ESTUDIOS DE INGENIERÍA EN INFORMÁTICA

DESIGN, IMPLEMENTATION AND VALIDATION OF A MOBILE FRAMEWORK FOR AGILE DEVELOPMENT OF MHEALTH APPLICATIONS

REALIZADO POR:

RAFAEL GARCÍA FERNÁNDEZ

DIRIGIDO POR:

ORESTI BAÑOS LEGRÁN E IGNACIO ROJAS RUÍZ

DEPARTAMENTO:

ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

Granada, 14 de Diciembre de 2013

Design, implementation and validation of a mobile framework for agile development of mHealth applications

por

Rafael García Fernández

PALABRAS CLAVE: Framework, Android, Dispositivos portables biomedicos, Dispositivos móviles, Sensores cinemáticos, Sensores fisiologicos, Electrocardiograma, Bluetooth, Reconocimiento de actividades, Inferencia de conocimiento, Extracción de características, Selección de características, Clasificación.

RESUMEN: la esperanza de vida se ha visto drásticamente aumentada en el último siglo, lo que ha provocado un fenomeno de envejecimiento de la población mundial y un crecimiento en la aparición de enfermedades crónicas. Esto, unido a la falta de recursos médicos de calidad en los lugares más desfavorecidos del mundo obliga a buscar alternativas para paliar estos problemas. La tecnología móvil aparece en este contexto debido a las numerosas ventajas que ofrecen las aplicaciones biomedicas. Para facilitar su desarrollo, en este trabajo se ha desarrollado un framework sobre la plataforma Android, con el que se pueden desarrollar aplicaciones biomédicas de calidad de manera rápida Las principales funcionalidades que el framework provee son comunicación a dispositivos biomédicos portables, almacenamiento local y remoto de la información recogida, visualización online y offline, inferencia de conocimiento de carácter biomédico y guidelines. Para probar las funcionalidades del framework se ha desarrollado una app y se ha realizado un estudio con un conjunto de inviduos para construir un modelo de reconocimiento de actividades físicas.

KEYWORDS: Framework, Android, Portable biomedical devices, Wearable biomedical devices, Portable mobiles, Kinematics sensors, Physiological sensors, Electrocardiogram, Activity recognition, Knowledge inference, Feature extraction, Feature selection, Classification.

ABSTRACT: the human population life expectancy has drastically risen in the last century, translating into a larger elder population with increased appearance of chronic deseases. This, along with the lack of quality health facilities in developing countries, makes essential the seek of alternatives to solve these problems. Mobile technology demonstrates as the key to encounter this challenges due to many adventages that biomedical applications offer. In order to facilitate its development, a framework for the agile and flexible development of biomedical applications is here provided. The main functionalities provide for this framework are communication with portable biomedical devices, local and remote data storing, data visualization either online or offline, medical knowledge inference and guidelines support. To validate the framework functionalities an app has been developed and a study has been conducted to collect a dataset used to build a model for activity recognition.

D. Oresti Baños Legrán y D. Ignacio Rojas Ruiz,
Profesores del departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada, como directores del Proyecto Fin de Carrera de D. Rafael García Fernández

Informan:

que el presente trabajo, titulado:

Design, implementation and validation of a mobile framework for agile development of mHealth applications

Ha sido realizado y redactado por el mencionado alumno bajo nuestra dirección, y con esta fecha autorizamos a su presentación.

Granada, a 14 de Diciembre de 2013

Fdo. ____________________________

Los abajo firmantes autorizan a que la presente copia de Proyecto Fin de Carrera se ubique en la Biblioteca del Centro y/o departamento para ser libremente consultada por las personas que lo deseen.

Granada, a 14 de Diciembre de 2013

Oresti Baños Legrán (cotutor)	DNI:	Firma:
Ignacio Rojas Ruíz (cotutor)	DNI:	Firma:
Rafael García Fernández (autor)	DNI:	Firma:

A mi padre, mi madre, Iga Rzeszewska y mis amigos.

Acknowledgements

Me gustaria agredecer a la gente que me ha ayudado en el desarrollo de este proyecto. A Ignacio Rojas por animarme y permitirme realizarlo. A Oresti Baños por toda su ayuda, un ejemplo como tutor y persona. A todos mis amigos, en especial a aquellos que pusieron su granito de arena participando en los experimentos. A mi padre por ser un padre ejemplar y por siempre estar ahi. Por ultimo, a Iga Rzeszewska por todo su apoyo y paciencia.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

1.1 Motivation and Context

The human population life expectancy is constantly increasing since centuries. This increase has been specially noteworthy in twentieth century, which according to a study of Max Planck Institute for Demographic Research in Rostock, life expectancy has risen faster than it did in the previous 200.000 years (11). Since 1840, the life expectancy of a newborn baby in a western industrialized nation has risen by approximately 3 months per year. This trend continues today and is surprisingly constant, which means that it has not achieved its upper limit (Figure 1.1). This is due to advanced medical knowledge in developed countries, better health habits and infectious disease control.

According to Kevin G. Kinsella (12), in the middle of twentieth century countries belonging to Western Europe had the longest life expectancy, standing out Norway, The Netherlands and Sweden with 71-73 years old (average male-female). Closely follow, Canada and Australia with a 70-71 years old of expectancy. Regions such as Southern and Eastern Europe barely reached 65 years old, whereas United States of America had a life expectancy of 69 years old. In 2011 according to the World Health Organization (WHO) (13), a significant enhancement took place over the world, reaching the longest life expectancy with 83 old in Japan, San Marino and Switzerland. Countries like Spain, France or Italy with 82 years old experimented an enhancement of 15 years. The rise of the life expectancy in Japan is worth to mention, since it was around 61 in the fifties, meaning an increase of more than 20 years.

However, most of developing countries in Africa and Asia have kept lifespans shorter

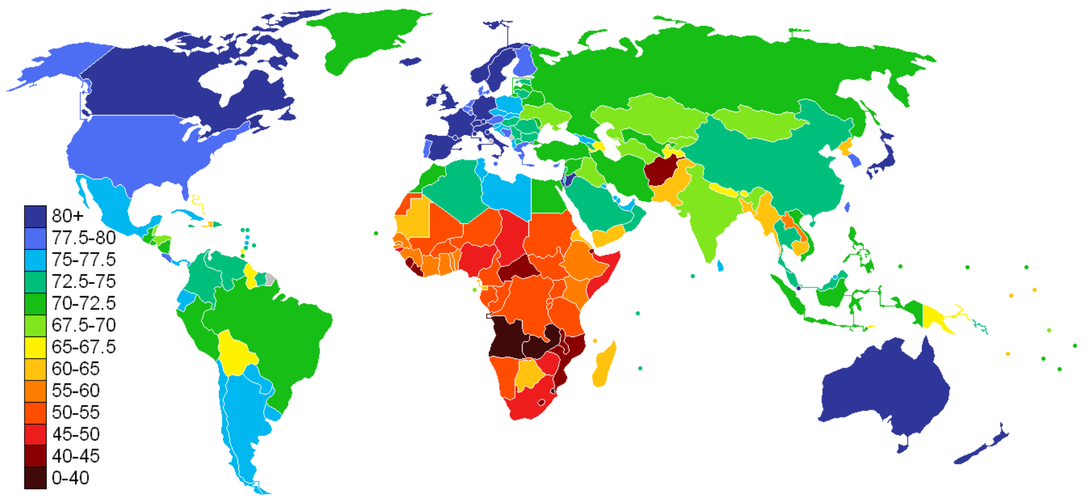


Figure 1.1: Life expectancy over the world in 2011. Figure obtained from (1)

than developed countries, except for some countries of Asia as India where a huge increase has been observed. This is consequence of several factors, such as lack of laboratory equipments -that leads to poor diagnostics- or pandemics diseases. For example, in some sub-Saharan countries severely affected by the HIV/AIDS pandemic, life expectancy has decreased in the past two decades due to increases in premature death (14). An interesting measure tool is the Service Availability and Readiness Assessment (SARA) developed by the WHO that was conducted in many developing countries to asses and monitor service readiness, capacity and health facility levels. The results of the assessments indicates that although the basic equipment score tends to be high, the laboratory diagnostic score is usually the lowest, as can be seen in examples in Sierra Leona SARA 2011 report (15) and Zambia SARA 2010 report (16). The WHO reveals that a 66 % of Africa population not have access to health quality services, whereas in some areas of Asia this percentage is 57 %. Therefore, the provision of health facilities within the acceptable range of access remains a challenge.

While in developing countries the main problem stays the lack of quality health facilities, in western countries a good health service has lead to a drop of mortality in several aspects, such as chronic diseases and hazards to human health. In its report of 2009, the WHO estimates that the leading global risks for mortality in the world are high blood pressure (responsible for 13% of deaths globally), tobacco use (9%), high blood glucose (6%), overweight and obesity (5%). These risks are responsible for

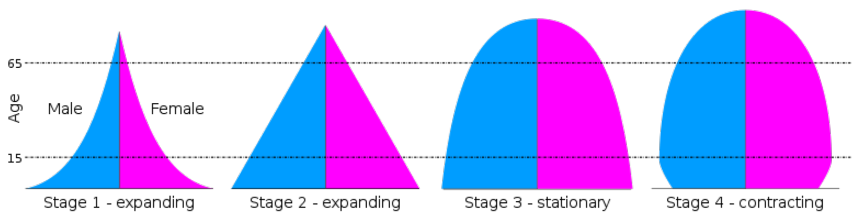


Figure 1.2: Population pyramids for four stages of the demographic transition model. Figure obtained from (2)

raising the probability of chronic diseases such as heart disease, diabetes and cancer. They affect countries across all income groups: high, middle and low.

By contrast, the increase of the life expectancy has meant a phenomenon in world population. The so-called population pyramid is a graphical illustration showing the distribution of various age groups in a population, which forms the shape of a pyramid when the population is growing. In the Figure 1.2 are shown the different stages that the global population is going through. The first stage is similar to the population pyramid found at the beginning of the twentieth century, with a high birth rate, rapid fall in each upward age group due to high death rates and by last short life expectancy. Second stage shows an slightly improvement in the life expectancy and a fall in death rate, living more people in middle age (around 1950). The third stage shows declining birth rate, low death rate and more people living to old age (expected for twenty-first century, the current population pyramid is changing into this shape). Lastly, stage four shows the pyramid contracting, representing low birth and low death rate, higher dependency ratio and longer life expectancy (around 2100) (2).

These statistics bring a fact: nowadays humans live longer, but not in a healthy way. With the current life expectancy, short-term diseases have turned into chronic ones and people with degenerative diseases live longer, however suffering severe health conditions. Patients, specially elder people, should pay special attention to their health if they suffer from one condition, due to their poor health, they can be more prone to suffer others related ones. For example, patients with diabetes or obesity conditions usually tend to have heart problems. This is why healthy habits are important not only to fight against one condition but also to prevent patients from related diseases. A healthy diet is always important to fight against diabetes or obesity conditions. Moreover, walking and running not only help keep fit, but also avoid heart problems.

1. INTRODUCTION

Unfortunately, quality health resources are still far from having a complete coverage, due to the lack of resources and the existence of financial boundaries. According to the 2010 WHO report (17) , no country has yet been able to guarantee everyone immediate access to all the services that might maintain or improve their health. They all face resource constraints of one type or another, although these are most critical in low-income countries. However, WHO also claims that every country could raise additional domestic funds for health or diversify their funding sources if they wished to. Some of these options include governments giving higher priority to health in their budget allocations, collecting taxes or insurance contributions more efficiently and raising additional funds through various types of innovative financing. Adding taxes on harmful products such as tobacco and alcohol are a good example, since this would reduce consumption, improve health and increase the resources governments can spend on health.

As consequence, different alternatives have been carried through because of the need of quality healthcare. Home Care, (also referred to as domiciliary care, social care, or in-home care) is supportive care provided in home. This may be offered by licensed healthcare professionals who provide medical care needs or by professional caregivers who help to maintain daily activities. Home care is not only about supervision and assistance, but also a philosophy of independence and dignity for the patient. Nowadays, work of home care providers can be supported or even partially replaced by modern technologies. Portable biomedical devices offer new alternatives for patients to receive assistance without leaving home or need assistance. Thus, they play an important role in this field, due to the many advantages they offer.

Portable biomedical devices provides us with a new concept of monitoring that goes beyond occasional doctor visits or periodical medical check ups. More accurate and timely diagnosis may be obtained through the continuous monitoring of patients. In some cases patients do not need to stay in hospitals in order to be under medical control, so this way they can stay at home living normal life and if some problem is detected, an ambulance can transport them to the hospital as soon as possible. It is important to realize that this is not only more comfortable for the patient, but also an enormous save of money for hospitals.

Portable biomedical devices also remove geography and time barriers, thus allowing care providers to diagnose and treat illnesses remotely. This takes a special importance in developing countries, where a lack of care providers and medical tools exists (according to the WHO statistics previously described). However, also in some highly developed countries exists rural areas, where this advantage of portable biomedical devices offers many new possibilities.

1.2 Objectives

The objective of this thesis is the design and development of a framework which allows for the rapid and easy development of health applications in Android. One of the key aims of *eHealthDroid* is to abstract the user from low level programming details and to support any kind of biomedical monitoring technology. Functionalities implemented through the *eHealthDroid* should include receiving data from biomedical device, storing data in a local database or/and a remote server, data visualization in real time, knowledge inference and guidelines, since these are normally required in general health applications.

The specific objectives of the framework are:

- Rapid development of medical, health and wellbeing applications.
- Communication between portable biomedical devices and portable mobile devices. For this version of the framework, Shimmer devices (18) and mobiles phones with its own sensors are available to be used as biomedical devices, but any device could be easily incorporated.
- Define mechanisms to work efficiently with the data sent by the portable biomedical devices.
- Define storage procedures to locally or remotely store health and physiological data.
- Visualization tools to show the received data from the portable biomedical device.
- Medical knowledge inference mainly procured through machine learning and statistical models that can be used for triggering alerts, recommend healthier lifestyles or activity recognition.

- Other features as guidelines (YouTube, Audio, and Video), mobile device setup (Brightness, Wi-Fi, and Bluetooth), services (calls, messages, and schedule notifications), user login, etc.

1.3 Contributions

The framework has been developed in order to facilitate the development of biomedical applications. Moreover, to show its possibilities, a biomedical application has been designed and implemented. This exemplar app makes use of some functionalities of the framework, such as inference knowledge to perform activity recognition, visualization of data received from portable biomedical devices, data uploading to a remote storage, manage notifications or YouTube guidelines.

1.4 Thesis and Structure

This thesis is divided into the following chapters:

- *Chapter 1. Introduction.* It is the current chapter which is composed by four sections. The first section titled *Motivation and context* describes the importance and need that urges in modern world to develop biomedical frameworks, that is the subject of this thesis. The second section, *Objectives*, contains an overall summary of the proposal objectives at the beginning of the thesis. The third section, *Contributions* and describes the contributions made in this work. Finally, the last section is *Thesis and Structure* and presents the structure and composition of the thesis.

- *Chapter 2. State of the Art.* This chapter presents a review of four fields related to biomedical engineering: Biomedical Applications, Portable Biomedical Devices, Biomedical Software and Biomedical Methods.

- *Chapter 3. eHealthDroid.* This chapter, composed by three sections, describes the developed framework. The first sections shows an overview of the framework. While, the second section comprise an explanation of Android OS and the reasons why it was chosen. The last section deeply describes the modules and components of the developed framework.

- *Chapter 4. APP*. This chapter describes the application developed to demonstrate the framework capabilities and its use.
- *Chapter 5. Conclusion and future work.* This chapter contains the conclusions reached in this thesis and proposal of future work.

2

State of the Art

The portable medical devices industry is a fast growing market. With the advent of mobile technology this medical field has become one of the most interesting with the promise of great potential for future healthcare. With the appearance of modern smart phones, several biomedical applications focused on users' health have been developed. They vary from presenting different measures like blood pressure and heart rate to much more specific helping patients suffering from degenerative diseases so common in our aging population. Some applications use distinct devices that are demanded to collect data. Biomedical methods are used to inference medical knowledge from the collected data, for example to inform patients about their state or alert them immediately in case if some anomaly is detected. Biomedical software remains an important field, not only to create medical software to be used in computer but also to provide tools to develop biomedical applications easily, such as frameworks or libraries.

2.1 Biomedical Applications

One of the main objectives of biomedical applications is called Patient Monitoring (19), which means collecting and evaluating (remotely in some cases) physical and physiological variables from mobile or fixed devices around the patient. In case that health problem is detected, the user can be warned. Other objective is known as Patient Education (20), applications which allow the user to monitor his own health and educate him how to keep good health habits. Another important objective is achieved by applications that provide medical quality services, which are especially

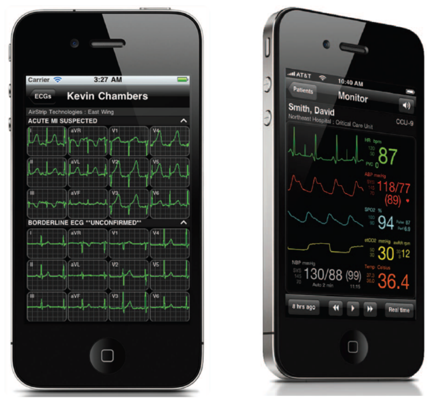


Figure 2.1: AirStrip Cardiology and AirStrip Patient Monitoring Apps. Picture from (3)

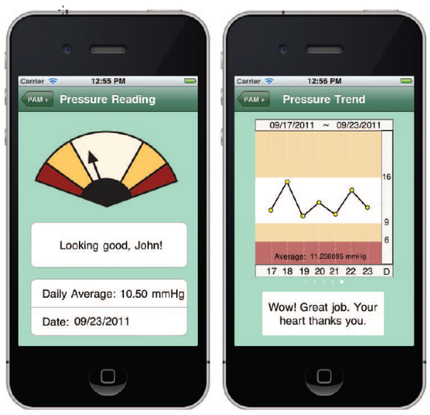


Figure 2.2: Pam+ app. Picture from (3)

important in places where those are not easily available (21). A few examples of each group are presented.

Airstrip Cardiology (22) and AirStrip Patient Monitoring (23) are two applications worth mentioning (Figure 2.1). These apps are able to digitalize ECGs and visualize in real time this information in a portable device, enabling health providers to monitor their patients in a remote way.

PAM+ app (Figure 2.2) works in conjunction with an external device which obtain blood pressure information and is designed to encourage patients to become active health-care participants. Patients adjust their medications depending on the obtained information to maintain a correct blood pressure (3).

Mobisante Company developed a complete phone-and-ultrasound package (Figure 2.4) that offers low cost, facility in use and portability. This app could make a real difference in rural settings and developing countries where medical equipment, including imaging machines, is usually beyond reach (24).

Another interesting biomedical application is able to do online detection of freezing of gait for people with advanced Parkinsons disease (25). This application uses mobiles accelerometers to recognize a freezing of gait and stimulates the patient via rhythmic auditory cuing or vibrotactile feedback to resume walking.

2.2 Portable Biomedical Devices

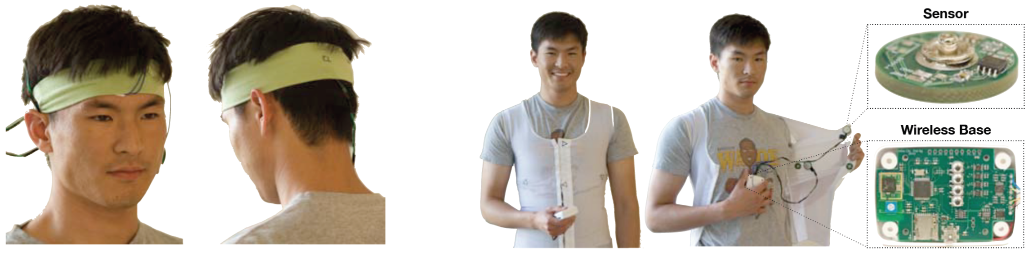
The majority of applications presented in the previous section requires an external device to work with. In the last years, due to the hardware improvements along with the smart phone boom, the portable medical devices industry is constantly growing (26).

An exciting and crucial concept to understand portable biomedical devices is the Smartclothing (27) (28). It consists of embedding biomedical devices into normal clothes. These devices are able to perform different functionality like body kinematics or physiological signals acquisition while they are being worn in a practical way. The tendency will be to use them in a close future without even realizing that are being worn. Different measures that portable biomedical devices are capable of monitor are described below:

- ECG: Zhou et al (29) present a portable wireless ECG monitoring system composed of fabric electrodes and a monitoring terminal, which essential to continuous ECG monitoring. The ECG signal acquired could be processed and stored in the monitoring terminal. This system is able to obtain the heart rate signal and transmit it to any device through Bluetooth protocol.

- EMG: Walters et al (30) describe a low-cost (around 100 $) portable EMG device specially developed to assess muscle activity during free-living situations, which has been used to investigate work-related pain, advancing age, and clinical disorder. It has been made using specialized concentric bipolar electrodes.

- EEG: there exists a powerful wearable device described by Chi et al (4) that unlike most of the smart clothes devices, is not in direct contact with the human



(a) EEG Headband (b) ECG Vest

Figure 2.3: Wearable devices using non-contact electrodes.Picture from (4)

body through contact sensors (wet or dry) cause it uses non-contact electrodes (Figure 2.3). It is able to capture cerebral (EEG) and cardiac activity (ECG) .

- Oxygen in blood: Cleven et al (31) introduce an implantable wireless sensor system for monitoring hypertension patients. This sensor system is intended for long-term monitoring of hypertension patients, designed for implantation into the femoral artery with computed tomography angiography. It consists of a pressure sensor and a telemetric unit, which is wirelessly connected to an extracorporeal readout station for energy supply and data recording. The system delivered stable measurement in initial animal trials like sheep.

- Kinematic information: many devices are able to obtain kinematic information, such as SHIMMER (18), the device used for this work. SHIMMER (Sensing Health with Intelligence, Modularity, Mobility and Experimental Reusability) is a platform for measurement of biomechanical and physiological variables related to functional movement. Some of its features are integrated peripherals, open software, modular expansion, specific power management hardware and a library of applications supported with platform validation.

- Ultrasound: Mertz (3) and Wojtczak et al (24) describe a device used for the Mobisante Company App commented in the previous section. This portable device is able to acquire not only fetal monitoring but also other clinically useful images (3) .

Other kind of sensors used to empower the patient health is the exosensor, typically associated to smarthomes (32). These devices are not attached to the body, but local-

Figure 2.4: Mobisante ultrasound package. Picture from (5)

ized in human environment, such as cars, houses or beds. A disadvantage to consider is that they can only be applied to one person in the place. Two interesting examples are the following:

- Breathing: mobile interrupter module, described by Jablonski et al (33), is dedicated to the enhanced interrupter measurement of respiratory mechanism in a home environment and capable of cooperation with a telemedical system. It is characterized by noninvasiveness and minimal requirement regarding patient cooperation and is specially suitable for newborns, preschool children and patients suffering from respiratory muscle impairment.
- Sleeping: Air Cushion or EarlySense Mattress (34) (Figure 2.5), although not exactly portable devices but worth mentioning, are the ones positioned on the top of the bed and under the mattress respectively. They record various measures like respiration rate, snoring, body movement, coughing, etc. .

2.3 Biomedical Software

The biomedical software field is not only composed by computer programs but also frameworks (35).

Frameworks development have a great potential for the biomedical general field. Frameworks facilitate creating biomedical applications (for computers or portable devices), even for people who are not really familiar with the field.

Although applications for desktop or traditional computers could seem a bit pushed into the background due to the appearance of portable devices, they still have a big

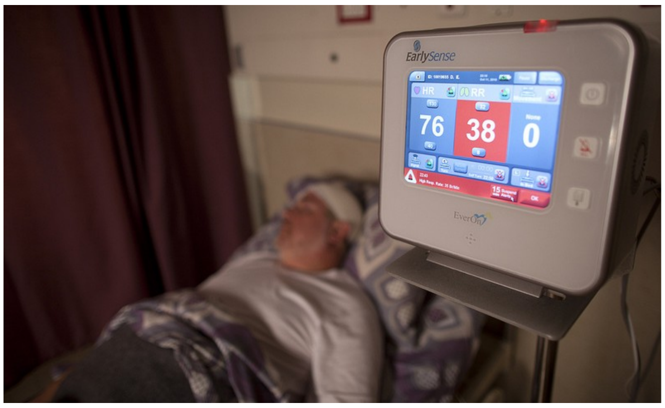


Figure 2.5: EarlySense Mattress device. Picture from (6)

importance. Computers capacity is greater, which makes them faster to evaluate big amounts of data. Even with the exceptional increase in the last years of portables devices capacity, approaching to the one provided for computers, the emergence of the Cloud Computing is a key role to play for computers. Cloud Computing is a compelling paradigm for managing and delivering services over the internet and is rapidly changing the landscape of information technology (36) .

There exists a wide variety of biomedical software. TheraGnosos is an interactive blended learning, simulation and training system, useful for students of biomedical engineering university courses (37). BioSig is an open source framework for biomedical signal processing (38). DtiStudio is a software to perform fiber tracking (39), while ScanImage manipulates laser scanning microscope (40). Finally, there is a framework called HaploView that provide tools to work with human genome structures (41).

2.4 Biomedical Methods

Biomedical devices consist of sensors capturing signals, which are transformed by a module A/D and are sent to a receiver. This signals are not only used for monitoring purpose, but also for biomedical knowledge inference that allows to classify signals.

Figure 3.5 represents the different phases to perform biomedical knowledge inference and how data flows through them. Once the data has been acquired, often a preprocessing phase is needed to remove existing noise in the signal, which makes it clean and ready to be processed. To achieve this, a denoising technique or filtering is applied over

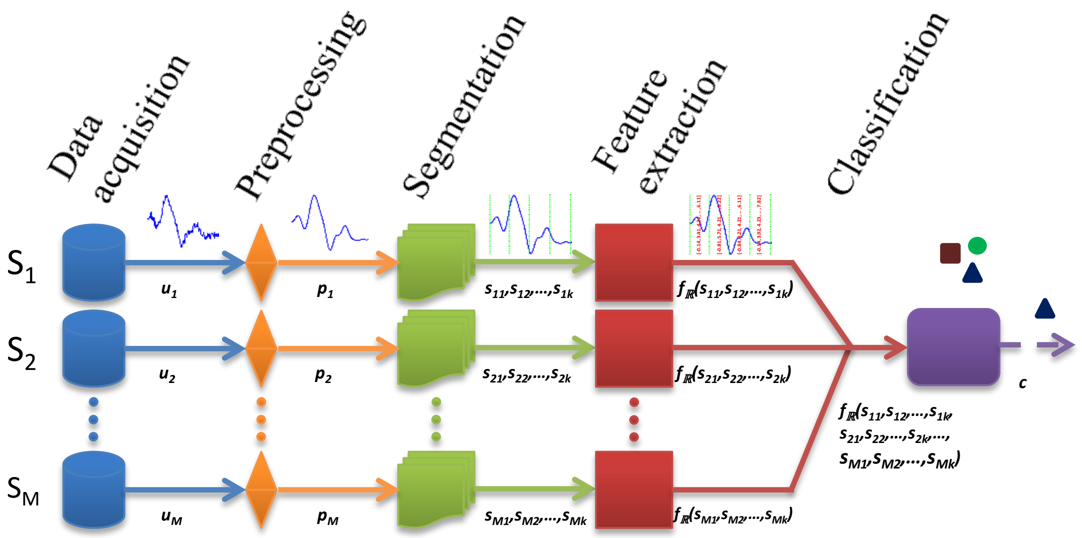

Data acquisition
Preprocessing
Segmentation
Feature extraction
Classification
S_1
S_2
S_M
u_1
u_2
u_M
p_1
p_2
p_M
$s_{11},s_{12},\dots,s_{1k}$
$s_{21},s_{22},\dots,s_{2k}$
$s_{M1},s_{M2},\dots,s_{Mk}$
$f_{\mathbb{R}}(s_{11},s_{12},\dots,s_{1k})$
$f_{\mathbb{R}}(s_{21},s_{22},\dots,s_{2k})$
$f_{\mathbb{R}}(s_{M1},s_{M2},\dots,s_{Mk})$
$f_{\mathbb{R}}(s_{11},s_{12},\dots,s_{1k}, s_{21},s_{22},\dots,s_{2k},\dots, s_{M1},s_{M2},\dots,s_{Mk})$
c

Figure 2.6: Classical data flow in biomedical data analysis. Picture from (7)

the signal, such as the wavelet transform (42). A segmentation phase is usually also applied to divide the signal into smaller time segments by applying windowing techniques. Then, these segments of data are characterized through different kind features. These range from general statistical and mathematical features such as mean, variance, median, standard deviation or domain specific features. Examples of the latter are, the RR interval has an enormous utility to inference the nervous cardiac level using ECG (43). Some famous feature extraction methods are MILCA or PCA (44). The last phase in the inference knowledge process is classification, where the previously obtained features are classified using models based on different techniques. Some of the most used classification techniques are Support Vector Machine (45), Fuzzy logic (46), Neural Networks (47), Decision Trees (48) or more recently Extreme Learning Machine (49).

3

eHealthDroid

3.1 Overview

eHealhDroid is a framework designed to develop applications in an easy and rapid way, under the Android operating system and to be used along with portable biomedical devices. This framework has been developed independently from the device type or communication protocol. Owing to that, it is necessary to create an intermediate driver for every portable biomedical device to be used which will work between the device and the framework. The drivers are included in the framework and remain unnoticed to the user. Thanks to this approach, an application can run simultaneously with different kinds of devices, providing a great flexibility.

The framework offers a wide range of functionalities. The most important ones are the following:

– Connection management and data streaming from the portable biomedical device (or portable mobile itself).

– Storage of the received data and other kind of data into a database.

– Functions to consult and to retrieve data from the database.

– Data uploading to a remote server.

– Online and offline data visualization.

– Data processing and knowledge inference: signal pre-processing, signal segmentation, feature extraction and data classification. It could be done either online or offline.

– Reproduce diverse media guidelines (embedded videos, audios, or online resources such as Youtube playlists).

– Make calls and send text messages particularly intended for emergency events.

– Send and scheduled notifications.

– Manage Wi-Fi, Bluetooth and other mobile device functionalities.

This framework is used through the managers that composed it, all of them singleton classes:

1. *Communication Manager*
   This is one of the most important managers. It is responsible for, among other things, the communication management with the biomedical devices, the data streaming and the broadcast of the received data to the rest of the framework. It is composed by three modules:

   (a) Data receiver: includes the drivers needed to use a specific portable biomedical device or the mobile itself with the framework.
   (b) Data structure: defines the proprietary data structure used by the framework.
   (c) Storage: contains classes for the database management.

2. *Server Manager*
   It is responsible for the data uploading to a remote server. The manager access to the local database, retrieving the data required by the user and uploading it to a server. The manager works for all kind of Internet connection (Wifi, 3G, etc).

3. *Visualization Manager*
   It is in charge of the data visualization. It is possible to perform an online visualization of the data streaming, as well as an offline visualization of the data

retrieved from the storage unit. It also allows to build graphics with different purposes such as to show the usage of a portable biomedical device in the last month.

4. *Data Processing Manager*

   This is probably the most powerful manager of the framework. It is responsible for processing the received data and applying medical knowledge inference. It is possible to use it either online or offline. It is composed by five modules:

   (a) Acquisition: defines methods to acquire data in a suitable way to be processed later. This is only necessary for the offline inference mode. For the online mode, the data is obtained directly from the Communication Manager.

   (b) Pre-processing: defines methods to carry out what is called the signal preprocessing. That is, to perform a downsampling or an upsampling.

   (c) Segmentation: defines methods to achieve the segmentation of a signal. In other words, to establish windows. This module should be used only in the offline mode.

   (d) Features extraction: defines methods to perform features extraction and features arrays for classification.

   (e) Classification: defines method to build, train, test or load classifiers, among other classification functionalities.

   All modules can be used independently, but using them one after the other, they form the sequence shown in Figure 3.5

5. *System Manager*

   This is a miscellaneous manager that offers methods to do different kind of tasks through the Android operating system. It is composed by three modules:

   (a) Services: defines methods to execute a call, send a text message or scheduled a notification.

   (b) Setup: defines methods to manage the Wi-Fi connection, Bluetooth and the screen brightness.

(c) Guidelines: module to play videos, audios and Youtube videos or playlists. It is composed by three sub modules:

i. Audio

ii. Video

iii. Youtube

In figure 3.1 the "big picture" the framework is presented, with all the existing managers and how they interact with each other.

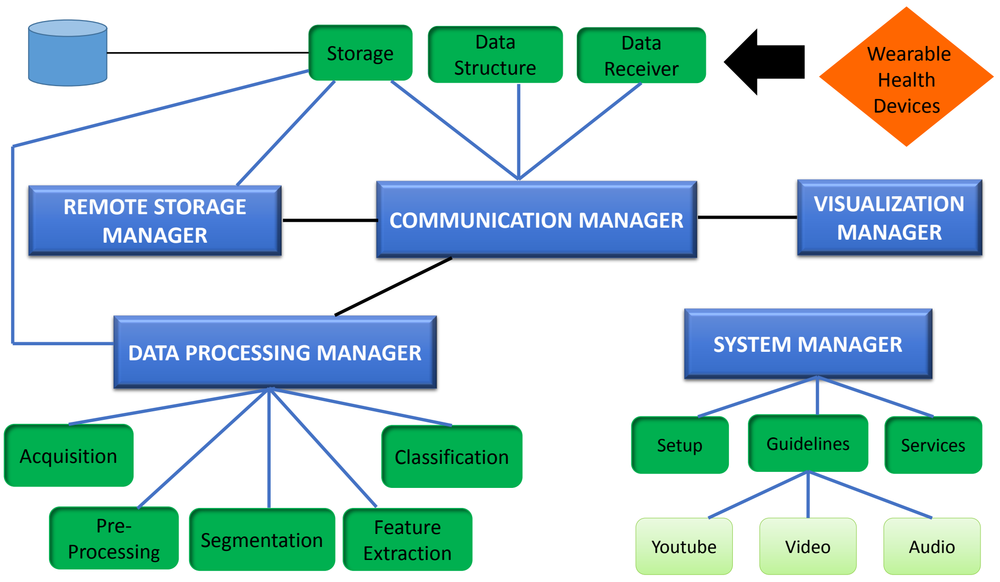


Figure 3.1: Framework's diagram

3.2 Android

3.2.1 Operating System

Android is an operating system based on the Linux Kernel, specially designed for touchscreen mobile devices such as smart-phones and tablets computers. It was initially developed by Android Inc., which was backed financially by Google and later bought in 2005. Google announced Android in 2007 along with the founding of the Open Handset Alliance (50). The first Android-powered phone was sold in October 2008.

Figure 3.2: Android logotype. Picture from Google Inc.

The user interface of Android consist in using touch input corresponding to real-world actions such as swiping, tapping or pinching to manipulate screen objects. Android allows users to customize their home screens with shortcuts to their applications or widget, which offers users to display live content. Applications can send notifications to the users to inform them of something relevant, such new messages, alarms or appointments. Most Android-powered devices own built-in sensors which are used by applications to provide different functionality to the user, such as measuring motion, orientation or environmental conditions. These sensors are capable of providing raw data with high precision and accuracy.

One of the strongest features of Android is the fact that it is an open source and its code is released under the Apache License (51). This allows the software to be freely modified and distributed by device manufactures or enthusiast developers. Android has a large community of developers writing applications (apps), that provides Android devices with almost boundless possibilities. In November 2013, there were almost 900.000 apps available for Android (878632 exactly, with a 22 % of low quality apps) (52), and the estimated number of apps downloaded from Google Play (Androids app store) was 25 billion (53). A survey conducted in April-May 2013 revealed that Android is the most popular platform for developers, used by 71% of the mobile developer population (54).

Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It is the largest installed base of any mobile platform and growing fastevery day another million users power up their Android devices for the first time and start looking for apps, games, and other digital content (55)

The Android architecture is formed by the following layers:

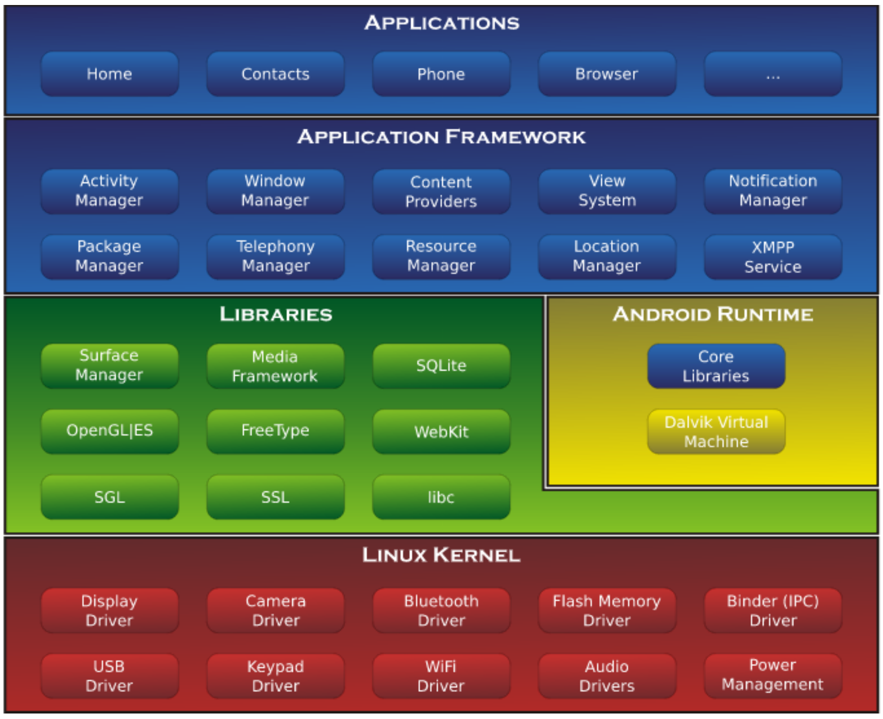


Figure 3.3: Android Architecture Diagram

1. **Applications**. Android applications are not exactly a part of the Android operating system and lies on the top of the three layers that form the Android Software Architecture. Android devices are delivered with a set of applications capable to provide basic services, such as web browsing, sending SMS or making phone calls. This is accomplished and managed by the underlying layers like the Applications framework one, just under this application layer. Android applications are written in Java and executed in Dalvik (DVM), a specific Java runtime engine.

2. **Application Framework**. The upper layer of the diagram. Provides framework services and libraries to interact with the application layer. These are written mostly in Java. Some examples are the Activity Manager, Location Manager, Notification Manager or View System.

3. **Libraries**. Special libraries written in C or C++. These libraries offer services to applications like SQLite for Databases, OpenGL or WebKit for accessing website.

4. **Android Runtime**. Consisting of two components, the DVM and core libraries responsible for process management. The DVM is a Java runtime engine which processes are isolated from the environment. It executes program written in Java by using a DX tool to convert it into bytecode that is readable by the DVM.

5. **Linux Kernel**. Is the lowest layer of Android Architecture and is the abstraction layer between software stack and hardware. It includes drivers for hardware, networking, security, energy consumption file, system access, processes and memory management.

3.2.2 Advantages/Drawbacks

Some advantages and drawbacks of Android Operating System are presented to show what Android may provide to an user and Android weaknesses:

3.2.2.1 Advantages

- Providing the code to everyone by making it open and free source makes it easy for developers to implement their own libraries.
- Populist Operating System. Android Phones, opposite to iOS which are limited to the IPhone from Apple, have many and potential manufacturers such as Sony or Samsung. That is the reason why there exists a wide variety in phones (style and capacities), suiting user budget.
- Easy access to the Android App Market (Google Play), where Android users can download thousands of free apps.
- Multitasking, which means that many applications may be running at the same time.
- Frequent OS updates, for improving performance.
- Installing ROM modification. There are many custom ROM installable on Android phones, guarantying they will not harm the Android device.

- USB full facilities, which means user may replace battery, mass storage, disk driver or USB tethering.
- Support all Google Services.
- And many other like easy in terms of notifications, widgets, router functionality...

3.2.2.2 Drawbacks

- In order to provide full functionalities it needs an active Internet connection.
- Some phones become slow after installing many apps. However, new Android versions and mobiles devices which offer better capacities deal better with this problem.
- Many applications contain malware, even if they are available in the Google Play Market users need to be careful. Besides, advertising is usually presents in applications.
- Wasteful battery. Android is said to be more wasteful than any other mobile operating system, due to constant processing in the background that lead to quickly drains batteries.
- Sometimes applications, external devices or just protocols as Blue-tooth presents strange behaviors depending on the mobile device where are being used, due to the wide variety of mobile devices existing (manufacturers are free to insert modifications in Android).

3.2.3 Why Android?

It is aim of this work to provide an open tool for developing biomedical applications at the reach of as many people as possible. Having this in mind, Android seems to be the mobile operating system which is the most appropriate, owing to the amount of users and mobiles devices Android powered. The fact that is free and open source converts this into a software platform for mobile development rather than just a mobile operating system.

3.3 Managers

3.3.1 Communication Manager

This manager, like the rest of the managers, is a singleton class and is responsible for the communication between the devices and the Android application. Its main function is to manage the data received by the devices, and deliver them to the rest of the framework's package/managers that need them. It is also in charge of the storage in the local database. The manager is composed by 3 modules: Data Structure, Data Receiver, and Storage.

3.3.1.1 Description

In order to do its work properly, the manager should run in background. Therefore it is defined as a *Service* (56). The data acquisition is handled by the drivers. They also create the framework's data structure along with them, so it is necessary to design some kind of mechanism for sending the data structure to the manager. After a deep study of all the possibilities, it was decided that the drivers would send the data structure by *Message* (57) through a *Handler* (58) (both tools are provided by the Android API). It was also decided that the manager would not do all the work. Independent threads would be created instead. It was done like this to take advantage of the benefits of the approach "divide and conquer". Having said that, it is clear that a thread will be in charge of just its own device. Thus, there will be as many threads as devices connected. The threads are defined within the manager, namely *CommunicationThread*. They only have the *Handler*, as mentioned before, and a *Looper* (59), which is a class used to run a message loop for a thread. Threads receive the message with the framework's data structure and work accordingly: store data, send data to the rest of the framework, or check whether the communication has failed. But there were other details that needed to be controlled, such as linking the thread with its device. To do this, the *ObjectCommunication* object was created. This object is an inner class within the manager and is formed by some flags, an array with the graph's name (this will be explained in 3.3.3), a *CommunicationThread* and a *Device* (the driver previously mentioned).

The manager has, among other things, a Hashtable of String and *ObjectCommunication*. This Hashtable stores the devices. The device name is the key (so it has to be unique) and the *ObjectCommunication*, which make all the work described previously, is the object. To add a device to this hashtable the manager provides a pattern function: AddDeviceSomething. In particular, the manager provides *AddDeviceShimmer* and *AddDeviceMobile*.

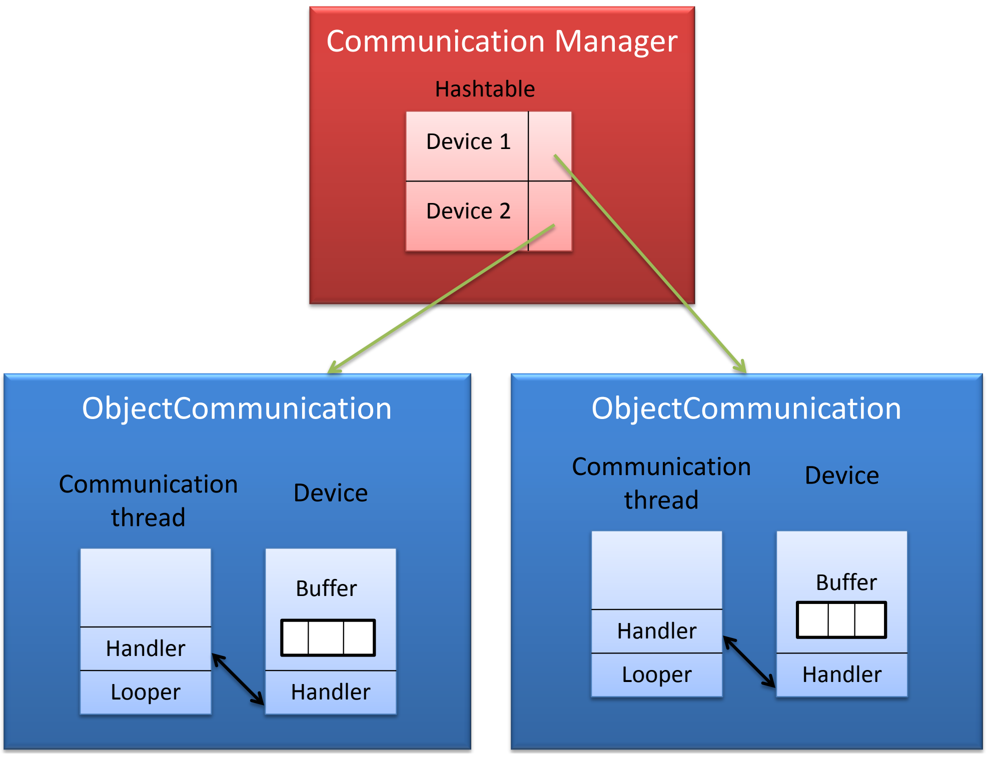


Figure 3.4: Graph showing how the Communication Manager works

When the appropriate flags are activated, this manager also sends the data to the *Storage* package for storing them into the local database (which will be explained later), to the *Visualization Manager* for the data representation or to the *Data Processing Manager* for processing and knowledge inference. Since the data can be stored into a database, the API demanded a buffer to store them temporally. To do this, different kind of buffers were considered. After evaluating their advantages and disadvantages, the circular buffer was considered for implementation.

The two most important functions in the manager are *startStreaming* and *stop-Streaming*.

startStreming checks whether the device is connected or not. If so, the database tables are created, and the *CommunicationThread* is started. Then, and just in the first session, a database consistency check is performed in case the application stopped owing to any exception. Finally, the *startStreaming* function defining in the device driver is called (a device driver is needed in order to use the framework, but this will be explained later). The Communication thread runs in background receiving the messages sent by the device driver. When the first sample is received, the session metadata are stored in the database. In case the proper flags are activated, the thread also stores the samples into the database and sends them either to the *Visualization Manager* or to the *Data Processing Manager*. If the connection is lost, the thread stores the rest of the session metadata, and in case the *Store* flag is activated the last block of samples is stored.

stopStreaming calls the device driver *stopStreaming* function. Then, it calls to the *storeLastInstance* function, which kills the *CommunicationThread* and, in case the *Store* flag is activated, stores into the database the last block of samples . Finally, the session metadata is updated.

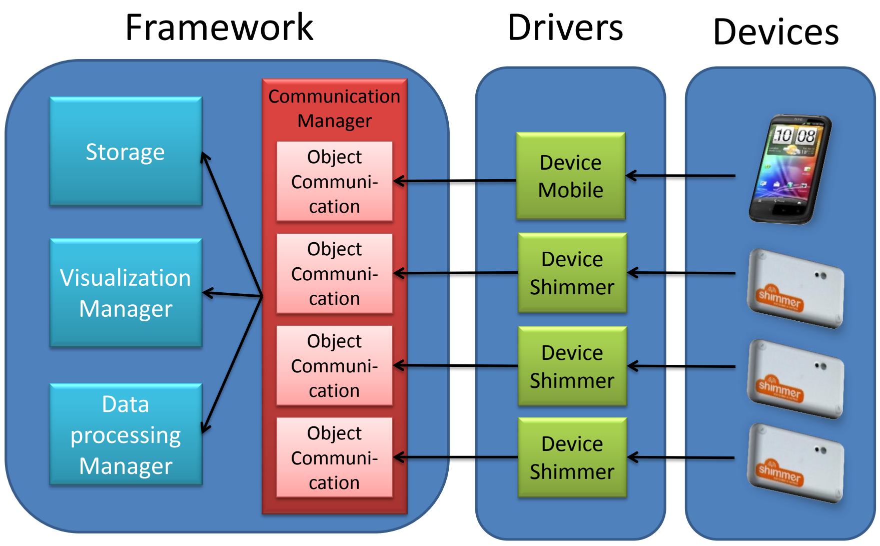


Figure 3.5: Data flow diagram from the devices to the framework.

The manager also provides other functionalities such as to set whether to store or not the data, to set the sensors enabled of a specific device, to set a label (this will be explained later), to set and get the device sample rate and to get among others.

This manager contains three packages: Data receiver, Data structure and Storage.

Data receiver

The Data receiver package contains the "drivers" for the devices that will be connected with the Android application. The aim of these drivers is to manage the connection with the device and fill the framework data structure with the sample received. Other kinds of functionalities offered by the device are also managed.
The package is composed by 1 interface and 2 classes:

Device (interface)

This interface is the pattern for creating the drivers, so obviously they must inherit from Device. Some of the most important functions in the interface are *connect/disconnect* which define the connection between the device and the Android application. *Start/stop streaming* are also very important since they define the beginning and ending of the data streaming. There are also other functions defining this pattern. It is possible to change the sample rate with the function *setRate* or consult it with *getRate*. In the same way, it is possible to change and consult the enabled sensors with *writeEnabledSensors* and *getEnabledSensors*. Since each device has its own buffer, there are other functions to set and get the number of samples to be stored. Something similar happens with the metadata, so that there are functions that get and set the session metadata of the device. Sensor device drivers are required for their use within the framework.

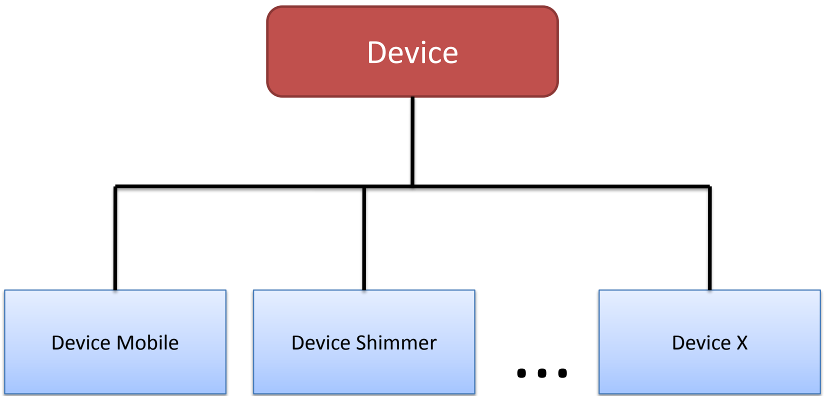


Figure 3.6: Patterns for the drivers.

DeviceShimmer

This is the driver developed for the Shimmer device. Its aim is to manage the connection between the Shimmer device and the Android OS, to receive the data sent by the device and to create the framework data structures.

The company which provides the Shimmer devices also provides a driver. As our manager, it is defined as a *Service* so that it can work in background. It is responsible for managing the connection with the Shimmer device. To do this, first create and start the *ConnectThread*. Once the connection has succeeded, the *ConnectedThread* is created and started. This thread manages all the incoming and outgoing transmissions. It receives the raw samples from the device. Then, the Shimmer driver converts them to calibrated and uncalibrated values. A hashtable is created with these values and sent in a message. It also comprises a wide range of functions for controlling all the device's features. From now on, this driver will be referenced as either *Shimmer driver* or *intermediate driver*. Thus, the driver defined for us, and explained in this section, will be referenced just as *driver*.

The driver is composed, among other things, by a *BluetoothAdapater* object, a *BluetoothDevice* object, a circular buffer, an instance of the Shimmer driver, and a handler. The BluetoohAdapter and BluetoothDevice objects (both provides by the Android API) are responsible for the management of the Bluetooth connection. The circular buffer is defined as an ArrayList. Its size is four times the number of samples to store. Thus, the buffer stores four blocks of samples. The number of samples to stored can be set in order to make the buffer more suitable for each situation. The handler is in charge of to receive and handler the messages sent by the Shimmer driver. Depending on the message received, the handler will make a different task. When the handler receives a message with a sample, first it is checked to see if it is the first one. If so, an offset timestamp and the beginning of the session are set to this moment. Then, the framework data structure (which is stored in the buffer), is filled with the sample values. Finally, a data structure reference is sent to the handler of the Communication Manager.

The most important functions in this class are *connect/disconnect* and *start/stop streaming*. *connect* and *disconnect* are not complex functions since the hard work is done by the Android OS and the intermediate driver.

StartStreaming is a function that requires more work since it is necessary to initialize the buffer. To do this, first the enabled sensors are consulted, then the appropriate data structure is created and the buffer fills with that data structure. Finally, the intermediate driver's *startStreaming* function is called and session's metadata are set.

StopStreaming function calls to the intermediate device's *stopStreaming* function and updates sessions metadata.

The driver also has to implement the other functions of the interface *Device* mentioned previously. Other features, that are specific of the device, are controlled by other functions such as *getFormat*. This function sets whether the data received by the device must be calibrated or uncalibrated.

DeviceMobile

This is the driver class for mobile devices. Most Android devices have built-in sensors that measure motion, orientation, and environmental conditions, depending on the mobile device and the available sensors. Since the biomedical device used in this case is the mobile itself, there is no need to establish a Bluetooth connection or other similar connection. To acquire raw sensor data, the Android Sensor Framework is used. The components used are similar to the *DeviceShimmer* class, but in this case, there is only one Handler (the manager handler) and there is not Bluetooth variables. A *SensorManager* object is necessary. This is a class which provides functionality to manage sensors such as listening/accessing them, registering/unregistering sensor event listener, and sensor constants that are used to report sensor accuracy, set data acquisition rates, or calibrate sensors. Also, an array list of Sensor is used in order to manage each sensor. In this class, *connect* and *disconnect* methods are empty since there is no connection to be done. *Start/stop streaming* methods behavior changes with respect to the *DeviceShimmer* class. After initializing the buffer (checking the enabled sensors and filling the buffer with the appropriate structure in a similar way than for the Device Shimmer), a listener for every sensor is registered. This is done using the function *registerListener* belonging to the *SensorManager* class, which needs a *Sensor* object (the sensor to register) and an acquisition data rate. Four *SensorManager* constants exists to set the acquisition data rate to be used with all sensors. These constants rates are the following:

- SENSOR_DELAY_FASTER. Rate: 100 Hz
- SENSOR_DELAY_GAME. Rate: 50 Hz
- SENSOR_DELAY_UI. Rate: 16.7 Hz
- SENSOR_DELAY_NORMAL. Rate: 5 Hz

These rates are approximated due that all of them depend on the mobile device where the app is used. Once the listeners are registered, the *methodonSensorChanged* which receives as parameter a *SensorEvent* object, is called every time a sensor reports a new value. In other words, it sends a new sample (inside the *SensorEvent* object). In order to not block this listener, like is recommended in the Android Sensor Framework, all the work to be done with the new sample is done in an external method called *gatherData*. Each *sensorEvent* object has a raw sensor data (the type of sensor which generates the event), the accuracy of the data and, the timestamp of the event. The *GatherData* method aims at receiving this *sensorEvent* and convert the data to the framework data structure. Once this is done, the method puts this into the class buffer and sends a message to the manager handler with the reference of the new *ObjectData*. The work done in this method is problematic because the Android Sensor Framework does not send raw data synchronously. It only does so when a sensor changes its value, resulting in *ObjectData* containing only information about one sensor. Thus, the framework data structure would lose sense hence the hashtable would be almost empty. The solution chosen to solve this problem is shown in 3.3.1.3.

The last thing done in the *startStreaming* method concerns metadata sessions. The starting session time and device rate attributes belonging to *ObjectMetadata* reset with the current time, and the device current acquisition data rate. Then, the method *set-Metadata* is called. This method is aimed to fill the *ObjectMetadata* hashtable with metadata (sensors units) of every registered sensor. The *stopStreaming* function simply unregisters every listener registered and updates *ObjectMetadata* ending session time attribute.

Data structure

Here is defined the framework's data structure. This topic also was deeply studied, taking into account that it had to be very flexible because of the wide range of devices that exitsexists and the variety of sensors. It also had to be scalable because in the future other devices with new features or sensors can be released.
The data structure is composed of four objects (classes) and one enumeration:

- Data: It is composed by a double. It represents the value return by the sensor.
- ObjectData: This class stores the device name and a Hashtable where the type of sensor is the key and Data is the object. It represents a complete sample sent by the device. It means that if the sample rate is 50Hz, 50 ObjectData will be generated in 1 second.
- Metadata: This class is composed by a String. It represents the units of measure used by the devices.

- ObjectMetadata: This class is composed by a Hashtable where SensorType is the key, and Metadata is the object. It also contains two Time variables, two Int variables, and one Double variable. The Hashtable is filled with the enabled sensors of the device and their units of measure. The two Time variables represent when the device starts to stream and when it finishes. The two Int variables represent the first index of the database table which belongs to this device during a session, and the last index of this session. The period of time that goes from when the device starts to stream until the streaming is stopped is called *session*. Finally, the Double variable represents the devices sample rate (measured in Hz) during a determinate session.

- SensorType: This is not a class, but an enumeration. It represents the type of sensor.

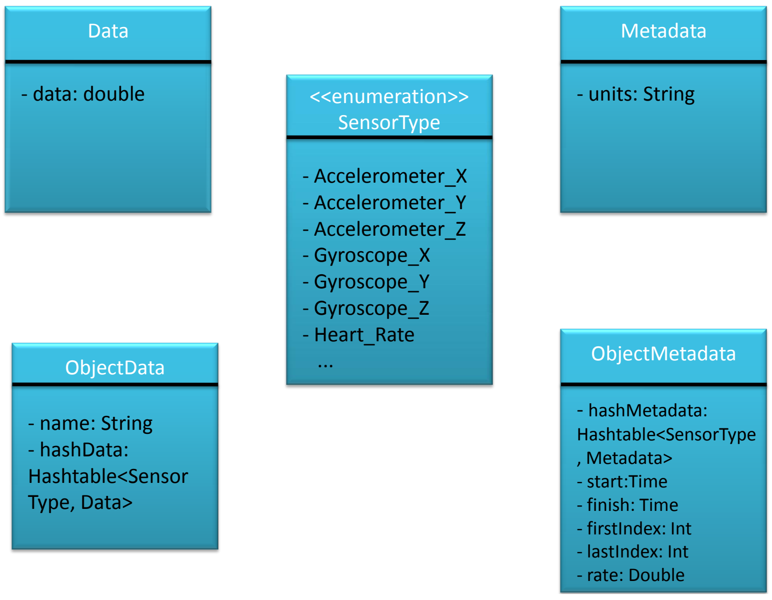


Figure 3.7: Class diagram of the framework data structure

Storage

This module offers all the functionalities to manage everything related with the database.

To generate the most appropriate database environment, various experiments were performed, which are explained in 3.3.1.3, and test which design was better. Taking into account the results, it was desgined a database composed by 4 tables: Table_MacAddress, Metadata_MacAddress,

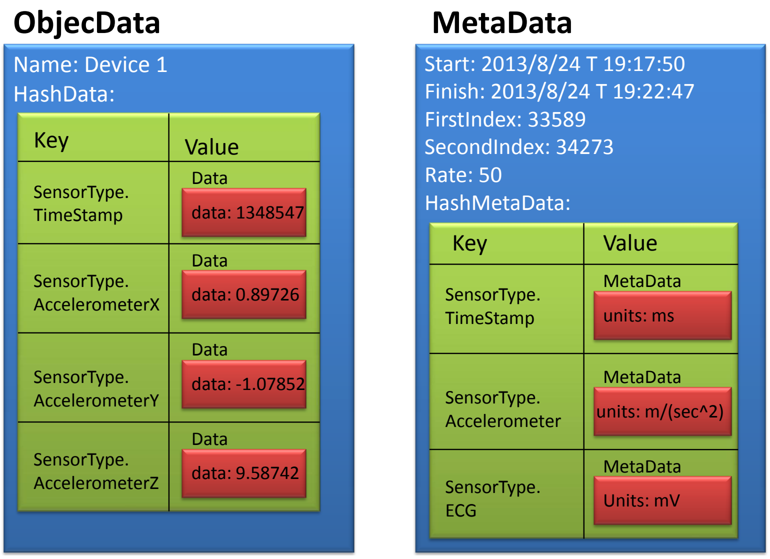


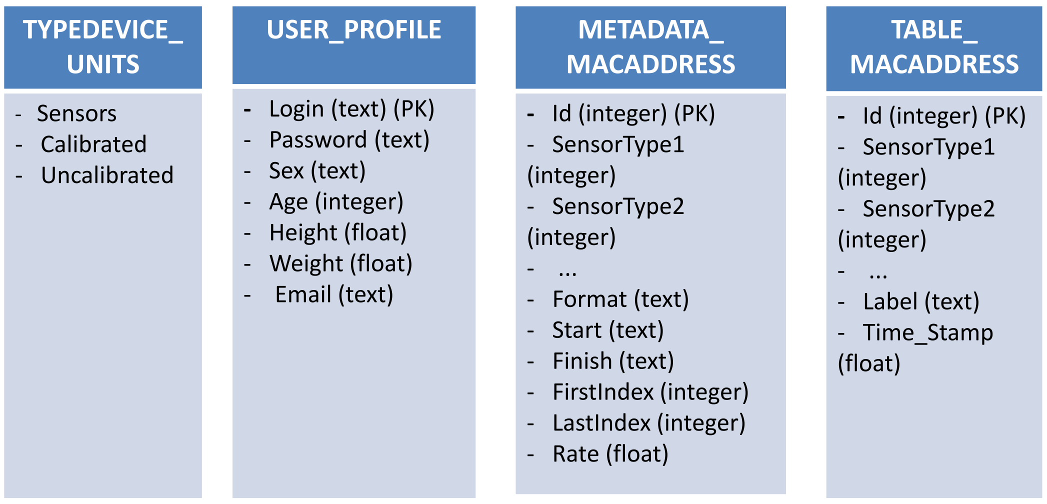
Figure 3.8: Example of an ObjecData and Metadata object.

TypeDevice_Units and User_Profile. When Table_MacAddress and Metadata_MacAddress tables are created, the part "MacAddrses" is replaced by the device's MAC address to make the table name unique, easy to identify and relate this with its corresponding device. When the Units tables are created, the part "TypeDevice" is replaced by the type of device, either Mobile or Shimmer so far.

- Table_MacAddress: In these tables are stored the samples sent by the devices. *Id* field is an auto increment integer and the primary key. *SensorTypeX* fields are all the sensors available in the device. Since different devices may have different sensors, the tables definitions can be different for different kinds of devices. *Label* field is the labeling given to that streaming data. For instance, if storing the activity of the person who is wearing the device is wanted, the *Label* could take values such as "walk", "run"or "stand". The *Time_stamp* field stores the time, by hour and date, when the sample is received. It is expressed in milliseconds starting from January 1, 1970 UTC.

- Metadata_MacAddress: These tables store the session metadata. *Id* field is an auto increment integer and the primary key. *SensorTypeX* fields are all the sensors available in the device. These fields will be either 1 if the sensor is enabled or 0 if not. *Format* field denotes whether data is calibrated or uncalibrated. *Start/Finish* fields indicate when the data streaming started/finished . *First/Last Index* fields represent the first/last index of the table's row where the samples of this session are stored. *Rate* field stores the sample rate of the session.

- TypeDevice_Units: These tables store the sensors units of measure for each kind of device. *Sensors* field indicates the type of sensor. *Calibrated/Uncalibrated* fields represent the calibrated/uncalibrated units of measure.
- User_Profile: This table stores a profile of each user. *Login* field is the login for the user. It is the primary key so it must be unique. *Password* field stores the password of the user. *Sex, Age, Height, Weight* and *Email* fields are, like their names indicate, the sex, age, height, weight and email of the user.



TYPEDEVICE_ UNITS
- Sensors
- Calibrated
- Uncalibrated

USER_PROFILE
- Login (text) (PK)
- Password (text)
- Sex (text)
- Age (integer)
- Height (float)
- Weight (float)
- Email (text)

METADATA_ MACADDRESS
- Id (integer) (PK)
- SensorType1 (integer)
- SensorType2 (integer)
- ...
- Format (text)
- Start (text)
- Finish (text)
- FirstIndex (integer)
- LastIndex (integer)
- Rate (float)

TABLE_ MACADDRESS
- Id (integer) (PK)
- SensorType1 (integer)
- SensorType2 (integer)
- ...
- Label (text)
- Time_Stamp (float)

Figure 3.9: Design of the database.

The module is composed by 3 classes:

DataBaseHelper

This is a class specifically created for helping carry out the management role such as create, open, close or update the database.

DBAdapter

All the queries needed are made in this class. All of the functions used to create the databases tables include the following: *createShimmerTable, createShimmerMetadataTable, createShimmerTableUnits, createMobileTable, createDeviceMetadataTable, createMobileTableUnits, createUserProfileTable* In order to insert all the data storage in these tables the following are used: *insertShimmerSignal, insertShimmer-*

Metadata, insertMobileSignal, insertMobileMetadata, fillMetadataRow, fillTableShimmerUnits, fillTableMobileUnits, insertUserProfile. In addition, there are a wide range of functions to retrieve any kind of data from database such as *retrieveInformationById, getMaxIndex, getColumnsName, getShimmerUnitsTable, getSessionsIds, getPassword,* etc.

To evaluate the consistency of the metadata tables the *consistencyTestMetadata* function was created. It consists in retrieving the last metadata row and checking whether the *LastIndex* field is not set to 0. If the *LastIndex* field is set to zero, it means the application was forced to close due to a bug. To fix this, *LastIndex* is set to the index of the last row of the data table.

Storage

This class could be defined as the upper layer of this module. Thus, it is an intermediate class between low level database definition and the manager. For some tasks, functions just get the input values and call to the DBAdapter functions (*getMobileUnitsTable, getShimmerSignalsTable, consistencyTestMetadata,* etc). However for other tasks, functions get the input values, adapt them, and then call to the DBAdapter functions (*insertShimmer, insertShimmerMetada, insertMobile, insertMobileMetadata*).

3.3.1.2 How to use it

As it was said before, this class is a singleton. It means that first, it is necessary to get its instance like this:

```
CommunicationManager cm = CommunicationManager.getInstance();
```

Then, if one wants to storage data into the database, the variable storage must be initialized this way:

```
cm.CreateStorage(getApplicationContext());
```

To add devices to the manager, one just needs to call the appropriate functions:

```
cm.AddDeviceShimmer(getApplicationContext(), "Device Shimmer", true);
cm.AddDeviceMobile(getApplicationContext(), "Mobile Device");
```

Since the mobile device uses the device's sensor, it is not necessary to connect it. But for the Shimmer devices, once they have been added, they need to be connected. This work is done by the function *Connect*, giving as parameter the mac address:

```
cm.connect("Device Shimmer", address);
```

Before the device starts to stream, the enabled sensors of the devices can be set:

```
ArrayList<SensorType> sensors = newArrayList<SensorType>();
sensors.add(SensorType.ACCELEROMETER);
sensors.add(SensorType.MAGNETOMETER);
sensors.add(SensorType.GYROSCOPE);
cm.writeEnabledSensors("Mobile Device", sensors);
cm.writeEnabledSensors("Device Shimmer", sensors);
```

The number of samples to be stored in the buffer can be chosen too:

```
cm.setNumberOfSampleToStorage(100);
```

After all these things are done, it is the suitable moment to start (and stop whenever is wanted) to stream:

```
cm.startStreaming("Mobile Device");
cm.startStreaming("Device Shimmer");

cm.stopStreaming("Mobile Device");
cm.stopStreaming("Device Shimmer");
```

There are a couple of features that can be set either before or during the data streaming such as to select what device is going to store its data (by default, it is set to *true*) or to set a label.

```
cm.setStoreData("Mobile Device", false);
cm.setStoreData("device Shimmer", true);
cm.setLabel("walking");
```

There is a way to get the device in case it is wanted to access to its specific functionalities:

```
DeviceShimmerdS = (DeviceShimmer) cm.getDevice("Device Shimmer");
DeviceMobiledM = (DeviceMobile) cm.getDevice("Mobile Device");
```

To insert a new users profile there are a couple of things to consider. The database must be open and the table created. It is also possible to check whether the login already exists in order to avoid a SQL exception. Then the database should be closed. All of this is possible by doing:

```
cm.openDatabase();
cm.createUsersTable();
if(!cm.existsLogin(login))
```

```
   cm.addUserProfile(login, password, age, sex, weight, height, email);
else
   Toast.makeText(getApplicationContext(), "this login already exist",
       Toast.LENGTH_SHORT).show();
cm.closeDatabase();
```

In case there are devices streaming, it is advisable to check whether any of them store data before closing the database. It should be done in order to avoid a SQL exception. To do that, it is only necessary to call the function of the Communication Manager named *isStoring()*:

```
if(!cm.isStoring())
   cm.closeDatabase();
```

What has been explained is the main use of this manager. For further information, the reader is referred to the manager implementation.

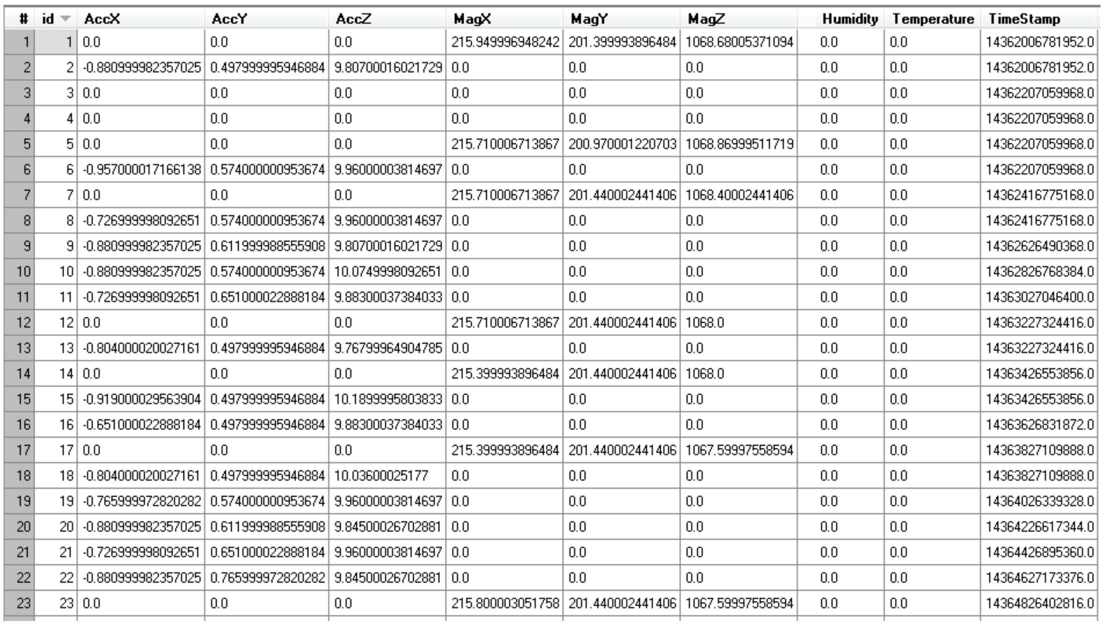
3.3.1.3 Difficulties and solutions applied

Buffer

In the first version of this manager, the buffer was defined as part of it. The Shimmer driver received objects with the data captured by the device. For every sample received, a new object with the framework data structure was created. After several tests were made, a problem was identified: too many objects were created since it did not send a reference to the object. A new object was created and its reference sent instead. The adopted solution was to define the buffer in the driver and fill it with empty *ObjectData*. Then, before the streaming begins, the objects are given the appropriate structure taking into account the enabled sensors. Thus, always an object reference is sent from the driver to the Communication Manager as well as from the Communication Manager to the rest of the framework.

Asynchronous raw data transmission from Android Framework Sensor (Mobile Device driver)

As was briefly commented in the mobile driver class chapter, the Android Sensor Framework does not send raw data synchronously. Only does it when a sensor changes its value. This emerged from a bad utilization of the framework data structure, since every *ObjectData* had information about only one sensor instead of information of multiple sensors. Also, this problem ended up storing a disproportionate number of rows for any streaming session. The whole problem can be visualized in the next image where there are different rows with the same timestamp value. So, the desired objective



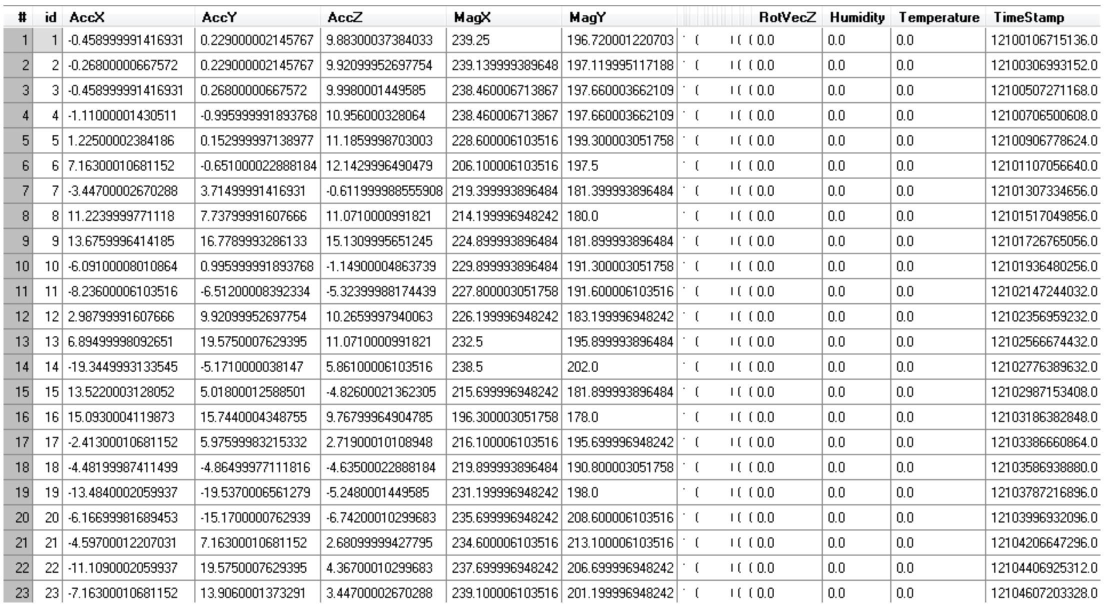
#	id	AccX	AccY	AccZ	MagX	MagY	MagZ	Humidity	Temperature	TimeStamp
1	1	0.0	0.0	0.0	215.949996948242	201.399993896484	1068.68005371094	0.0	0.0	14362006781952.0
2	2	-0.880999982357025	0.497999995946884	9.80700016021729	0.0	0.0	0.0	0.0	0.0	14362006781952.0
3	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	14362207059968.0
4	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	14362207059968.0
5	5	0.0	0.0	0.0	215.710006713867	200.970001220703	1068.86999511719	0.0	0.0	14362207059968.0
6	6	-0.957000017166138	0.574000000953674	9.96000003814697	0.0	0.0	0.0	0.0	0.0	14362207059968.0
7	7	0.0	0.0	0.0	215.710006713867	201.440002441406	1068.40002441406	0.0	0.0	14362416775168.0
8	8	-0.726999998092651	0.574000000953674	9.96000003814697	0.0	0.0	0.0	0.0	0.0	14362416775168.0
9	9	-0.880999982357025	0.611999988555908	9.80700016021729	0.0	0.0	0.0	0.0	0.0	14362626490368.0
10	10	-0.880999982357025	0.574000000953674	10.0749998092651	0.0	0.0	0.0	0.0	0.0	14362826768384.0
11	11	-0.726999998092651	0.651000022888184	9.88300037384033	0.0	0.0	0.0	0.0	0.0	14363027046400.0
12	12	0.0	0.0	0.0	215.710006713867	201.440002441406	1068.0	0.0	0.0	14363227324416.0
13	13	-0.804000020027161	0.497999995946884	9.76799964904785	0.0	0.0	0.0	0.0	0.0	14363227324416.0
14	14	0.0	0.0	0.0	215.399993896484	201.440002441406	1068.0	0.0	0.0	14363426553856.0
15	15	-0.919000029563904	0.497999995946884	10.1899995803833	0.0	0.0	0.0	0.0	0.0	14363426553856.0
16	16	-0.651000022888184	0.497999995946884	9.88300037384033	0.0	0.0	0.0	0.0	0.0	14363626831872.0
17	17	0.0	0.0	0.0	215.399993896484	201.440002441406	1067.59997558594	0.0	0.0	14363827109888.0
18	18	-0.804000020027161	0.497999995946884	10.03600025177	0.0	0.0	0.0	0.0	0.0	14363827109888.0
19	19	-0.765999972820282	0.574000000953674	9.96000003814697	0.0	0.0	0.0	0.0	0.0	14364026339328.0
20	20	-0.880999982357025	0.611999988555908	9.84500026702881	0.0	0.0	0.0	0.0	0.0	14364226617344.0
21	21	-0.726999998092651	0.651000022888184	9.96000003814697	0.0	0.0	0.0	0.0	0.0	14364426895360.0
22	22	-0.880999982357025	0.765999972820282	9.84500026702881	0.0	0.0	0.0	0.0	0.0	14364627173376.0
23	23	0.0	0.0	0.0	215.800003051758	201.440002441406	1067.59997558594	0.0	0.0	14364826402816.0

Figure 3.10: Different rows with same timestamp value.

was to send *ObjectData* to the manager with information about multiple sensors, as many as sensors sent information in the same timestamp value. In order to do this, the GatherData method is used to receive information from the Android Sensor Framework and process it. The method receives a sensorEvent, a class which contains fields for event accuracy, the Sensor which generates the event, a timestamp and a values array (which length and contents depends on which sensor type is being monitored).

GatherData creates a new objectData in every execution, filling the objectData using the sensorEvent received. In order to put together into an objectData all the sensorEvent with a same timestamp, every time a new sensorEvent is received, it is necessary to check if some sensorEvent with the same timestamp has already been received. If so, a fusion of both objectData is done. Otherwise, the objectData is stored for a while, expecting that maybe some other sensorEvent will have its same timestamp value. To do this, an auxiliary Arraylist of *ObjectData* is used, called *bufferSignals.* The first *ObjectData* (in other words the first *sensorEvent*) goes directly into *bufferSignals.* The following ones are processed in a different way. When they are received, the *bufferSignals* Arraylist is checked for the existence of some *ObjectData* with the same timestamp value. If this exists, the sensor information in the new *ObjectData* is inserted into the Hashtable of the other *ObjectData* (fusion is done). By contrast, if there is not any *ObjectData* inside the *bufferSignals* that has the same timestamp value, the new *ObjectData* is also inserted in the *bufferSignals*.

The *bufferSignals* is circular in order to avoid making it too big, which would slow down the search for an *ObjectData* with same timestamp value. When *bufferSignals* has five elements, and another *ObjectData* wants to be inserted, the oldest one is inserted in the main buffer class and removed from *bufferSignals* (which shift the rest of the elements to the left), inserting the new *ObjectData* at the end of the *bufferSignals* Arraylist. The size of five elements for the Arraylist is large enough, due that *sensorEvents* with the same timestamp than the first *ObjectData* in the Arraylist will not be received after receiving five different *sensorEvent* timestamp values. Finally, the reference to the *ObjectData* inserted in the main class buffer (also circular) is sent using a message to the manager.



#	id	AccX	AccY	AccZ	MagX	MagY	RotVecZ	Humidity	Temperature	TimeStamp
1	1	-0.458999991416931	0.229000002145767	9.88300037384033	239.25	196.720001220703	0.0	0.0	0.0	12100106715136.0
2	2	-0.26800000667572	0.229000002145767	9.92099952697754	239.139999389648	197.119995117188	0.0	0.0	0.0	12100306993152.0
3	3	-0.458999991416931	0.26800000667572	9.9980001449585	238.460006713867	197.660003662109	0.0	0.0	0.0	12100507271168.0
4	4	-1.11000001430511	-0.995999991893768	10.956000328064	238.460006713867	197.660003662109	0.0	0.0	0.0	12100706500608.0
5	5	1.22500002384186	0.152999997138977	11.1859998703003	228.600006103516	199.300003051758	0.0	0.0	0.0	12100906778624.0
6	6	7.16300010681152	-0.651000022888184	12.1429996490479	206.100006103516	197.5	0.0	0.0	0.0	12101107056640.0
7	7	-3.44700002670288	3.71499991416931	-0.611999988555908	219.399993896484	181.399993896484	0.0	0.0	0.0	12101307334656.0
8	8	11.2239999771118	7.73799991607666	11.0710000991821	214.199996948242	180.0	0.0	0.0	0.0	12101517049856.0
9	9	13.6759996414185	16.7789993286133	15.1309995651245	224.899993896484	181.899993896484	0.0	0.0	0.0	12101726765056.0
10	10	-6.09100008010864	0.995999991893768	-1.14900004863739	229.899993896484	191.300003051758	0.0	0.0	0.0	12101936480256.0
11	11	-8.23600006103516	-6.51200008392334	-5.32399988174439	227.800003051758	191.600006103516	0.0	0.0	0.0	12102147244032.0
12	12	2.98799991607666	9.92099952697754	10.2659997940063	226.199996948242	183.199996948242	0.0	0.0	0.0	12102356959232.0
13	13	6.89499998092651	19.5750007629395	11.0710000991821	232.5	195.899993896484	0.0	0.0	0.0	12102566674432.0
14	14	-19.3449993133545	-5.1710000038147	5.86100006103516	238.5	202.0	0.0	0.0	0.0	12102776389632.0
15	15	13.5220003128052	5.01800012588501	-4.82600021362305	215.699996948242	181.899993896484	0.0	0.0	0.0	12102987153408.0
16	16	15.0930004119873	15.7440004348755	9.76799964904785	196.300003051758	178.0	0.0	0.0	0.0	12103186382848.0
17	17	-2.41300010681152	5.97599983215332	2.71900010108948	216.100006103516	195.699996948242	0.0	0.0	0.0	12103386660864.0
18	18	-4.48199987411499	-4.86499977111816	-4.63500022888184	219.899993896484	190.800003051758	0.0	0.0	0.0	12103586938880.0
19	19	-13.4840002059937	-19.5370006561279	-5.2480001449585	231.199996948242	198.0	0.0	0.0	0.0	12103787216896.0
20	20	-6.16699981689453	-15.1700000762939	-6.74200010299683	235.699996948242	208.600006103516	0.0	0.0	0.0	12103996932096.0
21	21	-4.59700012207031	7.16300010681152	2.68099999427795	234.600006103516	213.100006103516	0.0	0.0	0.0	12104206647296.0
22	22	-11.1090002059937	19.5750007629395	4.36700010299683	237.699996948242	206.699996948242	0.0	0.0	0.0	12104406925312.0
23	23	-7.16300010681152	13.9060001373291	3.44700002670288	239.100006103516	201.199996948242	0.0	0.0	0.0	12104607203328.0

Figure 3.11: Rows with same timestamp values combined.

Mobile timestamp format

Another problem faced was the format to be used for the timestamp values. In mobile devices, the timestamp value can be given in two different formats: epoch time given in milliseconds, or the called Uptime time, which is the milliseconds passed since the operating system was loaded. The format given for every device depends on the mobile manufacture.

In order to have the same format regardless of the mobile device manufacturer, when every *ObjectData* is going to be introduced in the main driver buffer, its timestamp is set to the system current time, using the system method *System.currentTimeToMillis()*.

Shimmer timestamp

The timestamp of the Shimmer devices is not given as the timestamp of a mobile phone. That is, the timestamp is not the milliseconds since January 1, 1970 UTC. It was discovered that the timestamp of a sample was the milliseconds since the device started to stream. A solution must be found in order to store a legible timestamp. The solution was to create an initial timestamp. It was set with the function *System.currentTimeToMilis()* in the moment when the device start to stream. Thus, when the framework's data structure is been filled with the sample received, the final timestamp is calculated making the addition of the initial timestamp and the timestamp of the sample. Unfortunately this led us to another problem. A set of activities were captured by the devices; but, when the data received was graphically represented, an unusual out-of-time was noticed. Eventually, it was discovered that the problem was that the device took a few seconds between the call to the *StartStreaming* function and the sending of the first sample. Owing to that, the initial timestamp was wrong. To fix this, it was decided to calculate the final timestamp in a different way. When the first sample is received by the handler of our driver, the variables *offsetSessionTimeStamp* and *offsetShimmerTimeStamp* are set. The first one is set with the function *System.currentTimeToMillis()* and the second one is set to the value of the timestamp gets from the first Shimmer's sample. Then, to calculate the timestamp to stored, the following equation is applied:
timeToStore = SessionTimeStamp + (timestampLastSample - timestampFirstSample).

Timestamp values

When *Data* object was first created, its variable *data* was defined as a float trying to optimize the framework since float is a single-precision 32-bit data type. It was thought that this would be enough. When the manager was tested and database checked, Timestamp column had wrong values. After a deep search, the problem was found. Function *System.currentTimeMillis()* is used to get the sample's timestamp. This function returns a long value, which is 64-bit signed two's complement integer data type. Unlike what we thought, conversion from long to float was not working well. Therefore, it was necessary to change the variable *data* from float to double, a double-precision 64-bit data type.

Consistency of the database

When a device starts to stream, the session metadata is stored, but only those which are known until this moment (start, first index and sensors enabled). The rest of them (finish and last index) are stored when the device stops streaming. To set the first index, the last index of the datas table is retrieved, and this number is incremented one value. The problem appears when the Android application fails while streaming data since the last index of the session remains to zero. To fix this, *consistencyTestMetada* function was made. An explanation of how this works can be found in the DBAdapter's description.

Closing the database

Several devices can be streaming at the same time. When their buffers are filled, they store the data. To do this, *CommunicationThread* opens the database, writes the data, and closes the database. This is done by separate threads in a concurrent mode. The database may be closed by a thread while the other is still writing which would produce an exception. To avoid this, it was necessary to create some mechanism. After studying different possibilities, it was decided to create the flag *isStoring*. This flag is inside of each *ObjectCommunication*. Before a thread starts to write in the database, the flag is set to *True*, and when it is done, the flag is set to *False*. Then, a function checks whether any thread is writing in the database. If the function returns *True*, it means someone else is writing, and therefore the database is not closed. Thus, it will only be the last thread that will close the database.

Database experiment

Due to the need to store the data received by the device into a database (SQLite), it is highly important to do it as fast and efficient as possible. In order to choose the best option, an experiment was done by testing its design, how to store the data, and where to save the database.

1. Database's design

    - Design A

      For the experiment only the *Signals* table was implemented. In this design, every Signal table's row keeps the value returns of a particular sensor and device.

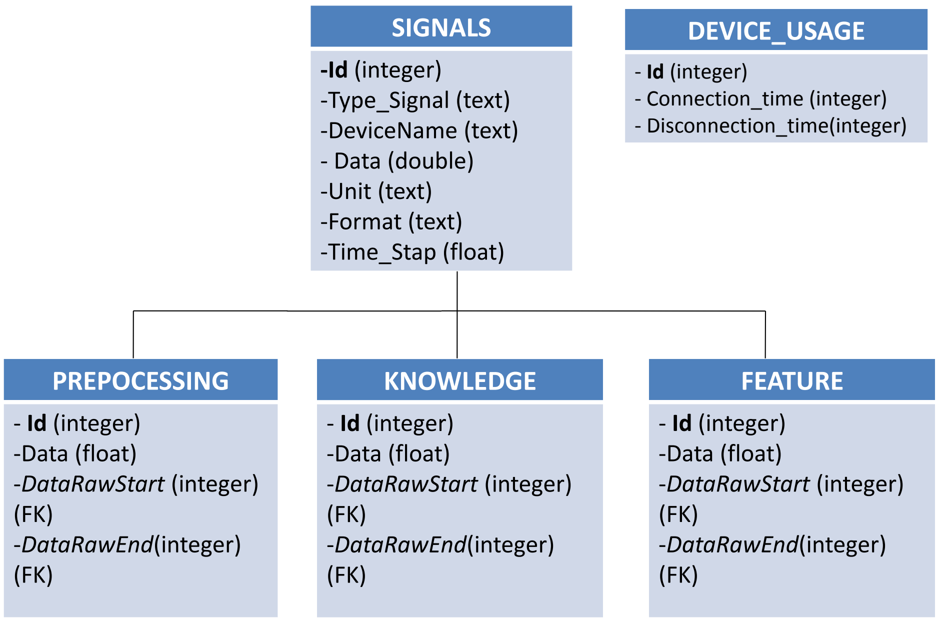
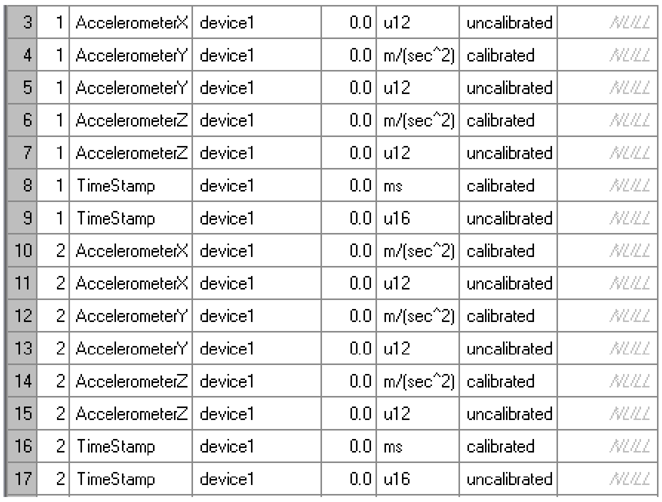


Figure 3.12: Database's design A.



3	1	AccelerometerX	device1	0.0	u12	uncalibrated	NULL
4	1	AccelerometerY	device1	0.0	m/(sec^2)	calibrated	NULL
5	1	AccelerometerY	device1	0.0	u12	uncalibrated	NULL
6	1	AccelerometerZ	device1	0.0	m/(sec^2)	calibrated	NULL
7	1	AccelerometerZ	device1	0.0	u12	uncalibrated	NULL
8	1	TimeStamp	device1	0.0	ms	calibrated	NULL
9	1	TimeStamp	device1	0.0	u16	uncalibrated	NULL
10	2	AccelerometerX	device1	0.0	m/(sec^2)	calibrated	NULL
11	2	AccelerometerX	device1	0.0	u12	uncalibrated	NULL
12	2	AccelerometerY	device1	0.0	m/(sec^2)	calibrated	NULL
13	2	AccelerometerY	device1	0.0	u12	uncalibrated	NULL
14	2	AccelerometerZ	device1	0.0	m/(sec^2)	calibrated	NULL
15	2	AccelerometerZ	device1	0.0	u12	uncalibrated	NULL
16	2	TimeStamp	device1	0.0	ms	calibrated	NULL
17	2	TimeStamp	device1	0.0	u16	uncalibrated	NULL

Figure 3.13: Example of the table from the design A.

Advantages

- Scalability: If a new device with new sensors is added to the API, only a few changes would be necessary in the Storage class.

- Completeness: With this design, it is ensured that rows are completed and there will not be *Null* values.

Disadvantages

  - Time and size: Since each row stores just one value, it takes a lot of time and needs more space.

- Design B

  This was the database picked, so its design and structure is explained above.

  Advantages

  - Time and size: Fewer rows are inserted so that less time and space is required. For instance, if the data is received from 4 different sensors, this design would insert 4 times less rows.

  - Completeness: With this design it is ensured that rows are completed and there will not be Null values.

  Disadvantages

  - Scalability: Creating a specific table for each kind of device is required, which implies a bigger programming effort.

2. Data insertion

- Mode 1

  Insertion in this mode is made one by one.

  - Code for design A:

```
ContentValues initialValues = new ContentValues();
initialValues.put(ID, index);
initialValues.put(TYPE_SIGNAL, signal);
initialValues.put(NAME_DEVICE, nameDevice);
initialValues.put(DATA, data);
initialValues.put(UNIT, unit);
initialValues.put(FORMAT, format);
initialValues.put(TIME_STAMP, timeStamp);
db.insert(TABLE_SIGNALS, null, initialValues);
```

  - Code for design B:

```
ContentValues initialValues = new ContentValues();
initialValues.put(ACCELEROMETER_X, array[0]);
initialValues.put(ACCELEROMETER_Y, array[1]);
initialValues.put(ACCELEROMETER_Z, array[2]);
initialValues.put(MAGNOMETER_X, array[3]);
initialValues.put(MAGNOMETER_Y,array[4]);
initialValues.put(MAGNOMETER_Z, array[5]);
initialValues.put(GYROSCOPE_X, array[6]);
initialValues.put(GYROSCOPE_Y, array[7]);
initialValues.put(GYROSCOPE_Z, array[8]);
initialValues.put(GSR, array[9]);
initialValues.put(ECG_RALL, array[10]);
initialValues.put(ECG_LALL, array[11]);
initialValues.put(EMG, array[12]);
initialValues.put(STRAIN_GAUGE_HIGH, array[13]);
initialValues.put(STRAIN_GAUGE_LOW, array[14]);
initialValues.put(HEART_RATE, array[15]);
initialValues.put(EXP_BOARDA0, array[16]);
initialValues.put(EXP_BOARDA7, array[17]);
initialValues.put(TIME_STAMP, array[18]);
initialValues.put(NAME_DEVICE, nameDevice);
initialValues.put(FORMAT, format);
db.insert(TABLE_SIGNALS, null, initialValues);
```

- Mode 2
  Insertion is done using transactions.
  - Code for design A:

```
String sql = "INSERT INTO "+TABLE_SIGNALS+" ("+ID+",
    "+TYPE_SIGNAL+", "+NAME_DEVICE +", "+DATA+", "+UNIT+",
    "+FORMAT+" , "+TIME_STAMP+") VALUES (?, ?, ?, ?, ?, ?, ?)";
db.beginTransaction();
SQLiteStatementstmt = db.compileStatement(sql);

for(inti=0; i<200; ++i){
   stmt.bindLong(1, id);
   stmt.bindString(2, type);
   stmt.bindString(3, name);
   stmt.bindDouble(4, data);
   stmt.bindString(5, unit);
   stmt.bindString(6, format);
   stmt.bindDouble(7, time);
   stmt.executeInsert();
   stmt.clearBindings();
}

db.setTransactionSuccessful();
db.endTransaction();
```

- Code for design B:

```
String sql = "INSERT INTO "+ nameTable+ " (" + ACCELEROMETER_X
    + ", " +ACCELEROMETER_Y + ", " + ACCELEROMETER_Z + ", " +
    MAGNOMETER_X + ", " +MAGNOMETER_Y + ", "+ MAGNOMETER_Z +
    ", " + GYROSCOPE_X+ ", " + GYROSCOPE_Y + ", "+
    GYROSCOPE_Z+ ", " + GSR + ", "+ ECG_RALL + ", " + ECG_LALL
    + ", " + EMG + ", "+ STRAIN_GAUGE_HIGH + ", " +
    STRAIN_GAUGE_LOW + ", " + HEART_RATE + ", "+EXP_BOARDA0 +
    ", " + EXP_BOARDA7  + ", " + TIME_STAMP + ", "+ LABEL+ ")
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?)";

db.beginTransaction();
SQLiteStatementstmt = db.compileStatement(sql);

for (inti = 0; i<array.length; i++) {
   stmt.bindDouble(1, array[i][0]);
   stmt.bindDouble(2, array[i][1]);
   stmt.bindDouble(3, array[i][2]);
   stmt.bindDouble(4, array[i][3]);
   stmt.bindDouble(5, array[i][4]);
   stmt.bindDouble(6, array[i][5]);
   stmt.bindDouble(7, array[i][6]);
   stmt.bindDouble(8, array[i][7]);
   stmt.bindDouble(9, array[i][8]);
   stmt.bindDouble(10, array[i][8]);
   stmt.bindDouble(11, array[i][10]);
   stmt.bindDouble(12, array[i][11]);
   stmt.bindDouble(13, array[i][12]);
   stmt.bindDouble(14, array[i][13]);
   stmt.bindDouble(15, array[i][14]);
   stmt.bindDouble(16, array[i][15]);
   stmt.bindDouble(17, array[i][16]);
   stmt.bindDouble(18, array[i][17]);
   stmt.bindDouble(19, array[i][18]);
   stmt.bindDouble(20, array[i][19]);
   stmt.executeInsert();
   stmt.clearBindings();
}
```

```
            db.setTransactionSuccessful();
            db.endTransaction();
```

3. Data storage

   - Internal memory: It is usually faster but has less capacity. It is difficult to get the database file.
   - External memory: It is usually slower but has more capacity. It is easy to get the database file.

Experiment

It is supposed that the following signals are received: Accelerometer_X, Accelerometer_Y, Accelerometer_Z, Magnometer_X, Magnometer_Y, Magnometer_Z, Gyroscope_X, Gyroscope_Y, Gyroscope_Z, TimeStamp, ECGRall and ECGLall. 12 signals in total. Taking into account the storage, two experiments were done. The first was done with SD card type 4 (4 MB/s), and the second one with SD card type 10 (10MB/s).

Table 3.1: Database experiment with HTC Sensation Z710e. Android 4.0.3 (SD card type 4)

Insertion	**Design**	**Storage**	**Time(s) *N=50**	**Time(s) N=100**	**Time(s) N=200**	**Time(s) N=500**	**Time(s) N=1000**
Mode 1	**A**	**Internal**	17.45	34.63	67.94	172.15	339.07
		External	77.84	156.17	329.63	776.12	1642.57
	B	**Internal**	1.76	3.14	6.63	15.04	32.98
		External	6.80	16.07	31.19	78.86	154.62
Mode 2	**A**	**Internal**	0.31	0.43	0.83	1.83	3.72
		External	0.97	1.35	1.71	3.52	4.56
	B	**Internal**	0.15	0.25	0.35	0.75	1.45
		External	0.89	1.07	1.11	1.64	2.24

*N = number of sample received. Design A would be: 50 samples * 12 signals = 600 rows whereas design B would be: 50 samples = 50 rows.

At the end of the first experiment, database 1 had 156600 rows and 7940KB.
At the end of the first experiment, database 2 had 13450 rows and 655KB.

Table 3.2: Database experiment with HTC Sensation Z710e. Android 4.0.3 (SD card type 10)

Insertion	Design	Storage	Time(s) N=50	Time(s) N=100	Time(s) N=200	Time(s) N=500	Time(s) N=1000
Mode 1	**A**	**Internal**	17.77	34.53	69.87	170.30	350.71
		External	18.65	39.24	77.25	172.41	385.25
	B	**Internal**	1.78	3.22	6.91	16.20	34.58
		External	1.65	3.99	6.86	16.93	35.12
Mode 2	**A**	**Internal**	0.29	0.46	0.82	1.85	4.19
		External	0.26	0.44	0.77	1.90	4.48
	B	**Internal**	0.12	0.19	0.24	0.52	1.10
		External	0.12	0.22	0.31	0.53	1.18

Taking into account that the sample rate was 50Hz:
655 KB / 13450 rows = 0.048 kb/row
T(1s) = 0.048 kb/row * 50 rows = 2.41 kb (50 rows)
T(3600s) = 0.048kb/row * 50 rows * 3600 seconds = 8640 kb = 8.4375 MB
T(86400s) = 0.048kb/row * 50 rows * 86400 seconds = 202.5 MB (a whole day)

Conclusions of the experiment

It is quite obvious that mode 2 of data insertion (transaction) is much faster than mode 1 (one by one). Regarding the database's design, it may seem that design B is faster than design A, mainly because design A does 12 times more insertions than design B. Thus, with N=500, design A inserts 6000 rows (12 signals * 500 samples) whereas design B just inserts 500 rows. In addition, the experiments show that the SD card type 10 is as fast as the internal memory, so using this kind of SD card gets all of the advantages and none of the disadvantages.

In conclusion, the best combination is to use design B: inserting rows with mode 2 and storing the database file into external memory. In the end, it was decided to implement this combination in order to make the framework the most efficient possible.

3.3.2 Remote Storage Manager

The Remote Storage Manager provides remote storage and uploading functionalities for the information stored in the mobile local database.

3.3.2.1 Description

For the sake of simplicity we build this section upon the two remote communication elements (i.e., client and server). Specifically, on the client side the eHealthDroid library functions and classes defined for the data uploading and retrieval are presented while some customized PHP templates are provided for the server side.

Client side

There are three main functions in the RemoteStorageManager class: *uploadUnitsTable, uploadSignalsTable* and *uploadMetadataTable.* These functions are used to upload Units, Signals and Metadata database tables (3.3.1.1). For this purpose, acquisition and dispatch of the desired data table to the thread used to transmit the data to the remote storing is done.

These functions receive as parameter the ID of the device from which the data is wished to upload. This ID is used to identify the corresponding database and tables. Once this is achieved, it is necessary to retrieve all the data (table rows) contained in the table. To that end, some of the functions defined for the Local Storage Manager are used. These are: *getMobileSignalsTable, getShimmerSignalsTable, getMobileMetadataTable, getShimmerMetadataTable, getMobileUnitsTable* and *getShimmerUnitsTable.* These functions call to functions of the DBAdapter class, here used to proceed with the database queries for obtaining the data.

To show the procedure to extract the data from a table is described the function *getMobileSignalsTable* of DBAdapter class, used to retrieve the signal table of a portable mobile. The data are returned using the structure *Hashtable<SensorType, ArrayList<Double>>*. The hash key is a signal type used within the table (columns), whereas the *ArrayList<Double>* comprises all the data values belonging to that signal. Using a SQL query, all the information is obtained and then, using a cursor an array list for every *SensorType* is filled with the data.

For the rest of the functions the process is similar, just extracting different columns depending on the device and the table selected (in some cases also different data structures). The functions to obtain the signal units are slightly different, since the tables used for the mobile data are different to the ones used for the external sensors.

Back to the RemoteStorageManager class, now that the data have been acquired is time to send them to the remote storage. A new thread is needed to avoid blocking the main one during the process of uploading, and this thread needs a Message Queue which is implemented by a Looper. A Looper is a message handling loop which reads and processes items from a MessageQueue, where messages are posted using a Handler. After the uploading process is important not to leave this thread running to avoid the waste of resources. Due to killing or stopping threads in Android is unsafe, the function *finishProcess* uses the *interrupt* Thread function, which tells the system to stop the thread when system needs to kill some threads. It also quits the threads looper. The data are sent to the thread inside messages by:

```
Message messageToSend = thread.myHandler.obtainMessage(3);
messageToSend.obj = data;
messageToSend.arg1 = MOBILE_DEVICE;
thread.myHandler.sendMessage(messageToSend);
```

The parameter *thread* is a RemoteStorageManager class variable of RemoteStorageThread type, class which extends of Thread and is responsible of the upload processing. This thread starts when the *RemoteStorageManager* class variable is created. The *obtainMessage* argument is just an ID (*message.what*) used to recognize what the thread must do with the received data. *Data* is the information that will be uploaded to the remote storage and *arg1* a number to recognize if the data belongs to a mobile device or a biomedical sensor (e.g., Shimmer device). This changes the PHP scripts to be executed in the server side.

Once the message with the data is sent, the thread needs to receive the created message using a Handler and depending on the ID code proceed in consequence. This is done by a switch-case, where the cases are the following:

- Case message.what = 0 → To upload mobile units table
- Case message.what = 1 → To upload shimmer units table
- Case message.what = 2 → To upload mobile/shimmer signals table
- Case message.what = 3 → To upload mobile/shimmer metadata table

Once inside a case, the data sent in the message need to be placed into a structure. As an example, it looks like this for the case that upload signals tables:

```
Hashtable<SensorType, ArrayList<Double>> data2 = (Hashtable<SensorType,
    ArrayList<Double>>) msg.obj;
```

Http Post requests are necessary to be executed on the server. The POST request method is designed to ask a web server for accepting the data enclosed in the request messages body for storage and is often used to upload files or submitting web forms. Thus, it is necessary an instance of an *Http Client* and *Http Post*.

```
DefaultHttpClient httpClient = new DefaultHttpClient();
HttpPost httpPost = new HttpPost(url);
```

The url parameter of the *HttpPost* constructor is really important. The url varies depending on the case being executed, for example:

```
url = serverIP + "/" + shimmerUnitsPath;
```

The ServerIP is the IP address where the remote storage is hosted. This attribute is a class attribute set through the function *setServerIP*. The rest of the url is formed by a url that contains the name of the script file that will be executed.

An *Http Post* is performed for every case. The data themselves are given to the web server using the Android object *NameValuePair* and *BasicNameValuePair*, which both together are used to encapsulate an attribute/value pair. In order to speed up the data uploading time, JSON (60) has been used as tool to pass the data to the server. A JSON Object is filled with all the information to be uploaded and once this is done, this is sent through a *NameValuePair* object by a post execution. For example, in case 1 corresponding to uploading the units table of the device Shimmer the code looks this way:

```
List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(2);
nameValuePairs.add(new BasicNameValuePair("TableName",
    tableNameShimmerUnits));
JSONObject json = new JSONObject();
JSONArray sensors = new JSONArray();
JSONArray calibrated = new JSONArray();
JSONArray uncalibrated = new JSONArray();

for (int i = 0; i < data1.size(); i++) {

   ArrayList<String> aux = data1.get(i);
   sensors.put(aux.get(0));
   calibrated.put(aux.get(1));
   uncalibrated.put(aux.get(2));
}
```

In this case, the data (*data1*) are encoded into an *ArrayList<ArrayList<String>>* structure. The table Shimmer Units contains three attributes: Sensors, Calibrated format and Uncalibrated format. This is the reason why three *JSONArray* are created and filled within the loop. These JSON arrays needs to be placed into the *JSONObject* which will be parsed in the server side after doing the post execution.

```
json.put("Sensors", sensors);
json.put("Calibrated", calibrated);
json.put("Uncalibrated", uncalibrated);

String jsonString = json.toString();
nameValuePairs.add(new BasicNameValuePair("Json", jsonString));
httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
HttpResponse httpResponse = httpClient.execute(httpPost);
```

Uploading signals tables from Shimmer and portable mobiles is slightly different. The data are enclosed into a *Hashtable<SensorType, ArrayList<Double>>* structure. That is why it is necessary to loop the hashtable as many times as different *SensorType* contains and creating a *JSONArray* with all the data belonging to a sensor. Once this is done, every *JSONArray* is inserted into the *JSONObject* that will be sent to the server and the process starts again with other sensor. As some values can be NaN, is necessary to check whether this happens and, if so, insert a null value instead. The code looks like follows:

```
List<NameValuePair> nameValuePairs = new
    ArrayList<NameValuePair>(2);nameValuePairs.add(new
    BasicNameValuePair("TableName", tableNameSignals));
JSONObject json2 = new JSONObject();
Set<SensorType> signals = data2.keySet();

for(SensorType s: signals){

   JSONArray aux = new JSONArray();
   for(int i = lastID; i < data2.get(s).size(); i++){

      if(data2.get(s).get(i).isNaN())
         aux.put(null);

      else
         aux.put(data2.get(s).get(i));
   }
   json2.put(s.getAbbreviature(), aux);

}

String jsonString = json2.toString();
nameValuePairs.add(new BasicNameValuePair("Json", jsonString));
httpPost2.setEntity(new UrlEncodedFormEntity(nameValuePairs));
HttpResponse httpResponse2 = httpClient.execute(httpPost2);
```

The reason why the loop goes from *lastID* to the end of the data structure will be explained later. In short, this is the way of preventing that information already stored in the server will not be uploaded twice.

The framework allows for the logging of the execution results using the function *logStatusAndJson*, which creates a *JSONObject* with information received by the server side and contains three tags to check if everything is done correctly: Success, Message and Rows (number of rows uploaded)

Difficulties and solutions applied. Client side

Partial table uploading

One of the faced problems was to avoid uploading information already available on the server. The employed solution differs depending on the data table that is being uploaded.

- **Mobile&Shimmer Units table.** The data to be upload are always the same, since the sensors formats do not change. Therefore, there is no need to upload this table more than once. To achieve this, the field ID (a string with the sensor name) is set as Primary Key (besides of not null), so there will not exist repeated keys.

- **Mobile&Shimmer Signals table.** To avoid uploading data which is already on the server, a PHP script called *GetLastID* is used to get the last ID of a specific table in the server. The loop where the JSON arrays are filled goes from the last ID to the size of the data acquired.

- **Mobile&Shimmer Metadata table.** It solves the problem in the same way as for the signals table.

Data transmission to server

As will be presented in section 3.3.2.3, the selection of JSON to do the data transmission to the web server provides a great performance. However, this was not the first implementation done. At the beginning, to upload every single row of a table, an *Http Post* execution was done, with as many *BasicNameValuePair* inside the *nameValuePairs* list as attributes were necessaries. For example, uploading Shimmer Units database tables looked like this:

```
List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(4);

for (int i = 0; i < data1.size(); i++) {

   ArrayList<String> aux = data1.get(i);
   nameValuePairs.add(new BasicNameValuePair("TableName",
       tableNameShimmerUnits));
   nameValuePairs.add(newBasicNameValuePair("Sensors", aux.get(0)));
   nameValuePairs.add(new BasicNameValuePair("Calibrated", aux.get(1)));
   nameValuePairs.add(new BasicNameValuePair("Uncalibrated", aux.get(2)));
   httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
   HttpResponse httpResponse = httpClient.execute(httpPost);
}
```

The performance is really poor in comparison with the one using JSON, due to the amount of *Http Post* requests performed in this implementation.

Server Side

The server side development is not part of the framework. However, it is provided a possible implementation that may be used or not, depending on the user requirements. For this work XAMPP has been used (61). XAMPP is a free and open source cross platform web server solution stack package, consisting of the Apache HTTP Server, MySQL database and interpreters for scripts written in PHP and Perl programming languages. Since it is essential to interact with the server in order to create tables or insert data into them, the server-side scripting language PHP (62) is used. Every PHP file is stored in a subfolder of XAMPP called *htdocs*, providing a functionality of the following ones:

- *DB_config.php*. Set variables as user, password, database and server address to be used for other scripts to connect with the database.

- *insert_mobile_signals.php.* Provides functionality to create (if necessary) a mobile signals table and insert new rows.
- *insert_mobile_metadata.php.* Provides functionality to create (if necessary) a mobile metadata table and insert new rows.
- *insert_mobile_units.php.* Provides functionality to create (if necessary) a mobile units table and insert new rows.
- *insert_shimmer_signals.php.* Provides functionality to create (if necessary) a mobile metadata table and insert new rows.
- *insert_shimmer_metadata.php.* Provides functionality to create (if necessary) a shimmer metadata table and insert new rows.
- *insert_shimmer_units.php.* Provides functionality to create (if necessary) a shimmer units table and insert new rows.
- *getLastId.php.* Provides functionality to get the last ID of a table.

These scripts are written using the PHP API *MySQLi* (63). Details of the implementation are described further ahead in the text.

Difficulties and solutions found. Server side

SQL Injection and Bobby tables. From MySQL to MySQLi API

At the beginning, the original PHP API *mySQL* (64) was used to write PHP scripts. This API has security problems allowing hacking techniques as SQL injection to be really easy.

Bobby tables (Figure 3.14) or its formally name SQL injection is a real-world exploit that is used by hackers and malicious users on daily basis (65). SQL injections work by inserting additional queries or code into your application. It does this by putting malicious code into form fields. The application receives the form field filled and executes the query, ignoring that actually may be executing more than one query. When someone is able to execute code this way in an application, it takes control over it, which allows him to do all kind of exploits like hosting fake paypal sites or child pornography. Here it is encountered of worth necessity to have control of the data. The

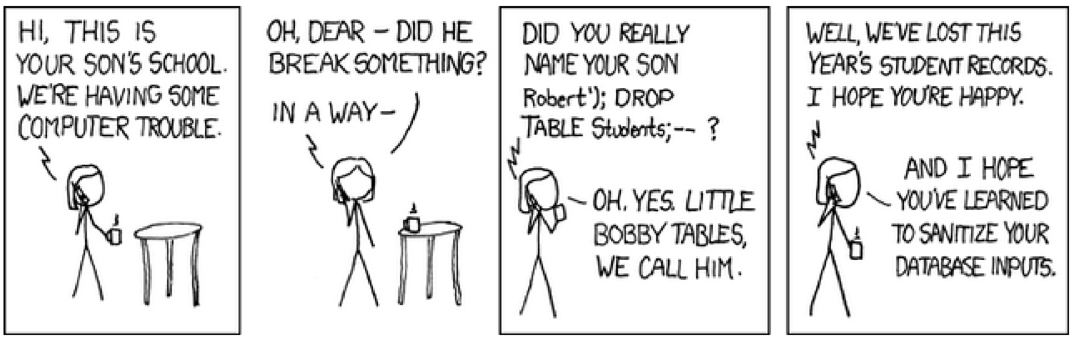

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 3.14: Bobby Tables/SQL injection comic. Consequences of allowing malicious code into form fields. Picture from (8)

importance of patients or users data makes absolutely necessary to guarantee privacy and security procedures to avoid any kind of disclosure.

It is important to point out how crucial is the prevention of servers from SQL injection, since SQL syntax is pretty much the same across different brands of databases, so the same exploit could run successfully against Oracle, MySQL, PostgreSQL, etc.

A simple example of a potencial SQL statement which could turn into SQL injection follows:

```
Select * FROM users WHERE user_name = $_POST['user'];
```

In this case, if the user variable passed to the PHP script is 'Juan', everything works fine and there is no problem. However, problems appears when the script input starts being malign. If the input is `"'Juan' OR 'a' = 'a'"` the SQL query looks like follows:

```
Select * FROM users WHERE user_name = "'Juan' OR 'a' = 'a'";
```

This would obtain all the information inside the users table due to the evaluation of `'a' = 'a'` is always true. But what happens when the input relinquished to the application is actually malicious?

```
Select * FROM users WHERE user_name = "'Juan' ; DROP TABLE
    users";
```

Once the query is submitted, the whole table users is dropped. This is the basic idea of SQL Injection.

At the beginning, the fraemework PHP scripts had this dangerous issue, looking SQL queries like follows:

```
    $id = $_POST['ID']; $accX = $_POST['AccX'];  $accY =
        $_POST['AccY']; $accZ = $_POST['AccZ']; $magX =
        $_POST['MagX']; $magY = $_POST['MagY']; $magZ =
        $_POST['MagZ']; $gyrX = $_POST['GyrX']; $gyrY =
        $_POST['GyrY']; $gyrZ = $_POST['GyrZ']; $light =
        $_POST['Light']; $pressure = $_POST['Pressure'];
        $proximity = $_POST['Proximity']; $gravX =
        $_POST['GravX']; $gravY = $_POST['GravY']; $gravZ =
        $_POST['GravZ']; $linAccX = $_POST['LinAccX']; $linAccY =
        $_POST['LinAccY']; $linAccZ = $_POST['LinAccZ']; $rotVecX
        = $_POST['RotVecX']; $rotVecY = $_POST['RotVecY'];
        $rotVecZ = $_POST['RotVecZ'];  $humidity =
        $_POST['Humidity'];  $temperature = $_POST['Temperature'];
        $timeStamp = $_POST['TimeStamp'];

$sql2 = "INSERT INTO mobile_signals(ID, AccX, AccY, AccZ, MagX,
    MagY, MagZ, GyrX, GyrY, GyrZ, Light, Pressure, Proximity,
    GravX, GravY, GravZ, LinAccX, LinAccY, LinAccZ, RotVecX,
    RotVecY, RotVecZ, Humidity, Temperature, TimeStamp) VALUES
    ($id, $accX, $accY, $accZ, $magX, $magY, $magZ, $gyrX, $gyrY,
    $gyrZ, $light, $pressure, $gravX, $gravY, $gravZ, $proximity,
    $linAccX, $linAccY, $linAccZ, $rotVecX, $rotVecY, $rotVecZ,
    $humidity, $temperature, $timeStamp);";

$result2 = mysql_query($sql2);
```

How SQL injection can be solved? It is important to ensure that the input data are actually what are supposed to be. In other words, never trust the network. In practice it is just a matter of checking the appropriate data types of the statement, rejecting or cleaning strange characters that can be found, as back ticks, semicolons or words as DROP, SELECT or UPDATE. Thus, to avoid SQL injection problems has been used the PHP API *MySQLi*, that is able to execute what is called Prepared Statements.

Mysqli API, Prepared Statements and Transactions

Prepared Statements (66) tend to increase security by separating the SQL logic from the data supplied to the application. In the previous example the data was placed into the SQL logic by building the query as a string on the fly.

On another note, transactions is a sequential group of database manipulation operations, which is performed as if it were one single work unit. This means a transaction will never be complete unless each individual operation within the group is executed successfully. If any operation belonging to the transaction fails, the transaction will fail and the database will not change its status.

To show the entire process of uploading a table using prepared statements and transactions by means of *MySQLi* API, the script to upload a shimmer signals table has been illustrated.

First of all, it is necessary to obtain a MySQLi object with all the information needed, such as user, password, database and server address. Once this is done, connection and creation of the table (if necessary) where data will be upload is done. In order to unable the automatic modifications of the database (to be able to use transactions), the function *autocommit* with parameter false is used.

```
require_once __DIR__ . '/db_config.php';
$mysqli = new mysqli(DB_SERVER, DB_USER, DB_PASSWORD,
   DB_DATABASE);
if ($mysqli->connect_errno)
   die('Connection Error (' . $mysqli->connect_errno . ') ' .
      $mysqli->connect_error);

$mysqli->autocommit(FALSE);
$tableName = $_POST['TableName'];
$sql1 = "CREATE TABLE IF NOT EXISTS" . "$tableName" . "(ID INT
   NOT NULL PRIMARY KEY AUTO_INCREMENT, Acc INT, Mag INT, Gyr
   INT, GSR INT, ECG INT, EMG INT, SG INT, HeartRate INT, ExpBA0
   INT, ExpBA7 INT, Format varchar(50), Start varchar(50),
   Finish varchar(50), First INT, Last INT, SampleRate INT);";
$result1 = $mysqli->query($sql1);
```

Once all this is done and if *$result1* is true (meaning was not any problem checking/creating the table) it is moment to set the transaction using prepared statements, having in mind that all the data is contained in the JSON object sent from the client side. For this, a loop is used and in every iteration a new insert SQL operation is added to the transaction. The most relevant code looks like follows:

```
$data = $_POST['Json'];
$json = json_decode($data, true);
$continue = true;

for($i = 0; $i < sizeof($json['AccX']) && $continue; $i++){
   $sql2 = "INSERT INTO " . "$tableName" . "(Acc, Mag, Gyr, GSR,
        ECG, EMG, SG, HeartRate, ExpBA0, ExpBA7, Format, Start,
        Finish, First, Last, SampleRate) VALUES
        (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?);";
    $stmt = $mysqli->prepare($sql2);
    $stmt->bind_param('iiiiiiiiiisssiii', $json['Acc'][$i],
        $json['Mag'][$i], $json['Gyr'][$i], $json['Gsr'][$i],
        $grav['Ecg'][$i], $json['Emg'][$i],
        $json['SG'][$i],$json['HeartRate'][$i],
        $json['ExpBA0'][$i], json['ExpBA7'][$i],
        $json['Format'][$i], $json['Start'][$i],
        $json['Finish'][$i], $json['First'][$i],
        $json['Last'][$i], $json['Sample_Rate'][$i]);
    $continue = $stmt->execute();
    $stmt->close();
}
```

As can be noticed, now the data have not been supplied to the query as it was done before. Every character of the first parameter of *bind_param* function denote the variable type which will be introduced. To finish, the variable $continue informs if any trouble occurs during the execution of operations. If this happens, a rollback (`$mysqli->rollback()`) must be done to cancel the whole transactions. Else, it proceeds to commit the transaction (`$mysqli->commit()`) and to send a JSON object containing information about the uploading process to the client side.

Logging JSON

As was commented in the client-side chapter, there exists the functionality to log the process of uploading information. Thus, a JSON is created at the beginning of the script and is being filled depending on the server responses after requesting executions. For instance, when the process goes well, the array is filled by doing this:

```
if($continue){
   $mysqli->commit();
   $mysqli->close();
   $response["Success"] = 1;
   $response["Message"] = "Data was upload successfully!";
   $response["Rows"] = "Rows uploaded: " . $i;
    echo json_encode($response);
}
```

By contrast, if a problem occurs, for example creating the transaction or creating the table, the JSON is filled like this:

```
else{
  $mysqli->rollback();
  $response["Success"] = 0;
  $response["Message"] = "Oops! Couldnt upload data! (Shimmer
     Signals)";
  $response["PhpError"] = mysql_error();
  echo json_encode($response);
}
```

3.3.2.2 How to use it

The way to use this module is quite simple. RemoteStorageManager is a singleton class (class instantiation restricted to one object), so to obtain a RemoteStorageManager instance is necessary to proceed as follows:

```
ServerManager sm = ServerManager.getInstance();
```

It is also necessary to initialize the Storage variable belonging to the RemoteStorageManager class called storage. This is done by:

```
sm.CreateStorage(getApplicationContext());
```

Then the server IP address is set by means of the function *setServerIP*. Also, it is compulsory to set the name and path (if necessary) of the scripts that will be executed in the server side. These scripts must be accessible from the server IP address. For example, using XAMPP like in this example the scripts inside the *Htdocs* can be accessible by the address *http://serverIP/scriptName*.

```
sm.setServerIP("http://192.168.1.11");
sm.setMobileMetadataPath("insert_mobile_metadata.php");
sm.setMobileSignalsPath("insert_mobile_signals.php");
sm.setMobileUnitsPath("insert_mobile_units.php");
sm.setShimmerMetadataPath("insert_shimmer_metadata.php");
sm.setShimmerSignalsPath("insert_shimmer_signals.php");
sm.setShimmerUnitsPath("insert_shimmer_units.php");
sm.setLastIDPath("get_last_ID.php");
```

If Wi-Fi connection is used, Wi-Fi must be enabled. The user can turn it on through the *setWifiEnabled* function defined on the System Manager.

```
sysm.setWifiEnabled(true, getApplicationContext());
```

There are three functions for uploading, one for each database table: data, metadata or units. For example, the uploading of the the database tables for the mobile device ("Mobile device") and the ones for a given wearable monitoring device ("device 1") is defined as follows:

```
// To upload shimmer tables
sm.uploadUnitsTable("device 1");
sm.uploadMetadataTable("device 1");
sm.uploadSignalsTable("device 1");

// To upload mobile tables
sm.uploadUnitsTable("Mobile Device");
sm.uploadMetadataTable("Mobile Device");
sm.uploadSignalsTable("Mobile Device");
```

To know when the data are upload, it is needed to run within a thread the function *isProcessFinished()*. An example of how to do this follows:

```
Handler mHandler = new Handler();
mHandler.post(isFinished);

Runnable isFinished = new Runnable(){

   @Override
   public void run() {
      if(rsm.isProcessFinished()){
         Toast.makeText(getActivity(), "Uploading completed",
             Toast.LENGTH_LONG).show();
      }
      else
         mHandler.postDelayed(this, 1000);
   }
};
```

Once all the data are uploaded, the thread responsible for the uploading must be stopped, to save the waste of resources.

```
sm.finishProcess();
```

To finish, according to users preferences, the WiFi connection could be turn off.

```
sysm.setWifiEnabled(false, getApplicationContext());
```

To upload the local database, some PHP scripts are necessaries, provided along with the framework. Different scripts could be used, but the framework user must be careful with the scripts name and the paths where these are located. By default, if the user does not change scripts names or paths, the files *get_last_ID.php, DB_config.php, insert_mobile_metadata.php, insert_mobile_signals.php, insert_mobile_units.php, insert_shimmer_metadata.php, insert_shimmer_signals.php* and *insert_shimmer_units.php* must be located in a path where can be accessible by the address *http://serverIP/nameFile.php* (for example, in the directory Htdocs for XAMPP).

Consequently, if the user wants to use different scripts or just for server reasons needs to change their path, it can be achieved by means of the following functions: *setMobileUnitsPath(String path), setMobileSignalsPath(String path), setMobileMetadataPath(String path), setShimmerUnitsPath(String path), setShimmerSignalsPath(String path), setShimmerMetadataPath(String path)* and *setLastIDPath(String path).*

3.3.2.3 Uploading test

The elapsed time to upload data depends on the upload speed of the internet connection used and of the number of rows that will be uploaded. As was commented, two different implementations have been developed. The first one uses as many *Http Post* executions as rows needed to be uploaded and the second one passes the data into a JSON.

For testing the first model, a connection with an upload speed of 749 kbps (93.6 KB/s) was used. The local database looked like follows:

- Mobile Units Table: 11 rows
- Metadata Table: 5 rows
- Signals Table: 601 rows

It gives a total of **617 rows** to upload. The elapsed time for the whole uploading process was **11 minutes 30 seconds**.

For the second test corresponding to the final setup a connection with an upload speed of 701 kbps (87,6 KB/s, slightly slower) has been used. In this case, the number of rows was higher than in the first test, 1021 rows to upload:

- Mobile Units Table: 11 rows

- Metadata Table: 5 rows
- Signals Table: 1005 rows

The elapsed time for the whole process of uploading this time was **8.768 seconds**, way faster than the first model.

3.3.3 Visualization Manager

This is the manager in charge of the data graphic representation. The data representation can be dynamic or static; in other words, the manager offers the possibility of the online and offline data representation. In the offline mode, it can represent both data acquired from the device and, furthermore, other kinds of data such as the session length. The manager is composed, besides by the own manager, by a class called *Plot*.

3.3.3.1 Description

Since Android does not provide a good API for making charts, it was decided to look for a library which could help us with this task. Several libraries were considered: Androidplot (67), Graphview (68), Chartdroid (69), Afreechart (70), Achartengine (71). Finally, Graphview was the library chosen to make this manager. It was chosen due to the fact that it is a free, open source (anything can be changed in case it is demanded), and easy to use. It is also possible to implement online visualization.

Graphview library

GraphView library has five important classes:

- GraphView: This is the main class. It is abstract and extends from LinearLayout. This object represents the frame of the graph and almost everything is defined and implemented here.
- GraphViewData: It is an inner class defined inside of the *Graphview* class. It has only two variables: valueX and valueY. This object represents a point in the graph and their variables are the coordinates.
- GraphViewSeries: All the 2D graphs are composed by a set of coordinates (Y axis and X axis). This class represents the sets of data to be drawn. The set of data is defined as an array of *GraphViewData*. It is possible to choose the color and thickness of the series as well as to store a little description of it. This class also implements functions to either append data or reset them.
- LineGraphView: This class extends from the *GraphView* class, and it is only in charge of painting specifically a line graph.
- BarGraphView: This class extends from the *GraphView* class and it is only in charge of painting specifically a bar graph.

The library had to be modified in order to make the visualization online, but this will be explained in 3.3.3.3.

Manager

At this point, a class which deals with both the library and the manager was demanded. Thus, the class *Plot* was created. The class *Plot* can be defined as a middle layer, and it represents a graph with its series. It has three variables: a *Graphview* object and two Hashtables. The *Graphview* object is the object necessary to use the library. One of the Hashtable is made to store the list of the series belonging to the graph, a list of *GraphViewSeries* objects. The other Hashtable is made to store the counter of each series. These counters are the values of the highest X coordinates of the graphs. The class defines an enumeration: *GraphType*. This enumeration is made to choose what kind of graph is desired, either line graph or bar graph.

There are some methods (*addSeries, appendData, resetData, removeSeries, removeAllSeries*) that modify the graph and the class variables in some way, and call the library functions.

- *addSeries* receives the series name and the data to be represented and converts them into the coordinates of the graph by creating an array of *GraphViewData*. Then, a counter with the number of coordinates is stored, the series is created, and added to the graph. The function returns either True in a case of success or False in a case of not success. There is another version of this function, which receives as parameters the color and the thickness of the series.

- *appendData* receives the series name, the value to be inserted at the end of the series, and a *Boolean* indicating whether or not the graph must scroll to the last value. The function increments its counter and adds the value in the end of the series.

- *appenData2* is a function that was necessary to implement for the online visualization. It is actually defined in the *GraphViewSeries* class and introduces a value at the end of the series. But unlike *appenData*, this function does not increase the size of the array with the values. It moves the values one position to the left in the array and introduces the new value in the last position. It could be expressed as follows: X[0] = X[1], X[1] = X[2], ... X[n-1] = X[n], X[n] = NewValue.

- *resetData* receives the series name and an ArrayList with the new values. The function creates a new array with the values given and replaces the current array for the new one. Finally it replaces the counters.

- *removeSeries* and *removeAllSerie* removes a single series or removes all the series respectively.

The rest of the methods of this class -*setHorizontalLabels, setViewPort, setScalable, isScrollable, getLegendAlign ...*- are made to mask the most relevant functions of the library. That is, the methods only call to the library functions.

The Visualization Manager was created to make a representation of any kind of data, but overall, to make possible the online visualization of the data received. It extends from *Activity* (the why will be explained later) and has four variables: an instance of itself since it is a singleton class, an instance of the Communication Manager in order to get the data sending by the sensors, a Hashtable which contains all the graphs, and finally, an array with eight different colors.

Thinking about the online visualization, and in a similar way of how it was done in the communication manager, a new object was created: *ObjectVisualization*. This object is an inner class which has three variables: a *Plot* object, an ArrayList and an Int. The *Plot* object is the graph, the ArrayList is a list of the device sensors to be visualized, and the Int is a counter. The last two variables are only used for the online visualization.

As it happens with all the managers, the function *getInstance* must be used to get the unique instance of the manager in order to use it. Creating the graph is the first thing to do in order to visualize the data. To do that, this manager offers the function *addGraph*, which needs as parameters the name of the graph (it must be unique), the type of the graph, and the application context. Another important function in this topic is *paint*, which needs as parameter the graph name and a LinearLayout. What this function does is add the customized graph to the LinearLayout in order to be painted. Taking that into account, this function should be called at the end, when everything in the graph is already set. Once the graph is created, to make the offline visualization or create simple graphs, the manager offers the *addSerie* function. This function needs as parameters the name of the graph, the name of the series, and the data to be represented. The function checks whether the graph exits and call the function *addSeries* of the class *Plot*. The *appendData* function is also interesting for the statics graphs since it allows values to be added at the end of the series. It receives as parameters the graph name, the series name, the value, and a *Boolean* indicating if the graph must scroll to the end. The function first checks whether the graph exists and then calls the *appendData* function of the *Plot* class.

To make the online visualization, it was decided to take the same approach used for the Communication Manager. That is, the data received by the communication manager would be sent to the visualization manager by messages. The *ObjectCommunication* of the Communication Manager defines an ArrayList with the names of the graphs which will represent the data. A flag is also defined, which is activated whenever the data must be sent to this manager. In the Visualization Manager, it was necessary to define a *Handler*, which receives all the messages. Since the manager is going to change the user interface, the code of the Handler has to be defined inside of the *runOnUiThread* function. This function makes the code inside of it run in the thread of the user interface. But the function is defined in the *Activity* class. This is why the manager inherits from this class. The messages sent by the Communication Manager contain an *ObjectData*. This *ObjectData* is the sample sent by the device to the Communication Manager. The device name is retrieved from the *ObjectData* and with it, the *ObjectCommunication* is retrieved from the Communication Manager. The list with the names of the graphs, which will represent any data of the sample, is retrieved from the *ObjectCommunication*. Then, for each graph in the list, the sample is checked to see if there are any values of the sensors to be represented. In case there are, the new value is added to its corresponding series. There are two cases when the data are added and it depends on the viewport. When the number of samples received is below of the viewport, the function *appendData* is used. Otherwise, the function *appendData2* is used. It is coded like this to get the data representation from being continuous. Finally, the counter of the samples received is incremented, and the graph is redrawn.

To start the online visualization, the manager offers the function *visualizationOnline*. It receives the graph name, the device name, and a list of the sensors which will be represented in the graph. The function creates a series for each sensor given and activates the flag of the *ObjectCommunication* for sending the data to the Visualization Manager. It is important to note that the data of a device can be handled by different graphs, but a graph can NOT handle the online visualization of different devices.

To stop the online visualization, the manager offers the function *stopVisualizationOnline*. It receives the graph name and the device name. The function removes the graph name from the list of the graph defined in its corresponding *ObjectCommunication*. Then, the flag for sending the data to the visualization manager is deactivated in case the list of the graph is empty.

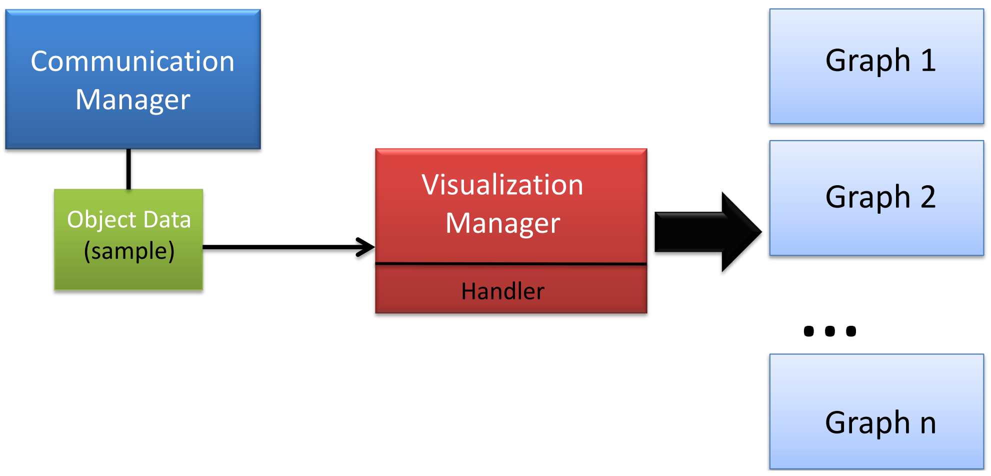


Figure 3.15: Passing message from the Communication Manager to the different graphs.

The functions *removeSerie* and *removeAllSerie* remove a particular series or all of the series respectively from statics graphs as well as from a graph with online data visualization. It does this by calling to their homonymous functions in the class *Plot*. The rest of the functions of this manager, *setScalable, setVerticalLabels, getLegendAlign, isScrollable, isShowLegend, etc*, are made to set the graph features, and basically, just call to the functions of the library through the class Plot.

3.3.3.2 How to use it

Since it is a singleton class, first it is necessary to get the instance:

```
VisualizationManager vm = VisualizationManager.getInstance();
```

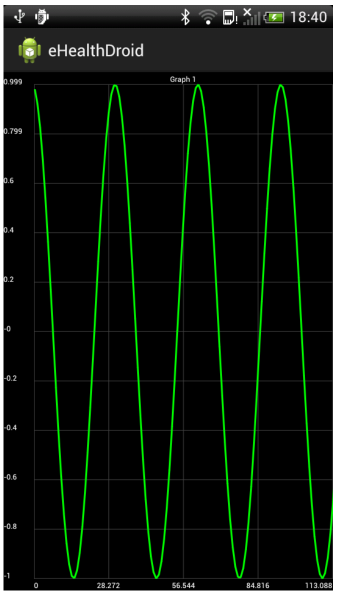
Then, a graph must be created. A unique name to the graph and its type (Line or Bars) must be selected. The UI context must also be indicated.

```
vm.addGraph("graph", GraphType.LINE, getApplicationContext());
```

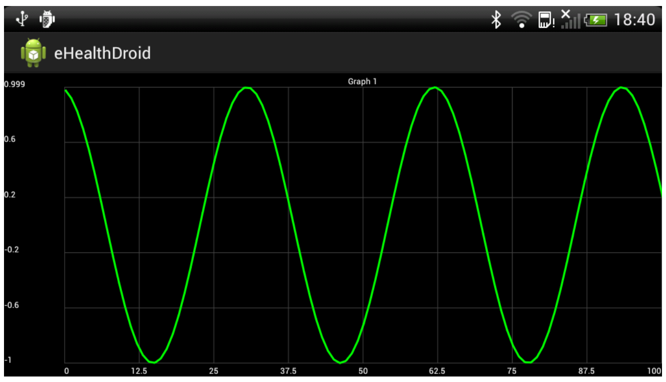
Now, it depends on the approach desired, either online or offline. Here, both of them are explained.

- Offline
  To create a series, first it is necessary to create an array with the values of the Y coordinates. For this example, the Sine of a set of numbers will be used:

  ```
  float[] array = new float[250];
  float a = 0;
  ```

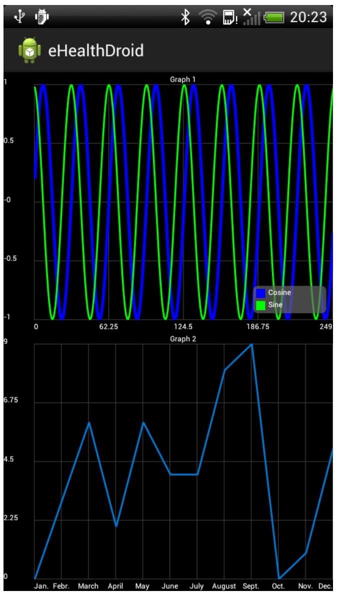


(a) Horizontal view

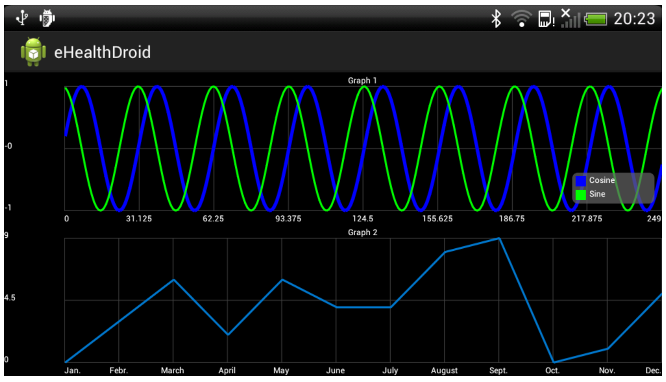


(b) Vertical view

Figure 3.16: Simple grah with one series.

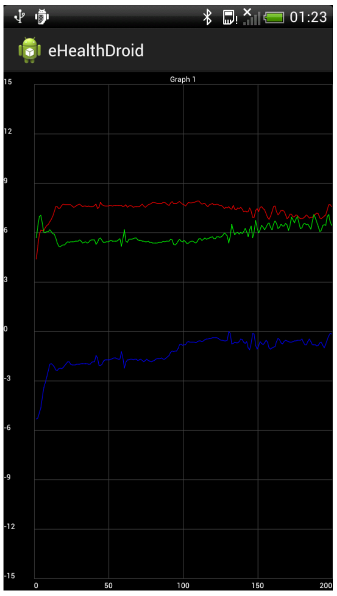


(a) Horizontal view

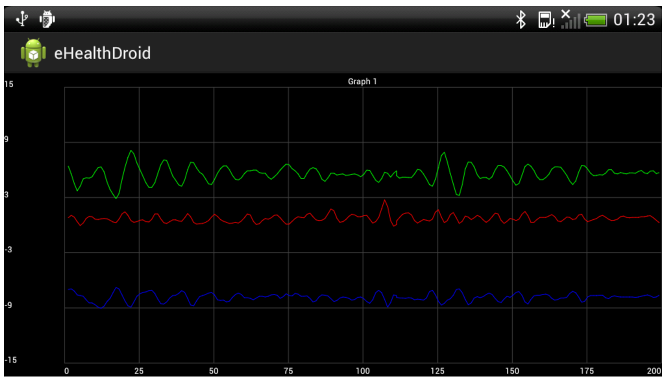


(b) Vertical view

Figure 3.17: Two graphs in the same view. The first one with two series and legend. The second one with customized labels.



(a) Horizontal view



(b) Vertical view

Figure 3.18: Online visualization of the 3D accelerometer signals.

```
for (int i = 0; i < 250; i++) {
   a += 0.2;
   array[i] = (float) Math.sin(a);
}
```

To introduce the data into the graph, the following procedure must be used:

```
vm.addSerie("graph", "Sin", array);
```

or

```
vm.addSerie("graph", "Sin", array, Color.BLUE, 1, "Sin from 0 to 250");
```

A series can be introduced with a description. To show it, one must activate the option. Its alignment in the graph can be set as well:

```
vm.setShowLegend("graph", true);
vm.setLegendAlign("graph", LegendAlign.TOP);
```

The graph must be included in a layout In order to be painted:

```
LinearLayout layout = (LinearLayout) findViewById(R.id.graph);
vm.paint("graph", layout);
```

Horizontal and vertical labels are created automatically, but they can be set like this:

```
String [] month = {"January", "February", "March", "April", "May",
    "June", "July", "August", "Sept.", "Oct.", "Nov.", "Dec."};
vm.setHorizontalLabels("graph", month);
vm.setVerticaltalLabels("graph", month);
```

- Online

  Like in the offline mode, it is necessary to add the graph into a layout:

```
LinearLayout layoutOnline = (LinearLayout)
    findViewById(R.id.graphOnline);
vm.paint("graph", layoutOnline);
```

To visualize online data, it is necessary indicate the set of sensors desired:

```
ArrayList<SensorType> sensors = new ArrayList<SensorType>();
sensors.add(SensorType.ACCELEROMETER_X);
sensors.add(SensorType.ACCELEROMETER_Y);
sensors.add(SensorType.ACCELEROMETER_Z);
sensors.add(SensorType.GYROSCOPE_X);
sensors.add(SensorType.GYROSCOPE_Y);
sensors.add(SensorType.GYROSCOPE_Z);
sensors.add(SensorType.MAGNETOMETER_X);
sensors.add(SensorType.MAGNETOMETER_Y);
sensors.add(SensorType.MAGNETOMETER_Z);
```

The *scrollable* feature of the graph MUST be activated. Otherwise online visualization will NOT work:

```
vm.setScrollable("graph", true);
```

Now it is the time to use the function, which does all the work, to visualize the data:

```
vm.visualizationOnline("graph", "nameDevice", sensors);
```

To stop the visualization process, the following method may be used:

```
vm.stopVisualizationOnline("graph", "nameDevice");
```

There are some features that are advisable to set in order to get a better visualization in both modes:
- Make the graph scalable (advisable for offline mode):

```
vm.setScalable("graph", true);
```

- Set the init and the viewport (advisable for online mode):

```
vm.setViewPort("graph", 1, 200);
```

- Set the values of the Y axis (advisable for both modes):

```
vm.setManualYAxisBounds("graph", 10, -10);
```

3.3.3.3 Difficulties and solutions applied

Online visualization

As it has been said before, one of the most important points of this manager was the online visualization. The first version developed was tested on a mobile phone with Android 2.3.3. It worked well, but when it was tested on a mobile phone with Android 4.0.3, it did not work. To make sure which was the problem, the application was tested in different phones. Then, it was noticed that in versions of Android over 4.X it did not work. As it was explained in the previous section, the online visualization lies in the addition of data at the end of the series with the function *appenData*. This function scrolls the graph to the end and redraw it using the function *redrawAll* defined in the class *GraphView*.

```
public void redrawAll() {
   verlabels = null;
   horlabels = null;
   numberformatter = null;
   invalidate();
   viewVerLabels.invalidate();
}
```

The work of redraw is done with the function *invalidate* which belongs to the Android API, specifically to the class *View*. But this function did not work properly; it should redraw the graph eventually, and it was only done when the screen was touched. Since this was an internal problem of Android, it was not possible to fix it. To make the online visualization work, the following solution was applied:

```
public void redrawAll() {
      verlabels = null;
      horlabels = null;
      numberformatter = null;
      this.invalidate();
      ViewGroup v;
      v= ((ViewGroup)getParent());
      v.removeView(this);
      v.addView(this);
      viewVerLabels.invalidate();
}
```

The View (the whole graph) is removed from its parent and is added right after.

When this problem was fixed, it was possible to make an intensive battery test. Then, a new problem was noticed. As it is shown in the next piece of code, the function *appenData* makes a copy of the current series. Then, the new value is added at the end of the array. This means that the array just gets bigger and bigger.

```
public void appendData(GraphViewData value, boolean scrollToEnd) {
   GraphViewData[] newValues = Arrays.copyOf(values, values.length+1);
   newValues[values.length] = value;
   values = newValues;
   for (GraphView g : graphViews) {
      if (scrollToEnd) {
         g.scrollToEnd();
      }
   }
}
```

The function is called every time the Visualization Manager receives a message. That is, for a sample rate of 50 Hz, the function is run approximately 50 times per second. In 10 minutes of online visualization, the array would have 30000 elements, and the function would have done 30000 memory reallocation (owing to the Array.copyOf function). This would be only for one series/signal. Despite the fact the mobile phones are equipped with a lot of RAM memory nowadays, Android just gives a small amount of it to the applications, and in case they need more (when the garbage collector is called), it is given little by little. Therefore, the system would not be able to handle a long visualization.

During the test, when the array had a size of around 10000 elements, the application started to collapse, and eventually it failed. Since the library needed many changes to fix it, an alternative solution must to be found. It was decided to not let the array get bigger than the viewport. To do this, it was necessary to create a new *appenData* function:

```
public void appendData2(GraphViewData value, boolean scrollToEnd) {
      GraphViewData[] newValues = Arrays.copyOfRange(values, 1,
          values.length+1);
      newValues[values.length-1] = value;
      values = newValues;
      for(int i=0; i<values.length; i++)
         values[i].valueX--;
      for (GraphView g : graphViews) {
         if (scrollToEnd) {
            g.scrollToEnd();
         }
      }
}
```

In this function, a copy of the series from the index 1 to the arrays length plus one is created first. This is done like so to validate that the copy has the same size and to discard the last element (first index) at the same time. Then, the new element is added at the end of the array. In order to make the graph continuous, all the X coordinates are decremented. Finally, the graph is scrolled to the end and redraws. Thus, the *appenData* function is used until the number of samples received is equal to the viewport. Then, the *appenData2* function is used after that to avoid the array increase and to create the illusion of continuity.

Viewport variable

In order to make the online visualization, getting the size of the viewport was necessary; but it was not accessible since it was defined as a *private* variable. A new function in the class *GraphView* had to be created to get the viewport.

```
public double getViewportSize(){
   return viewportSize;
}
```

3.3.4 System Manager

The System Manager aims are helping the user to manage intrinsic aspects that specifically corresponds to the mobile device. It is composed by the following modules:

1. *Services* → Provides functionality to make a phone call, sending messages or schedule notifications.
2. *Setup* → Provides functionality to set WIFI or Bluetooth or adjust the screen brightness.
3. *Guidelines* → Provides functionality to reproduce YouTube videos as well as local stored ones. Audios stored locally can be reproduced too. It is composed by three sub-modules: YouTube, Audio and Video.

3.3.4.1 Description

To develop these modules Android APIs has been used. Thus, encapsulating them in a proper way was in some cases the main task to worry about. By contrast, the YouTube Guidelines sub-module has been developed using the Android YouTube Player API and is development is way more different.

Services

The functionalities provided for this module are to make a phone call, to send a text message and to create notifications that may be scheduled at a specified time.

For the function *call* used to make a phone call is necessary to create a new Intent, setting the type parameter with *Intent.ACTION_CALL* and the phone number to call. Once this is done, a new activity with this intent as parameter can be launched. In the case of *sendSMS* function used to send a text message, a *SmsManager* must be obtained to use its function *sendTextMessage*, giving as parameter the phone number and the text message to be sent.

With respect to notifications, there were multiple ways to implement them due to each Android version provides different functionality to create them. Since the framework developed is compatible with the Android SDK Version 10, notifications implementation uses the functions provided for this Android version. To create a notification, it is needed to obtain a Notification Manager (72). Then a new Notification object must be created and set properly. Some notifications features like flags and sounds are optional, which means that if these are not specified their default values are set automatically. However, others parameters are compulsory, such as context or the Intent to be launched when the user clicks the notification. Finally, the created notification is added to the Notification Manager.

Since there are different features available to create notifications, three different functions are offered. The function *sendSimpleNotification* send a simple and default notification with a title, text and icon. With *sendComplexNotification* is possible to specify also the Intent that will be launched when the user clicks on the notification. Finally, with *sendComplexNotificationCustomSound* is also possible to specify a sound reproduced when a notification is launched.

Notifications can also be scheduled. To do that, a new class called ScheduledTask was created. It extends from BroadcastReceiver (73) in order to be able to receive intents. It contains only one function called *onReceive*. It checks what kind of task is demanded -simple notification, complex notification, complex notification- and run it. In order to use it must be declared as a receiver in the Android manifest. Thus, to schedule a notification, first a new Intent with the ScheduledTask class as parameter is created. The Intent will be launched at the time scheduled. Secondly, a PendingIntent, which will perform the broadcast, is created. Then, an AlarmManager (74) is retrieved from the system. Finally, the alarm is registered in the system at the specified time. The ScheduledTask is a class specifically created to schedule tasks.

Due to the fact that there exist three different functions for notifications, there also exist three functions to schedule these notifications: *scheduledSimpleNotification*, *scheduledComplexNotification* and *scheduledComplexNotificationCustomSound*. It is also possible to schedule the playing of an audio or soundtrack using the function *scheduledAudio*.

Setup

This module provides functionality to set the Wi-Fi ,Bluetooth protocol or to adjust the screen brightness.

For the *setWifiEnabled* function offered to set the Wi-Fi, a WifiManager is obtained in order to use its own function *setWifiEnabled*. With Bluetooth the process is similar, but using a mBluetoothAdapter and its functions *enable* and *disable*. With respect to the *setScreenBrightness* used to adjust the screen brightness, a WindowManager is obtained and the screen brightness is modified changing its value *screenBrightness*.

Guidelines

YouTube

The guideline module offers two functionalities: reproducing a YouTube video in a *YoutubePlayerView* and loading in a *listView* every video of a YouTube playlist reproducing the one selected by the user.

The main class of this module is the Youtube class. The function *reproduceSingleVideoMode* is in charge of reproducing a YouTube video. This function calls to *getYouTubeVideoID* to obtain the real YouTube ID video (just in case the API user does not provide it directly). The Youtube ID video is inside the url body, so just doing some matching pattern recognition the ID is be extracted.

Once this is done, the method *initialize* of the *youtubePlayerView* is called, needing a key developer and a *YoutubePlayer.onInitializedListener*, which is an interface for callbacks which are invoked when player initialization succeeds or fails. If the initialization is successful, the function *onInitializationSuccess* is executed, reproducing the video with *cueVideo*. Otherwise, if the initialization fails, *onInitializationFailure* is executed, showing a toast error message.

The development of the function of reproducing a playlist with its corresponding *listView* is way more complex. The function to be used is *reproducePlayListMode*, which needs as parameter a *listView*, a layout with the format for each entry (*video description, url and thumbnail image*) and the playlist *url*.

The first thing to be done is to acquire all the information about every video in the playlist. This process is done in an auxiliary thread (to not block the main one) and is accomplished by the class GetYoutubeUserVideosTask (which implements Runnable). The method *run* performs the following steps to obtain the information of every video in the playlist:

1. Execution of an HttpUriRequest.

   ```
   HttpClient client = new DefaultHttpClient();
   HttpUriRequest request = new
       HttpGet("https://gdata.youtube.com/feeds/api/playlists/" +
       playlist + "?v=2&alt=jsonc");
   HttpResponse response = client.execute(request);
   ```

   The parameter *playlist* is the playlist ID provided by the API user.

2. Creation of a JSONObject using the obtained response.

   ```
   BufferedReader reader = new
       BufferedReader(newInputStreamReader(response.getEntity().getContent(),
       "UTF-8"));
   ```

```
String jsonString = reader.readLine();
JSONObjectjson = new JSONObject(jsonString);
```

3. Creation of a JSONArray using the tags “data” and “items” belonging to the JSONObject and creation of an ArrayList where videos will be introduced. The ArrayList contains Video objects, where Video is a class with 3 attributes: *title*, *url* and *thumbnail url*.

   ```
   JSONArrayjsonArray = json.getJSONObject("data").getJSONArray("items");
   ArrayList<Video> videosList = new ArrayList<Video>();
   ```

4. Loop the JSONArray creating Video objects and introducing them in the Videos ArrayList.

   ```
   for (int i = 0; i<jsonArray.length(); i++) {

      JSONObjectjsonaux =
          jsonArray.getJSONObject(i).getJSONObject("video");
      String title = jsonaux.getString("title");
      String url = null;

      try {
         url = jsonaux.getJSONObject("player").getString("mobile");
      }  catch (JSONException ignore) {
         url = jsonaux.getJSONObject("player").getString("default");
        }

        String thumbUrl =
            jsonaux.getJSONObject("thumbnail").getString("sqDefault");
     videosList.add(new Video(title, url, thumbUrl));
    }
   ```

5. Creation and filling of a bundle to send it to the Youtube class.

   ```
   Bundle data = newBundle();
   data.putSerializable("VideosList", videosList);
   Message msg = Message.obtain();
   msg.setData(data);
   replyTo.sendMessage(msg);
   ```

Back to the Youtube class, there exists a handler with the aim to obtain the message sent by the GetYoutubeUserVideosTask class and to call to the method *fillListView* with the message as parameter. This method set an *adapter* for the *listView*, which is in charge of populate the *listView* inserting every video in an entry of the list.

```
videosListView.setAdapter(newAdapterList(context, videoEntryViewID,
    listVideos) {

   @Override
   publicvoidonEntry(Object entry, View view) {

      if (entry != null) {
         TextViewsuperiorText = (TextView)
             view.findViewById(R.id.textView_superior);

         if (superiorText != null)
            superiorText.setText(((Video) entry).getTitle());

            TextViewinferiorText = (TextView)
                view.findViewById(R.id.textView_inferior);

            if (inferiorText != null)
            inferiorText.setText(((Video) entry).getUrl());

            ImageView thumb = (ImageView)
                view.findViewById(R.id.imageView_imagen);
          String aux = ((Video) entry).getThumbUrl();
         ImageDownloader downloader = newImageDownloader();
         downloader.download(aux, thumb);
      }
}
```

ImageDownloader is a class used to download asynchronously the thumbnail image of the videos.

At this stage, the *listview* is correctly filled with every video. An *onClickListener* is necessary, this way when an user application will click in a video, this will be reproduced in the same way as for the single video case.

Audio

The Audio Guideline module has been developed using the Media Player Android

API. Only some functions has been encapsulated in the module, since the goal of this module is to provide the basic functionality. The API user can find about the entire functionality that this API provides in the Android Developer webpage (9)

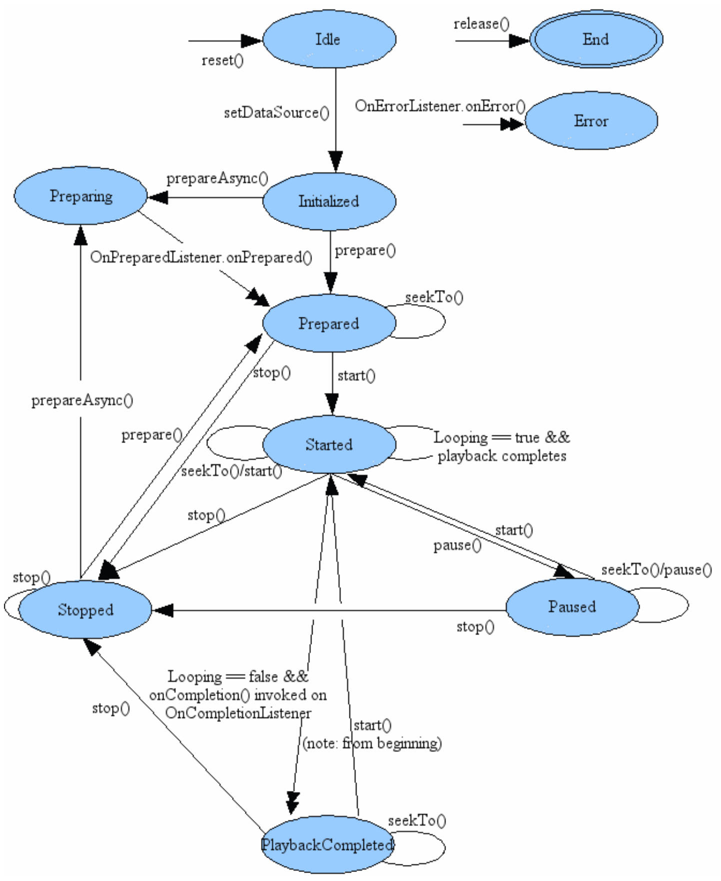


Figure 3.19: Media Player Android API State Diagram. Figure from (9)

To use MediaPlayer functions in a proper way, it is important to keep in mind its state diagram. For the encapsulated functions, this is not important since there has been extracted the complexity that the state diagram could bring.

Video

All the functionality provided is done through the VideoView class. The functions encapsulated are only the basic ones to be able to display a video. The API user can use the entire functionality of the VideoView class which can be found in the Android Developer Webpage (75).

3.3.4.2 How to use it

System Manager is a singleton class. Thus, to obtain the unique instance that could exists in an application it is necessary to use the *getInstance* function.

```
SystemManager sm = SystemManager.getInstance();
```

Services and Setup

Services and Setup modules are quite simple to use. To perform a mobile call or to send a text message it could be done like follows:

```
sm.call(number, activity);
sm.sendSMS(number, message);
```

Where *number* parameter is the mobile number to call or send a message and *message* the text message to be sent.

There are three kind of notifications which can be used as follow:

```
sm.sendSimpleNotification(title, text, notificationID, icon, context);

PendingIntent contentIntent =
    PendingIntent.getActivity(getApplicationContext(), 0, Intent, 0);
int flags = Notification.FLAG_AUTO_CANCEL;
sm.sendComplexNotification(title, text, notificationId, icon, flags,
    contentIntent, context);

Uri soundUri =
    RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
sm.sendComplexNotificationCustomSound(title, text, notificationId, icon,
    flags, contentIntent, soundUri, context);
```

Where *title* parameter is the notification title, *text* the text to be shown, *notificationId* the notification ID, *icon* an notification icon and *context* the application UI context. For the last two notification type, *contentIntent* is the Intent which is run when the notifications are clicked on and flags parameters are the notifications flags available for user usage (76).

NOTE. The minimum Android SDK Version required is 10th so some flags belonging to newer Android SDK versions may do not work properly.

These same notifications may be scheduled. It is necessary to declare in the Android Manifest the class ScheduledTask as a receiver.

```
<receiver android:name="systemManager.services.ScheduledTask"/>
```

```
sm.scheduledSimpleNotification(title, text, notificationID, icon, context,
    date);
sm.scheduledComplexNotification(title, text, notificationId, icon, flags,
    contentIntent, context, date);
sm.scheduledComplexNotificationCustomSound(title, text, notificationId, icon,
    flags, contentIntent, soundUri, context, date);
```

All parameters play the same role than explained previously except *Date*. It is a Calendar type which indicates when the notification is scheduled. It is also possible to schedule an audio to be reroduced by this:

```
sm.scheduledAudio(path, context, date);
```

Where *path* is the files path, *context* is the application UI context and *date* is a Calendar type indicating when the audio is scheduled to be reproduced.

To turn on/off Wi-Fi or Bluetooth:

```
sm.setWifiEnabled(true/false, context);
sm.setBluetooth(true/false);
```

The first boolean parameter set the Wi-Fi/Bluetooth state and *context* is the application UI context. Finally, the screen brightness could be adjust as follows:

```
sm.setScreenBrightness(brightness, activity)
```

The parameter of *brightness* is a float number between 0 and 1 (from dark to bright). It is also possible to set it to less than 0, which means that the user default screen brightness is used.

Guidelines

The manner of using any of the guidelines modules is slightly different. First, it is necessary to obtain an instance of the class which will be used and once this is done, just use its functions.

YouTube

To build the YouTube module has been necessary the YouTube Android Player API (77). To employ this API is compulsory to provide at least a *YoutubePlayerView*. An example of how to use the YouTube Guideline module is shown next:

```
setContentView(R.layout.youtube_layout);
```

```
YouTubePlayerViewyoutubeView =
    (YouTubePlayerView)findViewById(R.id.youtube_view);
youtube = sm.getYoutubePlayer(getApplicationContext(), youtubeView,
    DEVELOPER_KEY);
```

The developer key is a variable that identifies the YouTube developer submitting an API request. In the case when only playing a YouTube video is wanted, it could be done by:

```
youtube.reproduceSingleVideoMode(url)
```

The *URL* parameter is the YouTube video URL to be reproduced, that consists of the last part of the video link.

It is also possible to reproduce YouTube playlist videos, selecting the video to be reproduced using a *listView* view. In order to do this, it is necessary to have in the current layout a *listView* view.

```
ListViewvideosListView = (ListView)findViewById(R.id.listListView);
youtube.reproducePlaylistMode(videosListView, R.layout.entry, PLAYLIST_URL);
```

The parameter *entry* is a layout with every entry format of the *listview*.

Audio

The Audio module has been developed using the Media Player Android API (9). To utilize this module, the first thing to do is to obtain an instance of the Audio class.

```
Audio audioPlayer = sm.getAudioPlayer();
```

Now, to play a stored audio file in the mobile phone, this must be done:

```
String path = Environment.getExternalStorageDirectory() + "/" + "song.mp3";
audioPlayer.loadFile(path);
audioPlayer.prepare();
audioPlayer.play();
```

More functions are available, such as pause, resume, getDuration, etc. However, every function of the MediaPlayer API may be used.

Video

The Video module is based on the functionality provides by the VideoView view (75). To reproduce a video which is stored in the mobile external card, the following should be done:

```
setContentView(R.layout.video_layout);
VideoViewvideoHolder = (VideoView)findViewById(R.id.videoView);
Video video = sm.getVideoPlayer(videoHolder);
String path = Enviroment.getExternalStorageDirectory() + "/" + "video.mp4";
video.setVideoPath(path);
video.play();
```

More functions are available, such as pause or resume. Every function of the VideoView class may be used.

3.3.5 Data Processing Manager

This manager is in charge of the data processing and knowledge inference, making it is one of the most powerful managers of the framework. The knowledge inference can be done online as well as offline. In order to make the framework flexible, and following the sequence shown in the Figure 3.5, the manager was divided in five modules or packages. The *Acquisition* package is the responsible for the data collection. The *Pre-processing* package has the functions to make the pre-process of the data. The *Segmentation* package is in charge of splitting into segments the data. The *Features Extraction* package is responsible for obtaining the signal features. The *Classifcation* package is in charge of the knowledge inference.

3.3.5.1 Description

In a similar way to the *Communication Manager*, this manager must be able to run in background. Therefore, it is also defined as a *Service*. The manager has, among other things, an instance of the *Communication Manager*, an instance of *Storage* module, and an instance of each module that composed it (*acquisition*, *preprocessing*, *segmentation*, *featuresExtraction* and *classification*), in order to use the functions that they offer. It also has a bunch of data structures where the data acquire/process/calculate will be stored:

- *hash* stores the acquired data and is the following data structure:

  ```
  ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>>
  ```

- *hashProcessed* stores the processed data and is the following data structure:

  ```
  ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>>
  ```

- *indexSegmentation* stores the index of the data segments and is the following data structure (the *ObjectSegmentation* object will be explain later):

  ```
  ArrayList<Pair<String, ArrayList<ObjectSegmentation>>>
  ```

- *featuresVector* stores the extracted features and is the following data structure:

  ```
  ArrayList<Double>
  ```

- *featuresMatrix* stores a matrix of extracted features and is the following data structure:

```
ArrayList<ArrayList<Double>>
```

In order to indicate what kind of data must be used (processed or not processed) for some of the methods, the *DataType* enumeration was created. It has two values: *Processed* and *Raw*. This is translated, in programming terms, in the use of the variable either *hash* or *hashProcessed*.

The manager is mainly composed by methods that define the functionalities offered by its modules (*retrieveByID, retrieveBySession, upSampling, downSampling, windowing_overlap, windowing_NoOverlap, features_extraction, trainClassifier, setClassifier,* etc). All these methods will be explained later in their corresponding module. There are methods with other purposes such as the *getInstance* function, to get the instance of the manager or the *createAndSetStorage* function, which needs the UI's context as parameter, and initialize the *storage* variable needed to use the functions defined in the *Acquisition* module.

As it was mentioned at the section's beginning, it is possible to make the knowledge inference either online or offline. To make offline knowledge inference, the user only needs to call the different methods offered by the manager. To make the online knowledge inference, it was necessary that the Data Processing Manager got the samples sent by the portable biomedical devices. Hence, it was decided to implement the approach used for the *Communication Manager* and the *Visualization Manager*. That is, the samples will be sent by messages. It was also necessary to create a buffer for each device connected, and a synchronization mechanism in order to know when all the buffers are filled, and thus, to begin the knowledge inference. To make this work, a *Handler* and a new object was defined. The new object is named *ObjectProcessing* and is defined as an inner class. It has six variables: *buffer* is an array where the samples received are stored. *windowSize* is the size of the window. In other words, it is the number of samples that will be use in the knowledge inference. *maxSize* is an Int indicating the size of the buffer. *cont* is an Int which represents a counter. *hashOnline* is a Hashtable where the key is a *SensorType* and the object is an array containing the values of a specific sensor. *sensors* is an array with the device's sensors whose values will be used in the knowledge inference. The object has, besides the constructor, two functions. The first one is *setWsize*. It is called in the object's constructor and it sets the *windowSize* variable, multiplying the device's sample rate by the seconds given (the user must indicate the window's size in seconds). The method responsible for starting the online knowledge inference is *inferenceOnline*. This is the only method that the user needs to call. It receives as parameter all the information needed to make the

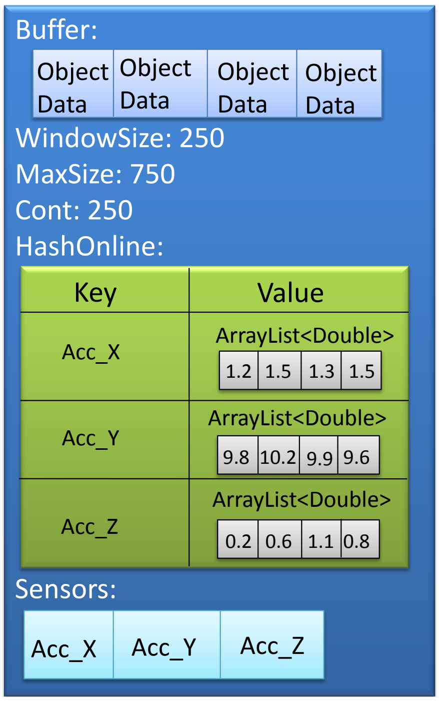


Figure 3.20: Example of an ObjectProcessing filled.

whole process. Its parameters are:

- *windowInSeconds*: It is the window's size measured in seconds.
- *sensorsAndDevices*: It is a list with the devices and its sensors where the data will be acquired to make the knowledge inference. Its structure is the following:

  ```
  ArrayList<Pair<ArrayList<SensorType>, String>>
  ```

- *preprocessingType*: It is the type of pre-processing to be done. It should be Null when is not desired to do the pre-processing.
- *factor*: In case the pre-processing is done, this is the factor to be used. Otherwise this variable is discarded.
- *typeClassAtribute*: It indicates the kind of the variable to be returned when the classification is done. It can be either String or Double.
- *[mHandler]*: This parameter is only for one of the two function versions. It is the handler (which must be defined by the framework's user) where the classified data will be sent through a message.

The function creates an *ObjectProcessing* for each device given in the parameter *sensorsAndDevices*. Then, for each device too, a flag in its *ObjectCommunication*, called *dataProcessing*, is enabled. This produces that the *Communication Manager* sends the samples received to this manager. Finally, the rest of the variables needed to make the knowledge inference are stored in their corresponding class variables. The *Handler* of the manager is responsible for handling the messages received and for making the synchronization mechanism. When a message is received by this manager, the name of the device, that sent the sample, is retrieved. The *ObjectProcessing* linked to that device is obtained, the sample received is added into its buffer, and its counter is incremented one value. When the amount of received samples is equal to the window's size, an array is created for each sensors specified in the *sensors* variable. These arrays contain the values of the samples for a specific sensor. Then, the *hashOnline* Hastable of the *ObjectProcessing* is filled with these arrays, and the class variable called *buffersReady* is incremented one value. When all the buffers are full, all the *hashOnline* Hastable are introduced in the *hash* variable (which has been explained previosly). That is, the data structure used for the knowledge inference is filled. Finally, a *Runnable* object is added to the message's queue through the *post* function. The *Runnable* object, named *runnableOnline*, has the code for making the knowledge inference. It checks whether there is a pre-process to do. If so, the pre-process is done using the corresponding function. Then, the features extraction is done to the data either processed or raw, and the data is classified. In order to stop the online knowledge inference, the function *endInferenceOnline* is developed. It receives as parameter a list of devices names. The method, for each devices in the list disables the *endInferenceOnline* flag of its *ObjecCommunication*, and clears the buffer and the Hastable of its *ObjectProcessing*.

This manager is divided in five packages in order to make its implementation and use easier:

Acquisition

This module is responsible for acquiring data from the local database when offline inference is performed. This module is not used when online inference is carried out, due to the Communication Manager which is the one that provides the data.

There are six different ways of acquiring data: by row IDs, by session, by interval sessions, by X last seconds streamed, by interval dates or just by all the available data. All of them need to use the Storage Module in order to execute queries into the local database to obtain the data. Every function may be used by one or more biomedical devices and they all return the same structure object: *ArrayList<Pair<String,*

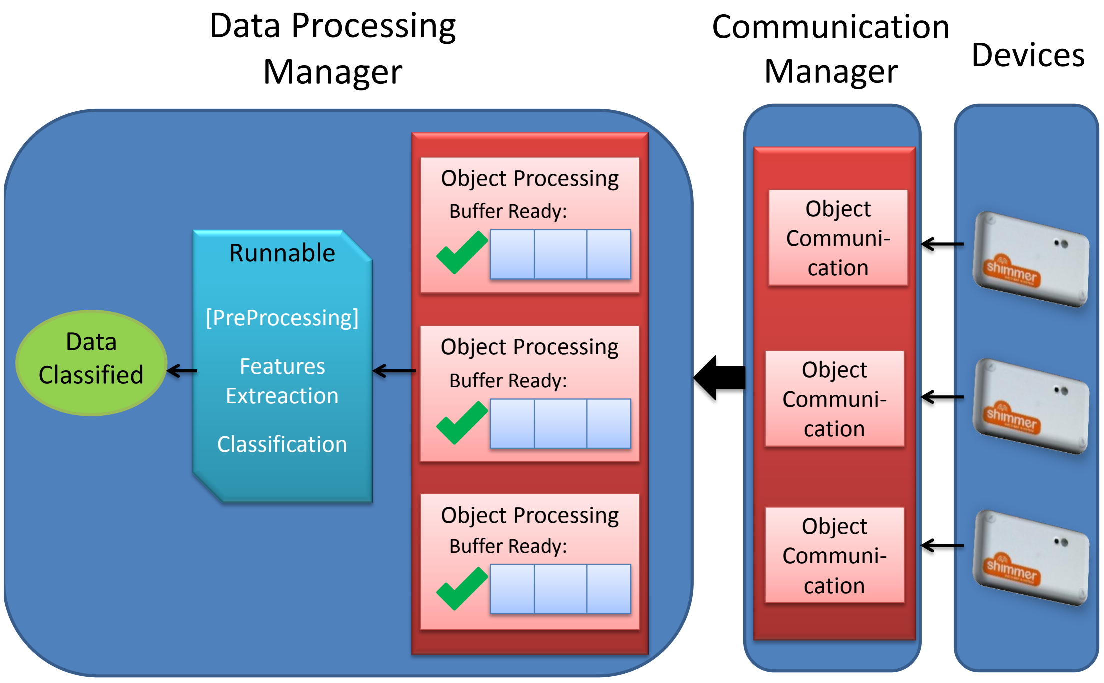


Figure 3.21: Graph showing the process to make the knowledge inference.

Hashtable<SensorType, ArrayList<Double>>>>. Inside this *Arraylist* of *Pairs*, the first element is the device name and the second element corresponds to the data, stored inside a hash with *SensorType* as keys. They also receive a common parameter, which is the structure *ArrayList<Pair<ArrayList<SensorType>, String>>*, representing a list of the devices which the data wants to be extracted from along with the signal sensors to be acquired. An example of how this structure could be filled:

- *Device Shimmer Chest* → Accelerometer X, Accelerometer Y, Accelerometer Z, Gyroscope X, Gyroscope Y, Gyroscope Z, ECG Right, ECG Left, Timestamp.

- *Device Shimmer Arm* → Accelerometer X, Accelerometer Y, Accelerometer Z, Magnetometer X, Magnetometer Y, Magnetometer Z, Timestamp.

- *Device Portable Mobile* → Accelerometer X, Accelerometer Y, Accelerometer Z, Humidity, Temperature, Light, Timestamp.

A detailed explanation of the developed functions follow:

- *retrieveInformationByID*: it retrieves data by giving two IDs of the Signal Table, so all the data (signals) inside that range are acquired. It receives as parameter starting and ending IDs and the *sensorAndDevices* structure (devices and sensors to be acquired). The function code looks like this:

```
public ArrayList<Pair<String, Hashtable<SensorType,
    ArrayList<Double>>>> retrieveInformationByID(
      int start, int end, ArrayList<Pair<ArrayList<SensorType>,
          String>> sensorsAndDevices) {

   ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>>
       data = new ArrayList<Pair<String, Hashtable<SensorType,
       ArrayList<Double>>>>();

   if ((start != 0) && (start < end)) {
      storage.open();
      for (int i = 0; i < sensorsAndDevices.size(); i++) {
         Hashtable<SensorType, ArrayList<Double>> hashAux;
         ArrayList<SensorType> sensors =
             sensorsAndDevices.get(i).first;
         String nameDevice = sensorsAndDevices.get(i).second;
         String nameTable = cm.getDevice(nameDevice).getTableName();
         hashAux = storage.retrieveInformationByID(sensors, nameTable,
               start, end);
         Pair<String, Hashtable<SensorType, ArrayList<Double>>>
             pairAux = new Pair<String, Hashtable<SensorType,
             ArrayList<Double>>>(
               nameDevice, hashAux);
         data.add(pairAux);
      }
      if (!cm.isStoring())
         storage.close();
   }
   return data;
}
```

After checking that IDs are correct and opening the database, it is proceeded to acquire the data for each device (looping sensorsAndDevices structure). Thus, it is necessary to extract from this structure each device name and its sensors to be acquired, plus the Signals table name (necessary for the SQL query). After doing this, the function *retrieveInformationByID* defined in the Storage Module is used to execute the corresponding queries into the database for the data acquisition of a device. This functions builds an SQL query and executes it. Then, using a Cursor it inserts all the data in a hash which is returned to the *retrieveInformationByID* function defined in Acquisition module. This is done as follows:

```
public Hashtable<SensorType, ArrayList<Double>>
    retrieveInformationByID(ArrayList<SensorType> sensors,
         String nameTable, int start, int end) {

      Hashtable<SensorType, ArrayList<Double>> hash = new
          Hashtable<SensorType, ArrayList<Double>>();
      int numKeys = sensors.size();
      List<List<Double>> arrays = new ArrayList<List<Double>>();
      for (int i = 0; i < numKeys; i++) {
         ArrayList<Double> aux = new ArrayList<Double>();
         arrays.add(aux);
      }

      String columns = getColumnsName(sensors);
      String sql = "select " + columns + " from " + nameTable + " where
          " + ID + " BETWEEN " + start + " AND " + end + ";";
      Cursor c = db.rawQuery(sql, null);
      c.moveToFirst();

      while (!c.isAfterLast()) {
         for (int i = 0; i < numKeys; i++) {
            if (c.isNull(i))
               arrays.get(i).add(Double.NaN);
            else
               arrays.get(i).add(c.getDouble(i));
         }
         c.moveToNext();
      }

      for (int i = 0; i < sensors.size(); i++)
         hash.put(sensors.get(i), (ArrayList<Double>) arrays.get(i));

      return hash;
   }
```

The aim of the function *getColumnsName* is to build a string with all the sensors names to be acquired using its database tables names. As it is possible that sometimes there exists a null value for some ID (table rows), it is important to check if this happens and if it does, a NaN value is inserted.

Once this process has been done for every device, the obtained *Hash* is returned to the Data Processing Manager.

- *retrieveInformationBySession*: it retrieves data streamed in a particular session. The method needs as parameter the session number and the *sensorAndDevices* structure. The process is similar to the one seen for *retrieveInformationByID*, in fact the function *retrieveInformationByID* of the Storage Module is used as well. The existing difference is that before calling to this function it is necessary to obtain the starting and finishing ID of the particular session to be acquired. This information is stored in the Metadata table with the same ID as the session to retrieve. Thus, a Storage Module method called *getSessionIDs* is used, which executes a simple query to obtain the starting and finishing IDs and returns them using a *Pair* structure. Now that these IDs are obtained, the rest of the process is the same for the *retrieveInformationByID*.

- *retrieveInformationByIntervalSessions*: it retrieves the data belonging to consecutive sessions. Its parameter are starting and finishing sessions IDs and *sensorAndDevices* structure. The process is really similar to the one for a particular session. This time, the function *getSessionIDs* is called twice. The first call is to obtain the starting ID of the starting session, and the second call to obtain the finishing ID of the finishing session. These two parameters are used to call to the function *retrieveInformationByID*.

- *retrieveInformationByDates*: it retrieves the streamed data inside of a range of two dates. Its parameters are the starting time, ending time and *sensorAndDevices* structure. For every device, once the Signal Table name has been obtained, the function *retrieveInformationByDates* defined in the Storage Module is called to acquire the data. This method converts both dates to its Epoch Time format [REF](using Android function *toMillies* defined in the Time class),which is the same format of the database signals timestamp. Then, the SQL query is built (using the WHERE clause with both dates) and executed.

- *retrieveInformationLastSeconds*: it retrieves the last data streamed according to a number of seconds. Its parameters are *sensorsAndDevices* and seconds, which is an integer number meaning the number of seconds to be acquired. A Storage

Module method called *retrieveInformationLastSeconds* is used. It converts the number of seconds to be acquired to milliseconds and executes a SQL query to obtain the last ID of the Signal table and its timestamp value. This is the upper timestamp value limit and the lower is obtained subtracting to this the number of seconds (in milliseconds format).

```
int maxId = getMaxIndex(nameTable);
String sql = "select " + TIME_STAMP + " from " + nameTable + " where
    " + ID + "=" + maxId + ";";
Cursor c = db.rawQuery(sql, null);
c.moveToFirst();
long finishingInterval = c.getLong(0);
long secondsMs = seconds * 1000;
long startingInterval = finishingInterval - secondsMs;
if (startingInterval < 0) {
   startingInterval = 0;
}
```

Opposite to the previously seen functions, there is one more thing to deal with. For the segmentation stage in the inference knowledge process, it is important to use windows of the same size. Because of this, the function must always return the same number of signals (sometimes biomedical devices may send less or more signals than they should according to its rate). Thus, if the number of rows acquired is marginally lower or just higher than the esteemed rows, it is necessary to fill with NaN values or delete some rows. By this process, all windows will hold the same size.

```
//Number of esteemed rows
double esteemedRows = cm.getDevice(nameDevice).getRate() * seconds;
double percentage = 5;
double differenceAccepted = (esteemedRows / 100) * percentage;
int obtainedRows = hashAux.get(sensors.get(0)).size();
// Less rows, therefore a fill up of NaN values is done in every
    arraylist inside the hash
if ((obtainedRows < esteemedRows) && (obtainedRows > esteemedRows -
    differenceAccepted)) {
   for (int j = obtainedRows; j < esteemedRows; j++) {
      for (int h = 0; h < sensors.size(); h++)
         hashAux.get(sensors.get(h)).add(Double.NaN);
   }
```

```
    }
    // More rows, therefore is necessary to delete rows
    if (obtainedRows > esteemedRows) {
       for (int j = obtainedRows - 1; j >= esteemedRows; j--) {
          for (int h = 0; h < sensors.size(); h++)
             hashAux.get(sensors.get(h)).remove(j);
       }
    }
```

- *retrieveAllInformation*: it retrieves all the information stored in the database. Its unique parameter is *sensorsAndDevices* structure. For the data acquisition, the method *retrieveAllInformation* defined in the Storage Module is used. This builds and executes a SQL query and returns the data.

Pre-Processing

This module is responsible for the signal pre-processing. The implemented methods are Upsampling and Downsampling.
Upsampling is the process of increasing the sampling rate of a signal. The Upsampling factor is usually an integer or a rational fraction greater than unity. This factor multiplies the sampling rate. *Downsampling* is the process of reducing the sampling rate of a signal. The Downsampling factor is usually an integer or a rational fraction greater than unity. This factor divides the sampling rate.

The module is composed only by the *preProcessing* class. It has an enumeration called *PreProcessingType*. It was created in order to choose the pre-process method desired. The enumeration has two values since there are two different methods: *Upsampling*, and *Downsampling*. The class has four methods, two *upSampling* and two *downSampling*.

- The first *upSampling* function is designed to be used only for one portable health device. It must receive as parameter a data structure like this:

  ```
  Hashtable<SensorType, ArrayList<Double>>
  ```

  In this Hashtable, the key is a *SensorType*, and the object is an array with the values of this sensor. The other parameter is the Upsampling factor. The Upsampling is calculated using the following loop:

  ```
  for (int i = 0; i < data.size() - 1; i++) {
     newData.add(data.get(i));
  ```

```
    double n = (data.get(i+1) - data.get(i)) / factor;
    for (int f = 1; f < factor; f++)
        newData.add(data.get(i)+f*n);
}
newData.add(data.get(data.size() - 1));
hash.put(s, newData);
```

In the fragment of code, the *data* variable represents an array of the Hashtable given as parameter. The *newData* variable is the array which contains the Upsampling. The *hash* variable is a new Hashtable containing the Upsampling arrays calculated. This Hashtable is the data returned by the function. The other *upSampling* function is designed to be used for multiple portable health devices. It must receive as parameter a data structure like this:

```
ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>>
```

This data structure represents a list of Pair, where the first element is the name of a portable health device, and the second element is a Hashtable. In this Hashtable, the key is a *SensorType*, and the object is an array with the values of that sensor. The other parameter is the Upsampling factor. This function calculates the Upsampling of the list of devices, one by one, using the function described previously. Finally, a data structure with the Upsampling of all the devices is returned. This data structure is equivalent to the data structure given as parameter.

- The first *downSampling* function is designed to be used only for one portable health device. The parameters that must be received are the same as for the *upSampling* function (the one designed to be used only for one device). The Downsampling is calculated by taking one value for each Downsampling factor values. That is, if the Downsampling factor is 2, then 1 value for each 2 is taken. If the Downsampling factor is 3, then 1 value for each 3 is taken, and so on. The function returns a Hashtable with the Downsampling arrays. The Hashtable's structure is equal to the Hashtable given as parameter. The other *downSampling* function is designed to be used for multiple portable health devices. The parameters that must be received are the same as for the *upSampling* function (the one designed to be used for multiples devices). This function calculates the Downsampling of the list of devices one by one, using the function described previously. Finally, a data structure with the Downsampling of all the devices is returned. This data structure is equal to the data structure given as parameter.

Segmentation

This module is in charge of the signals segmentation. The implemented methods are the Windowing, and the Windowing with overlap. *Windowing* is the process of segmenting a signal. That is, to split the signal. These pieces are called segments or windows. The size of these windows can be set in different units of measures (seconds, number of samples, etc). The Windowing can be done either by overlapping other windows or without overlapping. Also, a window can be complete or incomplete. It is completed when its size is the size set for the window. It is incomplete when its size is smaller than the size set for the window.

The module is composed only by the *Segmentation* class. Since the signals are always stored in arrays, it was decided to make the Windowing by storing the windows' indexes instead of splitting the arrays. In order to make this task easier, the *ObjectSegmentation* was created. This object has three variables: *start*, *finish*, and *complete*. *start* is an Int which stores the index where the window begins. *finish* is an Int which stores the index where the window finishes. *complete* is a Boolean which defines whether the window is either complete or incomplete. The class has four functions, two *windowing_noOverlap*, and two *windowing_Overlap*.

- The first *windowing_noOverlap* function is designed to segment the data of only one portable health device. It must receive as parameter a data structure like this:

  ```
  Hashtable<SensorType, ArrayList<Double>>
  ```

  In this Hashtable, *SensorType* is the key, and the object is an array with the values of that sensor. The other two parameters are the window's size (measure in seconds), and the device's sample rate (measure in Hz). To make the Windowing, the function calculates the window's size, measure in number of samples, and stores in the *windowInSamples* variable. Then, the number of windows that are contained in the acquired data (the signal) is calculated. The windows' indexes are calculated as follow:
  start = i * *windowInSamples*
  finish = i * *windowInSamples* + (*windowInSamples* - 1)
  Finally, the function returns an array containing the windows' indexes. The other *windowing_noOverlap* function is designed to segment the data of multiples portable health devices. It receives as parameter the following data structure:

```
ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>>
```

This data structure represents a list of Pair, where the first element is the name of a portable health device, and the second element is a Hashtable. In this Hashtable, a *SensorType* is the key, and the object is an array with the values of that sensor. The other parameters are the window's size (measure in seconds), and an array with the devices sample rates. This function calculates the Windowing without overlap for a list of devices, one by one, using the *windowing_noOverlap* function designed for one device. Finally, an ArrayList with a list of the windows' indexes for each device is returned.

- The first *windowing_Overlap* function is designed to make the overlap segmentation for the data of only one portable health device. It receives as parameter a data structure like this:

  ```
  Hashtable<SensorType, ArrayList<Double>>
  ```

  In this Hashtable, *SensorType* is the key, and the object is an array with the values of that sensor. The other parameters are the window's size (measure in seconds), the device's sample rate (measure in Hz), and the seconds of the previous window to overlap. To make the Windowing with overlap, the function calculates the window's size, measure in number of samples, as well as the the number of samples from the previous window to overlap. The windows' indexes are calculated as follow:

  ```
  int start = 0;
  int finish = windowInSamples - 1;
  while(finish < sensors.get(s).size()) {
     index.add(new ObjectSegmentation(start, finish, true));
     start = finish - samplePreviousWindow;
     finish = start + (windowInSamples - 1);
  }
  if(start < sensors.get(s).size() - 1)
     index.add(new ObjectSegmentation(start, sensors.get(s).size() - 1,
         false));
  ```

  The indexes for the first window are set and stored. The following indexes are calculated until the windows end is out of the array. Then, the last windows indexes (which are uncompleted), are stored. Finally, the function returns the array which contains the windows' indexes. The other *windowing_Overlap* function is designed to segment, with overlap, the data of multiples portable health devices. It receives as parameter the following data structure:

```
ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>>
```

This data structure represents a list of Pair, where the first element is the name of a portable health device, and the second element is a Hashtable. In this Hashtable, a *SensorType* is the key, and the object is an array with the values of that sensor. The other parameters are the window's size (measure in seconds), the seconds of the previous window to overlap, and the array with the devices' samples rate. This function calculates the Windowing with overlap for a list of devices, one by one, using the *windowing_Overlap* function designed for one device. Finally, an ArrayList with a list of the windows' indexes for each device is returned.

Features extraction

This is the module in charge of the features extraction. It is only composed by the *FeaturesExtraction* class, but an enumeration named *FeatureType* is also defined in order to choose the feature desired. Their possible values are: *Mean, Variance, Median, Standard_Deviation, Zero_Crossing_Rate, Mean_Crossing_Rate, Maximum and Minimum.* They represent the type of features that the module is able to extract.

The features are extracted from a specific signal of a particular wearable health device. Thus, in order to make easier the specification of this information, the *ObjectFeature* object was created. *ObjectFeature* is composed by three elements: a String with the device's name, a *SensorType* with the type of sensor (or signal), and a *FeatureType* with the type of the feature to extract.

The module has two public functions, both of them called *feature_extraction.* These are the only functions that can be used by framework's users.

- The first *feature_extraction* method is design for no segmented signals. It receives as parameter the following data structure:

  ```
  ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>
  ```

  This data structure represents a list of Pair, where the first element is the name of a portable health device, and the second element is a Hashtable. In this Hashtable, a *SensorType* is the key, and the object is an array with the values of that sensor. The other parameter is an array of *ObjectFeatres* with the features to be extracted. The function retrieves, one by one, all the features from the features array given as parameter, and calculates them calling to its specific function (which are explained later). Finally, an array with all the features extracted is returned. The other *feature_extraction* method is designed for splitting signals. It receives as parameter the following data structure:

```
ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>
```

This data structure represents the same as that in the first *feature_extraction* method. The others parameters are an array of *ObjectFeatres* with the features to be extracted, an array with the windows indexes, and a Boolean which indicates whether the uncompleted windows must be used to calculate the features. The function retrieves, one by one, all the features from the features array given as parameter. Then, the features are extracted from all the signal's windows by calling to its specific function (which are explained later). In case the Boolean given as parameter is True, the features from the uncompleted windows are extracted too. Finally, a matrix with all the features extracted is returned. Every row of the matrix is composed by all the features extracted for a single window.

The private functions, which cannot be used by the framework's users, are the following: two *Mean*, two *Variance*, two *Std*, two *Median*, two *ZeroCrossing*, two *MeanCrossing*, two *Maximum*, two *Minumun*,*getHashByNameDevice*, and *getIndexByNameDevice*.

- The first *Mean* is designed to calculate the mean of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is an array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the mean of the signal. The other *Mean* is designed to calculate the mean of a segmented signal for an specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is an array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the mean of each window.

- The first *Variance* is designed to calculate the variance of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the variance of the signal. The other *Variance* is designed to calculate the mean of a segmented signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the variance of each window.

- The first *Std* is designed to calculate the standard deviation of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the standard deviation of the signal. The other *Std* is designed to calculate the standard deviation of a split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the standard deviation of each window.

- The first *Median* is designed to calculate the median of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the median of the signal. The other *Median* is designed to calculate the median of a split signal for an specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the median of each window.

- The first *ZeroCrossing* is designed to calculate the zero crossing rate of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the zero crossing rate of the signal. The other *ZeroCrossing* is designed to calculate the zero crossing rate of a split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the zero crossing rate of each window.

- The first *MeanCrossing* is designed to calculate the mean crossing rate of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the mean crossing rate of the signal. The other *MeanCrossing* is designed to calculate the mean crossing rate of a split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the mean crossing rate of each window.

- The first *Maximum* is designed to calculate the maximum value of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the maximum value of the signal. The other *Maximum* is designed to calculate the maximum value of a segmented signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the maximum of each window.

- The first *Minimum* is designed to calculate the minimum value of a no split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameter is the type of sensor (or signal) on which the feature extraction is calculated. The function returns the minimum value of the signal. The other *Minimum* is designed to calculate the minimum value of a split signal for a specific portable health device. It receives as parameter a Hashtable which represents a device. In this Hashtable, a *SensorType* is the key, and the object is the array with the values of that sensor. The other parameters are the type of sensor (or signal) on which the feature extraction is calculated, and the windows indexes. The function returns an array with the minimum of each window.

- The *getHashByNameDevice* receives as parameter the following data structure:

  ```
  ArrayList<Pair<String, Hashtable<SensorType, ArrayList<Double>>>
  ```

  This data structure represents a list of Pair, where the first element is the name of a portable health device, and the second element is a Hashtable. In this Hashtable, a *SensorType* is the key, and the object is an array with the values of that sensor. The other parameter is a device's name. The function seeks, in the pair's array given as parameter, the device's name given as parameter too. When the device's name is found, its Hashtable is returned. In case it is not found, the function returns Null.

- The *getIndexByNameDevice* function receives as parameter the windows' indexes of a set of portable health devices. It also receives the device's name. The function seeks the windows' indexes of the device given as parameter and returns them. In case it is not found, the function returns Null.

Classification

It is in charge of the classification stage in the biomedical inference knowledge process. It is composed by the Classification class and has been developed using Weka (78). Weka is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand. It encapsulates the functionality of a stripped version of Weka for Android, offering functionalities like creating a model using different classifiers, training or testing these or classifying instances.

The classifications class defines an enumeration named *ClassifierType*, used to specify a classifier of the followings: Naive Bayes, J48, Adaboost, Zeror or Linear Regression. The defined methods are described, being important to note that most of them are meant to be used one after another.

- *readFile*: it reads a Weka file (arff format) and is loaded into the class variable *atf* of *ArffLoader* Weka type.
- *setTrainInstances*: method to set train instances using a previously loaded arff file. It loads the *trainInstances* class variable of *Instances* Weka type and *attributes*, a list of Weka *Attributes* objects.

```
    public void setTrainInstances() {

       if (atf != null)
          try {
             trainInstances = atf.getDataSet();
          } catch (IOException e) {
          e.printStackTrace();
          }
       for (int i = 0; i < trainInstances.numAttributes(); i++)
       attributes.add(trainInstances.attribute(i));
 }
```

- *setTestInstances*: method to set test instances using a previously loaded arff file. It loads the *testInstances* class variable of *Instances* Weka type.
- *getTrainInstancesSummary*: it returns a summary with train instances information.
- *trainClassifier*: essential method to perform knowledge inference, used to train classifiers. Before using it the class variable *trainInstances* must be loaded and it needs as parameters the index of the class attribute and the type of classifier to be used (the enumeration *ClassifierType*. When the classifier is built, it is loaded into the class variable called *cModel* of Weka *Classifier* type.

```
public void trainClassifier(int classIndex, classifierType
    classifier) {

  trainInstances.setClassIndex(classIndex);
  switch (classifier) {
  case NAIVE_BAYES:
     cModel = new NaiveBayes();
     break;
```

```
        case ADABOOST:
            cModel = new AdaBoostM1();
            break;
        case J48:
            cModel = new J48();
            break;
        case LINEAR_REGRESSION:
            cModel = new LinearRegression();
            break;
        case ZEROR:
            cModel = new ZeroR();
            break;
        }
        try {
            cModel.buildClassifier(trainInstances);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

- *testClassifier*: it tests a loaded classifier (*cModel*). Thus, it loads a class variable of *Evaluation* type called *eTest* using *testInstances* and evaluates the model performance.

```
    public void testClassifier() {

        if (testInstances != null && cModel != null) {
            try {
                eTest = new Evaluation(testInstances);
                eTest.evaluateModel(cModel, testInstances);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

- *getTestSummary*: it returns a summary description of a classifier evaluation. It needs *eTest* to be loaded.

- *getTestConfusionMatrix*: it returns the confusion matrix of a classifier evaluation. It needs eTest to be loaded.

```
public double[][] getTestConfusionMatrix() {

   double[][] confusionMatrix = null;
   if (eTest != null) {
      confusionMatrix = new double[eTest.confusionMatrix().length][];
      confusionMatrix = eTest.confusionMatrix();
   }
   return confusionMatrix;
}
```

- *featureVectorToInstances*: it converts a features vector into an *Instances* object.

```
public Instances featureVectorToInstances(ArrayList<Double>
    featureVector) {

   Instances instances = new Instances("Instances", attributes, 0);
   DenseInstance instance = new DenseInstance(attributes.size());
   for (int i = 0; i < featureVector.size(); i++)
      instance.setValue(i, featureVector.get(i));

   instances.add(instance);
   // Set class attribute
   instances.setClassIndex(attributes.size() - 1);
   return instances;
}
```

- *featureMatrixToInstances*: it converts a features matrix into a *Instances* object.

```
public Instances featureMatrixToInstances(
      ArrayList<ArrayList<Double>> featureMatrix) {

   Instances instances = new Instances("Instances", attributes, 0);
   DenseInstance instance = new DenseInstance(attributes.size());
   for (int i = 0; i < featureMatrix.size(); i++) {
      for (int j = 0; j < featureMatrix.size(); j++) {
         instance.setValue(j, featureMatrix.get(i).get(j));
      }
      instances.add(instance);
      // Set class attribute
      instances.setClassIndex(attributes.size() - 1);
   }
```

```
        return instances;
    }
```

- *classifyInstanceToDouble*: it classify an unlabeled Instance. It returns a double value with the classifying result.

```
    public double classifyInstanceToDouble(Instance unlabeled) {

        double clsLabel = -1;
        try {
            clsLabel = cModel.classifyInstance(unlabeled);
        } catch (Exception e) {
            e.printStackTrace();
        }
        unlabeled.setClassValue(clsLabel);
        return clsLabel;
    }
```

- *classifyInstanceToString*: it classify an unlabeled Instance. It returns a string with the classifying result.
- *loadModel*: it takes a *Classifier* Weka object as parameter and loads a model into the *cModel* class variable.
- *getAttributes*: it returns the class object *attributes*.

3.3.5.2 How to use it

Since it is a singleton class, first it is necessary to get the unique instance.

```
DataProcessingManager dpm = DataProcessingManager.getInstance();
```

It is also necessary to initialize and set (for the acquisition module) the Storage variable belonging to the RemoteStorageManager class called storage. This is done by:

```
dpm.createAndSetStorage(getApplicationContext());
```

Now it proceeds depending on the inference knowledge approach desired: online or offline.

Offline

Acquisition

To show how the data acquisition is done, some devices and sensors where data are retrieved must be selected. For example, it is shown how to acquire the signals accelerometer X, accelerometer Y, accelerometer Z and timestamp of a given wearable device called “Shimmer Chest” and the gyroscope X, gyroscope Y, gyroscope Z, humidity and timestamp of a portable mobile device called “Mobile Device”.

```
ArrayList<Pair<ArrayList<SensorType>, String>> sensorsAndDevices = new
    ArrayList<Pair<ArrayList<SensorType>, String>>();

ArrayList<SensorType> sensors1 = new ArrayList<SensorType>();
ArrayList<SensorType> sensors2 = new ArrayList<SensorType>();
sensors1.add(SensorType.ACCELEROMETER_X);
sensors1.add(SensorType.ACCELEROMETER_Y);
sensors1.add(SensorType.ACCELEROMETER_Z);
sensors1.add(SensorType.TIMESTAMP);
sensors2.add(SensorType.GYROSCOPE_X);
sensors2.add(SensorType.GYROSCOPE_Y);
sensors2.add(SensorType.GYROSCOPE_Z);
sensors2.add(SensorType.HUMIDITY);
sensors2.add(SensorType.TIMESTAMP);

String nameDevice1 = "Shimmer CHEST";
String nameDevice2 = "Mobile Device";
Pair<ArrayList<SensorType>, String> pair1 = new Pair(sensors1, nameDevice1);
Pair<ArrayList<SensorType>, String> pair2 = new Pair(sensors2, nameDevice2);

sensorsAndDevices.add(pair1);
sensorsAndDevices.add(pair2);
```

Now that *sensorsAndDevices* is ready, any Acquisition function may be used. To acquire all the selected data in the IDs range 100-500:

```
dpm.retrieveInformationByID(100, 500, sensorsAndDevices);
```

To retrieve data belonging to the first existing session:

```
dpm.retrieveBySession(1, sensorsAndDevices);
```

To retrieve data belonging to the first, second and third session:

```
 dpm.retrieveInformationByIntervalSessions(1, 3, sensorsAndDevices);
```

To retrieve the last 20 seconds of available data in the local database:

```
dpm.retrieveInformationLastSeconds(20, sensorsAndDevices);
```

Retrieving data using dates is slightly more complicated. An example of retrieving all the data streamed from the day 24th January 2014 at 22:00:00 hour to the day 25th January 2014 at 16:00:00 follows. It is important to keep in mind that month value goes in the interval [0-11]

```
Time start = new Time();
start.hour = 22;
start.minute = 0;
start.second = 0;
start.year = 2014;
start.month = 0;
start.monthDay = 24;

Time end = new Time();
end.hour = 16;
end.minute = 0;
end.second = 0;
end.year = 2014;
end.month = 0;
end.monthDay = 25;

dpm.retrieveInformationByDates(start, end, sensorsAndDevices);
```

To retrieve all the information available:

```
dpm.retrieveAllInformation(sensorsAndDevices);
```

Pre-Processing

From this point, it is assumed that the data has been already acquired and stored in the *hash* variable (which is defined in this manager). Thus, to calculate either the Upsampling or the Downsampling, one just needs to do the following:

```
dpm.downSampling(2);
dpm.upSampling(3);
```

The result will be stored into the *hashProcessed* variable, which is defined in this manager.

Segmentation

To make the segmentation, first it is necessary to get the devices' sample rate. For this example, an array with false sample rates is created:

```
ArrayList<Float> rates = new ArrayList<Float>();
rates.add((float) 50.0);
rates.add((float) 50.0);
rates.add((float) 50.0);
```

Then, the windowing no overlap can be calculated as follow:

```
dpm.windowing_NoOverlap(DataType.Raw, (float) 2, rates);
```

and the windowing overlap as follow:

```
dpm.windowing_overlap(DataType.Raw, (float) 2.5, (float) 0.5, rates);
```

Features Extraction

The features must be defined in order to make the features extraction. These are some features definitions:

```
dpm.addFeature("Device Shimmer 1", SensorType.ACCELEROMETER_X,
    FeatureType.MAXIMUM);
dpm.addFeature("Device Shimmer 2", SensorType.ACCELEROMETER_Y,
    FeatureType.MINIMUM);
dpm.addFeature("Device Mobile", SensorType.ACCELEROMETER_Z,
    FeatureType.VARIANCE);
dpm.addFeature("Device Shimmer 1", SensorType.GYROSCOPE_Y,
    FeatureType.STANDARD_DEVIATION);
dpm.addFeature("Device Shimmer 2", SensorType.MAGNETOMETER_X,
    FeatureType.ZERO_CROSSING_RATE);
dpm.addFeature("Device Mobile", SensorType.MAGNETOMETER_Z,
    FeatureType.MEAN_CROSSING_RATE);
```

Owing to the previous steps, the features extraction can be done from different data:
- From not processed data and not segmented:

```
dpm.feature_extraction(DataType.Raw, false, false);
```

- From processed data and not segmented:

```
dpm.feature_extraction(DataType.Processed, false, false);
```

- From not processed data and segmented:

```
dpm.feature_extraction(DataType.Raw, true, false);
```

- From processed and segmented data, using the uncompleted windows:

```
dpm.feature_extraction(DataType.Processed, true, true);
```

Classification

The way to use the Classification module does not change depending on the selected approach (offline or online). The first thing to do is to read a Weka file (arff format).

```
dpm.readFile(Environment.getExternalStorageDirectory(), "example.arff");
```

Now, test or training instances may be set. Also a summary of both train instances may be obtained.

```
dpm.setTrainInstances();
dpm.setTestInstances();
String TrainSummary = dpm.getTrainInstancesSummary();
```

A model may be built with the *trainClassifier* function. The first parameter is the class attribute and usually is the last attribute. The second parameter is the classifier type to be used, in this case J48 (Decision Tree). The *getAttributes* function returns a list of the existing Attributes.

```
int numAttributes = classification.getAttributes().size();
dpm.trainClassifier(numAttributes - 1, classifierType.J48);
```

The model may be directly loaded as well using the loadModel method:

```
dpm.loadModel(classifier);
```

Now that the model has been built, this can be evaluated by the method *testClassifier*. A summary about the evaluation may be obtained by *getTestSummary* method. The confusion matrix is obtained using the *getTestConfusionMatrix* function.

```
dpm.testClassifier();
String summaryEvaluation = dpm.getTestSummary();
String confusionMatrix = dpm.getConfusionMatrix();
```

To convert features vector or features matrix coming from the Feature Extracture stage into Instances Weka objects, the methods *featureVectorToInstances* and *featureMatrixToInstances* are available.

```
Instances instances = dpm.featureVectorToInstances(featureVector);
Instances instances = dpm.featureMatrixToInstances(featureMatrix);
```

To clasify and unlabeled instance, there are two functions available: *classifyInstanceToDouble* and *classifyInstanceToString*. For example, to classify the first instance of an Instances object called instances:

```
String label = dpm.classifyInstanceToString(instances.firstInstance());
Double label = dpm.classifyInstanceToDouble(instances.firstInstance());
```

Online

Acquisition and ***Segmentation***

These modules are not used in the online inference knowledge process, due to that the Communication Manager is the one which provides the data through windows with a determinate size.

Pre-Processing, ***Feature Extraction***, and ***Classification***

To make the online knowledge inference, it is necessary to define the features to be extracted. The classification model and the class attribute must be defined as well. The previous section, Offline knowledge inference, shows how to do it.

In order to execute the knowledge inference as a sequence, all the steps are defined under one function:

```
dpm.inferenceOnline(2, sensorsAndDevices, null, 0, 1);
```

The last parameter indicates which kind the class attribute is (and in consequence the value returned): String or Double. In case the last parameter is 0, the returned value will be a String and will be stored in the *stringClassified* variable (which is defined in this manager). In case the last parameter is 1 (like in the example), the returned value will be Double and will be stored in the *doubleClassified* variable (which is also defined in this manager).

There is also another function for the online knowledge inference:

```
dpm.inferenceOnline(2, sensorsAndDevices, null, 0, 1, mHandler);
```

4

APP

An app that demonstrates the usefulness and potential of the eHealthDroid framework is here implemented and presented. The app allows to collect data from Shimmer devices in order to obtain physiological and kinematic data, although existing sensors in the mobile device may be also used for monitoring. It offers functionality to visualize any of the received data streams from the portable biomedical devices, as well as uploading the collected data to a remote storage. It is also possible to recognize the physical activity that an app user is performing thanks to an expert system built as part of the app. To help the user to keep healthy habits, guidelines are supported through online video broadcasting and schedulable notifications.

4.1 App usage

It is possible to navigate around the app making use of the available tabs in the top of the screen. Each tab offers a functionality for the following: Connectivity, Visualization, Activity Recognition, Remote Storage, Notifications Manager and YouTube Guidelines.

4.1.1 Connectivity

This tab offers to the user all the connectivity features and the devices configuration. The tab has a Button to add the new devices, and a ListView to visualize them.

In order to add a new device, the "Plus" Button must be pressed. Then, a message will be thrown asking for the device name. The message also have two buttons in order to select the kind of device. It can be either *Mobile* or *Shimmer*.

Once the device is added, it will be displayed in the ListView. It is showed three fields for each device: The first one is the device's name, the second one is the device's

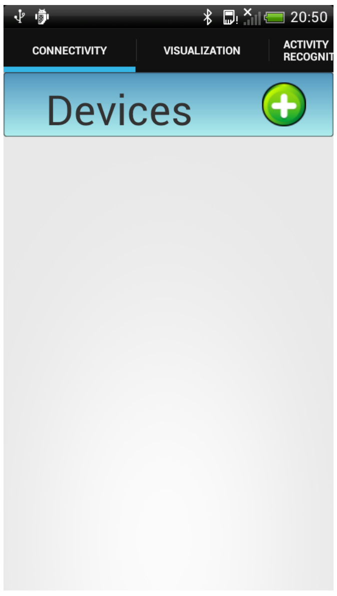


Figure 4.1: Connectivity tab.

type, and the third one is its state. The state is represented by a coloured circle. It will be red when the device is disconnected, orange when the device is streaming, or green when the device is connected but not streaming.

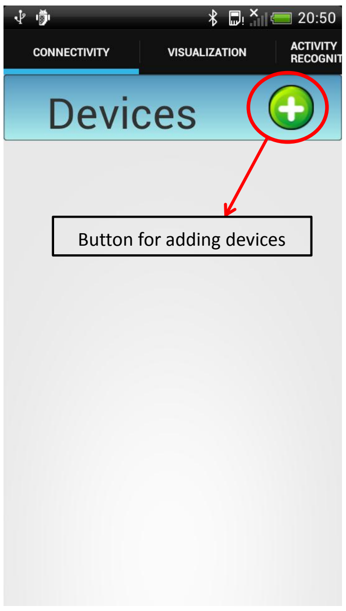
When a device is pressed, a menu with several options is displayed. These options will depend on the device's type and its status. The possible options are the following ones:

- Connect/Disconnect. With this option, the device is either connected (if it was disconnected) or disconnected (if it was connected). Since the mobile device does not need to be connected or disconnected, this function is only available for Shimmer devices.

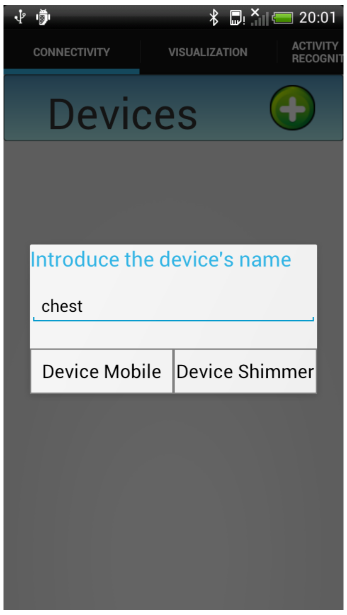
- Start/Stop streaming. With this option, the device either starts or stops streaming.

- Sensors. This option throws a window to set the enabled sensors. The sensors available can be different for each kind of device. It is only showed when the device is not streaming.

- Configuration. This option opens a window to set the device configuration. Here, it is also set whether the data must be stored into the database. The device configuration can be different for each kind of device. It is only showed when the



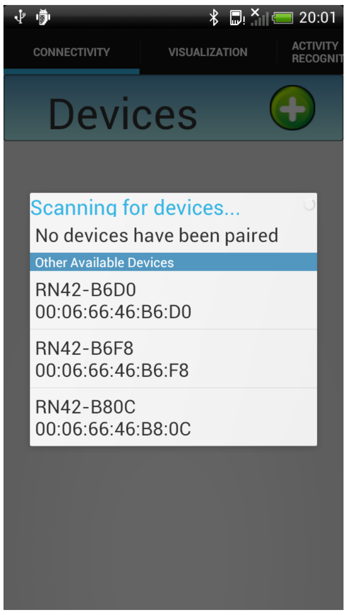
(a) Button "Plus"



Introduce the device's name

chest

(b) Selection of the device's name and its type



Scanning for devices...

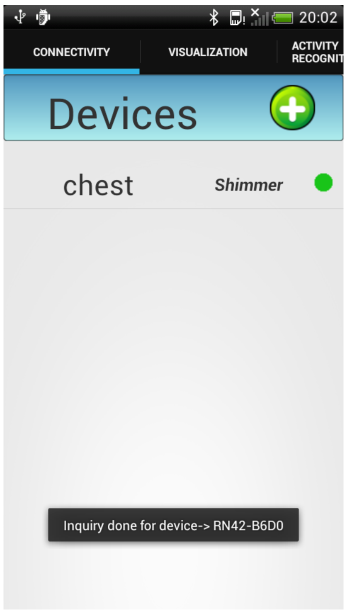
No devices have been paired

RN42-B6D0
00:06:66:46:B6:D0

RN42-B6F8
00:06:66:46:B6:F8

RN42-B80C
00:06:66:46:B8:0C

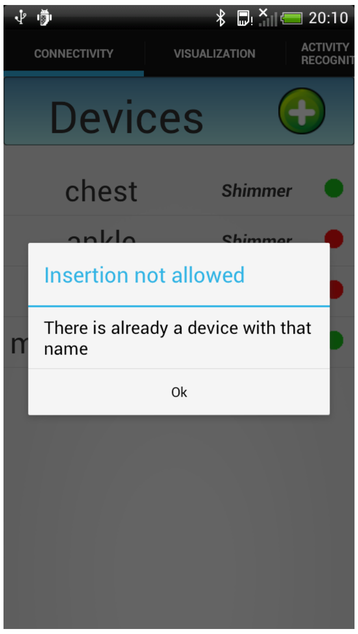
(c) Bluetooth scanning for portable health devices



Inquiry done for device-> RN42-B6D0

(d) Device added and connected

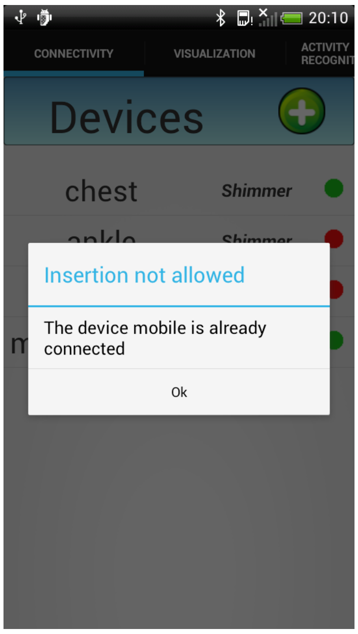
Figure 4.2: Steps to add a device.



Insertion not allowed

There is already a device with that name

(a) Identical name

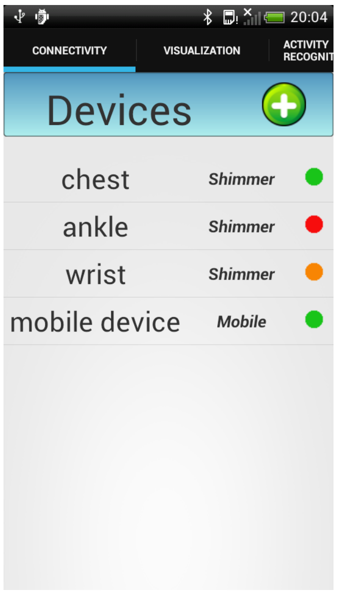


Insertion not allowed

The device mobile is already connected

(b) Multiple mobile devices

Figure 4.3: Error messages during insertion.

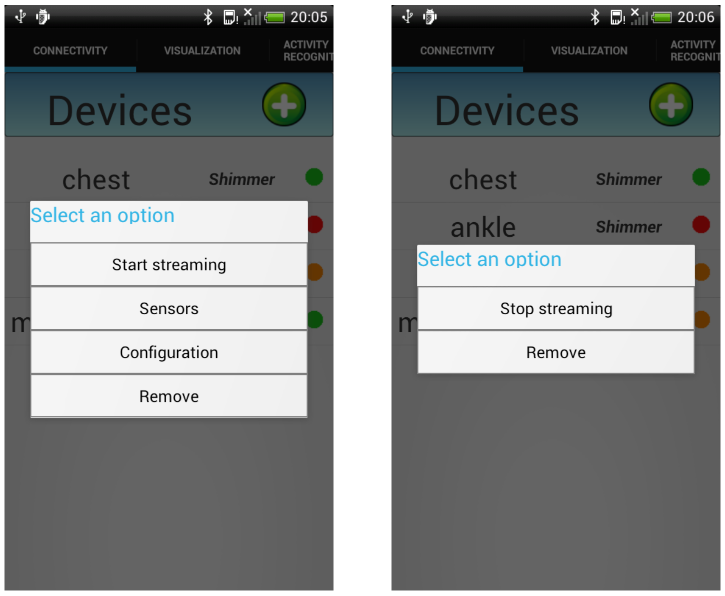


chest	Shimmer	●
ankle	Shimmer	●
wrist	Shimmer	●
mobile device	Mobile	●

Figure 4.4: Different type of devices in different status. Green: Connected. Orange: Streaming. Red: Disconnected

device is not streaming.

- Remove. This option removes a device from the devices list.



(a) Mobile device not streaming (b) Mobile device streaming

Figure 4.5: Menus for the mobile device.

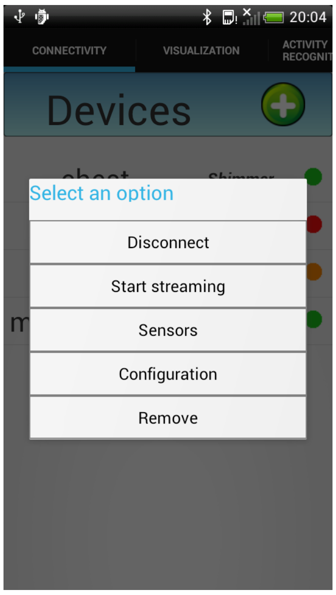
4.1.2 Visualization

This tab permits to visualize the signals recorded through the sensors. The tab is composed by a graph and two buttons: the *Configuration* button and the *Start* button.

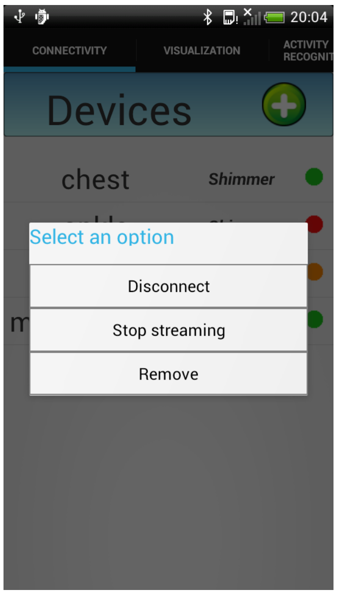
When the *Configuration* button is pressed, a menu is open in order to set the graph configurations. The options are initially disabled (the graph has a configuration by default). These have to be enabled in order to use them. The configuration menu is not available during the visualization, so the graph must be set before starting. The configuration options are the following ones:

- View Port. This option sets the graph view port. That is, the number of samples to be shown in the graph.

- Y coordinates. This option sets the maximum and minimum values for the Y axis.

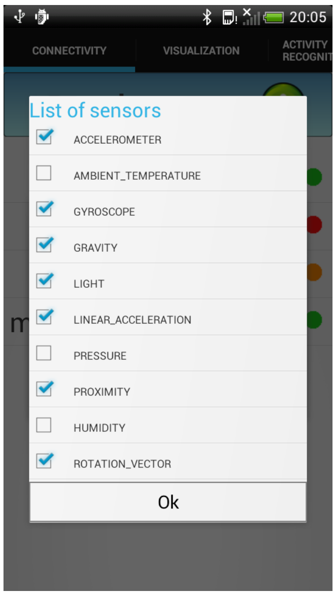


(a) Shimmer device not streaming



(b) Shimmer device streaming

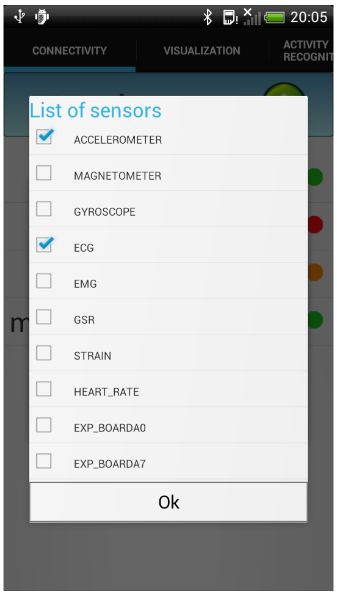
Figure 4.6: Menus for the Shimmer device.



List of sensors

- ✓ ACCELEROMETER
- ☐ AMBIENT_TEMPERATURE
- ✓ GYROSCOPE
- ✓ GRAVITY
- ✓ LIGHT
- ✓ LINEAR_ACCELERATION
- ☐ PRESSURE
- ✓ PROXIMITY
- ☐ HUMIDITY
- ✓ ROTATION_VECTOR

(a) Mobile sensors

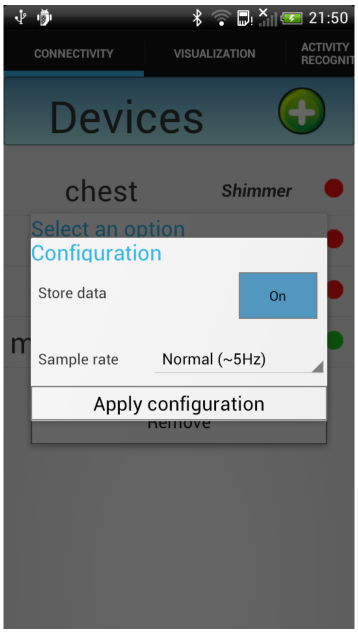


List of sensors

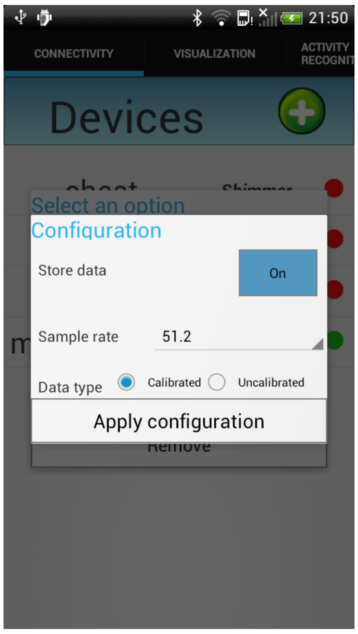
- ✓ ACCELEROMETER
- ☐ MAGNETOMETER
- ☐ GYROSCOPE
- ✓ ECG
- ☐ EMG
- ☐ GSR
- ☐ STRAIN
- ☐ HEART_RATE
- ☐ EXP_BOARDA0
- ☐ EXP_BOARDA7

(b) Shimmer sensors

Figure 4.7: Window to enable the different sensor devices



(a) Mobile device configuration



(b) Shimmer device configuration

Figure 4.8: Window to configure the different types of devices

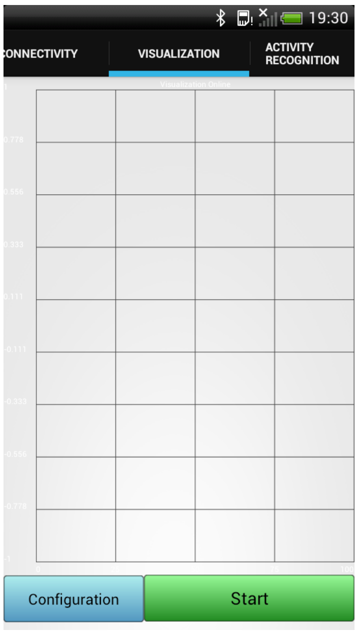
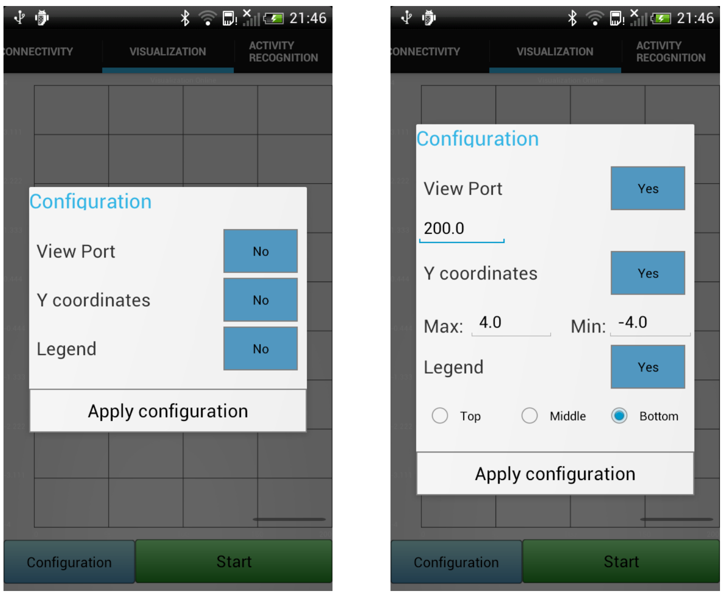


Figure 4.9: Visualization tab.

- Legend. This option shows the legend of the graph series. It can be aligned in three different positions. On the top of the graph, in the middle, or at the bottom.



Configuration
View Port No
Y coordinates No
Legend No

Configuration
View Port Yes
200.0
Y coordinates Yes
Max: 4.0 Min: -4.0
Legend Yes
Top Middle [✓ Bottom]

(a) No options selected (b) All options selected

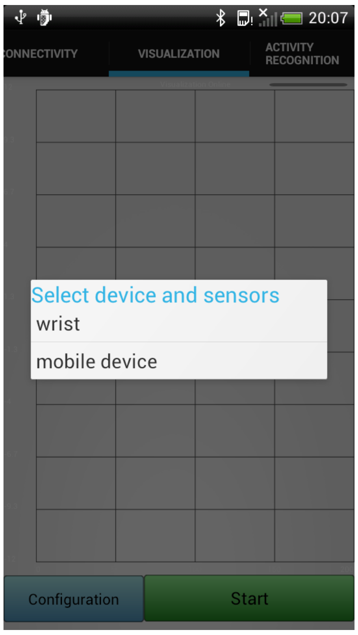
Figure 4.10: Menu of the visualization's configuration

When the Start button is pressed, a list of devices streaming is showed in order to select the device to be visualized. In case there are no devices streaming, an error message will be showed. When a device is selected, it is displayed a list with the device enabled sensors. When the device sensors to visualize are selected, and the OK button is pressed, the visualization starts.

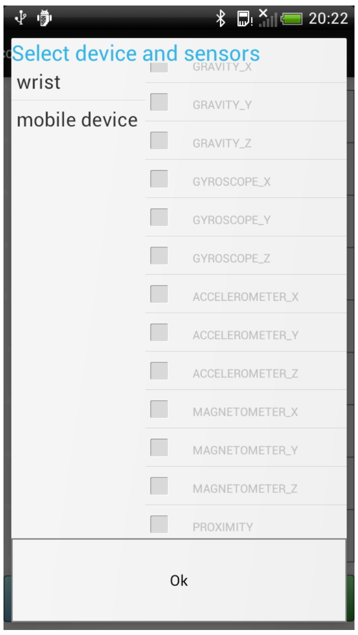
4.1.3 Activity Recognition

This tab uses inference knowledge module to perform activity recognition of a set of activities described in section 4.2. In the section 4.3 it is described how was built the model used to recognize activities.

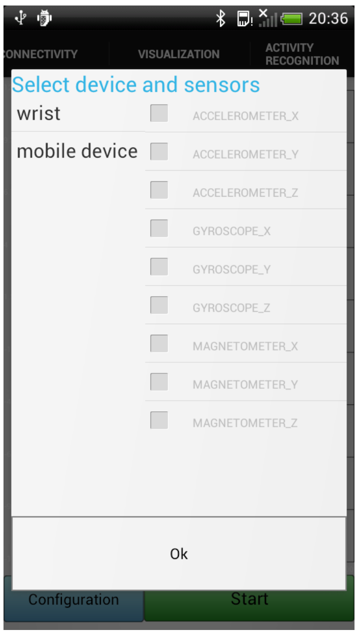
To start the activity recognition, it is necessary to set each connected device to its position in the body (chest, right wrist or left ankle). For this purpose there are three spinners on the screen, one for each position. Once this is done, the button start is used to begin the activity recognition. On the top of the screen there will appear an image and text representing the activity which the app user is performing. In case there are



(a) List of the streaming devices

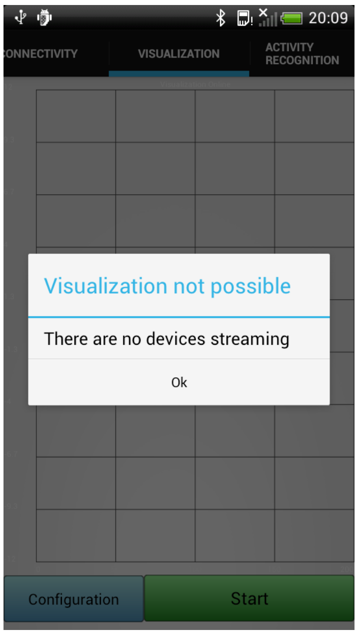


(b) Streaming sensors of the “mobile device”



(c) Streaming sensors of the “wrist” (Shimmer device)

Figure 4.11: Devices and its sensors available for the visualization.



Visualization not possible

There are no devices streaming

Ok

Figure 4.12: Error message when the visualization is not possible

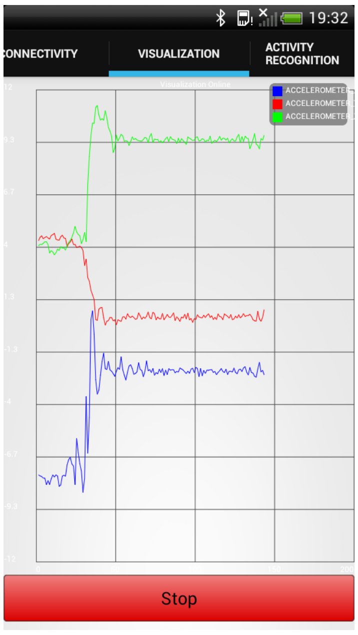


Figure 4.13: Visualization performance of three different sensors.

no devices streaming, an error message will be showed.

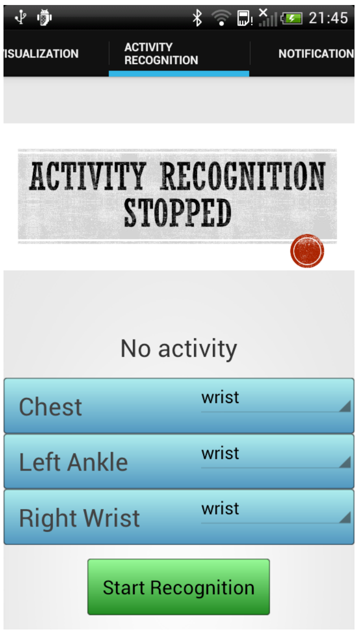
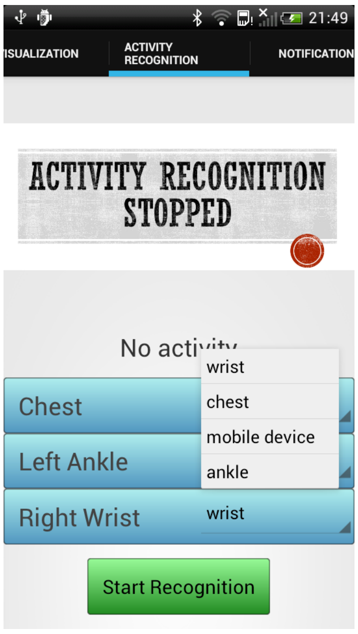
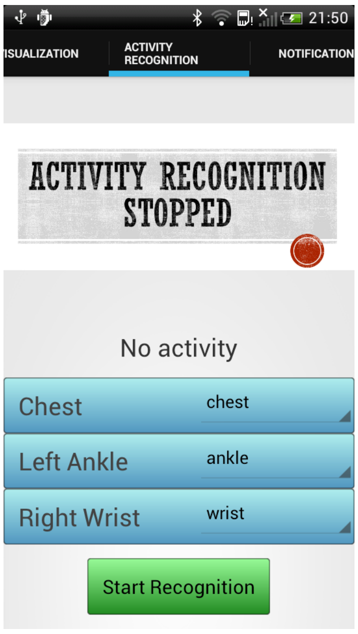


Figure 4.14: Activity recognition tab.



(a) Selecting the appropriate portable health devices



(b) Activity recognition ready to start

Figure 4.15: Setting up the activity recognition before to start.

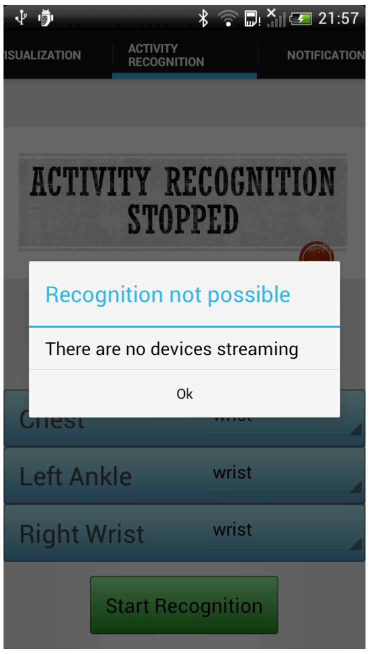


Figure 4.16: Error message when the activity recognition is not possible

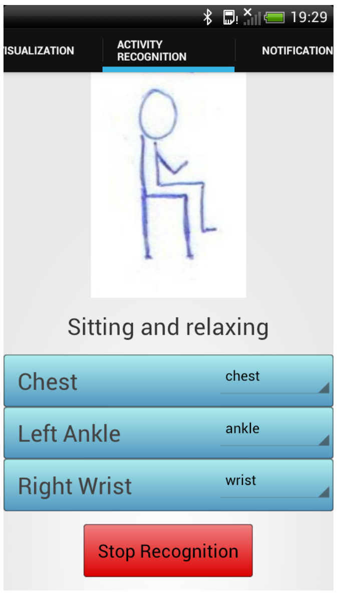


Figure 4.17: Activity recognition performance. In the figure, the Sitting activity is recognized.

4.1.4 Notifications

This tab permits to create different kind of notifications. When the tab is selected, it appears a form that it must be filled in order to create the notification. This field is composed:

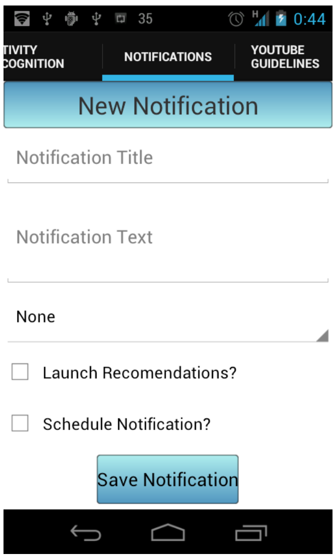


Figure 4.18: Notifications. General view

- Title. Title of the notification.
- Description. Full description text of the notification.
- Sound. A field where three different sounds may be selected to be reproduced when the notification is launched: Alarm, Ringtone and Notification sounds. There is also the possibility of set this sound to "None".
- Launch Recommendations. It is a checkbox that in the case of being checked, when the notification is clicked the YouTube guidelines is launched.
- Schedule Notification. It allows for the scheduling the notification to a specified date and time.

As an example, it can be used an app user with back problems that needs to be reminded about his daily exercises. Thus, for this purpose, the user can create a notification and customize it by putting the notification title and description, as well as setting the hour of its appearance. Moreover, if the user wants to visualize some

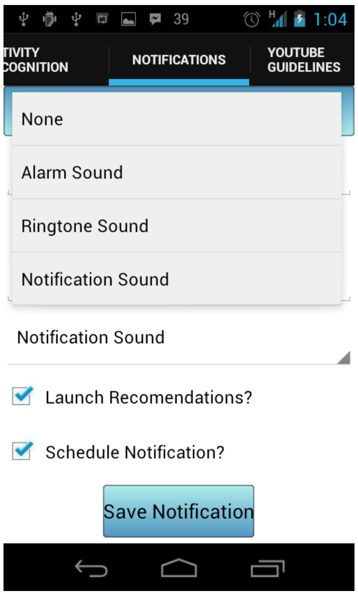
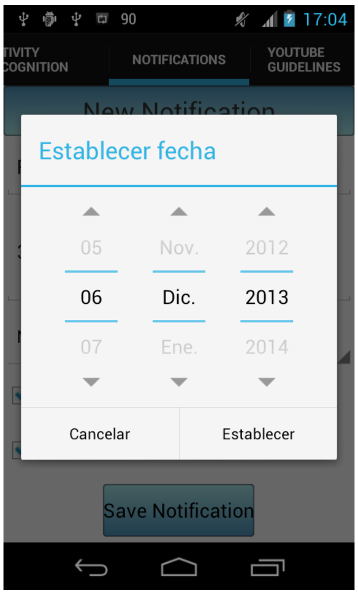
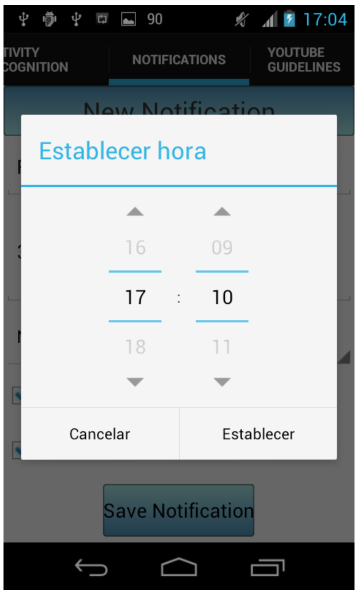


Figure 4.19: Notifications. Sound to be reproduced



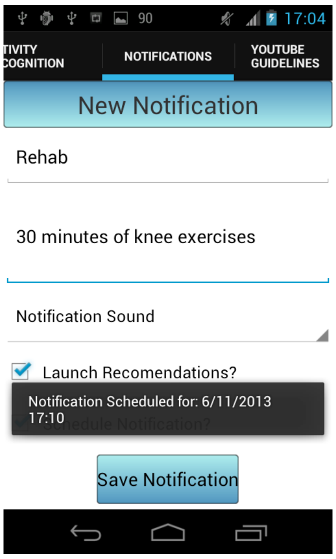
(a) Notification Date



(b) Notification Time

Figure 4.20: Notifications. Setting date and time

recommended exercises he can mark the *Launch Recommendations* option that after clicking on the notification will automatically open the YouTube guidelines 4.1.5



90 17:04

TIVITY COGNITION | NOTIFICATIONS | YOUTUBE GUIDELINES

New Notification

Rehab

30 minutes of knee exercises

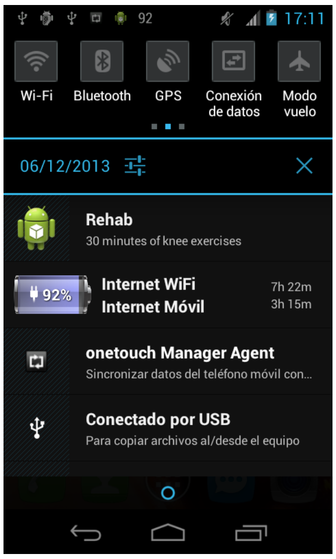
Notification Sound

✓ Launch Recomendations?

Notification Scheduled for: 6/11/2013 17:10

Save Notification

(a) Notification saved



92 17:11

Wi-Fi | Bluetooth | GPS | Conexión de datos | Modo vuelo

06/12/2013

Rehab
30 minutes of knee exercises

Internet WiFi 7h 22m
Internet Móvil 3h 15m
92%

onetouch Manager Agent
Sincronizar datos del teléfono móvil con...

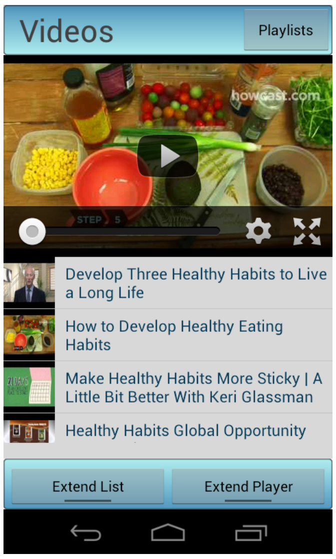
Conectado por USB
Para copiar archivos al/desde el equipo

(b) Notification launched

Figure 4.21: Notifications. Saving notification and notification launched

4.1.5 YouTube Guidelines

This tab allows the app user the visualization of videos in order to keep healthy habits or learn new exercises with physical therapy purposes. There are five different playlists that can be selected by the user using the button *Playlists*:



Videos | Playlists

Develop Three Healthy Habits to Live a Long Life

How to Develop Healthy Eating Habits

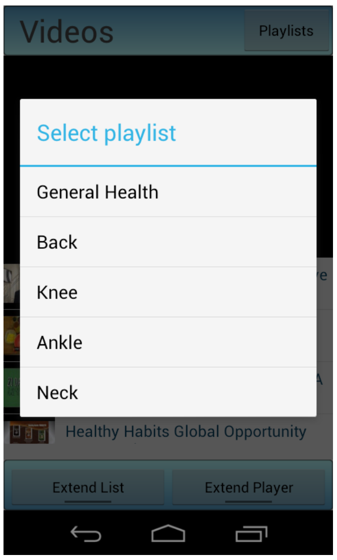
Make Healthy Habits More Sticky | A Little Bit Better With Keri Glassman

Healthy Habits Global Opportunity

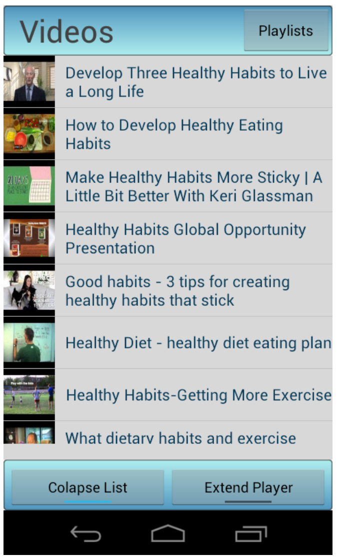
Extend List | Extend Player

Figure 4.22: YouTube Guidelines. General view

- General Health: playlist about how to keep healthy daily habits and how to maintain a healthy diet.
- Knee: playlist with videos of knee rehabilitation, orientate to user with knee problems.
- Back: playlist with videos of back rehabilitation, orientate to user with back problems.
- Ankle: playlist with videos of ankle rehabilitation, orientate to user with ankle problems.
- Neck: playlist with videos of neck rehabilitation, orientate to user with neck problems.



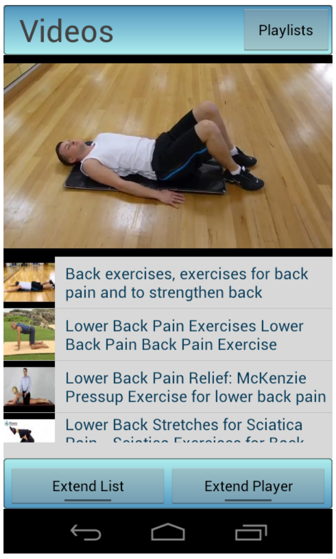
(a) Available playlists


Videos
Playlists
Develop Three Healthy Habits to Live a Long Life
How to Develop Healthy Eating Habits
Make Healthy Habits More Sticky | A Little Bit Better With Keri Glassman
Healthy Habits Global Opportunity Presentation
Good habits - 3 tips for creating healthy habits that stick
Healthy Diet - healthy diet eating plan
Healthy Habits-Getting More Exercise
What dietary habits and exercise
Colapse List
Extend Player

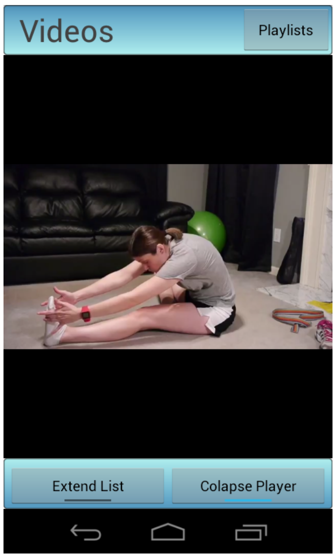
(b) Video list extended

Figure 4.23: Youtube Guidelines. Selecting and extending a playlist

Once that a playlist has been selected, a scrollable list of videos will appear on the screen. This list can be extended pressing on the *Extend List* button and collapse by pressing on it again. When the user selects a video to be reproduced, this appears on the player and can be reproduced when the start icon is clicked. The player may also be extended using the button *Extend Player* or collapse clicking on it again.


Videos
Playlists
Back exercises, exercises for back pain and to strengthen back
Lower Back Pain Exercises Lower Back Pain Back Pain Exercise
Lower Back Pain Relief: McKenzie Pressup Exercise for lower back pain
Lower Back Stretches for Sciatica
Extend List
Extend Player

(a) Reproducing video in normal size



(b) Reproducing video with player extended

Figure 4.24: Youtube Guidelines. Reproducing videos

4.1.6 Remote Storage

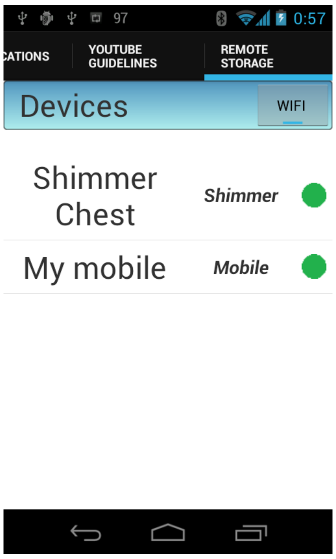
This tab allows the user app to upload to a remote storage the collected data from the portable biomedical devices. It is composed of a list with every of the available devices in the app and a button to set the WiFi connection (for the sake of simplicity, since 3G connection can be used as well).

When a device of the list is pressed on, it appears a dialog asking the app user to define which tables must be uploaded. There are three options: Units, Metadata and Signals. After pressing the OK button the uploading of the selected data starts. A message will appear in the screen to notify the end of the uploading process.

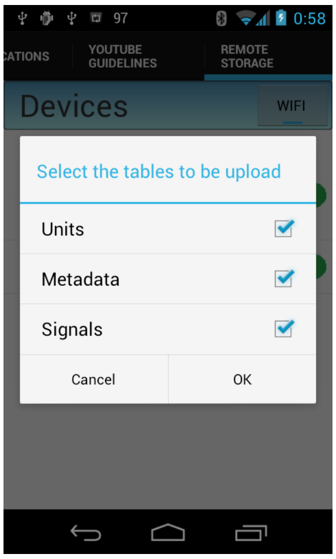
4.2 Activity Recognizer: Experimental Dataset

To demonstrate the system capabilities (section 4.3) a dataset is collected by using the Shimmer sensors together with the mobile device. This dataset is composed the following activities (from lower intensity to upper) and its length:

1. Standing still → 1 minute, 2. Sitting and relaxing → 1 minute, 3. Lying down → 1 minute, 4. Walking → 1 minute, 5 .Climbing stairs → 1 minute, 6. Waist bends forward → 20 repetitions, 7. Frontal (vertical) up/down → 20 repetitions, 8. Knees



(a) General view



(b) Choosing information to upload

Figure 4.25: Remote Storage. Setting the uploading

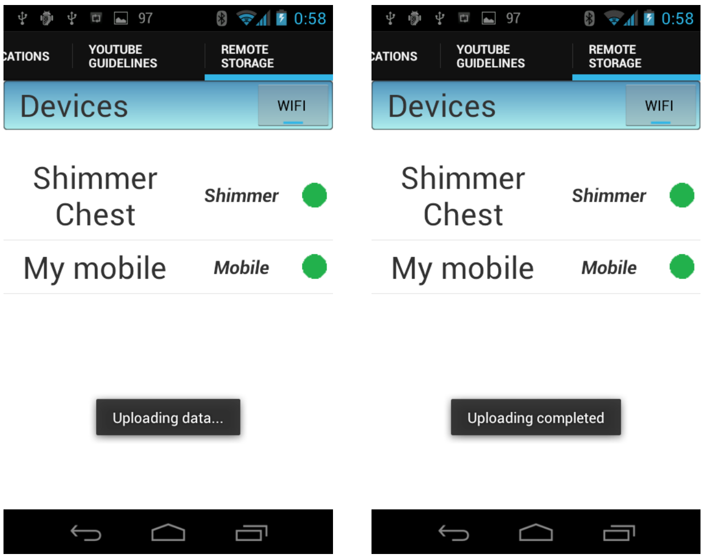


Figure 4.26: Remote Storage. Starting and finishing uploading

bended → 20 repetitions, 9. Cycling → 1 minute, 10. Jogging → 1 minute, 11. Running → 1 minute, 12. Jump forward and back → 20 repetitions.

The experiment setup was composed by a mobile device (situated for example in a pocket) and three SHIMMER devices placed on the chest (SHIMMER CHEST), wrist

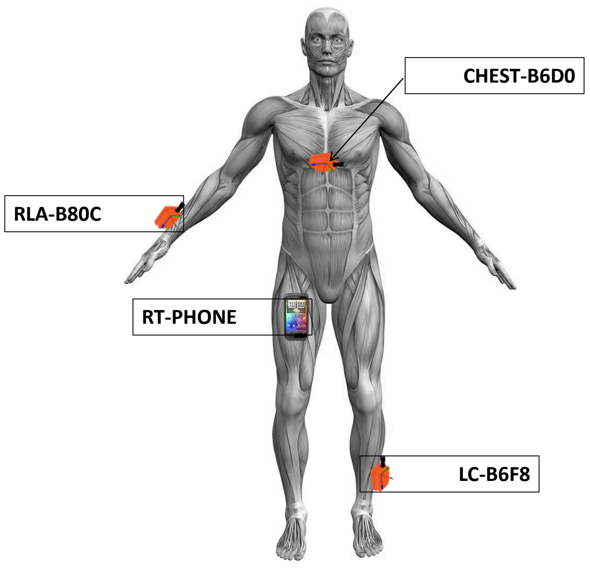


Figure 4.27: Positioning of the portable wearable devices

(SHIMMER RLA) and ankle (SHIMMER LC) as figure 4.27 shows.

The mobile device is used to manage and synchronize the SHIMMER devices, as well as store the physiological and kinematic data that these collect. The activated sensors for every device were:

- *SHIMMER Chest*: Timestamp, Accelerometer X, Accelerometer Y, Accelerometer Z, ECGRALL and ECGLALL.
- *SHIMMER RLA*: Timestamp, Accelerometer X, Accelerometer Y, Accelerometer Z, Magnetometer X, Magnetometer Y, Magnetometer Z, Gyroscope X, Gyroscope Y and Gyroscope Z.
- *SHIMMER LC*: Timestamp, Accelerometer X, Accelerometer Y, Accelerometer Z, Magnetometer X, Magnetometer Y, Magnetometer Z, Gyroscope X, Gyroscope Y and Gyroscope Z.

The ECGRALL and ECGLALL are used to acquire information about ECG. A SHIMMER ECG daughter board and four electrodes are used. These electrodes are situated in four positions (4.28): Left Arm (LA), Left Leg (LL), Right Arm (RA) and Right Leg (RL).

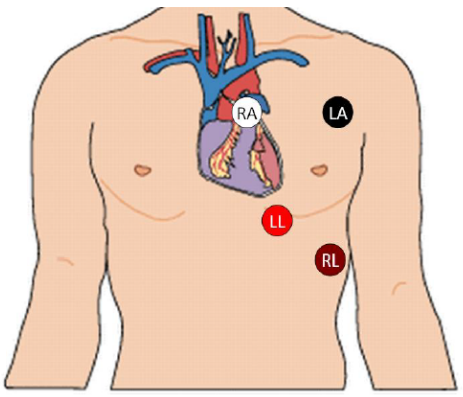


Figure 4.28: Positioning of the four electrodes. Picture from (10)

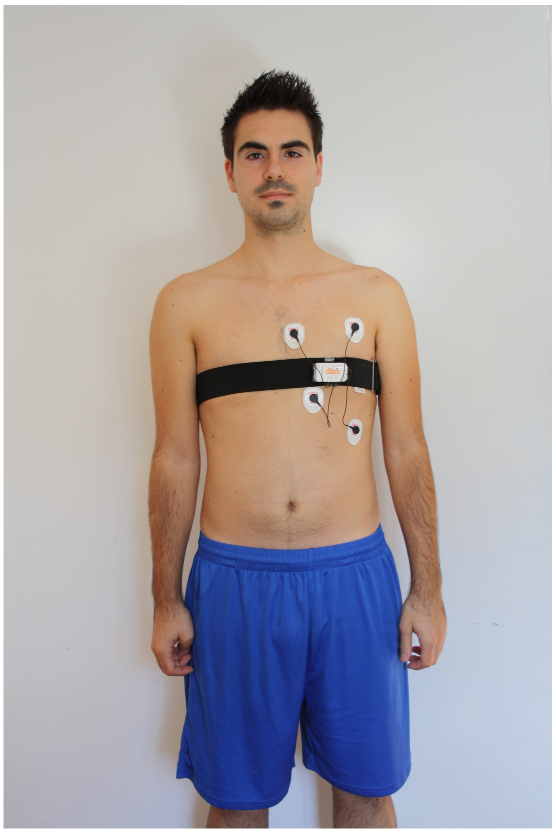


Figure 4.29: Electrodes in subject for ECG acquisition

The experiment was performed for ten subjects, seven males and three females of similar age and complexity. After the experiment, the recorded data were labeled according to time on which the activities were performed.

4.3 Performance evaluation

The performance of a system trained on these data is shown in the following. To that end, the recognition process is defined through a segmentation of two seconds window size. MATLAB (79) have been used to perform the feature extraction of the dataset.The following features were calculated for the data extracted from every SHIMMER device:

- Mean: Accelerometer X, Accelerometer Y, Accelerometer Z
- Standard Deviation: Accelerometer X, Accelerometer Y, Accelerometer Z
- Max: Accelerometer X, Accelerometer Y, Accelerometer Z
- Min: Accelerometer X, Accelerometer Y, Accelerometer Z

After this process, instances corresponding to two seconds of data were obtained composed of 36 features plus a label column which acts as attribute class. Putting these instances into WEKA (78) the model was built using the J48 classifier (80). To test the classifier performance, a cross-validation test with ten folds was used. Of 3353 instances, 3296 instances were correctly classifier, which gives a 98.3 % of accuracy performance. Some others statistics follow:

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        3296               98.3    %
Incorrectly Classified Instances        57                1.7    %
Kappa statistic                          0.9814
Mean absolute error                      0.0029
Root mean squared error                  0.0504
Relative absolute error                  1.923  %
Root relative squared error             18.26   %
Total Number of Instances             3353
```

The accuracy of the recognizer (per class) is:

```
=== Confusion Matrix ===

   a   b   c   d   e   f   g   h   i   j   k   l   <-- classified as
 300   0   0   0   0   0   0   0   0   0   0   0 |   a = 1
   0 300   0   0   0   0   0   0   0   0   0   0 |   b = 2
   0   0 299   0   0   1   0   0   0   0   0   0 |   c = 3
   0   0   0 300   0   0   0   0   0   0   0   0 |   d = 4
   0   0   0   0 296   0   1   2   1   0   0   0 |   e = 5
   0   0   0   0   0 270   1   7   0   0   0   0 |   f = 6
   0   0   0   0   0   0 288   0   0   0   0   0 |   g = 7
   0   0   0   0   3   8   1 273   2   0   0   0 |   h = 8
   0   0   0   0   0   0   0   0 300   0   0   0 |   i = 9
   0   0   0   1   0   0   0   0   0 288  10   1 |   j = 10
   0   0   0   0   0   0   0   0   0  17 283   0 |   k = 11
   0   0   0   0   0   0   0   0   0   0   1  99 |   l = 12
```

5

Conclusions

This work has presented a mobile framework for the development of biomedical applications specifically devised for the Android platform. The framework provides diverse functionalities, from which developers may levarage during the implementation of health monitoring apps. Any trustworthy and complete biomedical application needs to offer functionalities such as communication with portable biomedical devices, data streaming and visualization, local and remote storage, activity recognition, notifications and guidelines among others. All these functionalities can be incorporated to mobile applications using this framework in a really easy way, which makes the programmer task much easier and reachable for everybody with basic programming skills.

Personally, I think this work is a big contribution to the biomedical engineering field. There are many biomedical applications for different health purposes, however, there is a lack of biomedical frameworks to speed up, facilitate and provide to a wide range of programmers the development of biomedical apps.

5.1 Achievements achieved

All objectives established in this work has been achieved, at least to provide the basic functionality of every framework component. However, as will be commented in the section 5.2 every component can be extend or improved. The objectives achieved follow:

- Framework to allow rapid development of medical, health and wellbeing applications.

- Efficient and fast communication between portable biomedical devices and portable mobile devices to gather patient data like physiological or kinematic signals. For this version of the framework, SHIMMER devices and mobile sensors can be used as portable biomedical devices.

- Development of applications capable of working with different portable health devices simultaneously.
- Efficient data transfer across the framework managers.
- Fast data storage either local or remote.
- Visualization of any kind of data stream like patient's vital signs or kinematic data, either online or offline.
- Medical knowledge inference procured through machine learning and statistical model.
- Guidelines support through Audio, Video or YouTube playlists, as well as notifications procedures.
- System control and configuration tools to manage WiFi, Bluetooth, screen brightness, making phone calls or sending text messages.
- User management through login functionality, also supporting security and privacy.
- Multiple user dataset collection (kinematics+ECG) through the developed APP. This is partially used for the app activity recognizer.
- Development of an exemplary app which implements the most important framework functionalities.

5.2 Future work

Given the important magnitude of the engineering biomedical field and the fact that the health care is becoming essential in our lives along with the continuous release of wearable health devices, this framework could be envisioned as a key tool for future investigation and development could be carried on by companies on this field. Various framework functionalities could be improved and new ones may be added. Some ideas are described:

- To add more wearable health devices to the framework. This requires to implement new drivers for these. Also, could be necessary to create the intermediate driver between the framework and the device if this is not provided by the device manufacturer.
- Development of APPs for any particular health issue. Some ideas:
  - APP that performs activity recognition of daily basis activities along with health cardiac activity visualization. Principally devised for elders.
  - APP that performs activity recognition of knee rehab exercises. The app could also show the user how to correctly perform the exercise through the use of the built-in guidelines.
  - APP for people with obesity problems. The APP could encourage the user to keep healthy habits using guidelines, notifications promoting sport practice or recommending healthy diets, etc.
  - Any APP that uses a portable health device used for a specific health issue (i.e, oxygen in blood, ECG, EEG, EMG or chronic diseases).
- Add other communication protocols, such as ZigBee or Xbee, as the market starts providing new communication interfaces.
- To improve the medical knowledge inference process extending the pre-processing and features extractions modules belonging to the Data Processing Manager adding more pre-processing methods and features to be extracted.
- To add a new stage to the Data Processing Manager called fusion used to aggregate the classification decisions of every available device.
- To provide a mechanism for absolute synchronization of the wearable health devices connected.
- To improve the Visualization manager, with functionalities such as displaying different devices in the same graph.
- To extend the Remote Storage Manager, making possible the data transference in both ways and using the server side for more functionalities than storage.
- To extend the guidelines functionalities and the settings to manage the portable mobile.
- To update the notifications to the new ones provided by the last Android versions.

References

[1] **Wikimedia Commons**, 2013. http://commons.wikimedia.org/. v, 2

[2] **Population Pyramid**, 2013. http://en.wikipedia.org/wiki/Population_pyramid. v, 3

[3] L. Mertz. **Ultrasound? Fetal monitoring? Spectrometer? There's an app for that!: Biomedical smart phone apps are taking healthcare by storm**. *IEEE Pulse*, **3**(2):16–21, 2012. Cited By (since 1996):1. v, 10, 12

[4] Yu M. Chi, Patrick Ng, Eric Kang, Joseph Kang, Jennifer Fang, and Gert Cauwenberghs. **Wireless non-contact cardiac and neural monitoring**. In *Wireless Health 2010*, WH '10, pages 15–23, New York, NY, USA, 2010. ACM. v, 11, 12

[5] Mobisante. **The MobiUS SP1 System**, 2013. http://www.mobisante.com/products/product-overview/. v, 13

[6] EarlySense. **Florida hospital installs EarlySense to advance health care**, 2013. http://israel21c.org/news/florida-hospital-installs-earlysense-to-advance-health-care/. v, 14

[7] Oresti Banos, Miguel Damas, Hector Pomares, Fernando Rojas, Blanca Delgado-Marquez, and Olga Valenzuela. **Human activity recognition based on a sensor weighting hierarchical classifier**. *Soft Computing*, **17**(2):333–343, 2013. v, 15

[8] **SQL injection comic**. http://xkcd.com/327/. vi, 55

[9] **Media Player Android API**. http://developer.android.com. vi, 79, 82

[10] **Shimmer. Discovery in motion**. http://www.shimmersensing.com. vii, 130

[11] **The Evolution of mortality**, 2013. http://www.demogr.mpg.de/en/. 1

[12] Kevin G Kinsella. **Changes in life expectancy 1900-1990**. 1992. 1

[13] **World Health Organization**. http://www.who.int/en/. 1

[14] **UNAIDS. Joint United Nations Programme on HIV/AIDS**, 2013. http://www.unaids.org/. 2

[15] World Health Organization. **Sierra Leona. Service Availability and Readiness Assessment 2010**, 2013. http://apps.who.int/healthinfo/systems/. 2

[16] World Health Organization. **Zambia. Service Availability and Readiness Assessment 2010**, 2013. http://www.who.int/healthinfo/systems/. 2

[17] **World Health Organization 2010 Report**. http://www.who.int/whr/2010/en/. 4

[18] A. Burns, B.R. Greene, M.J. McGrath, T.J. O'Shea, B. Kuris, S.M. Ayer, F. Stroiescu, and V. Cionca. **SHIMMER. A Wireless Sensor Platform for Noninvasive Biomedical Research**. *Sensors Journal, IEEE*, **10**(9):1527–1534, 2010. 5, 12

[19] B. Karagozoglu. **Ambulatory monitoring**. In *2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering, TAEECE 2013*, pages 244–249, 2013. 9

[20] J.A. Garibaldi-Beltran and M. Vazquez-Briseno. **Personal Mobile Health Systems for Supporting Patients with Chronic Diseases**. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA), 2012 IEEE Ninth*, pages 105–110, 2012. 9

[21] D. R. Lustick and M. H. Zaman. **Biomedical engineering education and practice challenges and opportunities in improving health in developing countries**. In *2011 Atlanta Conference on Science and Innovation Policy: Building Capacity for Scientific Innovation and Outcomes, ACSIP 2011, Proceedings*, 2011. 10

[22] AirStrip. **AirStrip ONE Cardiology**, 2013. http://www.airstriptech.com/cardiology/care-continuum/. 10

[23] AirStrip. **AirStrip ONE Patient Monitoring**, 2013. http://www.airstriptech.com/cardiology/care-continuum/. 10

[24] J. Wojtczak and P. Bonadonna. **Pocket mobile smartphone system for the point-of-care submandibular ultrasonography**. *American Journal of Emergency Medicine*, **31**(3):573–577, 2013. 11, 12

[25] S. Mazilu, M. Hardegger, Z. Zhu, D. Roggen, G. Trster, M. Plotnik, and J. M. Hausdorff. **Online detection of freezing of gait with smartphones and machine learning techniques**. In *2012 6th International Conference on Pervasive Computing Technologies for Healthcare and Workshops, PervasiveHealth 2012*, pages 123–130, 2012. Cited By (since 1996):2. 11

[26] Kasey Panetta. **Impact of Portable Medical Devices**, 2013. http://www.mdtmag.com/articles/2013/06/. 11

[27] F. Axisa, P. M. Schmitt, C. Gehin, G. Delhomme, E. McAdams, and A. Dittmar. **Flexible technologies and smart clothing for citizen medicine, home healthcare, and disease prevention**. *IEEE Transactions on Information Technology in Biomedicine*, **9**(3):325–336, 2005. Cited By (since 1996):106. 11

[28] Michael Marschollek, Matthias Gietzelt, Mareike Schulze, Martin Kohlmann, Bianying Song, and Klaus-Hendrik Wolf. **Wearable sensors in healthcare and sensor-enhanced health information systems: all our tomorrows?** *Healthc Inform Res*, **18**(2):97–104, June 2012. 11

[29] P. Zhou, Z. Li, F. Wang, and H. Jiao. **Portable wireless ECG monitor with fabric electrodes**. In *IFMBE Proceedings*, **39 IFMBE**, pages 1495–1498, 2013. 11

[30] T. J. Walters, K. A. Kaschinske, S. J. Strath, A. M. Swartz, and K. G. Keenan. **Validation of a portable EMG device to assess muscle activity during free-living situations**. *Journal of Electromyography and Kinesiology*, **23**(5):1012–1019, 2013. 11

[31] Nina J. Cleven, Jutta A. Mntjes, Holger Fassbender, Ute Urban, Michael Grtz, Holger Vogt, Maik Grfe, Thorsten Gttsche, Tobias Penzkofer, Thomas Schmitz-Rode, and Wilfried Mokwa. **A Novel Fully Implantable Wireless Sensor System for Monitoring Hypertension Patients.** *IEEE Trans. Biomed. Engineering*, **59**(11-2):3124–3130, 2012. 12

[32] P. Bartolomeu, J. Fonseca, and F. Vasques. **Challenges in Health Smart Homes**. In *Pervasive Computing Technologies for Healthcare, 2008. PervasiveHealth 2008. Second International Conference on*, pages 19–22, 2008. 12

[33] I. Jablonski. **Wearable Interrupter Module for Home-Based Applications in a Telemedical System Dedicated to Respiratory Mechanics Measurements**. *Biomedical Engineering, IEEE Transactions on*, **58**(3):785–789, 2011. 13

[34] Jessica M. Kelly, Robert E. Strecker, and Matt T. Bianchi. **Recent Developments in Home Sleep-Monitoring Devices**. *ISRN Neurology*, **2012**, 2012. 13

[35] Ralph E. Johnson. **Frameworks = (components + patterns)**. *Commun. ACM*, **40**(10):39–42, October 1997. 13

[36] A. Shawish and M. Salama. **Cloud computing: Paradigms and technologies**. *Studies in Computational Intelligence*, **495**:39–67, 2014. cited By (since 1996)0. 14

[37] U. Morgenstern, A. Abdel-Haq, V. Barth, H. Dietrich, and I. Rudolph. **TheraGnosos: An interactive blended learning, simulation and training system for biomedical engineering university courses**. In *IFMBE Proceedings*, **22**, pages 2757–2759, 2008. 14

[38] A. Schlgl, C. Vidaurre, and T. H. Sander. **BioSig: The free and open source software library for biomedical signal processing**. *Computational Intelligence and Neuroscience*, **2011**, 2011. Cited By (since 1996):7. 14

[39] H. Jiang, P. C. M. Van Zijl, J. Kim, G. D. Pearlson, and S. Mori. **DtiStudio: Resource program for diffusion tensor computation and fiber bundle tracking**. *Computer methods and programs in biomedicine*, **81**(2):106–116, 2006. Cited By (since 1996):414. 14

[40] T. A. Pologruto, B. L. Sabatini, and K. Svoboda. **ScanImage: Flexible software for operating laser scanning microscopes**. *BioMedical Engineering Online*, **2**, 2003. Cited By (since 1996):183. 14

[41] J. C. Barrett, B. Fry, J. Maller, and M. J. Daly. **Haploview: Analysis and visualization of LD and haplotype maps**. *Bioinformatics*, **21**(2):263–265, 2005. Cited By (since 1996):6431. 14

[42] V. J. Barclay, R. F. Bonner, and I. P. Hamilton. **Application of Wavelet Transforms to Experimental Spectra: Smoothing, Denoising, and Data Set Compression**. *Analytical Chemistry*, **69**(1):78–90, 1997. Cited By (since 1996):185. 15

[43] Giovanni Acampora, Chang-Shing Lee, Autilia Vitiello, and Mei-Hui Wang. **Evaluating cardiac health through semantic soft computing techniques**. *Soft Computing*, **16**(7):1165–1181, 2012. 15

[44] M. Korrek and A. Nizam. **Clustering MIT-BIH arrhythmias with Ant Colony Optimization using time domain and PCA compressed wavelet coefficients**. *Digital Signal Processing: A Review Journal*, **20**(4):1050–1060, 2010. Cited By (since 1996):12. 15

[45] Corinna Cortes and Vladimir Vapnik. **Support-Vector Networks**. *Mach. Learn.*, **20**(3):273–297, September 1995. 15

[46] Lotfi A. Zadeh. **Fuzzy logic = computing with words**. *Fuzzy Systems, IEEE Transactions on*, **4**(2):103–111, 1996. 15

[47] R. Setiono and Huan Liu. **Neural-network feature selector**. *Neural Networks, IEEE Transactions on*, **8**(3):654–662, 1997. 15

[48] J.R. Quinlan. **Induction of decision trees**. *Machine Learning*, **1**:81–106, 1986. 15

[49] S. Karpagachelvi, M. Arthanari, and M. Sivakumar. **Classification of electrocardiogram signals with support vector machines and extreme learning machine**. *Neural Computing and Applications*, **21**(6):1331–1339, 2012. 15

[50] **Open Handset Alliance**. http://www.openhandsetalliance.com. 20

[51] **Apache License**. http://www.apache.org/licenses/. 21

[52] **Number of available Android Applications**. http://www.appbrain.com/stats/number-of-android-apps/. 21

[53] **Google Play hits 25 billion downloads**. http://officialandroid.blogspot.ca/2012/09/google-play-hits-25-billion-downloads.html. 21

[54] **Developer Economics Q3 2013**. http://www.developereconomics.com/reports/q3-2013/. 21

[55] **Android, the world's most popular mobile platform**. http://developer.android.com/about/index.html/. 21

[56] **Service. Android API component.** http://developer.android.com/reference/android/app/Service.html. 25

[57] **Message. Android API component.** http://developer.android.com/reference/android/os/Message.html. 25

[58] **Handler. Android API component.** http://developer.android.com/reference/android/os/Handler.html. 25

[59] **Looper. Android API component.** http://developer.android.com/reference/android/os/Looper.html. 25

[60] **JSON (JavaScript Object Notation)**, 2013. http://www.json.org/. 50

[61] **XAMPP**, 2013. http://www.apachefriends.org/en/. 53

[62] **PHP language**. www.php.net. 53

[63] **MySQL Improved Extension**. http://php.net/manual/es/book.mysqli.php. 54

[64] **Original MySQL (MySQL)**, 2013. http://www.php.net/manual/en/book.mysql.php/. 54

[65] **SQL Injection. Bobby Tables**, 2013. http://psoug.org/blogs/mike/2010/04/11/little-bobby-tables/. 54

[66] **Prepared Statements**. http://mattbango.com/notebook/code/prepared-statements-in-php-and-mysqli. 56

[67] **Androidplot**. 63

[68] **Graphview library**. 63

[69] **Chartdroid**. 63

[70] **Afreechart**. https://code.google.com/p/afreechart/. 63

[71] **Achartengine**. https://code.google.com/p/achartengine/. 63

[72] **Notification Manager**. http://developer.android.com/. 74

[73] **Broadcaster Receiver**. http://developer.android.com/. 75

[74] **Alarm Manager**. http://www.developer.android.com/. 75

[75] **Video View Class**. http://developer.android.com/. 79, 82

[76] **Notification Class**. http://developer.android.com/. 80

[77] **Youtube Android Player API**. https://developers.google.com/youtube/android/player/. 81

[78] **Weka: Data Mining Software in Java**. http://www.cs.waikato.ac.nz/ml/weka/. 101, 131

[79] **MATrix LABoratory**. http://www.mathworks.com/. 131

[80] **C4.5 algorithm (J48)**. http://en.wikipedia.org/wiki/C4.5_algorithm/. 131