all of these wearables and integrating them in their applications because most wearables are not using the same operating system and provide different *application program interfaces* (APIs) to include in an app as there's currently no global standard to access wearables. Even if they are running on the same system many manufacturers have created *software development kits* (SDKs) that have to be included in an app to fully utilize the functions of a wearable device. With the amount of different APIs and SKDs it becomes a lot of work for developers to add support for all devices and the number only keeps growing. In addition to that developers have to rewrite the wearable communication code that is used by basically any app supporting these devices, wasting a lot of time for general purpose code.

## 1.4 The Goal

The goal of this thesis is to create a fully modular Android framework, called AndroidSmart, that eases the process of creating new device drivers to communicate with wearables, but also the process of integrating said drivers into a mobile application. The finished framework will be shipped with a few drivers for certain devices and it can be extended nearly endlessly by adding new drivers for further devices. These new drivers should integrate effortless in the existing framework and are accessed as all previous ones, as the public interface is shielded from internal driver code so that differences between individual devices are irrelevant for a developer using the framework.

An application developer using the framework only specifies which device to connect to and which driver to use, as automatically detecting the type of the connected wearable is currently not supported by Android. Once the connection is established the developer can order the framework to register with the sensors available for that device. The connection status, data received and occurring errors are transmitted to the developer by the framework.

Any device supported by the framework should be accessed in the same way. Swapping devices on the fly should not stop the app from working. Furthermore a developer only has to implement what he wants to do with the data and not how he accesses them or in what format they are because the data output is standardized across all devices. The modular approach allows to add support for a basically unlimited amount of devices.

This thesis provides a general overview about the framework, example drivers for certain devices and a testing app as a proof of concept as well as an example to demonstrate what a developer has to do while using the framework.

# Chapter 2
# Related Work

This chapter introduces related papers and attempts that try to ease the work with external sensors. This includes theoretical works such as frameworks and toolkits, but also a technical realization with Bluetooth Low Energy.

## 2.1 ODK Sensors

Brunette et al. [3] analyzed the problems that occur while integrating external sensors in an application. Since users do not have administrative rights on their smartphones to include kernel level device drivers they explored ways to package software in a way that end-users can access external sensors on their locked devices. A hard-coded solution for a single device was not acceptable because the framework should support as many different sensors as possible. Their solution was a user level framework for Android to communicate with external sensors that has the following properties:

- Modularity, by lowering the workload to add new sensors to a minimum by abstracting communication, buffers and more, meaning a developer creating a new driver only has to implement low level driver code.
- Isolate application and sensor specific code from each other to avoid problems in the application when a sensor fails.
- Dealing with the trade-off of architectural approaches such as performance and modularity.
- Allow to add new sensors to an application by downloading the driver for the sensor from an application market in contrast to changing OS settings.

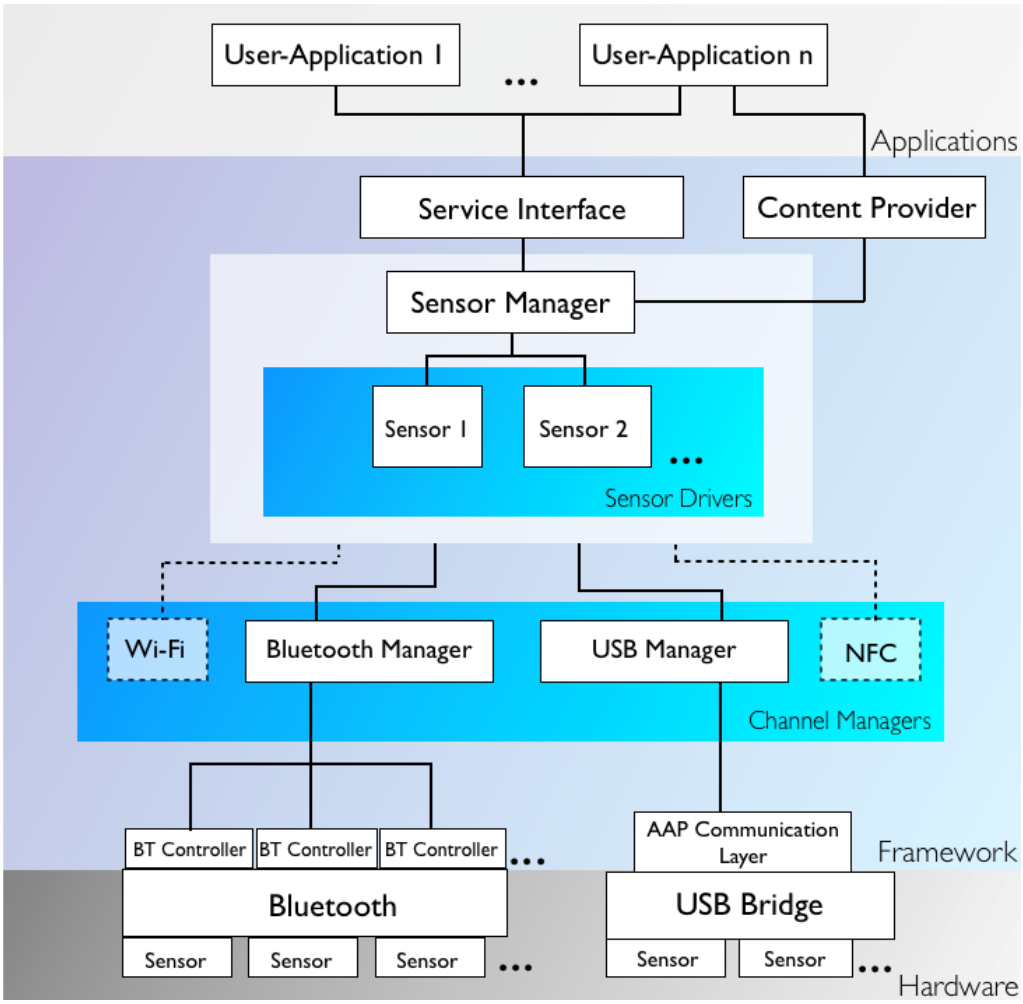



Figure 2.1: ODK Sensors framework architecture [3]

As seen in Figure 2.1 the framework consists of several layers that communicate with each other. All calls to the framework are forwarded by a *Sensor Manager* towards the corresponding *Sensor Driver* whereas each driver performs low level tasks. Each driver has access to different *Channel Managers* that abstract access to Wi-Fi, Bluetooth, USB and NFC. The chosen manager forwards the low level commands of each driver to the corresponding hardware device, receives the answers and passes them back to the driver.

New drivers therefore only have to implement the low level commands and send them to a channel manager, as sending data, threading or buffering is handled by the framework.

Drivers can be added to the framework in three different ways as shown in Figure 2.2. By default the framework has drivers integrated in itself. Adding the

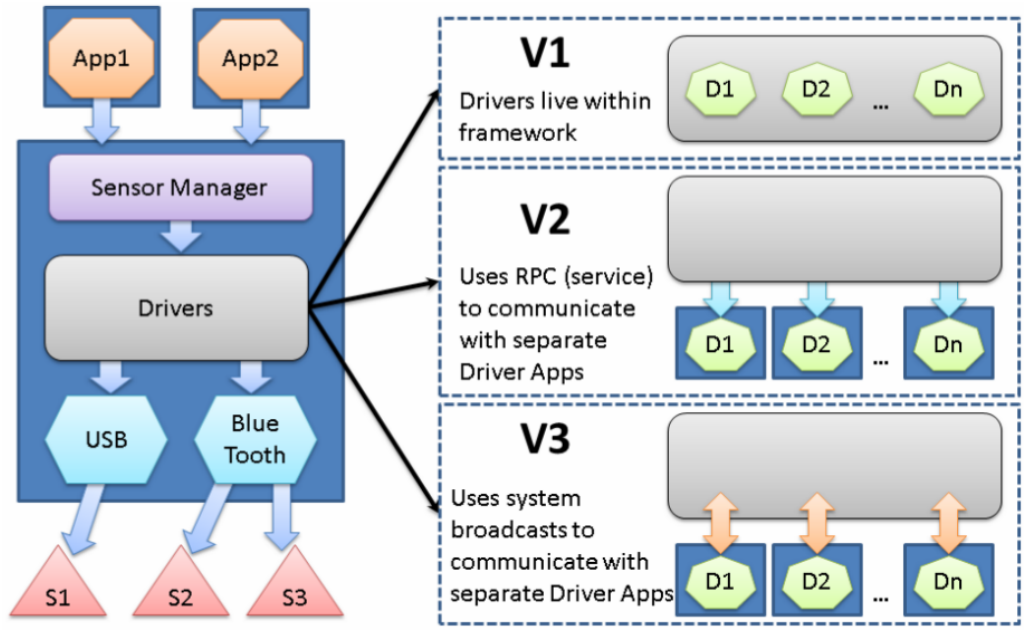



Figure 2.2: Possible ways of the framework to communicate with drivers [3]

framework to an app therefore adds the drivers for these external sensors to the app. If a user needs access to devices whose drivers are not part of the default framework, one can install additional external apps that provide these drivers. These apps have two different methods of communicating with the framework: RPC and system broadcasts.

At the time of writing there have been several real world projects that use ODK sensors. For example milk banks in South America are using ODK Sensors to monitor the pasteurization process of breast milk from donor mothers to deactivate pathogens [5]. The application uses ODK Sensors to read sensor values from the thermometer as well as to communicate with a printer to print the results of the process.

While ODK sensors ships an additional application to provide the background service, AndroidSmart will not do so but will work out of the box. While ODK Sensors focuses on low level drivers (mostly sending byte code) for devices such as external sensors with a Bluetooth module, AndroidSmart is going to focus on more intelligent devices such as smartwatches that do not accept byte strings as an input but rather use a manufacturer-provided SDK as ODK Sensors' design makes it hard to incorporate external SDKs into their framework.

## 2.2 Bluetooth Low Energy

Bluetooth Low Energy (BLE, also called Bluetooth Smart or Bluetooth 4.0) [9] is a standard developed by the Bluetooth Special Interest Group as an improved successor to Bluetooth 3.0 [15]. Although Bluetooth is primarily a communication protocol for different devices, it also includes a standard to transfer certain types of data defined by several Bluetooth profiles. Profiles in the current Bluetooth version include:

1. Blood pressure
2. Thermometer
3. Glucose
4. Heart rate
5. Running speed

and over 20 more.

BLE uses so called GATTs (Generic Attribute Profiles). They are descriptions for Bluetooth devices, specifying how to communicate with a device and what functions the device is capable of. Devices can run a GATT server, allowing external clients to request a list of GATT profiles of local sensors. Using these profiles clients can register with a sensor on the host device to receive messages containing sensor values or other information.

BLE is mostly used for irregular updates, meaning a devices stores sensor data locally and sends batches of data every certain minutes. Continuous transmission of data costs a lot of power and devices mostly return into a Bluetooth 3.0 transmission mode while doing so. This is probably also one of the reasons why BLE currently does not support accelerometer and gyroscope data, as these information are normally used in a real-time context (such as telling a runner how fast he is running, or how far he has been running up to now).

Although BLE support will not be provided by AndroidSmart as the main use cases are exactly accelerometer and gyroscope readings to provide linear acceleration values, BLE support will be a future goal once the hardware limitations will be overcome and profiles for these sensors are provided by BLE.

## 2.3 Wearable Sensing Framework for Human Activity Monitoring

Uddin et al. [18] created a wearable sensing framework for human activity monitoring with a focus on performance and a lowered battery consumption by following three design principles:

- Reducing the data communication overhead between the host and the wearable by pre-filtering the data on the wearable and only sending required data sets.
- Decoupling activity recognition from data processing to allow multiple applications to reuse the same preprocessed data.
- Providing flexibility to the activity application developer by creating APIs that ease the development process.

The framework is split up into parts on the wearable and the host device. The wearable contains the following components as seen in Figure 2.3:

- **Data collection**. Reads values from the sensors.
- **Data preprocessing**. Removes noise and sensor artifacts from the raw data.
- **Data segmentation**. Detects useful and useless data, meaning data that can be used to detect activities.
- **Sensing proxy**. Another filter based on rules send by the host device, allowing the host to accurately filter data on the wearable.
- **Local controller**. Communication channel between the wearable and the host.

The host contains the following components:

- **Activity application**. Application registering with the wearable sensing middleware that provides rules for the wearable to filter data.
- **Wearable sensing middleware**. Communication channel between the wearable and the host, also used for discovery of devices.

Applications can register with the middleware, telling it which device to connect to, what kind of data is wanted and how the data should be pre-processed to reduce the overhead. This information is sent to the wearable and processed, causing the device to only send required data to the host device. This obviously means that every supported wearable needs to have the application running.

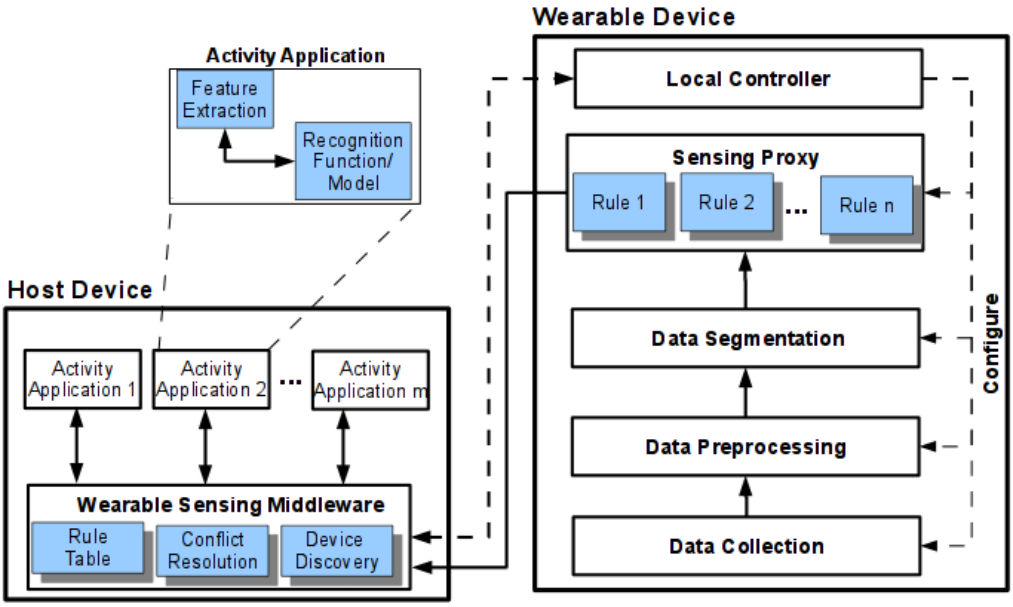



Figure 2.3: Wearable Sensing Framework for Human Activity Recognition [18]

Therefore the wearable application needs to be developed for every target wearable.

Although very useful, the rule based energy saving features of Uddin et al. are not going to be implemented in AndroidSmart as this would require to implement these rules on the companion apps for all devices. As several devices are usable without a companion app (because they have a service running providing the sensor values) and as simplicity is an important part of AndroidSmart creating additional apps is currently not desired.

## 2.4 Mscape

MScape is an extensible toolkit created by Clayton et al. [6] for Windows Mobile devices. Focusing on the same problems as ODK sensors, mainly development overhead for applications using sensors, MScape allows to create so called extensions using the *MScape Extension SDK*. It is capable of loading these extensions into an application, allowing these applications to use the sensors of connected wearables. MScape comes with three example extensions to show what the framework is capable of.

The **heart rate extension** is a driver for the Fraunhofer-Institute's *Pulseoximeter OxiSENS* that uses a *Bluetooth serial port profile* for communication. The extension connects to the device via the frameworks built-in serial port support and reads

the raw values from the sensor. Afterwards it uses an abstraction to classify the raw values, such as *resting, high activity* or *low activity*.

Using an external serial three-axis compass (built using an accelerometer and magnetometer), the **compass extension** provides an abstraction layer to translate raw sensor values into the closes 8-point compass direction (North, North east, South etc).

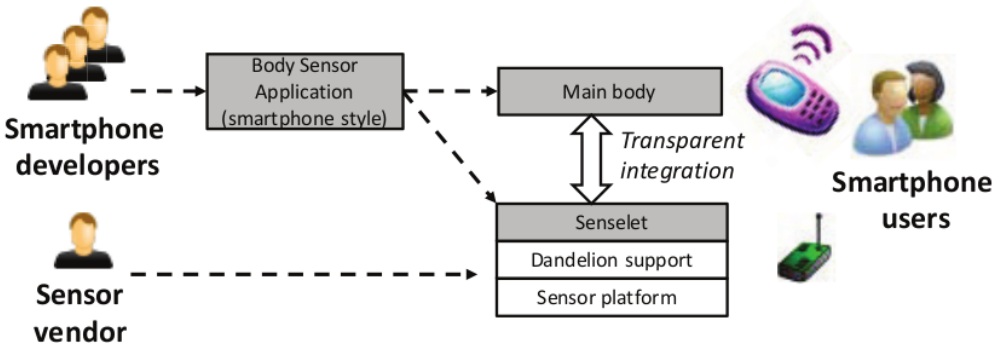
The **network discovery extension** is less a classical driver for an external sensor but rather a real network discovery tool supporting multiple network protocols such as Universal Plug and Play (UPnP) or Apple's Bonjour. The Bonjour part of the extension wraps the original Bonjour API into a smaller package, exposing only required values that allow to find, examine and bind with local network devices.

As MScape has been created for Windows Mobile, the framework and drivers are written in C#, making it unusable for Android. While porting it is possible, creating a new framework allows adding capabilities that MScape does not provide. MScape is also targeted at external sensors and network systems, not entire wearables, resulting in certain issues while using external SDKs to access them as it does with ODK Sensors.

## 2.5 Dandelion

Lin et al. [11] addressed the challenge of adding external sensors to mobile applications by creating Dandelion, a framework that allows smartphones to communicate with external sensors. Dandelion contains three core segments:

1. An abstraction layer called *senselet* to develop drivers for sensors.
2. A mechanism to integrate senselets into mobile applications.
3. A platform independent distribution and deployment tool for senselet executables.



Dandelion provides two runtimes that are capable of communicating with each other. The sensor runtime has to be included in a senselet running on a wearable while the smartphone runtime has to be included in the mobile application running on a phone. They communicate with each other using an XML-like interface description language.

Because senselets are heavily dependent on the device they run on, a two-phase compilation technique is used compile the code. Developers writing senselets compile it into an intermediate representation (IR) that gets shipped to the user. Once installed on the phone, a cross compiler compiles the IR into a native binary for the sensor the user wants to read data from.

Dandelion was prototyped for the Maemo Linux smartphone platform and uses the Rice Orbit body sensor platform[3] to communicate with external sensors. It is capable of integrating most external sensors on most mobile platforms, but it requires a cross-compiler for each and every sensor. This compiler simply does not exist for most of the current existing wearables as they are directly accessed by manufacturer provided SDKs. AndroidSmart is going to use exactly these SDKs to access the wearables sensor data.

[3] `http://www.ruf.rice.edu/~mobile/orbit/`, Retrieved November 30, 2015.

# Chapter 3
# Framework Structure

This chapter provides a general overview of the classes available in the framework. Afterwards important individual parts are explained in detail, showing available functions and illustrating the intended usage of the framework.

## 3.1 General Structure

The framework can be split up in three separate core segments as seen in Figure 3.1, with each being represented by a single class. Additionally there is a service or an application running on the wearable providing access to the sensor values.

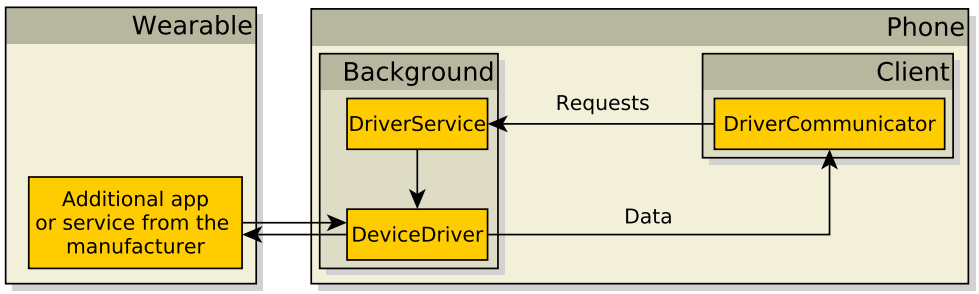



Figure 3.1: Framework structure

1. `DeviceDriver` is an abstract class that serves as a basic skeleton for all drivers. Drivers are running in the background thread of the `DriverService` and send data, connection and error messages to a connected client.

2. `DriverService` is the background service keeping track of running drivers. Clients always communicate with the service, while the service forwards requests and orders from each client to the appropriate driver.

3. `DriverCommunicator` is the main public class to be used and serves as a front-end client. It is abstract and contains various callback functions. A developer using the framework inherits a new class from the communicator and implements the callbacks that get triggered by messages from drivers and sometimes from the service.

### 3.1.1 DeviceDriver

The `DeviceDriver` class shown in Listing 3.1 is the abstract barebone for each implemented driver. The following public methods are supposed to be used by an implementation and are called by the service based on requests from a client.

```
1  public abstract class DeviceDriver {
2    // Methods called by the DriverService
3    public boolean registerSensor(Sensor sensor, Messenger
       messenger);
4    public int deregisterSensor(Sensor sensor, Messenger
       messenger);
5    public void deregisterSensors(Messenger messenger);
6    // Required functions
7    public void started();
8    protected void addSensorSupport(Sensor... sensors);
9    // Helper methods for an implementation
10   public int registeredCount(Sensor sensor);
11   public int clientCount();
12   public void stop();
13   public void sendData(Sensor sensor, float... data);
14   // Useful fields
15   protected final DriverService service;
16   protected final BluetoothDevice device;
17 }
```

Listing 3.1: Functions provided by the DeviceDriver class

`registerSensor`, `deregisterSensor` and `deregisterSensors` are called from the underlying service when receiving a message from a client. These functions should be implemented by a driver to actually register with the real sensors. `started` is mandatory to be called once the initialization of the driver is fully finished to notify the clients about the driver being ready. Without doing so, calls to register with a sensor are not processed and the driver is therefore not functional. Just as `started`, `addSensorSupport` needs to be called for the driver to function. Attempts to register with a sensor that has not previously been

added with this function are not processed and create an error. The remaining functions are helper functions to stop the driver (what is automatically done once all clients are disconnected) and to send sensor data to each client.

### 3.1.2 DriverService

The `DriverService` is a class inherited from a basic Android service. It is running in a background thread when at least one client attempted to connect to it. Its purpose is to instantiate drivers when requested and to redirect requests from a client to the corresponding drivers. The service is not supposed to be changed or extended. The only change affecting it are new drivers that are added to the `Drivers` enum, as the service instantiates drivers based on that enum.

### 3.1.3 DriverCommunicator

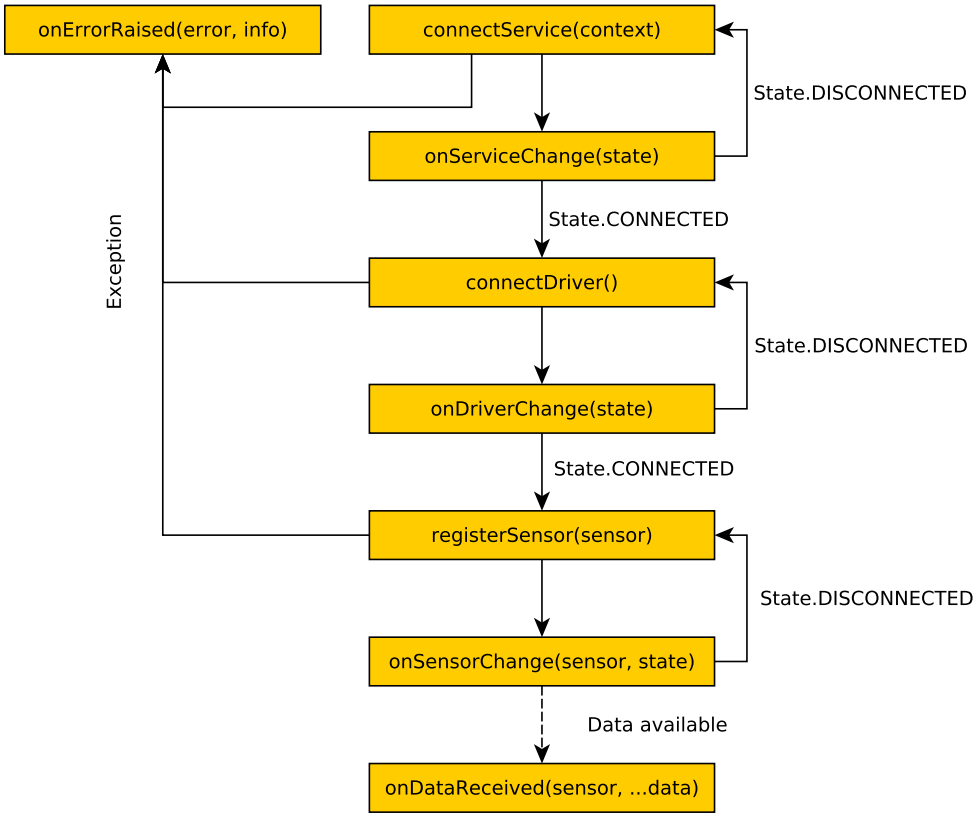
Listing 3.2 shows the `DriverCommunicator`, an abstract class providing several connection methods to register with the service, a driver or a sensor, each triggering a callback upon success or failure. It is the only class used by a developer integrating the framework into his or her application. Device and driver type are set upon initialization.

```
1  public abstract class DriverCommunicator {
2    public DriverCommunicator(String name, Drivers driver);
3    public boolean connectService(Context context);
4    public void disconnectService();
5    public abstract void onServiceChange(ConnectionState
       state);
6    public boolean connectDriver();
7    public void disconnectDriver();
8    public abstract void onDriverChange(ConnectionState
       state);
9    public void registerSensor(Sensor sensor);
10   public void deregisterSensor(Sensor sensor);
11   public abstract void onSensorChange(Sensor sensor,
       ConnectionState state);
12   public abstract void onErrorRaised(Error error, String
       info);
13   public abstract void onDataReceived(Sensor sensor,
       float ... data);
14 }
```

Listing 3.2: Functions of the public DriverCommunicator class

Using the framework happens completely asynchronously. Calling `connectService` starts the background service without an immediate response. Once the service has been started it sends a message back, triggering

`onServiceChange`. This is also true for the other connect/register/callback pairs: `connectDriver` and `onDriverChange` as well as `registerSensor` and `onSensorChange`. The expected work flow of a `DriverCommunicator` after instantiation is best explained by a flow diagram, starting at `connectService`:




## 3.2 Filters

The filter interface is an important addition to the framework that allows the preprocessing of data before being sent to an application. Filters can either be statically added to a driver implementation or dynamically to a communicator instance. Drivers are only supposed to use filters to provide basic functionality, for example to provide linear acceleration data for the accelerometer sensor or to deal with inaccurate hardware. Additionally drivers should not use filters to change the values of raw sensor data (for example when a client registers with the `ACCELEROMETER_RAW` sensor). The structure is very simple and can be seen in Listing 3.3.

```
1 public interface Filter {
2         public float[] filter(float ... data);
3 }
```

Listing 3.3: Filter interface

Each filter implements the filter method, taking an arbitary amount of data values and returning the processed data. Because the filter method takes a variable amount of values it can process data of every kind of sensor (for example, the GravityFilter could be used with a heart rate sensor to identify sudden changes in the heart rate). As most filters return the same amount of data values as they processed, filters can be combined in any way. The order of filters can also be defined to make use of filters that change the amount of values returned.

Clients on the other hand can use filters for anything. Currently there are the following filters available:

- GravityFilter to remove a static reoccuring value from sensor readings and provide linear acceleration values for the accelerometer.
- SmoothFilter to smooth data values over time by combining a part of current readings with old readings.
- High- and LowPassFilter as cut-off filters to remove all readings above or below a certain threshold (as an average of all values of one reading).

# Chapter 4
# Implemented Drivers

This chapter explains implementation details of each driver to demonstrate the work that was required to add it to the framework and contains information about advantages and issues of each driver and its corresponding device.

Some smart devices require companion apps that provide the data from the device itself, as in the case of the Pebble Watch or Android smartwatches. These companion apps are provided additionally to AndroidSmart and are introduced and explained in this chapter too.

## 4.1 Pebble Watch

The Pebble Technology Corporation provides an app for Android[4] that is required to use the Pebble and communicate with it.


They also provide an SDK[5] to develop Android apps that communicate with a Pebble companion app. These apps are running on the Pebble Watch and written in *C*. Only a few *C* libraries are available, but a custom Pebble library is included to handle communication, read sensor values and more. Communication between the apps is handled by sending `PebbleDictionary`s with the data inside.

[4] `https://play.google.com/store/apps/details?id=com.getpebble.android&hl=en`, Retrieved December 17, 2015

[5] `http://developer.getpebble.com/sdk/`, Retrieved December 17, 2015

### 4.1.1 Android Driver Implementation

The SDK for Android is available via Gradle[6].

```
// build.gradle
dependencies {
  compile 'com.getpebble:pebblekit:3.0.0'
}
```

The communication from `PebbleKit` with a Pebble Watch is fully asynchronous. Sending data is done by packing it in a `PebbleDictionary` and transferring it to a certain UUID. These UUIDs identify the corresponding companion app on the watch and are also used to start the companion app.

```
PebbleKit.startAppOnPebble(context, UUID);
PebbleDictionary data = new PebbleDictionary();
PebbleKit.sendDataToPebbleWithTransactionId(context,
   UUID, data, transactionId);
```

All messages which are sent have to be answered with an *ack* or a *nack*. These answers are received respectively by either a `PebbleAckReceiver` or a `PebbleNackReceiver`. In case of a *nack*, meaning the message was not accepted, the driver needs to react and attempt to resolve the situation.

Therefore it tries to redo the same call as before until the Pebble finally accepts the incoming message.

```
private class PebbleNack extends
   PebbleKit.PebbleNackReceiver {
  public void receiveNack(Context context, int
     transactionId) {
    switch (transactionId) {
      case REGISTER:
        sendRegisterMessage();
        break;
      default:
        break;
    }
  }
}
```

To avoid an endless loop, `sendRegisterMessage` internally tracks how often a connection attempt has been made, stopping the driver after several failures.

Messages sent by the watch are received by a `PebbleDataReceiver` and have to be answered as well.

[6] `http://gradle.org/`, Retrieved December 18, 2015

```
1 private class PebbleReceiver extends
     PebbleKit.PebbleDataReceiver {
2   public void receiveData(Context context, int
       transactionId, PebbleDictionary pebbleTuples) {
3     PebbleKit.sendAckToPebble(context, transactionId);
4   }
5 }
```

Once `receiveData` is called due to a message from the companion app the sensor values can be extracted from the `PebbleDictionary` and sent to connected clients.

The data values retrieved from the watch include the influence of gravity so a `GravityFilter` is used to provide linear acceleration values for the `ACCELEROMETER` sensor. Data values are also smoothed because the accelerometer of the Pebble Watch jitters a lot.

```
1 addFilter(Sensor.ACCELEROMETER, new GravityFilter(3));
2 addFilter(Sensor.ACCELEROMETER, new SmoothFilter(3));
```

Raw values can still be accessed by using the `ACCLEROMETER_RAW` sensor.

### 4.1.2 Companion App Details

The companion app listens to incoming messages and waits for a message to tell it to register with a sensor. Data values of the sensor are buffered in an array to reduce the overhead from sending data too often, as permanently sending data leads to connection interruptions. Currently ten readings are buffered before being sent to the client. As the companion app creates 50 sensor readings per second data is delayed by 200 miliseconds before being sent. While the Pebble could create 100 sensor readings per second to lower the delay to 100 miliseconds, this idea was abandoned to lower the power consumption on the watch.

### 4.1.3 Challenges and Advantages

The Pebble Watch caused certain issues while creating the driver. Although the Pebble is capable of notifying a developer in its app that an ACK has been received, sending new data immediately when this happens leads to a lot of connection issues. It appears that either the Bluetooth radio in the watch is unable to send the data fast enough (although it should be 'ready' when receiving an ACK because this means the old data was sent completely) or something else is going wrong in the software layer. This led to the requirement of buffering the data on the device and sending them in bulk. This obviously has the advantage of reducing the overhead from packet headers and connection initialization but it also means that the phone receives data delayed. The current delay of 200 miliseconds is noticeable when plotting the data on the phone but it is better than

a 500–1000 milisecond interrupt roughly every five seconds when using the old method.

Most importantly the reduced overhead from packet headers does not only affect the Pebble but obviously also the phone, saving battery life and reducing the current load on the device.

## 4.2 Microsoft Band

Microsoft also provides an app for Android called Microsoft Health. It is required to use the Microsoft Band and communicate with it. The SDK allows to directly register with the sensor, meaning there is no additional companion app required.


### 4.2.1 Android Driver Implementation

The SDK is not available via Gradle but can be downloaded manually[7] and added to the {module}/libs directory.

The communication can be done synchronously by waiting for the calls to return or asynchronously by implementing certain callbacks. The provided implementation waits for messages to return. This requires the driver to implement the `Runnable` interface and run in its own thread to not block the main thread of the framework.

Connecting to the band is easy by querying all available bands and selecting one.

```
1 BandInfo[] pairedBands =
     BandClientManager.getInstance().getPairedBands();
2 bandClient = BandClientManager.getInstance()
3         .create(service.getApplicationContext(),
             pairedBands[0]);
4 BandPendingResult<ConnectionState> pendingResult =
     bandClient.connect();
```

[7] `http://developer.microsoftband.com/bandSDK/`, Retrieved December 17, 2015

Afterwards one can register with a sensor.

```
1 gyroscopeListener = new GyroscopeListener();
2 bandClient.getSensorManager()
3         .registerGyroscopeEventListener(gyroscopeListener,
             SampleRate.MS32);
```

Data values are transmitted in a callback.

Just like for the Pebble Watch data values include the influence of gravity so a `GravityFilter` is used to provide linear acceleration values for the `ACCELEROMETER` sensor.

```
1 addFilter(Sensor.ACCELEROMETER, new GravityFilter(3));
```

Raw values can still be accessed by using the `ACCLEROMETER_RAW` sensor.

### 4.2.2 Challenges and Advantages

As the SDK allows direct communication with the Band without a companion app users have to download anything apart from the original app they wanted to use.

Microsofts API does not use the same format as Androids internal Bluetooth API, so matching names between the originally selected paired device and Microsofts device query is not possible (and Microsofts SDK does not expose the MAC address of the device). This means that the driver potentially uses the wrong device if multiple Microsoft Bands are connected with one phone. However, even Androids built in support for multiple wearables is still not stable, so this drawback has been accepted as the support relies in this functionality of Android. To work around this, the front-end DriverCommunicator would need a change to be able to query devices based on Microsofts SDK. This is not wanted, as driver code should be separated from the framework code itself. Potentially this could be solved by expanding the base driver code, allowing the framework to run certain parts of a driver before it has been instantiated (in this case query for available devices). Considering the issues arising with it and the fact that barely anyone uses multiple identical wearables with one phone this feature was not added but thought about for future work.

## 4.3 Android Wear

Android Wear has the advantage of being natively supported by Android. The connection is established by using the `GoogleApiClient` and communication is handled by the `MessageAPI`.

### 4.3.1 Android Driver Implementation

Although the wearable API is part of Android, it is not enabled by default but comes packed as individual libraries. They are available via Gradle.

```
1 dependencies {
2     compile 'com.google.android.support:wearable:1.2.0'
3     compile
          'com.google.android.gms:play-services-wearable:
          7.5.0'
4 }
```

The Android driver implements the `MessageListener` interface to receive messages from a watch as well as the `ConnectionCallbacks` and `OnConnectionFailedListener` interfaces to be notified about the connection status. After connecting a `GoogleApiClient`

```
1 googleApiClient = new GoogleApiClient.Builder(service)
2         .addApi(Wearable.API)
3         .addConnectionCallbacks(this)
4         .addOnConnectionFailedListener(this)
5         .build();
6 googleApiClient.connect();
```

the driver registers itself as a listener.

```
1 Wearable.MessageApi.addListener(googleApiClient, this);
```

After sending messages to the wearable

```
1 Wearable.MessageApi.sendMessage(googleApiClient, node,
     START, null);
```

responses are received in the `MessageListener` callback.

In comparison to the Pebble Watch or the Microsoft Band the basic driver for Android Wear does not use any filters. The sensor values are mostly jitter free on the device used for testing and Android Wear provides its own implementation of a virtual linear acceleration sensor. Filters can still be used in the `DriverCommunicator` to further filter the data before it is sent to the application (as in the case of worse devices to remove any jitter its sensors have).

### 4.3.2 Companion App Details

In comparison to the Android driver, the companion app does not use a `MessageListener` but a `WearableListenerService`. If this service is properly set up in the manifest, messages from Android apps with the same application id as the wearable app automatically cause the service to be instantiated to transmit the message.

If said service receives a `START` message it starts a secondary service and transmits the id of the watch to it.

```
1 Intent inte = new Intent(getApplicationContext(),
     DataService.class);
2 inte.putExtra(NODE, messageEvent.getSourceNodeId());
3 startService(inte);
```

The secondary service reads sensor values from the wearable

```
1 sensorManager.registerListener(instance, sensor,
     SensorManager.SENSOR_DELAY_FASTEST);
```

and sends them to the phone.

```
1 // buffer is a byte array containing the sensor values
     and an
2 // identifier for the sensor
3 Wearable.MessageApi.sendMessage(googleApiClient, node,
4         ConnectionListener.DATA, buffer.array())
5         .setResultCallback(new MessageCallback());
```

### 4.3.3 Challenges and Advantages

Just as Android allows to read sensor values from all Android devices, the Android Wear driver allows to read sensor values from all Android Wear devices. Therefore this driver provides support for a broad range of devices.

Creating the driver caused some issues, because the application id of an Android app and a wearable app have to be the same. This is intended as a safety feature to ensure that only the wearable app that belongs to a mobile app receives the corresponding messages. In this case however the companion app is supposed to be used by multiple mobile applications. It seems that this use case is not intended by Google and can not be circumvented for now. This means a mobile developer either has to inherit the id of the wearable application, or ship a new version of wearable application with a matching id.

# Chapter 5
# Testing Application

To test the framework and its drivers an Android app has been created using the framework to read accelerometer values of connected wearables.

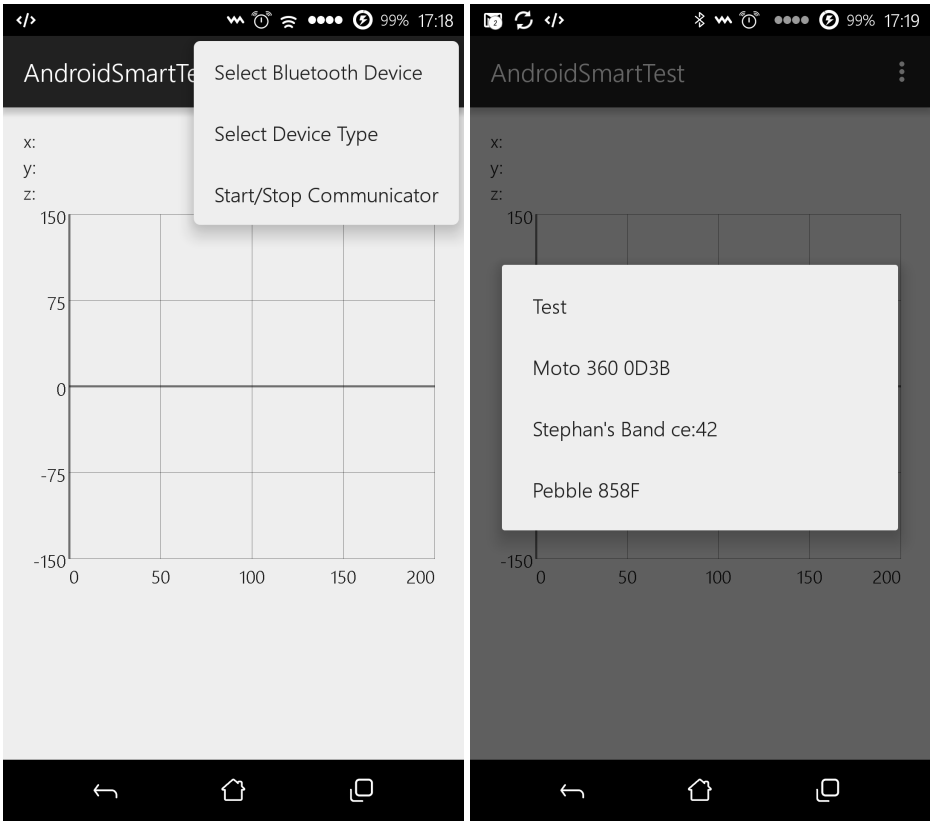
## 5.1 App Usage

The application has three options in a menu.

Figure 5.1b shows the device selection menu that allows to select a paired Bluetooth device to connect to. Currently devices have to be paired to be used by the framework, so showing all paired devices is sufficient. The application does not check if the devices on the list are currently available but considering the average amount of paired devices the list should normally be very short (most times only a single value).

In Figure 5.1c the *Select Device Type* creates a dialog that allows the user to select which type of device he wants to connect to. The list has been created by using the `Drivers` enum.

Pressing *Start/Stop Communicator* binds with the background service, starts the selected driver and registers with the accelerometer (filtered linear acceleration in this case). Once values are transmitted by the framework the app shows them as seperated x, y and z values and also plots them on a graph as seen in Figure 5.1d.




(a) App Menu (b) Driver Type Selection

(c) Device Type Selection (d) Driver running

Figure 5.1: Framework testing application

## 5.2 Implementation

The testing app contains the framework in its module/{libs} folder and added it to its Gradle file.

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile (':androidsmart@aar') {
        transitive = true
    }
    compile 'com.getpebble:pebblekit:3.0.0'
}
```

The transitive flag is required to automatically include libraries the framework uses. The Pebble SDK still needs to be added manually.

The app itself is a single Activity containing the graph and the menu. Once the selections are made and the starting command is selected, a new `DriverCommunicator` is created and a connection to the background service is established.

```
private void startStopCommunicator() {
    driverCommunicator = new DriverCommunicator(name,
       driver) { // callbacks shown afterwards
    }
    driverCommunicator.connectService(getApplication()
        .getApplicationContext());
}
```

As `DriverCommunicator` is abstract all callbacks need to be implemented. The implementations are shown in order of their sequence of actions.

Once the connection with the service is established the clients tries to connect to the corresponding driver.

```
@Override
public void onServiceChange(ConnectionState
   connectionState) {
    if(connectionState == ConnectionState.CONNECTED) {
    connectDriver();
    }
}
```

Once the connection to the driver is established the client tries to register with the accelerometer of the device (the accelerometer is used because all initial testing devices had such a sensor).

```
1 @Override
2 public void onDriverChange(ConnectionState
     connectionState) {
3     if(connectionState == ConnectionState.CONNECTED) {
4         registerSensor(Sensor.ACCELEROMETER);
5     }
6 }
```

The callback called `onSensorChange` that triggers once successfully connected is not used in this implementation as the app simply waits for incoming data and plots them on the graph.

```
1 @Override
2 public void onDataReceived(Sensor sensor, float... data) {
3         xText.setText("x: "+data[0]);
4         yText.setText("y: "+data[1]);
5         zText.setText("z: "+data[2]);
6         // Plotting the values on the graph
7 }
```

Once the menu is used to stop the application it first sends a stop request to the driver (that shuts itself down because there are no further connected clients) and then disconnects from the service.

```
1 if (driverCommunicator.serviceConnected()) {
2         driverCommunicator.disconnectDriver();
3         driverCommunicator.disconnectService();
4 }
```

The corresponding callbacks are triggered again (this time transmitting `ConnectionState.DISCONNECTED`) but are not required for this application and are therefore not used.

## 5.3 Testing

The application has been tested on a Nexus 5 device with several wearables: a Pebble Watch, a Microsoft Band and a Moto 360 Android Wear watch.

Connecting and disconnecting devices works without any problems. Selecting invalid combinations (such as a paired Android Watch but selecting the Band driver) leads to the driver in the background automatically shutting down as no Band could be found by the driver and no other client is connected (multiple applications can use the background service at the same time). This is the intended behavior. Physically moving the devices far away from the phone or shutting them down triggers a callback in the driver in all cases, causing the driver to shut down and notify the framework - and therefore the main application.

# Chapter 6
# Conclusions

As seen in the introduction, the wearable market (hardware as well as software) is a very fast growing market that has the capability to grow even further. As the manufacturers have neither a standard operating system nor a common SDK or API, developers that want to create apps for (but not limited to) the even faster growing market of healthcare and fitness applications face the issue that adding support for these wearables is a huge amount of work that gets replicated by each and every developer over and over again.

A structure for a framework solving this issue has been proposed and implemented. Drivers for three different devices have been created and added to the framework. Lastly an example application has been created to test the framework and show its general workflow. The application is capable of freely switching between wearables and has been tested with several different devices. It is capable of reading sensor values of all of them. Movements are reported on the same scale allowing a mobile developer to create code that is independent of the connected wearable.

While the framework and the testing app are in a functional state, there are certain existing limitations. The biggest limitation comes from the manufacturers themselves. Many devices use a proprietary protocol for the communication with their provided app. To add support for such devices it is required that the manufacturer provides an SDK that exposes the sensors. While for example the Microsoft Band and Pebble Watch do so, the fitness trackers from Garmin[8] or Jawbone[9] do not, making it impossible to add support without reverse engineering their protocol. This is not attempted in this thesis and therefore support for these devices is not provided.

[8] `https://explore.garmin.com/en-US/vivo-fitness/`, Retrieved December 13, 2015
[9] `https://jawbone.com/up/developer/faq`, Retrieved November 30, 2015

Another limitation comes from the hardware built into the devices. Although the framework tries to provide data in the same range and as accurate as possible, different quality accelerometers provide different data sets. In addition to that the Bluetooth transmission speed and quality also cause devices to provide different data values at different times. A good example for this issue is the Bluetooth radio of the Pebble Watch that causes regular short connection interruptions when transmitting data.

Although several manufacturers provide SDKs to develop applications for their devices and communicate with them from a smartphone, reading sensor values without creating a companion app is rarely possible. This means that after a user downloaded an application with AndroidSmart support, he still has to download an appropriate companion app or the smartphone app will not work.

While the limitations appear to be huge, these limitations affect everyone in the industry. If a manufacturer does not want to open their protocol there is not much a mobile developer can do about it and it does not matter if he uses the framework or not. For all other devices AndroidSmart provides an easy and standardized access to sensor data that otherwise would cause a huge amount of work to acquire.

# Chapter 7
# Future Work

As seen with the framework of Uddin et al. [18], a rule-based system to filter sensor data on wearables can reduce the overhead of data communications by 15-50%[18]. Although not practical for devices that do not require an additional companion app by default, adding additional filters in the form of a rule-based system can still be done.

While it is quite a lot of work if many devices are to be supported, companion apps can be extended by a protocol to accept rules from a phone. Each individual device would filter the data beforehand and only send requested data sets. For devices that do not require an additional application the rule based filtering can be done on the phone just as the current filter implementation does. This approach allows companion apps to save power while developers can filter data even further without noticing a difference between devices with a companion app and devices without.

Currently devices need to be paired with the phone for the framework to work. This is a requirement that was deemed to be acceptable as interacting with non-paired devices (such as certain BLE devices) requires constant scanning on the phone to find said devices, causing huge delays in the flow of the application. Adding support for BLE is desirable in the future to support a wider range of devices, even though the usage of these devices in this way is limited and slow.

As identification of devices is done using the Android Bluetooth API, scanning processes of external SDKs are currently not supported. This means that if multiple devices are present that can only be seperated using an external SDK, current drivers simply use the first available. As multi-wearable systems are rarely used, especially not with the same device multiple times, this is not a big issue. Still, an extension to the driver barebone including several static methods, such as device scan or information about the underlying code, can be added in the future.

AndroidSmart has only been tested in the presented example implementation for now. Real life tests with real users and different applications have to be done to prove the robustness of the framework and reveal potential flaws. This includes problems when accessing the framework from multiple applications at once but also other issues that can occur when using the framework a lot.

Especially mobile developers have to test the framework to find design flaws or missing functionality. While the available methods are sufficient for the testing application, developers might need other functions or information for their own application.

# Bibliography


[1] Number of available applications in the Google Play Store from December 2009 to February 2015. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2015. Retrieved August 23, 2015.

[2] Number of apps available in leading app stores as of July 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 2015. Retrieved August 23, 2015.

[3] Waylon Brunette, Rita Sodt, Rohit Chaudhri, Mayank Goel, Michael Falcone, Jaylen Van Orden, and Gaetano Borriello. Open Data Kit Sensors: A Sensor Integration Framework for Android at the Application-level. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 351–364, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1301-8. doi: 10.1145/2307636.2307669. URL http://doi.acm.org/10.1145/2307636.2307669.

[4] Bill Buxton. 31.1: Invited Paper: A Touching Story: A Personal Perspective on the History of Touch Interfaces Past and Future. *SID Symposium Digest of Technical Papers*, 41(1):444–448, 2010. ISSN 2168-0159. doi: 10.1889/1.3500488. URL http://dx.doi.org/10.1889/1.3500488.

[5] Rohit Chaudhri, Waylon Brunette, Bruce Hemingway, and Gaetano Borriello. ODK Sensors: An Application-level Sensor Framework for Android Devices. In *Proceedings of the 3rd ACM Symposium on Computing for Development*, ACM DEV '13, pages 30:1–30:2, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1856-3. doi: 10.1145/2442882.2442918. URL http://doi.acm.org/10.1145/2442882.2442918.

[6] B. Clayton, R. Hull, T. Melamed, and R. Hawkes. An extensible toolkit for context-aware mobile applications. In *Wearable Computers, 2009. ISWC '09. International Symposium on*, pages 163–164, Sept 2009. doi: 10.1109/ISWC.2009.42. URL http://dx.doi.org/10.1109/ISWC.2009.42.

[7] Sam Costello. How Many Apps Are in the iPhone App Store? http://ipod.about.com/od/iphonesoftwareterms/qt/apps-in-app-store.htm, October 2014. Retrieved August 23, 2015.

[8] D. Desjardins. Mobile Healthcare Apps Slated for Federal Oversight. http://healthleadersmedia.com/content.cfm?topic=TEC&content_id=288733, January 2015. Retrieved August 09, 2015.

[9] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. *Sensors*, 12(9), June-August 2012. doi: 10.3390/s120911734. URL http://dx.doi.org/10.3390/s120911734.

[10] A. Kailas, Chia-Chin Chong, and F. Watanabe. From Mobile Phones to Personal Wellness Dashboards. *Pulse, IEEE*, 1(1):57–63, July 2010. ISSN 2154-2287. doi: 10.1109/MPUL.2010.937244. URL http://dx.doi.org/10.1109/MPUL.2010.937244.

[11] Felix Xiaozhu Lin, Ahmad Rahmati, and Lin Zhong. Dandelion: A Framework for Transparently Programming Phone-centered Wireless Body Sensor Applications for Health. In *Wireless Health 2010*, WH '10, pages 74–83, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-989-3. doi: 10.1145/1921081.1921091. URL http://doi.acm.org/10.1145/1921081.1921091.

[12] Amanda Martin. GfK forecasts 51 million wearables will be bought globally in 2015. http://www.gfk.com/news-and-events/press-room/press-releases/pages/gfk-forecasts-51-million-wearables-sold-globally-2015.aspx, March 2015. Retrieved August 14, 2015.

[13] Ariel Michaeli. App Stores Growth Accelerates in 2014. http://blog.appfigures.com/app-stores-growth-accelerates-in-2014/, January 2015. Retrieved August 23, 2015.

[14] Nuria Oliver and Fernando Flores-Mangas. HealthGear: A Real-time Wearable System for Monitoring and Analyzing Physiological Signals. *Wearable and Implantable Body Sensor Networks, International Workshop on*, 0:61–64, 2006. doi: 10.1109/BSN200627. URL http://doi.ieeecomputersociety.org/10.1109/BSN.2006.27.

[15] M. Patel and Jianfeng Wang. Applications, challenges, and prospective in emerging body area networking technologies. *Wireless Communications, IEEE*, 17(1):80–88, February 2010. ISSN 1536-1284. doi: 10.1109/MWC.2010.5416354. URL http://dx.doi.org/10.1109/MWC.2010.5416354.

[16] I. Perez. All in Good Time. http://texascooppower.com/texas-stories/people/all-in-good-time, February 2012. Retrieved November 13, 2015.

[17] K. Taylor. Connected health: How digital technology is transforming health and social care. *Deloitte Centre for Health Solutions*, 2015.

[18] Mostafa Uddin, Ahmed Salem, Ilho Nam, and Tamer Nadeem. Wearable Sensing Framework for Human Activity Monitoring. In *Proceedings of the 2015 Workshop on Wearable Systems and Applications*, WearSys '15, pages 21–26, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3500-3. doi: 10.1145/2753509.2753513. URL http://doi.acm.org/10.1145/2753509.2753513.

[19] Neil Zhao. Full-Featured Pedometer Design Realized with 3-Axis Digital Accelerometer. *Analog Dialogue*, 44(06), June 2010. URL http://www.analog.com/static/imported-files/tech_articles/pedometer.pdf.

[20] Vincent W. Zheng, Yu Zheng, Xing Xie, and Qiang Yang. Collaborative Location and Activity Recommendations with GPS History Data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1029–1038, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772795. URL http://doi.acm.org/10.1145/1772690.1772795.