Smartphones for smart driving: 
a proof of concept

Author: Bernat Planas Puig
Supervisor: Josep Paradells Aspas

A thesis submitted in fulfillment of the requirements
for the degree of Master in Telematics Engineering
in the
Departament d’Enginyeria Telemàtica
Escola Tècnica Superior d’Enginyeria de Telecomunicació de Barcelona

November 2013
“Tot està per fer i tot és possible.”

Miquel Martí i Pol
Abstract

Nowadays people are owning vehicles for several years and on-board computers become outdated quite fast. Smartphones are renewed more frequently and improved constantly. Current Smartphones implement several communication interfaces and sensors that, combined with vehicles, can provide a lot of interesting features to the final user. The purpose of this project is to analyze the current capabilities of this scenario and to develop an Android application in order to implement several mechanisms to improve the environmental efficiency, the vehicle life and the driving safety.
Acknowledgements

I would like to thank Josep Paradells Aspas for the opportunity of doing this project and showing me his motivation on it. I will also thank him for trusting me all time and believing in what I have been doing.

I also want to thank the colleagues in the office. It was a pleasure to make new friends and share our concerns and achievements during this period. Working with them was really a great experience.

The people from the wireless network group have been also very helpful, specially for the beginner matters of working at university. It is a privilege to have kind people there to help you in everything and even for having a good time.

Finally, I want to thank my family for supporting and trusting me during all my studying career. Without them, I would have not reached this point.
Contents

Abstract

Acknowledgements

List of Figures

List of Tables

Abbreviations

1 Introduction
  1.1 Motivation
  1.2 Objectives
  1.3 Document structure

2 State of the art
  2.1 Overview of mobile applications currently available for the end users
  2.2 Related research works in the field
    2.2.1 Environmental efficiency
    2.2.2 Driving safety
    2.2.3 Accident detection and reporting
    2.2.4 Other automotive proposals

3 Context description
  3.1 Android
    3.1.1 Android application fundamentals
    3.1.2 Android Activities
    3.1.3 Android Services
    3.1.4 User interface
    3.1.5 Android Manifest file
    3.1.6 Android location strategies
    3.1.7 Android sensors
    3.1.8 Android connectivity
    3.1.9 Android SQLite
  3.2 On-Board Diagnostics
    3.2.1 J1962 connector
    3.2.2 Signal protocols
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3 Diagnostics Information</td>
<td>32</td>
</tr>
<tr>
<td><strong>4 Tools</strong></td>
<td>34</td>
</tr>
<tr>
<td>4.1 ELM327 adapter</td>
<td>34</td>
</tr>
<tr>
<td>4.1.1 ELM327 overview</td>
<td>34</td>
</tr>
<tr>
<td>4.1.2 How to use the ELM327</td>
<td>36</td>
</tr>
<tr>
<td>4.2 OBDSim (OBD GPS Logger)</td>
<td>36</td>
</tr>
<tr>
<td>4.3 Android Developer Tools overview</td>
<td>39</td>
</tr>
<tr>
<td>4.4 Extra Android libraries</td>
<td>40</td>
</tr>
<tr>
<td>4.4.1 GraphView</td>
<td>40</td>
</tr>
<tr>
<td>4.4.2 Android OBD Reader</td>
<td>42</td>
</tr>
<tr>
<td><strong>5 Project description</strong></td>
<td>45</td>
</tr>
<tr>
<td>5.1 Application functionalities</td>
<td>45</td>
</tr>
<tr>
<td>5.1.1 Read OBD data through Bluetooth</td>
<td>46</td>
</tr>
<tr>
<td>5.1.2 Provide real time driving behavior information</td>
<td>46</td>
</tr>
<tr>
<td>5.1.3 Record the trip statistics in a local database</td>
<td>47</td>
</tr>
<tr>
<td>5.1.4 Real time alerts related with the driving behavior</td>
<td>47</td>
</tr>
<tr>
<td>5.1.5 Accident detection and reporting</td>
<td>47</td>
</tr>
<tr>
<td>5.1.6 Traffic jam detection and reporting</td>
<td>49</td>
</tr>
<tr>
<td>5.1.7 Retrieve and take profit of the weather forecast</td>
<td>49</td>
</tr>
<tr>
<td>5.1.8 Environmental jam detection and reporting</td>
<td>49</td>
</tr>
<tr>
<td>5.1.9 Extending vehicle life summary</td>
<td>50</td>
</tr>
<tr>
<td>5.1.10 Driving safety summary</td>
<td>50</td>
</tr>
<tr>
<td>5.2 Navigation diagram</td>
<td>50</td>
</tr>
<tr>
<td>5.3 Source code overview</td>
<td>51</td>
</tr>
<tr>
<td>5.4 SQLite database structure</td>
<td>53</td>
</tr>
<tr>
<td><strong>5.5 Activities</strong></td>
<td>54</td>
</tr>
<tr>
<td>5.5.1 Main Activity</td>
<td>54</td>
</tr>
<tr>
<td>5.5.2 Settings Activity</td>
<td>56</td>
</tr>
<tr>
<td>5.5.3 Driving Panel Activity</td>
<td>59</td>
</tr>
<tr>
<td>5.5.4 Environment Activity</td>
<td>60</td>
</tr>
<tr>
<td>5.5.5 Vehicle Activity</td>
<td>62</td>
</tr>
<tr>
<td>5.5.6 Safety Activity</td>
<td>64</td>
</tr>
<tr>
<td>5.5.7 Statistics Activities</td>
<td>66</td>
</tr>
<tr>
<td><strong>5.6 Background processes</strong></td>
<td>71</td>
</tr>
<tr>
<td>5.6.1 TripService</td>
<td>73</td>
</tr>
<tr>
<td>5.6.2 ObdDataManager</td>
<td>79</td>
</tr>
<tr>
<td>5.6.3 LocationHelper</td>
<td>82</td>
</tr>
<tr>
<td>5.6.4 WeatherManager</td>
<td>82</td>
</tr>
<tr>
<td>5.6.5 CongestionDetector</td>
<td>84</td>
</tr>
<tr>
<td>5.6.6 Alerts</td>
<td>85</td>
</tr>
<tr>
<td><strong>6 Test and results</strong></td>
<td>88</td>
</tr>
<tr>
<td>6.1 Read OBD data through Bluetooth</td>
<td>88</td>
</tr>
<tr>
<td>6.2 Provide real time driving behavior information</td>
<td>92</td>
</tr>
<tr>
<td>6.3 Record the trip statistics in a local database</td>
<td>92</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.4 Real time alerts related with the driving behavior</td>
<td>94</td>
</tr>
<tr>
<td>6.5 Accident detection and reporting</td>
<td>96</td>
</tr>
<tr>
<td>6.6 Traffic jam detection and reporting</td>
<td>99</td>
</tr>
<tr>
<td>6.7 Retrieve and take profit of the weather forecast</td>
<td>100</td>
</tr>
<tr>
<td>6.8 Environmental efficiency summary</td>
<td>101</td>
</tr>
<tr>
<td>6.9 Extending vehicle life summary</td>
<td>102</td>
</tr>
<tr>
<td>6.10 Driving safety summary</td>
<td>102</td>
</tr>
<tr>
<td>7 Future work</td>
<td>103</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>107</td>
</tr>
</tbody>
</table>

Bibliography and References  109
# List of Figures

2.1 Waze route map capture .............................................. 6  
2.2 Waze event report capture ........................................... 6  
2.3 Congestion information in Inrix  ........................................ 6  
2.4 Torque Pro screen captures ........................................... 8  
2.5 DriSMo screen capture while driving  ......................... 9  
2.6 DriSMo statistics screen capture ......................... 9  
2.7 iOnRoad screen capture ........................................... 10  
2.8 Driving Coach implementation diagram ................ 11  
2.9 Estimating driving behavior flow diagram ................... 13  
2.10 WreckWatch accident detection and notification ........ 14  
3.1 World-wide Smartphone sales ......................................... 19  
3.2 Android activity lifecycle .......................................... 21  
3.3 Android service lifecycle ........................................... 23  
3.4 Android view hierarchy ............................................ 24  
3.5 Android sensor physical axis ......................................... 28  
3.6 OBD-II connector location ........................................... 30  
3.7 J1962 connector pinout ................................................ 30  
4.1 ELM327 pinout ............................................................. 35  
4.2 ELM327 implementations ............................................... 35  
4.3 OBDSim User Interface ............................................... 37  
4.4 GraphView usage examples ........................................... 40  
5.1 Car to smartphone communication scheme .................... 46  
5.2 Emergency message using mobile coverage ............... 48  
5.3 Emergency message using Wi-Fi  .............................. 48  
5.4 Congestion speed pattern example  ....................... 49  
5.5 Navigation diagram of the application ................... 52  
5.6 Main activity screenshot ............................................ 55  
5.7 Main activity screenshot while the trip is running ........ 55  
5.8 Settings activity layout ............................................. 57  
5.9 Device inclination preference dialog ....................... 57  
5.10 Bluetooth devices preference dialog ....................... 58  
5.11 Driving panel layout ................................................ 59  
5.12 Environmental statistics screenshot ....................... 61  
5.13 Fuel evolution graph screenshot ................................ 61  
5.14 Fuel over speed graph screenshot  ....................... 62
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.15</td>
<td>Vehicle life statistics screenshot</td>
<td>63</td>
</tr>
<tr>
<td>5.16</td>
<td>Engine speed over vehicle speed graph screenshot</td>
<td>64</td>
</tr>
<tr>
<td>5.17</td>
<td>Safety related events statistics report screenshot</td>
<td>65</td>
</tr>
<tr>
<td>5.18</td>
<td>Safety related events evolution graph screenshot</td>
<td>65</td>
</tr>
<tr>
<td>5.19</td>
<td>Main statistics panel screenshot</td>
<td>66</td>
</tr>
<tr>
<td>5.20</td>
<td>Trip durations graph screenshot</td>
<td>66</td>
</tr>
<tr>
<td>5.21</td>
<td>Trip list activity screenshot</td>
<td>67</td>
</tr>
<tr>
<td>5.22</td>
<td>Trip statistics menu screenshot</td>
<td>68</td>
</tr>
<tr>
<td>5.23</td>
<td>General trip statistics screenshot</td>
<td>69</td>
</tr>
<tr>
<td>5.24</td>
<td>Trip accelerations graph screenshot</td>
<td>70</td>
</tr>
<tr>
<td>5.25</td>
<td>Trip speed graph screenshot</td>
<td>70</td>
</tr>
<tr>
<td>5.26</td>
<td>Trip RPM graph screenshot</td>
<td>71</td>
</tr>
<tr>
<td>5.27</td>
<td>Interacting components during the trip</td>
<td>72</td>
</tr>
<tr>
<td>5.28</td>
<td>Status bar notification of the foreground service</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>OBDSim database sample</td>
<td>89</td>
</tr>
<tr>
<td>6.2</td>
<td>Speed graph from the OBDSim</td>
<td>89</td>
</tr>
<tr>
<td>6.3</td>
<td>RPM graph from the OBDSim</td>
<td>90</td>
</tr>
<tr>
<td>6.4</td>
<td>Real OBD connector</td>
<td>91</td>
</tr>
<tr>
<td>6.5</td>
<td>Real ELM327 device connection with the OBD</td>
<td>91</td>
</tr>
<tr>
<td>6.6</td>
<td>Smartphone location in the car</td>
<td>91</td>
</tr>
<tr>
<td>6.7</td>
<td>Capture of the application database file structure</td>
<td>93</td>
</tr>
<tr>
<td>6.8</td>
<td>Capture of the trip table from the application database</td>
<td>94</td>
</tr>
<tr>
<td>6.9</td>
<td>Capture of the accelerations table from the application database</td>
<td>95</td>
</tr>
<tr>
<td>6.10</td>
<td>Capture of the events table from the application database</td>
<td>96</td>
</tr>
<tr>
<td>6.11</td>
<td>Aggressive forward acceleration event graph</td>
<td>97</td>
</tr>
<tr>
<td>6.12</td>
<td>Accident alert screenshot</td>
<td>97</td>
</tr>
<tr>
<td>6.13</td>
<td>Emails received from the accident alert</td>
<td>98</td>
</tr>
<tr>
<td>6.14</td>
<td>Speed graph for a traffic congestion event</td>
<td>100</td>
</tr>
<tr>
<td>6.15</td>
<td>Weather forecast capture</td>
<td>100</td>
</tr>
<tr>
<td>6.16</td>
<td>Weather alert capture</td>
<td>101</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Average age of road vehicles ............................................. 1
3.1 J1962 connector pin description ..................................... 31
5.1 Structure of the local SQLite database ............................. 54
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VANET</td>
<td>Vehicular Ad-hoc Network</td>
</tr>
<tr>
<td>ITS</td>
<td>Intelligent Transportation Systems</td>
</tr>
<tr>
<td>OBD-II</td>
<td>On-Board Diagnostics, II generation</td>
</tr>
<tr>
<td>EOBD</td>
<td>European On-Board Diagnostics</td>
</tr>
<tr>
<td>ECU</td>
<td>Engine Control Unit</td>
</tr>
<tr>
<td>PCM</td>
<td>Powertrain Control Module</td>
</tr>
<tr>
<td>DTC</td>
<td>Diagnostic Trouble Code</td>
</tr>
<tr>
<td>PID</td>
<td>Parameter Identification Number (From OBD)</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

It is known that technology is evolving very fast, but not all products are renewed as frequently as they could. In the case of vehicles people do not buy new cars much frequently. The average vehicle age in Europe is of around 10 years [1] as is shown in table 1.1. We know that nowadays some recent manufactured cars are implementing integrated security and efficiency features that are combined with innovative full-featured on-board computers and user interfaces.

Otherwise, the time between a car is designed and it can be sold is quite long compared with technological evolution. In addition, if we consider the time period that a user owns a car it leads to a very long time. These facts show the possibility of using the user smartphone with the car. In this case, we know that smartphones are more frequently updated in software and even in hardware because the user renews the smartphone much more frequently compared with the vehicle. It also means that the possibility of using the smartphone as a part of the car can reduce the cost of manufacturing a car. This fact can be very interesting and useful for low or medium cost car manufacturers.

<table>
<thead>
<tr>
<th>Type of vehicle</th>
<th>Average vehicle age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger Cars</td>
<td>8,19 years</td>
</tr>
<tr>
<td>Light Duty Vehicles</td>
<td>11,67 years</td>
</tr>
<tr>
<td>Heavy Duty Vehicles</td>
<td>11,49 years</td>
</tr>
<tr>
<td>2 Wheelers</td>
<td>13,30 years</td>
</tr>
<tr>
<td>Buses</td>
<td>9,30 years</td>
</tr>
</tbody>
</table>

Table 1.1: Average age of road vehicles in 2009 in EEA member countries (that is EU-27 plus Norway, Switzerland, Turkey) [1]
1.1 Motivation

In the current vehicular scenarios we have seen very few proposals from the vehicle manufacturers that lead to the possibility of having upgradeable systems in our cars. As is just stated above, the fast evolution of the technology requires the possibility to upgrade the software or even some hardware features of the vehicles. A good approach is to take profit of the users smartphone or other similar devices like Tablets. Apart from that, the use of a smartphone gives the possibility of having additional sensors and communication interfaces, that permits to improve the driving features of the car without changing anything in the car.

This facts are the ones that motivate the development of this project, that pretends to proof the advantages of using the smartphone in automotive. There is the motivation to show the possibility of having intelligent cars without the need of any extra implementation on the vehicle.

1.2 Objectives

In a vehicular environment, there are a lot of features that can be implemented by the use of a smartphone. Possible smartphone applications could be map routing, providing road or traffic information, record and share car statistics, etc. However, this project is focused in the efficiency and safety features. In order to differentiate the application features there are defined three main blocks to classify them: Environmental efficiency, Extending the vehicle life, Driving safely.

- Environmental efficiency: The main goal is to provide mechanisms to save the maximum fuel as possible to avoid pollution. In the most of the cases, the fuel consumption will be directly related with the user driving skills. That means that the research topic must not only focus in the engine improvement, it should also focus on improve the user driving skills and a smartphone application can be a good tool to achieve this goal.

- Extending the vehicle life: It is important that the user drives the car correctly without damaging it. The car may be protected against bad user utilization. In
some cases, the user should be advised about some incorrect driving skills that may damage the engine or other car components. As in the former case, a smartphone application can be a good interface to interact with the user.

- Driving safely: In this case, the purpose is to avoid traffic accidents or risky situations due to a bad driving and in case of an accident the goal is to reduce the user damage as much as possible, for example, by sending an emergency message in case an accident is detected.

The objective of this project is to develop an Android application called Driving Helper that will integrate some features that will try to accomplish the requirements of the three classification blocks mentioned above. So that, the application will join several functionalities that can be provided by vehicle manufacturers, car renting companies, public transport managers, etc. to be able to check the user driving patterns and advice the user to improve the driving skills.

1.3 Document structure

This document has two differentiated parts. The first chapters are devoted mainly to contextualize the scenario and the following chapters are mainly focused on the description and evaluation of the proposed Android application.

The chapter 1 is a brief introduction that explains the motivation and the main objectives of this project. The chapter 2 corresponds to the state of the art about the existing work on the field. It contains an explanation of the existing smartphone applications and an analysis of the existing research works on using smartphones for vehicular environments specially for providing safety and efficiency. The chapter 3 contains the context description to facilitate the understanding of the proposal. It describes basically some features of the Android operating system and the on-board diagnostics interface. The chapter 4 explains the tools used to develop the proposed application that are the ELM327 adapter, the OBDSim tool, the Android developer tools and the extra Android libraries used in the application.
The following chapters are more focused on the practical part of this project. In \textbf{chapter 5} there is the project description that contains all the information about the application functionalities, structure, and implementation. The \textbf{chapter 6} contains the tests and results for the application that are mainly intended to proof that the stated functionalities work as expected.

In \textbf{chapter 7} there is the future work that may be done after this project. The main objective of this section is to propose the improvements and additional functionalities that can be implemented on the proposed application. Finally, in \textbf{chapter 8} there are the conclusions that are deduced from the development of this project.
Chapter 2

State of the art

As is stated in the introduction, there are many possible applications when using smartphones in automotive. A proof of that is the existing work on this field. In this chapter will be presented the most important of the existing work related with the purpose of this project. The existing work is divided in two main sections. The first section is about the existing smartphone applications that are already available for the users. The second section will be devoted to explain the most relevant proposals and articles related with this work.

2.1 Overview of mobile applications currently available for the end users

There are a lot of Android applications that implement features related with vehicular environments. If we search in the main smartphone application repositories we can find a wide variety of them. The purpose of this section is to show the wide range of application features that are already implemented in the market. In this way we can also see the capabilities of using a smartphone with the vehicle.

The most common vehicular feature that the drivers use with the smartphone is routing. People are being used on getting the routes from an smartphone application and get the indications when the driver does not know how to arrive to a desired destination. A widely used application that implements this feature is Google Maps, despite it is not only focused in automotive.
One of the most famous vehicular oriented applications is Waze [2]. This application is mainly a social network focused only in driving. It uses collaboratively made road maps in order to calculate routes and guide the users to the desired destination. This application uses the GPS system to locate the users and calculate the current speed. From this data, the application is able to compute the traffic situation from the users state and allow other users to recalculate the routes according to this data. In figure 2.1 there is a capture of the application map with the calculated route, the road events and the location of the other users that are currently using the application.

![Figure 2.1: A capture of the route map of the Waze application. [2]](image1)

Waze also allows the users to report road events such as an accident, the presence of the police, fuel stations and prices, a bad road state and multiple more customized alerts. In figure 2.2 there is a screen capture of the application showing how the user reports a
road event or situation. This application proofs the utility of using collaborative systems to provide road information.

Another application similar to the former one is Inrix [3]. This application uses also a collaborative way to get the traffic status and road events. It also analyzes traffic patterns to forecast future traffic situations. This information is used to compute routes according to the user driving patterns and usual trips. In figure 2.3 there is a screen capture for a detected congestion event.

![Congestion event information in the Inrix application.](image)

Figure 2.3: Congestion event information in the Inrix application. [3]

Apart from the applications that improve the user navigation and road information there are other vehicle oriented applications that focus on reading data from the car as well as reading failure and trouble codes. This type of applications are commonly developed to use an On-Board Diagnostics second generation (OBD-II) to Bluetooth device to extract the information from the car Engine Control Unit (ECU) or Powertrain Control Module (PCM). The most known application in this field is the Torque Pro [4]. This application is able to read data (e.g. vehicle speed, engine speed, intake air temperature, mass air...
flow...) and Diagnostic Trouble Codes (DTC) from the OBD-II interface. Torque Pro can also record the smartphone accelerometers data and measure the GPS location to provide track logs of the recorded data. In figure 2.4 there are two screen captures of the Torque Pro where we can see some real time data displays and the trouble codes list. What is interesting in this kind of applications is the possibility of getting real time data from the vehicle by only connecting a simple device on the OBD-II port of the vehicle. This is an economic way to improve the vehicle dashboard features wirelessly with a simple smartphone through the Bluetooth or Wi-Fi interface.

![Figure 2.4: Torque Pro screen captures showing the real time data displays and the trouble codes list. [4]](image)

The Android application DriSMo [5] is related with the safety while driving. This application uses the smartphone acceleration sensors to determine and monitor the driving skills of the user. In figure 2.5 there is a screen capture of the panel shown while driving that indicates the level of safety according to the measured accelerations. In figure 2.6 there is the statistics screen capture for the recorded trip where the user can check the safety grade while driving.
Chapter 2. State of the art

Figure 2.5: A screen capture of the DriSMo application while driving [5]

Figure 2.6: A screen capture of the statistics panel of the DriSMo application. [5]
Related also with safety there is also the Android applications iOnRoad [6] and Drivea [7] that measure the distance to the front car by using the smartphone camera and processing the image. In case the application detects any risk, it notifies the driver. In figure 2.7 there is a screen capture of the iOnRoad application where it is seen how the application computes the distance from the front vehicle and computes some statistics about it.

![Image of iOnRoad application](image)

**Figure 2.7**: A screen capture of the iOnRoad application while driving [6]

Apart from the former applications, there are other available ones. But we have only mentioned some examples to be aware of the many different possibilities of using the smartphone in automotive.

### 2.2 Related research works in the field

The use of smartphones for the vehicle driving scope has also been a topic under research. There are several proposals and publications related with car routing, accident detection, driving behavior estimating, fuel consumption efficiency, etc. In this section there will be shown some of the more interesting research works in this field. They are classified in different groups depending on the purpose.

#### 2.2.1 Environmental efficiency

Related with being efficient with the environment there is a proposal called Driving Coach [8]. In this publication, the authors state that a study made by Toyota said that
the user behavior can vary the fuel consumption up to a 20%. This is one of the reasons that they consider highly important to reduce the consumption as much as possible and they saw that using a smartphone is a good way to implement it. So, the authors have proposed an algorithm to determine the efficiency in fuel consumption of the vehicle user. To implement this algorithm, the authors have developed an Android application called Driving Coach. This application pretends to analyze the fuel efficiency and by using their algorithm and from that the application will give some tips to the driver in order to reduce the CO\textsubscript{2} emissions.

The Driving Coach application uses data from the OBD-II port of the car and also from the smartphone sensors as it is shown on figure 2.8. It reads in real time the vehicle speed, the throttle position, the accelerations, the altitude, the instant fuel consumption and the engine rotating speed. From all this data, the algorithm computes the driving condition to determine the type of road where the user is driving, the evaluation of the fuel consumption of the vehicle and a driving hint to be shown to the user in order to provide a recommendation to reduce the fuel consumption.

![Figure 2.8: Driving Coach implementation diagram.][8]

Another work related with fuel consumption efficiency is in a publication from Muñoz-Organero and Magana [9], where the authors implement and validate an application for this purpose. They propose the use of an Android mobile device with a camera to detect
and recognize the traffic signs to afterwards compute the optimal deceleration patterns to save fuel.

Once the application detects a known traffic sign that forces the driver to reduce the speed, the algorithm estimates the distance required to stop the vehicle without using the brakes according to the rolling resistance coefficient and the road slope angle. A feedback advice is given to the driver in order to release the accelerator pedal. It also records the information on a database to know the sign location in the future. A big advantage is that this application can be used without the need of any any extra road infrastructure and it can be used in any vehicle.

A different approach to improve the drivers fuel consumption is shown in a proposal called Carbon Recorder [10]. In this article, the authors propose a smartphone application to record the CO$_2$ emissions and share them through the social networks. The main purpose is to encourage an efficient driving behavior and raise the social awareness of the emissions.

Related to the environmental efficiency, in [11] there is a proposal to improve routing algorithms by including the energy consumption. The most common algorithms to find the optimal routes are only taking into account the distance, time or even the traffic, but it is not common to include the fuel consumption. So their proposal pretends to implement an algorithm to provide “eco-friendly” routes.

### 2.2.2 Driving safety

There are also proposals focused on provide driving safety by using a smartphone. This is the case of a publication titled *Estimating driving behavior by a smartphone* [12], where the authors propose an algorithm to be implemented on an iPhone device to estimate the driving behavior of the driver. In this case, the application will only use the smartphone sensor data, without the need of taking data from the car sensors or implementing any external sensors.

The proposed application uses the values of the accelerometers, the gyroscope and the magnetometer. With this data, the algorithm is able to detect risky driving situations due to inattention or traffic signs violation. To associate the measured values with a corresponding known driving pattern, the authors use the dynamic time wrapping
(DTW) algorithm and then they determine if the driving habits of the user are safe or risky by using a Bayesian classification scheme. So the application works in three steps, detecting an event by measuring a sharp or sudden maneuver, compare the recorded event with the recorded patterns using the DTW algorithm and finally classify the event by using the Bayesian scheme. These steps are shown in the flow diagram of figure 2.9.

![Flow diagram](image)

**Figure 2.9:** Estimating driving behavior flow diagram. [12]

An approach similar to the former one is the proposed by Johnson and Trivedi [13]. In this proposal, the authors focus also on the driving safety by analyzing the driving behavior using the smartphone sensors. In this case they use the camera, the gyroscope, the accelerometer and the GPS. They use the DTW algorithm as in [12] to determine the type of the events. This application is also only focused on determining if the driving pattern is aggressive or not by detecting aggressive turns, accelerations, braking or lane change.

In the field of driving safety there are also approaches that use the image captured from a camera that can be from a mobile device. In a proposal called *On-road vehicle detection: a review* [14], the authors deal with the feasibility of using the images captured from a camera at the front of the car in order to detect the other vehicles.

A similar approach is considered in the *SmartLDWS* system [15]. In this case they use the image captured from the camera to detect a lane departure. The authors propose a system to warn the driver in case a possible lane departure is detected.
2.2.3 Accident detection and reporting

Some research works related in introducing smartphones in the automotive sector focus on the detection and reporting of accidents. This is the case of WreckWatch [16] and the proposal from Zaldivar et al. [17]. Both of them focus on the importance of the quick response of the emergency services on a car accident in order to increase the probability of survival.

In WreckWatch [16] the authors have developed an smartphone application called WreckWatch that implements a mechanism to provide accident detection and notification by using the embedded smartphone sensors and communication interfaces as is shown in figure 2.10.

![Figure 2.10: WreckWatch accident detection and notification diagram. [16]](image)

This work includes a complete development organized in three different parts: presentation of a formal model for accident detection that combines sensors and context data; show how smartphone sensors, network connections, and web services can be used to provide situational awareness to first responders and provide empirical results demonstrating the efficiency of different approaches employed by smartphone accident detection systems to prevent false positives.

It is interesting the fact that they consider the possibility of sending with the accident emergency notification some information about the context and circumstances of the accident. This can help in the reconstruction of the accident. Some of the data that they consider to be sent on an accident detection is the GPS location, the speed before the crash, the accelerations measured before and during the crash and an acoustic record of the event.
This study is also useful for the empirical results. The authors have made several tests in order to determine the acceleration threshold to avoid false positive and false negative detections. The results are very promising in the sense that they have denoted that the difference between the acceleration of a car accident and any other situation that can occur on a smartphone is very different.

In the case of the proposal from Zaldivar et al. [17], they develop a similar system but using additionally data from the OBD-II interface of the car. The authors have developed an Android application that in case of accident detection sends an SMS with relevant data about the accident and starts an emergency call. To detect an accident it uses a very simple strategy that consists on checking through the OBD-II interface if the airbag has been triggered or not and checking if there is an acceleration detected greater than 5g.

Although, according to WreckWatch [16] the required acceleration to trigger the airbag deployment is above 60g. In this sense it seams that it is not worthy to keep tracking if the airbag is deployed or not through the OBD-II interface, detect accidents by sensing the smartphone accelerations in an appropriate way should be enough.

Related to this functionality, there is an European initiative called eCall [18] that pretends to bring rapid assistance response in vehicle accidents that take place anywhere in the European Union. The eCall-equipped vehicles will call automatically the nearest emergency center when an accident is detected. The system will send a minimum set of data even any passenger is not able to speak. This data will contain, at least, the precise location of the accident.

### 2.2.4 Other automotive proposals

There are a lot of research projects that deal with the communication between vehicles. The possibility of sharing information among the vehicles that are on a road can provide very useful information and opportunities. These mechanisms are in the field of Intelligent Transportation Systems (ITS), that is a term used to mention the advanced applications that pretend to develop smarter transport networks according to traffic management, safety, etc. It is also common the term Vehicular Ad-hoc Network (VANET) when referring to the scenario of deploying ad-hoc networks among vehicles
that are running by chance on a road to enable them to share information in real time. The IEEE (Institute of Electrical and Electronics Engineers) has already developed an standard for vehicular wireless communications that is the IEEE 802.11p, also known as WAVE (Wireless Access in Vehicular Environments).

The main problem of these solutions is that they have to face the difficulty of implementing them to the real vehicular environments. This is because there is the need of a wide deployment to take a good profit from the use of these intercommunication proposals.

The former situation leads on analyzing the possibility of using the users smartphones to share information and communicate among the users. This is what the authors do in [19]. As it is an interesting proposal, there is an explanation about the procedure that the authors followed. In this article the authors analyze the possibility of using the Wi-Fi interface implemented on an smartphone to communicate to other vehicles that also have an smartphone. Some of the recent Android smartphones (Above Android 4.0 version) are implementing a technology called Wi-Fi Direct, that is away to communicate between two devices through Wi-Fi without the need of an access point (AP).

So, in this article, the authors try to develop an application to experiment on using the Wi-Fi Direct interface to communicate between vehicles. They have faced the problem that, for security reasons, the Android operating system requires the user to accept the connection. Another drawback is the fact that it does not allow ad-hoc connections, it creates an infrastructure network instead of a distributed one. For these reasons and also to allow the application to work on previous Android versions, the authors tried to establish the connection by using the Wi-Fi AP/Client mode instead of using Wi-Fi Direct. In this case they were able to connect automatically by previously coding the AP password. By using this, the application could work from the Android 2.2 version. With this new scenario, the authors found that the time to set up the connection was between 1.7 and 3 seconds, that is quite long for a vehicular environment.

After that, they tried to use Wi-Fi working in an ad-hoc mode. As manufacturers have disabled the possibility of using the Wi-Fi in ad-hoc mode, they had to root the Android devices to allow it. With this configuration the devices could send and receive UDP broadcast packets immediately with the previous setup ad-hoc environment. This scenario allows multi-hop routing protocols. With this system they get good results,
but it has the difficulty of the required modifications on the default Android operating system that manufacturers provide.

There are other approaches that experiment the Wi-Fi interface to provide vehicular communication. In [20], the authors present an Android application to use Wi-Fi as a Vehicle to Infrastructure (V2I) communication. The idea that they present is about using the wireless network access points available in urban areas. The application will send useful information through Wi-Fi to a centralized server and this server will provide information to the user about the state of the vehicle surroundings. An important drawback of this application is that it requests a wide Wi-Fi infrastructure, that in some cases may be expensive to implement.

Related with using Smartphones to implement VANETs, in [21], the authors experiment on the feasibility of using the smartphone IEEE 802.11a/b/g instead of the IEEE 802.11p. In this article, they do not develop any smartphone application, they only measure in different scenarios the conditions of the Wi-Fi signals and evaluate the technical feasibility. They find it feasible only in some circumstances, mainly in highways.
Chapter 3

Context description

The purpose of this chapter is to introduce the context that involve this project. It consists of a description of the technologies used to develop the application. There are two main sections, the first one is dedicated to explain the Android operating system and how it works and the second is devoted to the description of the vehicle On-Board Diagnostics standards and utilities.

3.1 Android

Android is an operating system based on Linux that is designed for mobile devices. The Android platform comes from the company group called the Open Handset Alliance [22] and is led by Google. Google is in charge of the Android Open Source Project, that maintains and develops Android. Android is an open source platform available for everyone interested, including device manufacturers and software developers, to implement their ideas without any kind of restrictions or controls that any private company could try to impose. To ensure all this open permissions, Android is released mostly under the Apache License 2.0.[23]

The openness of the project that allows the implementation of the operating system by multiple mobile device manufacturers is what results in a big variety of Android devices with a wide cost and category range. As the SDK is also free and easy to install and use across multiple platforms, it is attractive for the application developers to implement
their own ideas leading to a very rich Android application market. All this facts result in an important growth of the Android mobile devices sales as it is shown in figure 3.1.

The wide variety of Android devices could be a drawback, but the truth is that all Android devices have a similar hardware, that means that they are implementing the same type of sensors, communication standards and other features. In addition, the Android operating system makes easy to develop Android applications in the sense of they can be easily adapted for each device.

The Android platform has been chosen for the development of this project due to the openness of the resources and the features or capabilities of the existing Android devices. It also has been chosen because of the popularity of the Android devices, that makes the applications to be released easily and to be used by a lot of users. In the following sections there are described the main features that will be used in this project.

3.1.1 Android application fundamentals

Android applications are mainly developed in the Java programming language. The application source code is compiled by the Android Software Development Kit (SDK) into an file with the .apk extension. This type of files are the ones that the Android devices use to install the applications. This file is required to have all the compiled code, data resources, images, etc. for a whole application. [25]
The Android operating system executes each application in a different process. Each process has its own virtual machine in order to do not interfere with each other. It is known that the Android platform is designed to be executed mainly on a mobile device, that means that it should be efficient in energy consumption. For this reason, the operating system will suspend all the applications in the memory when they are not active. The suspended applications will remain without consuming resources until they are requested again. However, any application can be killed permanently when the system detects that the idle memory is too low. The applications that have been more time idle are more likely to be killed.

Android applications are mainly made by different essential building blocks, that are called application components. Each component defines a way to enter the application. Android defines four types of components:

- **Activities**: Each activity corresponds to a screen of the application and implements the methods related with the layout.

- **Services**: A service works in a similar way as an activity, but without a user interface. It is used to run long time background operations.

- **Content providers**: This component is used to access or even modify data from outside the application. (e.g. to access and/or modify the contact information stored in the device)

- **Broadcast Receivers**: It is a component that can be executed when a system-wide broadcast message is sent from any point of the system. It can be used to wake up the application on a specific event (e.g. on a battery state change or time change).

In the following subsections the activities and services will be explained as they are the only components used in this project. There will be also the description of the layout files and the Android Manifest file.

### 3.1.2 Android Activities

The activity is the type of Android component that is in charge of the user interface items management and the user interaction events configuration. Each screen has an
associated activity. An application usually has a main activity that is the one presented when the application is launched. Typically, applications may have multiple activities, because they usually have multiple screens. Activities are usually started from other activities when the user presses a button or a specific event happen.

When an activity is started from another activity, the first activity is paused and the second activity shows up. The sequence of activities is stored in a stack where preserves the state of previous activities. When the user pushes the back button, the current activity is finished and the top activity of the stack is shown up again. The stack works as the well known last in first out queuing mechanism.

Each Android activity has a lifecycle. Activities should implement callback methods that are automatically executed on changing the state of the activity. In the figure 3.2 there is a diagram that explains graphically the Android activity lifecycle.

![Figure 3.2: The Android activity lifecycle.][25]

An activity can have three possible states that are *resumed*, when the activity is visible and has the focus; *paused*, when another activity goes to the foreground and gets the
focus but the activity is still visible, and stopped, when the activity is completely hidden by another activity.

When an activity is started for the first time, the methods `onCreate()`, `onStart()` and `onResume()` are executed. If another activity starts over the current activity, the first method that is called on the first activity is `onPause()`. When an activity is not on the foreground it is paused and will be kept on the memory. If a foreground activity does not have enough memory, the activity that is stored in the background will be destroyed. That means that if on an activity that goes to the background there is any critical information that needs to be saved is better to do it at the `onPause()` method.

At the same time, if there is some information on an activity that needs to be reloaded or reconfigured each time it gets the focus, it should be done at the `onResume()` method. This method will be executed when an activity recovers the focus, is started again or is created for the first time.

Android activities are directly related with a user interface layout. For this reason, on starting an activity it is needed to load the corresponding layout. It is commonly done at the `onCreate()` method. The layout is what contains the position of each view object. The layout is explained in section 3.1.4. After loading the view, the activity may configure the view objects such as buttons, spinners, text edits, text views and implement the corresponding listeners. The listeners will be in charge of executing the required operations upon an event has happened. For example, a button may call an `OnClickListener` when it is pressed. In this example, the `OnClickListener` will execute the desired commands such as starting a new activity or showing a specific message.

### 3.1.3 Android Services

An Android service is a component similar to an activity but without any user interface. This component is intended to execute long time operations in the background. Services can be started from other application components and these services can also be able to start other application components. Other components might be able to bind to a service in order to exchange information or interact with each other. It could be the case, for example, if a service is in charge of managing a connection to retrieve some data from Internet, an activity component could bind to the service to display this data.
As an activity, a service has a lifecycle. In figure 3.3 there is the Android service lifecycle. As it is shown in the figure, services can have two different types of lifecycles depending on the way that they are started.

![The Android service lifecycle](image)

**Figure 3.3:** The Android service lifecycle. [25]

The two different lifecycles that a service can have depend on the method that has been used to start them. There are two possibilities to start a service:

- Calling the `startService()` command
- Calling the `bindService()` command

If the service is started by calling the `startService()` command, the lifecycle is similar to an activity. When the service is started, the `onCreate()` and `onStart` callback methods will be executed. A service started in this way will be finished when it is required by itself or by a service client. Typically, this services run a single operation that do not require any communication with the starter and they stop themselves. When a service is stopped, it executes the `onDestroy()` method.

When a service is called by using the `bindService()` command, it has a different lifecycle. On binding a service from another component, a client-server interface is created to interact between them. A bound service is active only when there is another application
component that is bound to it. There can be multiple components bound to the same service. The first component that binds to the service is the one that starts the service. In this case situation is where the `onCreate()` method will be executed. The service will be destroyed automatically and call the `onDestroy()` method when any component is bounded to the service.

There is a specific implementation of an Android service that is called foreground service. This type of services are considered to be executed with the awareness of the user. This services are not candidates to be killed when the system is in a low memory situation. That is why, while the service is in the foreground, an ongoing notification must be provided in order to ensure the user awareness. An example of foreground service could be the music player, that the user may not be in a music player application activity but he or she is completely aware of its operation.

3.1.4 User interface

The user interface of an Android application is composed by View and ViewGroup objects. Each View object is an element of the user interface that a user can interact with, for example a button, a text view, a progress bar, an image, etc. A ViewGroup is an object that contains View objects and it is in charge of defining the layout interface. Each ViewGroup can contain other ViewGroup objects as is shown in figure 3.4.

![Figure 3.4: The Android view hierarchy that defines the user interface layout. [25]](image)

Developing an Android application there are two main ways of managing the user interface. The first one is by the use of eXtensible Markup Language (XML) files and the second one is at runtime, programmatically. Typically, the user interface is defined in XML layout files, as they can define the default properties of each View object. A
different XML file can be defined for each type of device screen or even different languages. In this way, the application can be installed and used successfully in a wider range of devices and users. By using the XML files to define the layout is good in order to separate the user interface presentation with the application code.

It is common to interact with the user interface also programmatically at the same time as using the XML files. This is done in order to dynamically modify the layout objects and configure the user interaction with the interface View objects. This operations could be, for example, changing the text of a text view or configuring a click listener of a button to define the operations to be executed on pressing the button.

There are other ways to interact with the user. There are the status bar notifications that can be shown at any time on the status bar to notify the user of some event or about an ongoing process. There are also the dialogs, that are small windows that are prompted over the current activity without filling the screen usually to request a user confirmation or information. Another user interface item type is the Toast. A Toast is a simple pop-up message that is shown for a limited amount of time.

3.1.5 Android Manifest file

The Android Manifest file is an XML file that is required in any Android application. It contains all the information needed to install and run the application. The main part of the Android Manifest is the component description. The Manifest must contain the list of all the Android components (Activities, services, content providers or broadcast receivers) that the application use. Each component will be related with the class that implements it and the conditions under these components will be executed, such as the screen orientation of an activity or the permission to execute a service from the system.

The Android applications, by default, are not able to access any other part of the system outside the application components. To have access to any system resource or content, the application must declare the required permissions in order to let the user be aware of the application scope. All these permissions must be declared in the Android Manifest file. Example of these type of permissions can be the ability to send SMS messages, access to Internet, manage the Bluetooth interface, etc.
This file will also contain other information like the Android versions that the application is available for. It will declare also the permissions that other applications can have to interact with the application. The main activity will also be set in the Manifest in order to be shown at first when the application is launched.

### 3.1.6 Android location strategies

If an application is able to know the location of the user, it can provide smart features that can be very useful to the user. For example, being able to locate the Android device, a weather forecasting application can provide a customized weather forecast for the specific user location. It also can be useful for navigating systems, where locating automatically the user can allow a map application to compute a route to go to another location.

Android powered devices have two ways of locating the device. The most known and the most accurate one is the GPS (Global Positioning System). However, it is known that the GPS locating mechanism can only be used outdoors and it consumes a lot of energy.

The other available location strategy is by using the network. The Android Network Location Provider is able to obtain an approximate user location from the cell towers and Wi-Fi signals. Locating the user with this strategy can work also indoors and the location response is much more faster than the GPS mechanism.

The Android location strategies basically provide the geographical location by giving the geographical coordinates (latitude and longitude). However, Android provides mechanisms to easily convert any coordinates into an address with several information like the locality, the country, the postal code or the altitude. It can be very useful for giving human readable information about a specific location.

As reading the user location can be a privacy concern, if an application uses any location strategy it is required to define the corresponding permission at the Android Manifest file.
3.1.7 Android sensors

A fact that contributes in making Android powered devices really powerful is the several built in sensors they have. Android devices are commonly manufactured with several sensors that can be used in any application. The Android operating system is already designed to manage all these built in sensors with an easy to use framework for Android application developers.

As there are many possible sensors to be implemented on an Android device, they are divided in three main groups:

- **Motion sensors**: These sensors measure the forces applied over the device, that means acceleration and rotation forces. The most known sensors are the accelerometer and the gyroscope.

- **Environmental sensors**: It is the type of sensors that measure environmental parameters like temperature, light, humidity, etc. Typical environmental sensors are the barometers, thermometers and photometers.

- **Position sensors**: Position sensors measure the parameters that determine the physical position of the device. the most common sensors are the magnetometer and the proximity sensor.

The Android operating system can provide parameters from sensors that are called either software based or virtual sensors. That means that the data acquired calling these sensors is not data measured directly from a hardware sensor. In this case, the values can be computed from several hardware based sensors.

An example of that is the orientation sensor, that combines the data from the magnetic field sensor and the accelerometer. Other cases of virtual sensors are the linear acceleration sensor and the gravity sensor that both compute their values from the accelerometer hardware based sensor.

A fact that makes easier to develop sensor based Android applications is the fact that the Android sensor framework provides very accurate values from the sensors and well formatted in units of the SI (International System of Units). For example, in the case of the accelerometer, the values are in $m/s^2$ in a three dimensions vector for the three
physical axis (x, y and z). The physical axis distributions with the device as a reference are as shown in figure 3.5.

![Figure 3.5: The Android sensors physical axis distribution. [25]](image)

### 3.1.8 Android connectivity

A good point of using Smartphones in vehicular environments is the multiple communication interfaces that are available in these devices. Android devices can have access to Internet at least through the mobile access networks (GSM, HSPA...) or Wi-Fi. They can also communicate for example through Bluetooth or Near Field Communications (NFC).

In Android, the Internet access is easy to implement, because the operating system is in charge of managing the mobile access networks. It is only needed to declare the Internet access permission in the Android Manifest File (explained in section 3.1.5) and the application is able to access Internet, for example using the HTTP protocol. The Android SDK has already some easy to use APIs to access the Internet.

It is also possible to manage the Wi-Fi connections from the application. By stating the convenient permissions on the Android Manifest file an application is able to change the Wi-Fi state by switching it on or off and connecting or disconnecting from a specific Wi-Fi access point by knowing the corresponding SSID and password.

Another useful communication interface is Bluetooth. It is very useful to communicate with devices that are in a short range. In this project, the Bluetooth interface will be used to communicate with an OBD-II adapter to get data from the vehicle ECU. The main advantage of using Bluetooth is that it is easy to communicate in a similar way
as a wired connections but with the advantage of that no wires are required. A wired connection could be annoying in some situations, specially when using smartphones. The Android SDK provides very useful APIs to manage the Bluetooth connections. It provides some methods to manage point-to-point communications between Bluetooth devices that will be very useful for the application developed in this project.

3.1.9 Android SQLite

A relevant feature of the Smartphones is the Storage capacity. To store data it is common to use a database, because it is easy to store and retrieve data from it. In the case of Android devices, the most commonly used database are the SQLite. The Android SDK provides some mechanisms to easily manage the SQLite databases and retrieve and store data to them. The SQLite databases have the feature that each database is stored in a single file. The SQLite databases will be accessible from any part of the application, but not from outside. In order to share the information among different applications, a content provider component should be used.

3.2 On-Board Diagnostics

The term OBD (On-Board Diagnostics) refers to the systems of self-diagnostic and reporting capability on modern vehicles. The OBD system is intended to give to the vehicle owner or the technicians access to the status of various vehicle subsystems. The first implementations of OBD systems took place in the early 1980s. At that time, manufacturers began installing processors to handle fuel and ignition functions on vehicles. From then, the functions of the on-board processors have been increasing. As at the beginning, the diagnostic systems only provided a simply malfunction indicator light, recent versions of the OBD can provide real-time data and standardized diagnostic trouble codes (DTC) that allow the fast and precise identification of the engine failures and problems. Most of the standards related with the OBD are defined by the SAE, that is the international Society of Automotive Engineers. [26, 27]

Was in 1996 when in United States of America there was the obligation for the car manufacturers to implement the OBD-II (On-Board Diagnostics, second generation) standard in all the cars. The OBD-II standard was an improvement of the first OBD
systems that was focused on the emission testing strategy. In Europe the standard, based on the OBD-II, was called EOBD (European On-Board Diagnostics) and it was imposed in Europe in 2001 for petrol cars and in 2004 for diesel cars. OBD-II and EOBD standards are essentially the same. From now on, in this written work, it may be used the OBD abbreviation to refer either OBD-II or EOBD standards. [28, 29]

### 3.2.1 J1962 connector

The OBD-II standardization forced to replace the wide variety of diagnostic connectors into a common 16-pin connector called J1962, which is generally in the same place on most vehicles, under the steering wheel or near the driver seat. In figure 3.6 there is how the connector looks like in a car.

![Figure 3.6: The location of the OBD-II connector in a car.][30]

In figure 3.7 there is the J1962 connector pinout and in table 3.1 there is the description of each pin. As we can see, there are some pins that are manufacturers discretionary, so that, the manufacturers can use them for any purpose. In the J1962 connector there are different pins dedicated to different communication standards. That is because the OBD-II standard allows five different signal protocols. Most of the vehicles use only one protocol, that can be deduced by looking at the J1962 connector and seeing which pins are present and which not.

![Figure 3.7: The pinout of the J1962 OBD-II connector in a car.][31]
3.2.2 Signal protocols

The OBD-II/EOBD standardized vehicles can choose between out of five signal protocols specified in the standard. Most of the vehicles use only one protocol. Depending on the used protocol, different pins of the J1962 connector are used. The five protocols are the following: [27]

- **SAE J1850 PWM**: pulse-width modulation — 41.6 kB/sec, standard of the Ford Motor Company. Uses the pin 2 (Bus +) and pin 10 (Bus -) of the J1962 connector.

- **SAE J1850 VPW**: variable pulse width — 10.4/41.6 kB/sec, standard of General Motors. Uses the pin 2 (Bus +) of the J1962 connector.

- **ISO 9141-2**: asynchronous serial data in 10.4 baud, similar to RS-232. Uses the pin 7 (K-line) and optionally, for wakeup, the pin 15 (L-line) of the J1962 connector.

- **ISO 14230**: uses the Keyword Protocol 2000 (KWP2000) with the same physical layer as ISO 9141-2 with a data rate between 1.2 to 10.4 kBaud. It also uses the pin 7 (K-line) and optionally the pin 15 (L-line) of the J1962 connector.

- **ISO 15765 CAN**: controller area network bus. It can provide 250 kBit/s or 500 kBit/s. Unlike the other protocols it is widely used outside the automotive industry. It uses the pin 6 (CAN High) and pin 14 (CAN Low) of the J1962 connector.

<table>
<thead>
<tr>
<th>Pin number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Manufacturers discretion</td>
</tr>
<tr>
<td>2</td>
<td>SAE J1850 Line (Bus +)</td>
</tr>
<tr>
<td>3</td>
<td>Manufacturers discretion</td>
</tr>
<tr>
<td>4</td>
<td>Chassis Ground</td>
</tr>
<tr>
<td>5</td>
<td>Signal Ground</td>
</tr>
<tr>
<td>6</td>
<td>SAE J2284 (CAN High)</td>
</tr>
<tr>
<td>7</td>
<td>K Line of ISO 9141-2 &amp; ISO/DIS 4230-4</td>
</tr>
<tr>
<td>8</td>
<td>Manufacturers discretion</td>
</tr>
<tr>
<td>9</td>
<td>Manufacturers discretion</td>
</tr>
<tr>
<td>10</td>
<td>SAE J1850 Line (Bus -)</td>
</tr>
<tr>
<td>11</td>
<td>Manufacturers discretion</td>
</tr>
<tr>
<td>12</td>
<td>Manufacturers discretion</td>
</tr>
<tr>
<td>13</td>
<td>Manufacturers discretion</td>
</tr>
<tr>
<td>14</td>
<td>SAE J2284 (CAN Low)</td>
</tr>
<tr>
<td>15</td>
<td>L Line of ISO 9141-2 &amp; ISO/DIS 4230-4</td>
</tr>
<tr>
<td>16</td>
<td>Unswitched Vehicle Battery Positive</td>
</tr>
</tbody>
</table>

Table 3.1: Pin description for the J1962 OBD-II connector. [26]
Nowadays, the CAN protocol is the fastest and the most powerful OBD-II/EOBD protocol. That is why many manufacturers from the United States have been migrating their cars to use CAN since 2003. In fact, all vehicles sold in the United States since 2008 are required to implement CAN as one of their signaling protocols.

### 3.2.3 Diagnostics Information

The Engine Control Unit (ECU) is the device that will provide the data to the OBD-II/EOBD interface. The method to request the data is defined by the SAE J1979 standard. This standard lists a big amount of parameters that can be available from the ECU. Not all the parameters are required to be implemented and the manufactures can include their own proprietary ones that are not listed in the J1979 standard. To request the parameters through the OBD interface the J1979 standard defines the parameter identification numbers (PIDs) to address them. By requesting and retrieving data through the OBD interface using PIDs, we can have access to real time performance data and to the recorded diagnostic trouble codes (DTCs). As well as the manufacturers can implement their proprietary parameters, they can also define and implement their own DTCs to freely enhance the OBD code set.

The PIDs are divided in ten operational modes. Each operational mode is defined to request different types of data. The PIDs are always preceded with a byte that is the mode identifier. The operational modes are the following:

- **Mode 01**: This mode is used to retrieve the real time performance data, for example the vehicle speed, the engine rotation speed, the mass air flow, the intake air temperature, the throttle position, etc.

- **Mode 02**: The 02 mode is used to retrieve freeze frame data. If a DTC is fired off, the freeze frame data will be showed in this mode.

- **Mode 03**: This mode is queried to retrieve the stored DTCs that have been already fired off.

- **Mode 04**: It allows to clear the DTCs and stored freeze frame data.

- **Mode 05**: Test results of the oxygen sensors monitoring only for non CAN vehicles.
• *Mode 06*: Test results for other components and systems. Also oxygen sensors in CAN vehicles.

• *Mode 07*: Show the pending Diagnostic Trouble Codes detected during current or last driving cycle.

• *Mode 08*: Control operation of the on-board components or systems.

• *Mode 09*: It is used to retrieve static information of the vehicle such as the Vehicle Identification Number (VIN), the calibration verification number, etc.

• *Mode 0A*: Lists the permanent diagnostic trouble codes, that are stored into a non-volatile memory. This corresponds to the DTCs that have turned the Malfunction Indicator Lamp on.

The DTCs have an specific format that indicates the type of failure and the specific fault. A DTC consists of five characters, a letter and four numbers, for example P0133. The first letter indicates if the trouble is related to the body (B), to the chassis (C) to the powertrain (P) or to the network (U). The codes related with the power train are the most common. The next character can be 0 (indicating that is a standardized generic code) or 1 (indicating that is manufacturer specific). The third character will be a number that indicates the specific vehicle system (fuel and air metering, ignition system, transmission, vehicle speed controls, etc.). The last two characters will be the specific fault identifiers.
Chapter 4

Tools

In this chapter, there are presented the main tools used in the development of the project and also the ones required to use the final application.

4.1 ELM327 adapter

In section 3.2 we have explained the On-Board Diagnostics standard, that all the vehicles must implement. However, the standard has the drawback that any communication protocol is not directly compatible with common computers or mobile devices. To overcome this, Elm Electronics developed a chip, the ELM327, to act as a bridge between the OBD connector and the standard RS232 serial interface. This chip is an appropriately programmed PIC18F2480 microcontroller from Microchip Technology. [31, 32]

4.1.1 ELM327 overview

The ELM327 chip is ready to be connected directly to the OBD connector with some external hardware that is already specified in the datasheet. As we can see in the ELM327 pinout (figure 4.1) there are pins for each OBD connector pin. We can also see that there are two pins (RS232 Rx and RS232 Tx) that are in charge to carry the RS232 communication to the computer or another device. The ELM327 chip has also several pins to be connected directly to the LEDs that will indicate the transmitting or receiving through the OBD or RS232 interfaces.
Recently, RS232 interfaces are not much supported in laptops, that are the computers most likely to be connected to an OBD interface. For this reason the most of ELM327 implementations include a bridge to adapt the RS232 to a most supported interface. Some implementations use USB but the most recent are implementing Bluetooth or Wi-Fi interfaces in order to be able to communicate to the ELM327 wirelessly.

In figure 4.2 there are two examples of the common implementation of the ELM327. On the left there is an implementation by adapting the RS232 to USB and on the right there is the implementation by adapting the RS232 to Bluetooth. To be used with smartphones, the most common implementations are using Bluetooth or Wi-Fi interfaces. In this project the Bluetooth will be used to communicate with the OBD interface so the application will be developed to use an ELM327 to Bluetooth adapter.
4.1.2 How to use the ELM327

The ELM327 based devices may implement RS232 adapters to other technologies. Anyway, the connection must follow a serial communication protocol emulating RS232. The communication with the ELM327 device is done by using ASCII characters. The ELM327 device will be in charge of adapting the received ASCII characters to the OBD standards.

When the ELM327 device prompts the ASCII ‘>’ character, it means that the device is in idle state and it is ready to receive any command. Each command must be sent with a carriage return at the end, otherwise, the command will not be executed. If only a carriage return is sent, the device will execute the last executed command. If an invalid command is sent, the ELM327 will return a single question mark. The chip is not case sensitive and it will ignore the space characters.

To send OBD commands, the characters have to be the corresponding ASCII characters of the hexadecimal numbers with a carriage return at the end. The response will also be send in the corresponding ASCII character for each number. For example, if we want to retrieve the value of the current engine speed we will send the command ‘01 0C’ in ASCII and we will receive two hexadecimal bytes coded also in ASCII.

The ELM327 is able to receive configuration commands, that are only dedicated for the chip configuration or interaction. This commands are called AT Commands. The AT commands are sent with the first two characters as ‘AT’ and followed by the desired command. For example, if we want to reset the ELM327 device, we look for the reset command in the ELM327 datasheet and we find that the reset AT command is ‘Z’. We will send the characters ‘AT Z’ and the device will do a reset. Other AT commands are for restoring default configuration, setting the desired OBD protocol (including automatic), defining the desired timeout, etc. [32]

4.2 OBDSim (OBD GPS Logger)

The OBD GPS Logger [33] is an application that contains several tools related with the OBD. The main feature of this application is the ability to log driving trips to an SQLite database. It requires a connection to the OBD interface of the vehicle and a
GPS module to track the trip. Some functionalities are available for the trip records like exporting it to Google Earth, but the one that is more interesting in our project is the OBDSim.

The OBDSim is a module of the OBD GPS Logger that its main function is to emulate an ELM327 device (explained in section 4.1) connected to the OBD interface of a vehicle. This tool is very useful for the development of OBD related applications because it enables the possibility of testing the ELM327 device communication without the need of a vehicle and an ELM327 adapter. A good feature of using this module is the fact that it will prompt in the terminal screen a log of the communication. Mainly it will show the connection or disconnection of the application to the ELM327 emulator, the ELM327 AT commands sent to the emulator and the total queries and samples rate for each ten seconds.

To emulate an ELM327 device connected to the OBD interface of a vehicle there is the need of a source of data that will be provided when the device is queried. There are different ways to provide this information. In figure 4.3 there is the user interface that can be used to provide input data to the OBDSim, this can be used for the user to provide specific values to the simulator. The problem of using the user interface is that the number of available parameters is limited and the provided data is not realistic like a real trip. Another way to get the real time information is in a random way, that is a way to keep the values changing in time, but the values received are not at all realistic and neither correlated.

![Figure 4.3: The user interface used to provide input data to the OBDSim tool.](image-url)
Another possibility, and the most realistic one, is to use a previously logged trip. The same SQLite file format recorded by the OBD GPS Logger application can be used by the OBDSim to provide realistic data with the simulator. At the application developer web page [33] there are some recorded trips available that can be used for this purpose.

There are several possibilities to connect to the OBDSim, but the one that we are interested in is to connect through Bluetooth from an Android smartphone. To do that, it is needed to have a Bluetooth interface or a Bluetooth dongle on the computer that runs the OBDSim. OBDSim can only run Bluetooth on a Linux operating system. As the Bluetooth must run on a free channel of the computer Bluetooth it might be needed to modify the source code of the OBDSim to use a different Bluetooth channel from the default one and also set the Bluetooth device address. After that it is needed to rebuild all the application. Lines changed from the source code where in the file ‘bluetoothimport.cc’. They were modified to match the following code:

```c
str2ba("00:80:5A:46:13:39", &loc_addr.rc_bdaddr);
loc_addr.rc_channel = (uint8_t) 14;
```

It is also needed to configure the Bluetooth to use the OBDSim. From the Linux terminal it is needed to execute the following commands as root to configure the Bluetooth channel to the Bluetooth interface:

```
/etc/init.d/bluetooth restart
rfcomm bind 0 00:80:5A:46:13:39 14 
sdptool add --channel=14 SP
```

To start the OBDSim it needs to be done from the command line with the ’obdsim’ command. For example, to execute the simulator using Bluetooth and a previously logged trip with a database file called ’ces2010.db’, we will have to execute the following command:

```bash
obdsim -b -g Logger -s ces2010.db
```

Option -b is used to indicate the use of the Bluetooth interface, -g Logger is used to configure the generator of the data and the -s ces2010.db is used to set the source of the logged data.
4.3 Android Developer Tools overview

The most common integrated development environment (IDE) used in Android application development is the Eclipse IDE. Android provides a plugin to be used with Eclipse, called Android Development Tools (ADT). This plugin integrates all the needed tools to develop Android applications, for example project creation, building, packaging, installation and debugging. It also integrates many Android SDK (Software Development Kit) tools with the Eclipse perspective, for example the *logcat*, that is used to retrieve the log of the applications or the *adb*, that is used to interact with the Android device for example to transfer files or access the device shell. So, to summarize, the ADT is mainly the junction of the Eclipse IDE with the Android SDK. An important advantage of these tools is that it can be used in multiple platforms and it is completely free.

As the Android applications are programmed in Java, the Android SDK integrates all the Java documentation, auto-completion, syntax checking for the Android framework. The ADT also provides a graphical layout editor. It allows to create the layout XML files graphically with a drag and drop interface. It also implement an interface to edit other Android files, for example the Android Manifest file or other XML resource files. The integrated documentation in the Android SDK and the on-line API guides and reference is very useful and makes it easier to develop Android applications. [25]

In order to test the Android applications there are two possibilities, by using an Android Virtual Device (AVD) or by using a real device. Using an AVD has the advantage that there is not the need of using a physical device and the developer has full permissions to access the operating system. In the case of using a real device, the main advantage is that the performance of the application is much more real and faster than an AVD. There is also the possibility of using the real smartphone hardware, like sensors or communication interfaces. In both cases, the debugging functionality is very useful to detect errors and repair them.
4.4 Extra Android libraries

In this section there is an overview of the two external libraries used in this project. These libraries are used to have additional functionalities on the Android SDK. In particular there are used to draw graphs and to communicate with the ELM327 device.

4.4.1 GraphView

The GraphView [34] is a library that implements several resources in order to easily draw graphs in an Android application. It has also the purpose to provide some simple mechanisms to customize and integrate the graphs in an understandable way. It is able to draw line charts and bar charts. the two type of charts may look like it is shown in figure 4.4.

![GraphView](image)

**Figure 4.4: Two examples of the usage of the GraphView library (a line chart on the left and a bar chart on the right)**

This library permits to draw multiple series of data in a single graph. In order to differentiate the series, the color of the graph can be customized and a legend can be displayed. The horizontal and vertical labels are generated automatically, but they can be also customized by overwriting them with a string array. It also has an option to enable the scaling of the graph, that means changing the horizontal scale by the use of the two finger scale gesture. As the scaling feature it is also interesting the scrolling option, that enables the user to move along the graph by touching and moving with one finger.
Appearance customizations are possible mainly in changing the colors of the lines and background or also to change the thickness of the lines. In order to modify the appearance of the text in the graph view, the source code of the library has to be modified as it is done in the library modification from [35], that implement some additional appearance features.

To use the GraphView library it is needed to create an GraphViewData object for each graph value and put it in an array. This array will be used to construct a GraphViewSeries object. This can be done as it is shown in the following example of code:

```java
GraphViewSeries exampleSeries = new GraphViewSeries(new GraphViewData[] {
    new GraphViewData(1, 2.0d)
    , new GraphViewData(2, 1.5d)
    , new GraphViewData(3, 2.5d)
    , new GraphViewData(4, 1.0d)
});
```

The GraphViewSeries object can be constructed with more parameters to define the text label or the color that will identify the bars or the line of this graph series. In order to show the graph series, it is required to construct a GraphView object. A GraphView object is the parent object of the LineGraphView and the BarGraphView objects. So that, on constructing the view object it is required to define the type of GraphView that we will draw, either line or graph. The parameters to create the object are the context and the heading of the graph. The code to construct the GraphView object will look like the following example:

```java
GraphView graphView = new LineGraphView(
    this  // context
    , "GraphViewDemo"  // heading
);
```

After the GraphView object is constructed the GraphViewSeries can be added to the view. The view object has a method to add the series as a parameter. Several series can be added to the same GraphView object. The view object has also several methods to modify the appearance, labels and other features of the graph.

For the development of this project it was also needed to modify the source code of the GraphView library to correct some by changing the source code in the `BarGraphView.java` file as it is shown in the following piece of code:
// BarGraphView.java
// float colwidth = (graphwidth - (2 * border)) / values.length;
float colwidth = graphwidth / values.length;

// GraphView.java
// horizontal labels + lines
int hors = horlabels.length;
for (int i = 0; i <= horlabels.length; i++) {
    paint.setColor(Color.DKGRAY);
    float x = ((graphwidth / hors) * i) + horstart;
    canvas.drawLine(x, height - border, x, border, paint);
    paint.setTextAlign(Align.CENTER);
    if (i < horlabels.length) {
        paint.setColor(Color.WHITE);
        canvas.drawText(horlabels[i], x + (graphwidth / hors) / 2,
                        height - 4, paint);
    }
}

4.4.2 Android OBD Reader

The Android OBD Reader [36] is a library developed for android that its main objective is to implement the main mechanisms to make it easy to configure and use the Bluetooth interface to communicate with an ELM327 Bluetooth device. It is also useful because the code is understandable and can be easily customized. As no documentation is provided, to use this library it is important to analyze and understand the source code in order to be aware about what is the library really doing and avoid unexpected problems.

The way this library works is by defining an abstract class called ObdCommand that implements the basic methods to send a command and to read the response and defines the required abstract methods to be implemented in the ObdCommand class implementations.

As it is explained in section 4.1, it is needed to send the commands in ASCII to the ELM327 device. To do that, the ObdCommand implementations define a String in the constructor with the corresponding command. For example, at the SpeedObdCommand class, the constructor will define the command String as '01 0D', that is the corresponding command to retrieve the instantaneous speed. Once the command object is constructed it can execute the run() method to execute the command and read the result. The run() method need as parameters the Bluetooth input and output streams.
The `run()` method does not return any result, the result is stored at the object attributes. To read the command result there are implemented methods like `getFormattedResult()`, `getResult()` or command specific methods like `getMetricSpeed()` for the SpeedObdCommand. What it is important to know is that the implementation of this library uses the `getFormattedResult()` method to compute the command response and format it in the proper units. That is why if we want to get the result from the `getResult()` method, we have to execute the `getFormattedResult()` method first.

Another issue that is important to know is that the implementation of the `run()` method adds the ‘\r’ character at the end of the command. As explained in section 4.1, the ‘\r’ character is used to indicate the end of the command and if we send this character without any previous command, it will execute the last executed command again. So, the problem of the library is that if we keep executing the `run()` method with the same command, the ‘\r’ character will keep adding and it will send commands like ‘01 0D\r \r \r \r \r’. To avoid that, it is needed to construct a new command object each time a command is going to be executed.

When the command reads the result from the input stream it tries to convert the received ASCII String to hexadecimal values. In some cases, the commands do not return any numerical result, specially when executing AT commands. That is why the `run()` method may throw an exception. A try/catch exception management is needed when OBD commands are executed.

Following the former stated requirements, each time we want to get, for example, the instantaneous speed value we will have to execute the following commands in the same order as they are in the following example:

```java
int speed;
try {
    SpeedObdCommand speedObdCommand = new SpeedObdCommand();
    speedObdCommand.run(in, out);
    speedObdCommand.getFormattedResult();
    speed = speedObdCommand.getMetricSpeed();
} catch (Exception e) {
    e.printStackTrace();
}
```

Another issue to take into account when using the Android OBD Reader library is the need of reading always the result from OBD commands. Even if the commands are not
intended to respond with a result, they always throw some response, for example the 'OK' characters to indicate that the operation has been executed correctly. In the case of the ResetObdCommand implemented on the library, the developers did an override to the superclass reading method and left it empty. The source code has to be modified in order to read and remove the reset command response from the input stream buffer.
Chapter 5

Project description

The main purpose of this project is to proof the concept of using the smartphones to provide features for an intelligent driving. In this chapter there is the description of an Android application that has been developed basically to proof this concept. It is important to say that it is not an application to be distributed directly, it is just a starting point for this new scenario to be considered by the car manufacturers, car rental companies, public transportation managers, etc.

In the following sections, there is the explanation of all the parts of the application. This application has been called Driving Helper. First, there will be the description of the main features and functionalities. After that, the navigation structure description will be presented. Finally there will be a more detailed explanation of the source code, interface and data structure of the application.

5.1 Application functionalities

The application developed in this project has several functionalities that have been chosen in order to proof the capabilities of using the smartphone with the vehicle. In the following subsections there is the description of the different functionalities that the application will implement.
5.1.1 Read OBD data through Bluetooth

The possibility of using the OBD interface (section 3.2) of a car to communicate with the smartphone leads to the feasibility of using the vehicle and the smartphone as a single device. This aggregation concept results on the chance of easily having intelligent cars. The good point of using the OBD interface is that it is available in a lot of vehicles, in fact, nowadays all new vehicles are required to implement this interface.

The application will require the use of an ELM327 to Bluetooth device (section 4.1) that will adapt the OBD protocols to a Bluetooth communication. The possibility of using the Bluetooth technology allows the user to connect the smartphone to the OBD interface without the need of wires. The connection will be used mainly to retrieve real time data from the vehicle, such as the vehicle speed or the engine rotation speed. The communication scheme will be as showed in figure 5.1.

![Scheme of the communication between the car and the smartphone through the ELM327 device.](image)

5.1.2 Provide real time driving behavior information

In order to proof the possibility of using the smartphone as an on board panel to provide information to the driver, the application will implement a driving panel that will provide real time information. This driving panel will contain the information that may be changing during the trip. Related with the driving behavior, the driving panel will retrieve data from the OBD, from the smartphone sensors and even from Internet. The real time data will be the vehicle speed, the engine rotation speed, the road slope, the
total distance of the trip, the lateral accelerations, the front and back acceleration and
the weather information.

5.1.3 Record the trip statistics in a local database

In order to afterwards analyze and take profit of the measured data during the trip, it
is needed to store all the real time data. The smartphone storage capacity is powerful
enough to record all the driving data. This is also a feature that reinforce the utility of
the proposed smartphone application. It will allow the user to revise the recorded trips
and also compute some statistics from the recorded data.

5.1.4 Real time alerts related with the driving behavior

The fact of keeping reading the real time data of the trip, allows the possibility to analyze
continuously the data to detect some events. There can be some risky situations that can
be detected by using the smartphone, for example aggressive turns or sudden brakings.
The main purpose of this functionality is to throw real time alerts in order to let the
driver know if any bad situation has occurred. There will also be an alert indicating a
long time driving.

5.1.5 Accident detection and reporting

Detecting and reporting accidents automatically is not an innovative feature, in fact there
are several proposals of implementing build-in systems in the vehicles with this feature
like the eCall [18]. As the application will be reading and processing real time data,
it is a good scenario to implement accident detection and reporting. This functionality
of the application could be used in parallel with the eCall to be able to inform third
parties about the accident event. The idea is to determine if an accident has happened
depending on the accelerometers values and send an emergency message with useful
information like the location. If it is possible, the location will be measured using the
GPS service, but if it is not possible, the mobile network will be used to locate the
accident, despite the accuracy will be worse.
Chapter 5. *Project description*

To send the emergency message, there are considered two different scenarios. The first one is the case that there is mobile coverage available. In this case, when an accident is detected, the application will send an SMS (Short Message Service) message to an emergency phone previously configured. An SMS message is chosen because it is the communication mechanism that requires less mobile coverage. This scenario is shown in the scheme on figure 5.2.

**Figure 5.2:** Emergency message using mobile coverage.

If there is not mobile coverage, it is considered that there could be a Wi-Fi infrastructure for this purpose. That is why the application, if it doesn’t find any mobile network, will try to connect to a predefined Wi-Fi infrastructure to send an emergency message through Internet. This is what is shown in figure 5.3.

**Figure 5.3:** Emergency message using Wi-Fi.
5.1.6 Traffic jam detection and reporting

By measuring and recording the vehicle speed, it is possible to analyze and detect some patterns. A very common pattern is the one that takes place when there is a traffic jam. In those cases, the speed is zero for some time and consecutive small accelerations are done. If a traffic jam can be detected, it can be useful to share this information to the road managers or to other drivers in order to avoid it or to manage it. That is why the application will send a message if a traffic jam speed pattern is detected.

An example of a speed pattern is shown in figure 5.4. In this example there is a speed graph over time. It shows consequent stops with small accelerations and decelerations reaching low speeds.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{speed_pattern_example.png}
\caption{An example of a speed pattern for a traffic congestion situation.}
\end{figure}

5.1.7 Retrieve and take profit of the weather forecast

The Internet access of the smartphones is something that can be very useful to get information from third parties. In the driving scenario, for example, it can be very useful to know the weather forecast during the trip. This is why in this application there is a functionality that will keep retrieving the weather forecast while driving. The idea is to know the prediction of the weather conditions at the vehicle location and showing the forecast data. It will also throw some alerts by correlating the weather forecast data with the driving patterns and analyzing if there may be any risk.

5.1.8 Environmental efficiency summary

An advantage of storing all the measured data during the trips is the possibility of afterwards analyze it and generate some useful statistics, graphs and tips. A useful
case is the possibility of generating a report about the environmental efficiency of the recorded trips. The application will include a section that will show information about the fuel consumption evolution and some other useful parameters and tips according to the CO$_2$ emissions.

5.1.9 Extending vehicle life summary

As a similar functionality of the former one is the vehicle life summary. This functionality will also use the stored statistics to generate some useful information about the driving skills of the driver according to the vehicle maintenance. These statistics may be, for example, the peak engine rotation speed, the aggressive jumps counter or the jump accelerations.

5.1.10 Driving safety summary

Another interesting functionality is to generate a report with driving safety statistics. As the application will generate alerts about risky events, those events will be also stored. The application will generate a summary about the event data of the events related with the driving safety including a graph with the risky events evolution over time.

5.2 Navigation diagram

The application is structured in a way pretending to be easy to use. In figure 5.5 there is the diagram with the navigation structure. It has a main activity that is the one that shows up on launching the application for the first time. From this activity the user is able to start and stop the trip. When the trip is started, the user will have access to the driving panel, where the real time data will be shown. The driving panel is designed to be the activity to be shown during the trip. The other parts of the application will only be accessible while there is not a trip running. From the main activity, the user can access the settings menu, where the user can configure some parameters, such as the ELM327 Bluetooth device, the device inclination, the emergency phone, the expected fuel consumption provided by the manufacturer, the emergency email address, etc.
The main activity is also the screen where the user will have access to the driving reports. One report is related to the environmental efficiency, that will contain the functionality explained in section 5.1.8. The second report is related with the vehicle life, explained in section 5.1.9. The last report is the one related with the driving safety, mentioned in section 5.1.10. Finally, from the main activity, the user can also have access to the main data recorded by the smartphone.

By clicking the “View Statistics” button in the main activity, the user will be able to see the recorded data ordered by trips. The statistics menu will contain an option to see the list of the trips, an option to view the graph of the duration of all the trips and a field that will indicate the size of the local database where all the data is stored. On pressing the option to view the trip list, a list of all the recorded trips will appear. At this point, the user can select any of the trips to see the specific statistics for the selected trip. For each trip, the user can see a summary of the main statistics, like averages or timing. It is also possible to see graphs of the vehicle speed, the engine rotation speed and the accelerations over time.

In the following sections, there will be a more detailed explanation about the specific implementation of each component, including the background processes and the database management.

5.3 Source code overview

The purpose of this section and the following ones of this chapter is to explain the main application components and their implemented features. As it is explained in section 3.1, the core of the application is programmed in the Java language. The application implements several classes that are organized in six different packages as follows:

- **edu.upc.drivinghelper**: This package will contain the source code for the main activities of the application.

- **edu.upc.drivinghelper.alerts**: This is the package that contains the classes that are in charge of managing the alerts that may be thrown during a trip.

- **edu.upc.drivinghelper.data**: In this package there are the classes that implement the methods to manage the local SQLite database.
**edu.upc.drivinghelper.res**: This package contains several classes with methods that are used as resources in different parts of the application.

**edu.upc.drivinghelper.service**: This is the package that contains the class that is implementing the service in charge of managing all the processes needed during the trip. It also contains the classes used by this service.

**edu.upc.drivinghelper.statistics**: This is a packet used to contain the activities that are accessed from the “View statistics” button on the main activity.

Apart from the Java packages, the application uses several XML files as resources. All these XML files are in the `res` folder of the application source code. In this folder, the XML files are classified depending on the purpose. The main XML files are the layout files, that contain the definition and position of the view components. As each layout file is associated with an activity, so that, the application will have an XML layout file for each activity.
There are also the drawable files that are the ones in charge of defining the appearance of the application. The drawable XML files contain the information about the default colors, sizes and other parameters of the application theme. The drawable folders also contain the images needed by the application theme, like buttons or icons. In our case, we needed to customize the progress bars for the driving panel. The new progress bar styles are also defined in the drawable folder.

The values folder is the one that has the XML files that contain some variables or other information. The values folder contains, for example, the strings file, that contain the text to be displayed on the application. It mainly contains the text of the buttons and the titles of the application screens. The values folder contains also the colors file that associates the RGB color values with a human readable name that may be used in the application.

All the source code of the application, including the one that will be explained in the following sections, can be found in the appendix volume.

5.4 SQLite database structure

In order to be able to analyze and process all the data from the previous trips, it is needed to store it somewhere. In our case, it is stored in the smartphone using an SQLite database, as this is the most used and easy to use database format in Android devices.

In our application, the database structure is as is shown in table 5.1. In the first column there are the table names and in the second column there are the column names for each table. As we can see, there is a table for each type of variable, it is done in order to be able to store the data only when it changes, for example, if the speed is zero for a long time, it will only store one zero and will not store any other value until it changes. For each recorded variable, it is stored the time stamp when the data has been measured and the trip identification number to be able to distinguish the data among the different trips.

In the case of the event table, the events will have an identifier that identifies the type of the event, that can be a jump, an aggressive turn, etc. The column called EventData has
the purpose to store the recorded data of the event, for example the lateral acceleration of an aggressive right turn.

<table>
<thead>
<tr>
<th>Table</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trips</td>
<td>ID, TimeStampStart, TimeStampStop</td>
</tr>
<tr>
<td>Accelerations</td>
<td>TimeStamp, TripID, Accelerations</td>
</tr>
<tr>
<td>Speed</td>
<td>TimeStamp, TripID, Speed</td>
</tr>
<tr>
<td>RPM</td>
<td>TimeStamp, TripID, RPM</td>
</tr>
<tr>
<td>FuelConsumption</td>
<td>TimeStamp, TripID, FuelConsumption</td>
</tr>
<tr>
<td>Distance</td>
<td>TimeStamp, TripID, Distance</td>
</tr>
<tr>
<td>Slope</td>
<td>TimeStamp, TripID, Slope</td>
</tr>
<tr>
<td>Events</td>
<td>TimeStamp, TripID, EventCode, EventData</td>
</tr>
</tbody>
</table>

Table 5.1: Structure of the local SQLite database, that stores all the data recorded during the trips.

In the application source code there are two classes to manage the SQLite database. One class is the `DataBaseHelper`, that is in charge of the table creation when the database is called for the first time. It contains the corresponding queries to create the database tables mentioned in table 5.1. The other class is the `DataBaseManager`, that contains all the static methods used to insert and retrieve information from the database, for example to insert a new speed value or to retrieve all the accelerations of a specific trip. There is also another class that only contains the constants related with the database, for example the table names or the column names.

5.5 Activities

As it is explained in section 3.1, the activities are components of an Android application that are in charge of the management of an application layout. In our application there are several activities that are described in the following subsections. You will see that all the screens are in a landscape orientation. That is because the application is designed to be used in a vehicle, where is more likely to have the mobile device horizontally oriented.

5.5.1 Main Activity

The main activity is the one that will be shown on launching the application. From this activity, the user will have access to the rest of the application screens. The layout of this activity is shown on figure 5.6.
On the right top of the screen there is the information about the GPS status and the Bluetooth status. It is to inform the user if the GPS and Bluetooth interfaces are activated or not. On the right of the action bar (the green bar on top of the application layout) there is a button to access the settings activity. As it is also seen at the main activity layout, there is a button to start the trip. On pressing this button and confirming the confirmation dialog, all the background trip operations are started and the driving panel is launched. While the trip is running, the start button will be changed to a stop button, that will be used to stop the trip processes. There is the possibility to close the driving panel during the trip, that is why in the main activity there is a button to access the driving panel, that will be accessible only while the trip is running. If the Main Activity is opened during the trip it will look like in figure 5.7, where the “Stop trip” and “Driving Panel” buttons are enabled and the rest of the buttons are disabled.
There are also three buttons to access the three different driving reports. As the reports use the recorded statistics, they will only be enabled when there is not a trip running. These three buttons are the following:

- **Environmental efficiency**: It launches a report with graphs and data related with the environmental efficiency, that is based on the recorded statistics.

- **Extending the vehicle life**: This button is used to show a report related to the car usage and maintenance while driving.

- **Safety while driving**: In this case the report is related with the safety skills of the driver mainly based on the recorded risky events.

At the bottom of the screen there is the “View statistics” button to access the recorded statistics for each trip. This button will also be enabled only when there is not any trip running.

### 5.5.2 Settings Activity

This activity is in charge of managing the preferences that are defined by the user. In the figure 5.8 there is the layout screenshot of the settings activity with all the available preferences. All these preferences will be stored in the mobile device and will be saved until the user deletes the whole application from the device or removes the application data from the application manager of the Android operating system.

In the “Global Preferences” section, there are the following preferences:

- **Device inclination**: This preference is used to configure the inclination of the mobile device when the device is fixed on the car. It will be used later to calculate the accelerations and the road slope. To configure this preference it shows a dialog like the one in figure 5.9. This preference uses the `DeviceInclinationPreference` class to manage its operations and layout.

- **Bluetooth Devices**: This preference is used to configure the ELM327 Bluetooth device. On pressing this preference element it shows a dialog with the list of the paired Bluetooth devices as is shown in figure 5.10. In order to let the ELM327
**Figure 5.8:** The settings activity layout with all the user preferences.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GLOBAL PREFERENCES</strong></td>
<td></td>
</tr>
<tr>
<td>Device inclination</td>
<td>Set up the inclination of your device</td>
</tr>
<tr>
<td>Bluetooth Devices</td>
<td>Select the ELM327 Bluetooth device.</td>
</tr>
<tr>
<td>Manufacturer fuel consumption</td>
<td>The expected fuel consumption provided by the manufacturer of the vehicle in l/100km.</td>
</tr>
<tr>
<td><strong>EMERGENCY PREFERENCES</strong></td>
<td></td>
</tr>
<tr>
<td>Emergency phone</td>
<td>The telephone number of the phone to send the emergency messages and calls automatically. Do not forget the country code.</td>
</tr>
<tr>
<td>Emergency email address</td>
<td>The email address where the emergency message will be sent when an accident is detected.</td>
</tr>
<tr>
<td>Emergency Wi-Fi SSID</td>
<td>The Wi-Fi SSID of the network that the application will try to connect to send an emergency email if no mobile coverage is found.</td>
</tr>
<tr>
<td>Emergency Wi-Fi password</td>
<td>The emergency Wi-Fi password.</td>
</tr>
<tr>
<td>Email user</td>
<td>The email user where the emergency message will be sent from.</td>
</tr>
<tr>
<td>Email password</td>
<td>The email password of the user where the emergency message will be sent from.</td>
</tr>
<tr>
<td><strong>DATA STORAGE PREFERENCES</strong></td>
<td></td>
</tr>
<tr>
<td>Delete data</td>
<td>Delete all internal stored data</td>
</tr>
</tbody>
</table>

**Figure 5.9:** The dialog that is showed on configuring the device inclination preference.
device appear on the list, the user has to previously pair the device with the smartphone.

![Image of Bluetooth Devices settings](image).

**Figure 5.10:** The dialog that is showed on configuring the Bluetooth devices preference.

- **Manufacturer fuel consumption:** This preference requests a numerical value that indicates the expected fuel consumption provided by the manufacturer. The value has to be in liters per 100 km. This value will be used to calculate if the user is consuming the expected fuel or not.

In the “Emergency Preferences” section there are the following preference items:

- **Emergency phone:** This preference requests a mobile phone that will be the phone number used to send the emergency SMS messages and to initiate a phone call when an accident is detected.

- **Emergency email address:** It is to configure the destination email address where the emergency email messages will be sent. The emergency messages will be sent through email only when there is not mobile coverage and the SMS message can not be sent. It will also be used to send the traffic congestion notifications.

- **Emergency Wi-Fi SSID:** In this preference field it is needed to configure the service set identifier (SSID) of the Wi-Fi network that the system will try to connect when there is no mobile coverage.

- **Emergency Wi-Fi Password:** To configure the password of the former defined Wi-Fi infrastructure.
• **Email User**: The email user where the emails will be sent from. As the application is designed to work with a gmail account, the user has to be in a format like example@gmail.com.

• **Email Password**: The password for the former email account.

In the “Data Storage Preferences” section, there is only a single item:

• **Delete all internal stored data**: In fact, this item is not a preference, it is used to delete all the data stored in the local database of the application. It basically deletes the SQLite database file stored in the smartphone memory. This preference uses the `DeleteDataPreference` class to manage its operations.

All these preferences are defined in the `preferences.xml` file stored in the `res` folder of the application.

### 5.5.3 Driving Panel Activity

The driving panel activity (`PanelActivity`) implements the layout to be showed while the trip is running. It manages the view objects to display the real time data of the trip. The driving panel screen looks like the figure 5.11.

![Figure 5.11: A screenshot of the driving panel layout.](image)

The information showed on the driving panel is from the `TripService` (explained in section 5.6), that is the one in charge of getting all the information. In order to communicate with the service, at the `onResume()` method of the `PanelActivity`, it binds to
the service and creates a connection with a messenger and defines a handler to manage the incoming messages. As the PanelActivity depends on the TripService to show the information, it will not be able to be launched if the service is not running. In fact, the PanelActivity implements only the functionality of receiving and displaying the data from the TripService, without the implementation of any smart feature.

The driving panel also has two buttons. One button is just to close the panel, it acts as the physical back button of the mobile device, that finishes the activity. The other button is to stop the trip, that shows the confirmation dialog for stopping the trip and does it in the same way as stopping the trip from the MainActivity.

5.5.4 Environment Activity

The EnvironmentActivity class implements an activity that will show information to the user related to the environmental efficiency. This information is mainly related with the fuel consumption. The layout of the activity has a list on the left where the user can choose which information wants to see. The selected information will appear on the right side of the layout as is shown in figures 5.12, 5.13 and 5.14. In this activity there are three possible information screens that can be selected by the user:

- **View environmental statistics**: In this screen the user can check the main statistics related with the fuel consumption like it is shown in figure 5.12. There is a message to indicate the user if the average fuel consumption is appropriate or not according to the one expected. The expected average fuel consumption provided by the vehicle manufacturer is configured on the settings menu. After the message, it provides some values about the average and total fuel consumption and the emitted CO$_2$.

- **Fuel consumption graph**: This graph (figure 5.13) shows the evolution of the fuel consumption average per distance. It computes the average fuel consumption (in liters per 100 km) per fractions of 20 km of trip distance. It allows the user to see if the fuel consumption evolves in a constant way or not, knowing if efficiency skills are improving or getting worse.

- **Fuel consumption over speed graph**: The purpose of this graph is to let the user know which are the optimal speeds to save fuel. To see this, a graph is plotted
Figure 5.12: Screenshot of the environmental statistics screen of the Environmental Activity.

Figure 5.13: Screenshot of the fuel evolution graph of the Environmental Activity.

with the average fuel consumption depending on the speed. In figure 5.14 there is a screenshot of this type of graph. In order to be fair about the fuel consumption over the speed it is computed only when the road slope is between a -3% and a 3%, that means when the road is nearly flat. In order to compute the statistics faster, it will be only computed for the last recorded data for a specified period of time.

At the implementation of this activity, the environmental statistics and the fuel consumption graph are computed at the same time. As the graph of the fuel consumption evolution is shown over the trip distance, it uses the fuel consumption and the trip distance tables of the database. To compute the graph it goes over the fuel table computing
the fuel consumption average for each distance section. The begin and end time stamps of each distance section are retrieved from the distance database table.

The fuel consumption over speed graph is computed in a different process as the former one. It uses the fuel consumption table, the vehicle speed table and the road slope table. The road slope table is used in order to be able to compute the fuel consumption over speed only for a flat road slope. In this case, the algorithm will relate the fuel consumption and speed tables to know the average fuel consumption for each speed range of values.

All the graphs of the application, including the ones in this activity are generated by using the GraphView library that is explained in section 4.4.1.

5.5.5 Vehicle Activity

The VehicleActivity class is in charge of generating a report about the vehicle life, that means analyzing the recorded data to evaluate the driving behavior of the driver related with the vehicle maintenance in order to extend the vehicle life. The main purpose is to show that with the proposed scenario of this project it is possible to generate this type of statistics. In this activity, as in the EnvironmentActivity, there is a menu on the left of the layout and on the right there will appear the selected graphs or statistics. There are two different screens that can be shown in this activity that are the following:
- **Vehicle statistics**: In this screen there are some numerical statistics related with the vehicle maintenance as it is shown in figure 5.15. There is a counter for the total number of aggressive jumps detected, the peak jump acceleration with the date and time and the average of aggressive jump accelerations. These jumps are commonly produced by going through potholes or speed bumps with a high speed. In this statistics screen there is also information about the engine rotation speed. There is the peak RPM (Revolutions Per Minute) value with the date of the event and the average RPM.

![Figure 5.15: Screenshot of the vehicle life statistics of the Vehicle Activity.](image)

- **Engine speed over vehicle speed graph**: This screen shows a graph that relates the engine rotation speed with the vehicle speed as it is shown in figure 5.16. This graph indicates the average engine rotation speed for each speed range. It will only take into account the data when the road slope is more or less flat. It pretends to proof the possibility of analyzing the data to know if the user drives with too much revolutions per minute. It may be useful if the vehicle manufacturer provides the expected driving behavior for the car.

To compute the vehicle statistics the `VehicleActivity` implementation uses the data from the RPM table and event table of the database. To compute the jumps statistics it retrieves the jump events from the event table and counts the number of jump events, computes the average from the event data and gets the jump with the biggest acceleration value with the time stamp of the event. For the RPM statistics it computes the average from the RPM values and also prints the biggest value with the time stamp.
Chapter 5. Project description

5.5.6 Safety Activity

Like the EnvironmentActivity and the VehicleActivity, the SafetyActivity implements a layout with a menu on the left to enable the user to choose which screen to see. The purpose of the SafetyActivity is to generate a report with statistics and graphs related with the driving safety, to proof the possibility of evaluating the driver behavior in terms of safety. This statistics are mainly based on risky events recorded during the trips. The two screens implemented in this activity are the following:

- **Event statistics report**: The main objective of this report is to provide information to the user about the risky events that have taken place while driving. It includes aggressive turns, aggressive speed up accelerations and aggressive brakings. The layout of this screen is like is shown in figure 5.17. It provides the counter of each type of events with the peak accelerations of each event.

- **Event evolution graph**: When selecting this item, a graph will be displayed with the evolution in time of the risky events recorded while driving. The number of
Figure 5.17: Screenshot of the safety related events statistics report at the Safety Activity.

risky events is shown in fractions of 30 minutes as it is shown in figure 5.18, that is a screenshot of this graph.

Figure 5.18: Screenshot of the safety related events evolution graph at the Safety Activity.

In order to compute either the statistics or the graph, the application just needs to retrieve the data from the event table, where all the risky events are stored. For the event statistics, the algorithm only needs to count the events and retrieve the peak values. In the case of the graph, as it is computed over time, the application also retrieves the trip table from the database in order to know when the trips start and stop and compute the events per time.
5.5.7 Statistics Activities

On the main activity of the application, there is the “View Statistics button”, that launches the main statistics activity (*StatisticsActivity*) as it is shown in figure 5.19. At the bottom of the list, there is the size of the application database. It is computed by checking the size of the SQLite database file, that is a “.db“ file.

![Figure 5.19: A screenshot of the main statistics panel, launched on pressing the “View statistics” button on the Main Activity.](image)

By pressing the “Trip durations graph” item, the user will open a bar graph implemented in the *TripDurationsGraphActivity* class that shows all the trip durations as it is seen in figure 5.20. This is generated by retrieving the database table that contains the trips and computing the duration from the start and the stop time stamps.

![Figure 5.20: A screenshot of the trip durations graph launched from the main statistics activity.](image)
The first item of the list is the “View trips” button. This is used to access the trip list, that looks like in figure 5.21, with the start time of the trip and the time duration. As it is done in the trip duration graph, the data to generate the list is taken from the trip table of the database.

In this activity, the user can access the main data recorded for each trip or delete the data from any trip. By long pressing a trip, a confirmation dialog for deleting a trip will appear. If the user confirms to delete the trip, all the data related with that trip will be deleted. The `DatabaseManager` class implements a method to delete all the trip data from the trip identifier, that removes all the rows from all the tables that correspond with the trip identifier.

![Figure 5.21: A screenshot of the trip list activity launched from the main statistics activity.](image)

When a trip from the trip list is pressed, the application will launch the trip statistics menu. In order to identify the trip where this activity is called from, on the calling intent the trip identifier is passed as a parameter, to afterwards be able to retrieve the data only for the selected trip. The layout of the trip statistics menu is a list with four options as it is in figure 5.22.

The four options in the trip statistics menu are the following:

- **General statistics**: This activity is implemented in the class called `TripGeneralStatisticsActivity`. The main purpose of this activity is to provide several statistics about a single trip. The layout of this activity is shown in figure 5.23. The data provided in this screen is as follows:
Chapter 5. Project description

68

Figure 5.22: A screenshot of the trip statistics menu launched from the trip list activity.

- **Trip duration**: The trip duration is computed from the start and stop time stamps of the trip table.

- **Total trip distance**: The total trip distance is retrieved from the distance table. The `DataBaseManager` class implements a method to retrieve the trip distance from the trip identifier.

- **Average speed from distance and duration**: This value is computed dividing the total trip distance by the total trip duration.

- **Average vehicle speed from recorded speed data**: This speed average is computed by using the data recorded in the speed table of the database. The application retrieves all the rows that correspond to the selected trip and computes the average. As the time periods between the different samples is not regular, the algorithm takes into account the time between two samples. For each interval of samples, the speed average is computed and weighted according to the time of the interval value. This speed average should be close to the former one.

- **Average moving vehicle speed**: This parameter indicates the speed average taking into account only when the vehicle is moving. That means that the time the vehicle is stopped will not be computed in the average. It is computed in a similar way as the former speed average. This value should be equal or greater as the two former ones.
– **Average engine speed**: The average engine rotation speed will be calculated in the same way as the “Average vehicle speed from recorded speed data” but using the engine speed (RPM) data from the RPM table of the database.

– **Average fuel consumption**: It is done as the “Average engine speed” but with the fuel consumption data of the selected trip.

– **Total time with the vehicle stopped**: This time value indicates the sum of the time periods that the vehicle has been stopped. It is done by analyzing the recorded speed data and computing a sum of the periods while the speed is zero.

**Figure 5.23**: A screenshot of the general trip statistics launched from the trip statistics menu.

- **Accelerations graph**: This menu item starts an activity implemented in the `TripAccelerationsGraphActivity` class. This activity generates a graph that shows the recorded accelerations during a trip as it is shown in figure 5.24. As is is seen, the graph shows the accelerations of the three axes that correspond to the side acceleration (related with the turns), the linear acceleration (related with the front accelerations and brakings) and the vertical accelerations (corresponding to the jumps). To generate this graph, the activity retrieves the stored accelerations for the corresponding trip and assigns them into a three different series of data that are the three ones shown in the linear graph. To let the user analyze in detail the graph it is scalable and scrollable.

- **Speed graph**: This item starts the `TripSpeedGraphActivity` that computes a graph to show the speed evolution during the selected trip. The algorithm only needs to
Figure 5.24: A screenshot of the trip accelerations graph launched from the trip statistics menu.

retrieve the data from the speed table of the database that corresponds to the trip and draw it into the linear graph. To let the user analyze in detail the graph, it is set as scalable and scrollable. The speed graph will look like in the figure 5.25.

Figure 5.25: A screenshot of the trip speed graph launched from the trip statistics menu.

• RPM graph: When selecting this menu item, the application starts a similar activity as the former one, implemented in the TripRPMGraphActivity class. It follows a similar process to generate the graph, but retrieving the data from the RPM table of the database. To let the user analyze in detail the graph it is scalable and scrollable. The resulting graph will look like in the figure 5.26.

To plot these three graphs over time, it is required to format the values of the time axis to a human readable value. The time stamps are stored in the database in milliseconds.
To format the values of the time axis a method of the \textit{GraphView} library needs to be overridden on the view generation as it is shown in the following code example:

```java
GraphView graphView = new LineGraphView(this // context,
        "Vehicle Speed (km/h)" // heading
)

@Override
protected String formatLabel(double value, boolean isValueX) {
    if (isValueX) {
        // transform number to time
        return Utils.formatTimeGraph(value);
    } else {
        return super.formatLabel(value, isValueX);
    }
}
```

By overriding this method, the time axis labels will dynamically be formatted when scaling and scrolling the graph. As it is seen on the former code example, to format the time it uses a method in the \textit{Utils} class called \textit{formatTimeGraph()} that will format the milliseconds to a string with a human readable format with hours, minutes and seconds.

### 5.6 Background processes

An important part of the application is the implementation scheme of the operations that take place during the trip. In figure 5.27 there is a diagram that shows how the application works during the trip. In this diagram, the boxes correspond to the different
Java classes used running the trip and the arrows indicate the flow of data. The main class that manages all the procedures during the trip is the **TripService** class. This class is an Android service (explained in section 3.1) that is created when the user presses the “Start Trip” button on the main activity. When the service is created, it constructs several objects and starts threads that will be used to get, process and store the trip data. It also configures the accelerometer sensors to be able to read the vehicle accelerations and the road slope.

![Diagram of the interaction of the application components during the trip.](image)

As it is seen in the figure, to retrieve the OBD data from the vehicle, the **TripService** uses a class called **ObdDataManager**, that automatically saves also the data to the SQLite local database. The **ObdDataManager** is a thread that will keep running until the **TripService** is stopped. The **DataBaseManager** is a class with static methods that are used to interact with the database from any point of the application. The **TripService** also starts a **CongestionDetector** that will keep checking periodically if there is any traffic congestion pattern. If it detects a traffic congestion, it will contact the **AlertManager** to let the user know that the congestion is detected. The **TripService** can also throw alerts through the **AlertManager** when it is needed depending on the computed data. The **AlertManager** is in charge of starting the **AlertActivity** for normal alerts or the **AccidentAlertActivity** for accident alerts.
Another object used by the TripService is the WeatherManager that uses the Location-Helper to get the approximate location and retrieve from Internet the Weather forecast for the actual time and location of the user. The Weather data is read and computed by the TripService.

As the most of the real time data will have to be showed on the driving panel, the TripService uses an object called DataToPanel that implements all the required methods to manage the messenger to send the data to the PanelActivity when it is binded to the service. When the DataToPanel object is constructed, the messenger needs to be sent as a parameter. This object implements a method for each type of data to be sent to the PanelActivity.

5.6.1 TripService

As it is stated, the TripService class is the main class that manages all the operations that take place during the trip. In order to be sure that the data sensing and recording during the trip is real, it should not be interrupted. That is why the TripService is started as a foreground service and run in a priority thread. Doing this, the process is not likable to be killed in case the system runs out of memory. The Android operating system requires, for foreground services, to implement a permanent status bar notification to be displayed while the service is running as it is shown in figure 5.28.

Figure 5.28: A screenshot of the status bar notification that will be displayed while the TripService is running, as it is set to a foreground service.

On starting the TripService by using the startService() command, the onCreate() and onStartCommand() methods are called. These two methods are implemented as follows:
As it is seen in the `onCreate()` method, the service is started on a priority thread. At the `onStartCommand()` method is where the important startup operations take place. First, the inclination parameter is taken from the shared preferences. This parameter is previously set on the settings activity explained in section 5.5.2 and it is used later to compute the accelerations and the road slope according to the inclination of the device. After that, the database object is retrieved to be able to insert and retrieve data from the local SQLite database. An `ObdDataManager` thread object is also created and started.
This object is in charge of configuring and managing all the communication between
the smartphone and the ELM327 Bluetooth device. The ObdDataManager implements
several public methods to retrieve the data taken from the OBD interface. This class
will be explained in more detail in the following sections. Then, the application executes
a private method that implements the commands to start the service as foreground, like
configuring the icon to be shown on the status bar, the actions to be taken on pressing
the status bar notification, etc. After starting the service as foreground, it creates an
AlertManager object, that will be used to manage and throw the alerts during the trip.
Finally the OperationPerformer thread object is constructed and started. This class is
nested in the TripService and its thread runs the main operations of the TripService.

The TripService also has a nested Handler class that is used to manage the incom-
ing messages from the messenger between the PanelActivity and the TripService. The
TripService can only receive two types of messages, the one that indicates that the driv-
ing panel activity is launched (with the MSG_REGISTER_CLIENT identifier) and the
one that informs that the driving panel is closed (with the MSG_UNREGISTER_CLIENT
identifier). The main purpose of receiving this two messages is to let the service know
if it has to send the data to the panel or not. That is why when the service receive the
client registration message it sets a boolean indicator to true and creates a DataToPanel
object that is in charge of managing the messages that go from the service to the driv-
ing panel. At this time, it also sends all the available data to the panel through the
DataToPanel object. When the service receives a unregistering message from the panel,
it will set the indicator to false and remove the DataToPanel object.

On the run() method of the OperationPerformer class, there are implemented the op-
erations that will be executed at the beginning of the trip, a while clause with the
commands to be executed periodically and the commands to be executed at the end of
the trip. In a simplified way, it will be structured as follows:

```java
public void run()
{
    // Commands to be executed at the beginning of the trip
    while (!finished){
        // Commands to be executed periodically during the trip
    }

    // Commands to be executed at the end of the trip
}
```
When the service is started from the *MainActivity* it also starts the driving panel. When the driving panel is started at the same time as the service, it will prompt a progress dialog while the starting trip operations are taking place. In the *DataToPanel* class there are implemented some methods to change the message of this progress dialog and to close it when the trip is started. It even has a method to send a stop message to close the progress dialog and the driving panel when any problem is detected on starting the trip. The longer operations are mainly the ones used to configure and start the OBD communication. Due to that, the service will wait until the OBD communication is ready before starting the trip operations.

When the OBD communication is started without error, the service will start the rest of the starting trip operations. It will schedule a timer task for the *WeatherManager* and the *CongestionDetector*, that are processes that will be executed periodically. These two classes will be explained in more detail in the following sections. The service will also setup the accelerometer sensors to keep measuring the accelerations during the trip. After that, it will retrieve the first data from the *ObdDataManager* and the sensors and will store it to the database using the *DataBaseManager*. It will also create a new trip row in the trips table of the database.

When all the beginning operations are done, the service will keep running in the while clause until the service is stopped. In this clause there are several parts. First, it retrieves the data from the *ObdDataManager* and stores it temporarily in local variables. It also checks if there is the weather forecast available from the *WeatherManager* and, if it is, it also stores it on local variables. The second step is to check if there is a client registered (that means the driving panel is on the screen) it sends the recorded data to the panel using the *DataToPanel* object. A third step is to check if any of the measured data has changed, comparing it to the previous measured value, and store it to the database if it has changed. In the same loop, there are also implemented three alerts that can be thrown based on different circumstances, that are the following:

- **Humidity alert**: This alert will be thrown by analyzing the humidity forecast and the vehicle speed. If both values are high, the user will be alerted that there may be a risk of the road of being wet and the speed should be reduced.
• *Precipitation alert*: This alert is similar to the former one, but in case of using the humidity value it will take into account the precipitation probability to notify the user.

• *Driving time alert*: In this case, the alert is implemented to warn the user if the driving time is too long. In some cases, if the driver has been driving for a long time, he should stop some time to take a rest.

When the trip is stopped, the `onDestroy()` method of the `TripService` is called. On this method the `WeatherManager` and `CongestionDetector` scheduled tasks are canceled, in order to do not be executed again, and the `finish` boolean variable is set to true to let the `OperationPerformer` go out of the `while` loop and execute the final commands. The final commands are basically to unregister the sensor listener, store the last measured data to the database and set the end trip time stamp to the trip table. It will also finish the running processes related to the trip, like the `ObdDataManager` or the `AlertManager`.

An important part of the implementation of the `TripService` class is the sensor data management. The service implements the methods of a `SensorEventListener`. The method used is the `onSensorChanged()`, that will be executed each time a sensor value is measured. In this application only the acceleration sensor is used, but the Android operating system enables to measure the linear accelerations (without taking into account the gravity) and the gravity (without taking into account the linear accelerations) as if they where different sensors. That is why when we register the sensor listener, we enable two different sensors, the `TYPE_LINEAR_ACCELERATION` and the `TYPE_GRAVITY`. The first one is used to know the accelerations of the vehicle in the three axes and the second one is used to calculate the road slope during the trip.

Each time the `onSensorChanged()` method is called, it checks which type of sensor the data comes from, whether the gravity or the linear acceleration. If the data comes from the linear acceleration, what the application does at first is to adjust the values to the real vehicle accelerations by using the device inclination value. After that, we will get the acceleration values according to the device inclination in the vehicle. Then, the application will check if the driving panel activity is registered to the service and, if it is, it will send the data to the panel by using the `DataToPanel` object.
The second step on the `onSensorChanged()` method is to analyze the linear acceleration values. The algorithm will check if there has been any excess of acceleration in any of the three axis. These high acceleration events will be based on different acceleration thresholds. The events will result on an alert that will be prompted to the driver to alert him about it. the different events that may be detected from the linear accelerations are the following:

- **Aggressive right turn**: This event is detected when a high side acceleration is detected to the right.
- **Aggressive left turn**: This case is the same to the former one but detecting the acceleration to the left.
- **Aggressive forward acceleration**: This is the case when a big forward acceleration is detected, that means that the driver has increased the speed too sharply.
- **Aggressive braking**: This event is registered when a big backward acceleration is detected, that usually corresponds to a sudden braking.
- **Jump**: A jump event is detected from the vertical acceleration of the vehicle, that usually happens when going through some road irregularity with too much speed.
- **Accident**: This is an event that is detected when an extremely high acceleration is detected in any of the directions. The acceleration threshold will be much higher than the former ones.

When an event of the former ones is detected, the application first throws the alert to the driver through the `AlertManager` and then it records the event on the local database through the `DataBaseManager`. On the database it stores the event time stamp, the trip identification number, the identifier that indicates the type of the event and some data related with the event, like, for example, the acceleration value that has caused the event detection.

After the events are checked, the algorithm will check if the acceleration values have changed compared with the previously ones stored in the database. If they have changed, it will record the new measured acceleration values to the database.

In case the `onSensorChanged()` method is called from the gravity sensor, it will also compute the acceleration values according to the device inclination. When this is done,
it will compute the gravity deviation angle from the car in a flat position. This angle value, will be converted to the road slope in percentage. The road slope percentage indicates the vertical distance over the horizontal distance, that means that a 50% of slope indicates that for each horizontal 100 meters there is a height difference of 50 meters.

As the accelerometer values are measured in a high frequency, the slope value should be smoothed in order to be not affected by vibrations or road irregularities. To smooth the road slope value, the application keeps storing the measured value in a vector that is used to compute the average. The average value of the vector will be the one that is showed on the panel and stored at the database. To store the slope on the database, the application will also previously check if the last stored value is different from the one measured.

5.6.2 ObdDataManager

The ObdDataManager class is implemented to manage the OBD communication and data management. It is mainly developed to lighten the work of the TripService. It extends the Thread class in order to run a loop with the OBD communications without interfering the main application processes. On constructing the object it requires the database, to be able to store the data, and the shared preferences, to retrieve the ELM327 Bluetooth device MAC address.

On starting the ObdDataManager thread it prepares the Bluetooth socket with an input stream and an output stream. These two streams are required to use the Android OBD Reader library explained in section 4.4.2. Once the Bluetooth interface is configured, the configuration commands will be sent. As it is required to configure the ELM327 device we have to send AT commands, that are the ones defined to interact only with the ELM327 device as it is explained in section 4.1. The AT commands used to configure the ELM327 device are the following:

- Set defaults: By sending the “AT D” it will reset all the settings of the ELM327 device to the default ones.
- **Reset**: It is done by sending the “AT Z” command and it will reset the ELM327 device. This command will last more time than the others as the device is going to be restarted.

- **Set echo off**: With the “AT E0” command the ELM327 will not send back the bytes that are sent to it.

- **Set line feed off**: The “AT L0” command configures the ELM327 in order to do not send linefeed characters after every carriage return character.

- **Disable spaces**: Sending the “AT S0” command the ELM327 device will be configured to send the responses without any spaces between the response characters.

- **Disable headers**: It is done with the “AT H0” command. The ELM327 device will not send the additional (header) bytes of information at the responses.

- **Set timeout**: It uses the “AT ST” command followed by the number of milliseconds. It indicates the time that the device will wait for a response from the vehicle before declaring that there is no data.

- **Select protocol**: To select the OBD protocol it uses the “AT SP” command followed by the protocol identifier. If it is set to zero, as it is in our application, the protocol selection will be automatic.

After configuring the ELM327 device, the thread sends a command to know the type of fuel of the connected vehicle (command PID: “01 51”). If the OBD interface of the vehicle does not provide this data, the application will use diesel as default. The fuel type is used to later compute the fuel consumption. The OBD interface usually does not provide the fuel flow, but it provides the MAF (Mass Air Flow) value, that indicates the air flow that enters the engine. To compute the fuel flow we have used an approximation of the calculation proposed in [37]. We have used an air to fuel rate of 14.65 that is approximately valid for gasoline or diesel. The type of fuel is used to take into account the density of the fuel, that is different between gasoline and diesel.

After knowing the type of fuel, the application will try to check if the required commands are available, it will try them twice before refusing. The required commands are the vehicle speed (command PID: “01 0D”), the engine rotation speed (command PID: “01 0C”) and the mass air flow (command PID: “01 10”). There is also an additional
command to be executed that is the ambient air temperature (command PID: “01 46”).
There are a few vehicles that have this command available, so if it is not available, the
application will use the intake air temperature (command PID: “01 0F”), that is more
common to be implemented in the vehicles. It is important to know that it can only be
considered as outside temperature to test the application, but it will not be a real value.
That is why if the intake air temperature is used as outside temperature, a message will
be shown to the user informing about that.

In the main loop of the thread, it will execute the commands in a separated way. The
outside temperature command will be executed much less frequently as the other three
commands, as it is expected to vary more smoothly. The other three commands, the
vehicle speed, the engine speed and the mass air flow will be measured continuously.
Each time the commands are executed, the application will check if they have changed
enough to be stored on the database and it will store them if it is the case. When the
process is stopped it just closes the Bluetooth socket.

The `ObdDataManager` is also in charge of computing and recording the trip distance.
The trip distance is computed from the measured speed values. To do it, it stores the
previous speed value with the corresponding time stamp and, on measuring a new speed
value, computes the average speed between the previous and the new speed value and
multiplies it by the time between the two speed samples. The distance value is added
to the total trip distance. The trip distance will be also stored in the database when it
change according to a threshold.

The `ObdDataManager` has several public methods in order to let the `TripService` retrieve
the measured data. It has a method for each variable, for example, to retrieve the speed,
the `TripService` will have to call the `getSpeed()` method. There are also additional public
methods to manage the object, for example the `setTripID()` to set the trip identifier once
the trip is started, in order to let the `ObdDataManager` store the data to the database
labeling it with the corresponding trip. There are methods to know if the communication
has failed or if the real outside temperature is available.
5.6.3 LocationHelper

The LocationHelper class is designed to manage all the operations to retrieve the user location. It is used for three different purposes in our application:

- **WeatherManager**: It is used by the WeatherManager to locate the user in order to provide a weather forecast for the actual location.

- **AccidentAlertActivity**: In this case, the location is used to be sent to the emergency services when an accident is detected. The emergency message needs to contain the user location in order to let the emergency services know where the accident has happened.

- **CongestionDetector**: It uses the LocationHelper to attach the user location at the message that is sent when a congestion is detected with the congestion data.

On creating a LocationHelper object, the location provider can be chosen between the network and the GPS, as it is explained in section 3.1.6. If the network location provider is used, the location will be computed according to the network, that is faster but less precise. If the GPS is chosen, the LocationHelper will be trying to retrieve the location from the GPS interface during a specific time. If it can not retrieve the location by using the GPS system, the application will finally retrieve the location by using the network location provider.

This class implements some public methods in order to know when the location is available and also to get the location data. To know if the location is available there is the isLocationAvailable() method, to retrieve the location the getLocation() method is used and to get the address there is a method called getAddress(). The getAddress() method uses the Google services that are available in the Android SDK.

5.6.4 WeatherManager

The WeatherManager is a class that implements the procedures and methods to retrieve the weather forecast. This class extends a TimerTask that is a thread to be run in a scheduled manner. From the TripService, this TimerTask will be scheduled to be executed each period of time. The weather forecast source is the AEMET ("Agencia
Estatal de Meteorología") that provides an XML file with the weather forecast. It will only work for the weather forecast in the Spanish bounds. To retrieve the XML file, it is required the locality code that will be the one corresponding to the user location. The locality codes are defined by the Spanish government and published by the INE (Instituto Nacional de Estadística) [38].

First of all, the thread will use the LocationHelper to get the user location. As the weather forecast does not require a fine location, the configured location provider will be the network, that is much faster. When the location is available, the application will retrieve the postal code from the address of the location. The postal code will be used to look up for the locality code. To get the locality code from the postal code, the application uses a text file from [39] that contains all the postal codes related with the locality codes. It will look up for the postal code and retrieve the corresponding local code.

To retrieve the XML file it is required to do it by using an HTTP (Hypertext Transfer Protocol) request to a URL (Uniform Resource Locator) that contains the local code. The URL will look like the following, where the 17189 is the local code:

http://www.aemet.es/xml/municipios/localidad_17189.xml

After receiving the HTTP response with the XML file, the application parses the XML code to retrieve the desired information. The weather forecast is organized per date, per concept and per hour range. That is why the application computes the local time and looks for the information in the actual date and time. It is important to note that the data from the XML is not empirical measured data, it is just a forecast, that means that it may not be the real weather state. The weather forecast information that is taken from the XML is the following:

- Precipitation probability
- Wind speed and direction
- Temperature (maximum and minimum)
- Humidity (maximum and minimum)

This class implements public methods to let the TripService look up for the retrieved values. There is a method to retrieve each one of the variables mentioned above. There
is also a method in order to know if there is new weather forecast information available, that is the \texttt{isNewDataAvailable()} method. Finally, there is a public method to retrieve the locality where the forecast is about.

\subsection*{5.6.5 CongestionDetector}

The \textit{CongestionDetector} class extends also a \textit{TimerTask} as the \textit{WeatherManager}. In this case, it is used to detect when there is a congestion or not. The main purpose of using this is to proof the capability of detecting congestion automatically by the use of a smartphone. In our implementation, the congestion detection will be based on a speed pattern.

The task of the \textit{CongestionDetector} is scheduled from the \textit{TripService}. There is a constant value that indicates the time period length of the analyzed speed pattern. That is why the scheduling of the task will start after this first predefined time period. After the task is executed for the first time it will keep executing more frequently.

When the task is executed it will first retrieve the stored speed data from the database for the last predefined period of time. By analyzing this speed data, the algorithm will look for the following information:

- Total time stopped
- Number of stops
- Average speed while moving

After computing these statistics, the algorithm will compare them with some predefined ones and take the decision if there is congestion or not. The decision thresholds are a minimum time stopped, a minimum number of stops and a maximum average speed while moving. These parameters that the decision is based on, may be agreed with the receivers of the congestion alert. By detecting and sharing the congestion event information there may be several advantages for road managers or for routing applications that can take into account the traffic status. To simulate this scenario, the application sends an email to a predefined email address with the recorded information. Before sending the email, the application uses the \textit{LocationManager} to locate the user and include at the email the
location to let the receiver know where the congestion is taking place. It is configured
to use the GPS if it is possible, as the location will be more accurate.

When all the data is ready, the email will be sent with the location, the congestion
parameters and all the speed record in order to let the receiver analyze the real speed
pattern itself if necessary. The CongestionDetector implements a public method to know
if a congestion event has been detected. It is used by the TripService to throw an alert
to the driver to let him know if congestion has been detected.

To send emails we use the resource explained in [40]. This resource requires to include a
.jar library called mail.jar and two Java classes that are the JSSEProvider.java and the
GMailSender.java. To use this resource it is needed a gmail account where the emails
will be sent from.

5.6.6 Alerts

From the user point of view, the alerts may be the most important part of the application
while running a trip. The alerts are the ones that are in charge of informing the driver
about the events that happen during the trip, that can be generated by an aggressive
driving pattern, an accident detection, a congestion detection or other circumstances.

The TripService is in charge of taking the decision if an alert has to be thrown. To
throw the alert, it uses the AlertManager class that implements the procedures and
public methods to start the alerts. The AlertManager extends the Thread class and at
the beginning of the trip, the thread is started. The implementation of the thread uses a
queue to store the alerts to be thrown, just in case there are several alerts to be thrown
in a very short period of time. In order to avoid the thread of being running all the
time and avoid concurrency conflicts on accessing the queue, the class uses synchronized
methods and the wait() command to stop the process while the queue is empty.

Basically, the main process contains a loop that will be active until the process will
be stopped by the TripService when the trip is finished. Inside the main loop, there is
another loop that will throw alerts while the queue of alerts is not empty. After each
alert, it will wait a predefined time to be sure that the alert is finished and the next one
can be thrown. When the queue is empty, the process will go out the while() and stop
at the wait() command. The code of the run() method of the thread is the following:
public void run() {
    // Loop to be executed while the TripService is not stopped
    while (!finished) {
        // Loop to be executed while the queue is not empty or the
        // TripService is not stopped.
        while (!alertQueue.isEmpty() && !finished) {
            synchronized (this) {
                startAlert(context, alertQueue.removeFirst());
            }
            // Time to wait until the Alert Activity is finished
            try {
                sleep(6100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        synchronized (this) {
            try {
                // It will wait until a new alert is added to
                // the queue.
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

The AlertManager has two public methods that are used from the TripService to throw the alerts. One of the methods is for the normal alerts that, on calling this method, they will be added to the alert queue. The type of alert is identified by a parameter that is passed in this method called throwAlert(). This method will also notify the thread to start throwing alerts, because it may be waiting at the wait() command. The other method is the throwAccidentAlert() that, as an accident alert has an important priority, it will throw the alert directly when the method is called.

The alerts are not strictly a background operation because they are implemented using activities. Otherwise, they are related and started from the background operations, that is why they are explained in this section. The normal alerts are thrown by using the AlertActivity. As the smartphone may be with the screen turned off or with another application, the first procedure will be to turn on, unlock the screen and start the activity in a new task. The activity is based on a text that will depend on the type of
Chapter 5. Project description

the alert, so that, on starting the activity, the type of alert will be send as a parameter of the Android Intent. The alert layout will only have TextView with the information about the alert. The activity also uses the text to speech functionality to inform the user orally about the alert. The AlertActivity has implemented all the alert types with the predefined texts.

There is a special activity for the accident alerts called AccidentAlertActivity. This activity implements the same functionality as the AlertActivity with additional features. The main purpose of detecting an accident is to send the accident alert to the emergency services. To avoid false positive accident detections, this alert implements a countdown chronometer to let the user to have some time to cancel the accident detection. To send the emergency message, this alert will use the LocationHelper to retrieve the location to be sent with the message. It will try to use the GPS location provider if it is possible, if not it will use the network provider. Once the location is available, the application will check if there is mobile coverage. If it is available an SMS message will be sent with the location and a phone call will be automatically started. The telephone of the SMS and the phone call will be predefined by the user on the shared preferences of the application.

If there is not mobile coverage available, the application will search for a Wi-Fi infrastructure with Internet access that ideally should be provided by the road managers. The SSID (Service Set Identifier) and password of the Wi-Fi network will be set on the settings activity of the application. As it is an emergency case, the application has enough permissions to enable the Wi-Fi interface of the smartphone if it is not enabled. When the mobile device is connected to the Internet, it will send an email to the predefined email address informing about the accident with the location coordinates. To send the email it will also use the method in [40] as it is done in the CongestionDetector.

The events that generate the alerts are stored in the database of the application. However, the event recording is not implemented in the alert activities, it is done at the TripService, when they are detected.
Chapter 6

Test and results

The purpose of this section is to proof if the functionalities implemented in the application work in the way that it is expected. The following sections correspond to each functionality exposed in section 5.1. On each of the following sections there are explained the scenarios and the results of the tests.

6.1 Read OBD data through Bluetooth

To test the OBD communication, the main tool used is the OBDSim, explained on section 4.2. This tool allows us to easily emulate a real trip without the need of a car. As we know, the OBDSim can use a SQLite database file with a real logged trip as the source for the data. By using this method we have been able to check that the existing data on the database is the same as the read by the application.

In figure 6.1, there is the screenshot of a sample of the database source of the OBDSim. This logged trip has been run and captured with the application. At the vs column, there are the speed values in km/h. As we can see in figure 6.2, there is the graph that correspond to the data capture of the application while the OBDSim was executing the log showed in figure 6.1.

As it can approximately be seen in both figures, the speed values are very similar in the table and in the graph. For example, it can be seen that the peak speed value in figure 6.1 is the same peak value in the graph of figure 6.2.
Chapter 6. Test and results

Figure 6.1: Sample of the database that the OBDSim uses to generate the trip data.

Figure 6.2: Speed graph that corresponds to the capture of the speed (vss column) from the OBDSim record showed in figure 6.1.
The same case is in figure 6.3, where the engine speed graph is showed for the same period in figure 6.1. In figure 6.1, the \textit{rpm} column corresponds to the engine speed values in RPM (Revolutions Per Minute). As it is also seen, the RPM values are very similar to the ones in the graph.

![Figure 6.3: RPM graph that corresponds to the capture of the RPM (rpm column) from the OBDSim record showed in figure 6.1.](image)

The application has also been tested with some real cars using a real ELM327 to Bluetooth device. It is seen that the application works correctly. Apparently, the sampling rate for the real ELM327 device is not as high as using the OBDSim emulator, but it is still quite good. The OBDSim only provides the most basic OBD commands, that is why the application is designed to work with them. For this reason, the application can work with the most of the cars that implement the OBD-II/EOBD interface.

In figure 6.4 there is the OBD connector of one of the cars used to test the application. In figure 6.5 there is the connector with the ELM327 to Bluetooth device. The smartphone is located in a place visible for the driver where it does not interfere the driver sight, as it is seen in figure 6.6. The smartphone \textit{y} axis, according to figure 3.5, must be horizontal and perpendicular to the driving direction.

As the application uses the MAF (Mass Air Flow) to compute the fuel consumption, the given value is just an approximation of the real one. The application has been tested with a modern vehicle, where the instantaneous fuel consumption appears on the dashboard. In this case, it is seen that the fuel consumption computed from the MAF is not much real in some situations. For example, when the user remove completely the foot from the throttle pedal, the real fuel consumption may be zero, although there may be still an air flow going through the engine.
Figure 6.4: Real OBD connector from one of the used cars to test the application.

Figure 6.5: Real ELM327 connection with the OBD connector in one of the used cars to test the application.

Figure 6.6: Real smartphone location and orientation to test the application in a real car.
Testing the application with a real vehicle, it is seen that the engine rotation speed value (RPM) is very similar to the one showed in the analog dashboard of the vehicle. Comparing the vehicle speed value from the OBD and from the analog dashboard, we have seen a quite important difference. It is seen that the vehicle dashboard indicator usually show a higher speed value than the OBD value.

As the distance of the trip is computed from the speed, it has been checked if the distance value computed by the application corresponds with the real distance. The highway location markers are traffic signs that indicate the road distance for each kilometer or 500 meters. This signs has been used to compare the recorded distance with the real distance and it has been proved that the computed distance is very close to the reality. This fact means that the OBD speed value is much more precise than the one in the dashboard.

6.2 Provide real time driving behavior information

This functionality is mainly implemented in the driving panel (PanelActivity), where the user can see the real time data. To test this feature, it has been done in both type of tests, using the OBDSim emulator and using the real ELM327 to Bluetooth device. We have seen that the response of the panel is fast and the user layout is friendly enough to be able to easily see the main real time data. No problems has been detected while the OBD communication and the smartphone sensors are working properly. That means that for the user interface, the sample rate of the measured data is enough to be seen in a smooth way.

6.3 Record the trip statistics in a local database

This functionality has been tested by showing the recorded statistics graphically and also by exporting the application SQLite database file. The graphs used to check that the data is correctly stored in the database are the ones implemented in the trip activities, where there are graphs for the vehicle speed, the engine rotation speed and the accelerations. We have seen that the graphs are coherent and smooth enough to proof
that the recorded data is well sampled and stored. We have observed that the pattern of data seen in the driving panel is similar to the one shown in the graphs.

The SQLite database file is only accessible from inside the application if the Android device is not rooted. In order to be able to analyze the database file, we have implemented a method inside the application that copies the database file to a folder accessible from other applications and attaches it to a new email to be sent by launching the smartphone email application.

To analyze the database file we have used a computer application that manages SQLite database files. In figure 6.7 there is a capture of the SQLite database file structure from the database file received by email. We can see that there are all the tables and columns defined in section 5.4. There is a `sqlite_sequence` table that is in charge of managing the `autoincrement` fields that, in our case, there is only the trip identifier, that appears in the trips table. There is also a `android_metadata` table that is automatically generated by the Android operating system to store some information of the application, for example the language.

![Figure 6.7: A capture of the application SQLite database file structure with all the tables and columns.](image-url)
In figure 6.8 there is a capture of the trips table, where we can see three recorded trips. It is seen that each trip has an identifier (id column), a starting time stamp (timeStampStart column) and a stopping time stamp (timeStampStop column. As it is the easiest way to compute time stamps in Android, they are in milliseconds since January 1, 1970 00:00:00.0 UTC. We can see that there are some trip identifiers that seem to be missing, for example after the trip number 4, comes the trip number 7. That means that the trips 5 and 6 have been deleted from the database.

![Figure 6.8: A capture of the trip table from the application SQLite database file.](image)

In figure 6.9, there is a capture of the accelerations table with the corresponding columns. It shows that the column with the accelerations values contains the three axes accelerations in a vector format. It is also seen that there is not any trip data between the trip 4 and the trip 7. This means that when the user deletes a trip, the data that belongs to the deleted trip is also deleted. It has been checked in all the database tables.

### 6.4 Real time alerts related with the driving behavior

This functionality of reporting alerts about risky events has not been tested in the sense of checking the reliability of the aggressiveness thresholds. Defining and calibrating the patterns and algorithms to decide if an event is risky enough to throw an alert is a complex topic under research that is not the main purpose of this project. Here, the purpose is to proof that this kind of detection can be done and managed by using a smartphone. That is why the tests are done by generating the risky events in a non real environment, just to check the correct alert management. The application implements a system to manage the alerts in order to throw one alert after another without losing anyone.

In this project, the main alerts are related with the accelerations, that when they exceed some thresholds, the alerts are generated. To test this system, we have originated alerts
by accelerating manually the smartphone to overpass the acceleration thresholds. The alerts are thrown with a good time response and, when there are several alerts generated at the same time, they are thrown one after the other without interruption. We have also tested that, when the screen of the smartphone is off and/or locked, it will turn on correctly, unlock the screen and show the alert. The text to speech system works also as expected only when the multimedia volume of the device is not configured to be silent.

All the aggressive events that generate an alert are recorded in the events table. In figure 6.10 there is a capture of this table from the application SQLite database file. As we can see, each event has a time stamp, the trip identifier, a code that determines the type of the event (defined in the $DBCons$ class) and a field with the data of the event. The event data column represents different parameters depending on the type of event, for example, for an aggressive right turn, the value will indicate the peak lateral

![Table: accelerations](image)

**Figure 6.9:** A capture of the accelerations table from the application SQLite database file.
acceleration on turning.

![Figure 6.10: A capture of the events table from the application SQLite database file.](image)

We have also checked if the detected events correspond with the recorded data that appears on the graphs. We have seen that it corresponds as expected. As an example, considering the first event in figure 6.10 we have seen that is from the first trip and it is a type 2 event. This type of event, according to the $DBCons$ class, corresponds to an aggressive forward acceleration. The accelerations graph of the first trip has been checked and we have looked for this peak. In figure 6.11 there is the piece of the accelerations graph that contain the forward acceleration event. As the acceleration is forward, we have to check the linear acceleration (blue line) and we see that the acceleration peak is at $9.075 \text{ m/s}^2$. If we look again at the first row in figure 6.10, we see that the event data is $9.075346$, that is the acceleration in $\text{m/s}^2$ in this event. It is seen that the two values correspond.

### 6.5 Accident detection and reporting

The detection of the accidents is done in the same way as the detection of the aggressive events that are related with the accelerations. The difference is in the threshold to
Figure 6.11: A piece of an accelerations graph that corresponds to the aggressive forward acceleration event of the first row in figure 6.10.

decide if the event is an accident or not. The acceleration thresholds to consider that the detected event is an accident will be much more higher.

Proofing that the application detects the accidents correctly is difficult and even not strictly necessary for our purpose, as the mechanism is the same as the normal aggressive events. What is more important to be tested is the AccidentAlertActivity implementation. To test it, fake accident alerts are generated by the application.

It is seen that the GPS locating system is started correctly just on starting the alert activity and it begins counting down the time while the user is able to cancel the alert, as it is shown in figure 6.12.

Figure 6.12: A screenshot of the accident detection alert, where appears the chronometer counting down the time that the user has to cancel the emergency message.
When the count down is finished, the application checks if there is an available location from the GPS, and if not, it will retrieve the location for the network. The location can be checked on the received emergency messages, that include the location coordinates and the accuracy of the locating mechanism. If there is mobile coverage, the application sends the information using an SMS message. If the location is retrieved from the GPS, the received emergency message text will look like the following:

**EM. Coord.: lat. 41.38870864983331 long. 2.112567306622016 / Accu: 16.0 m**

When the location is retrieved by the network provider, the accuracy is much worse. This is why the accuracy value increases a lot compared to the GPS accuracy value. In this case, the message may look like the following one:

**EM. Coord.: lat. 41.3887057 long. 2.1114285 / Accu: 558.0 m**

We can see that the latitude and longitude values are less precise in terms of decimal digits and the accuracy. In this example, is almost 35 times worse using the network than using the GPS. It also works correctly the functionality of initiating a call after sending the SMS message. The application just gives enough time to send the SMS message and it starts the phone call to the predefined number in the application settings. The only problem is that the automatic calls can not be emergency calls due to the Android permissions.

When there is no mobile coverage, the application will try to connect to a Wi-Fi infrastructure and send an email through Internet. The Wi-Fi name and password will be configured on the settings and it will connect even the Wi-Fi is disabled, because, as it is defined in the Android manifest file, this application has enough permissions to change the Wi-Fi state. The message will be the same but it will be sent to the email address configured on the application settings. The email will look like in figure 6.13, where it is seen that the content is the same as the SMS message. The advantages of sending short messages is that the system will require less resources to send them.

![Figure 6.13: Emails containing the emergency message that are received from the accident alert. They are sent when no mobile coverage is available.](image)
6.6 Traffic jam detection and reporting

To detect if there is a traffic jam or not, the application analyzes the speed record for the last five minutes during the trip looking for some patterns. Without focusing about if the decision thresholds or patterns are well defined or not, we have proved it with a predefined ones.

When a congestion event is detected, the application throws a simple alert informing the driver, but, from the background, it sends an email with the traffic jam information. An example of the content of an email that is received when a congestion event is detected is the following:

Location: lat. 41.3902364  long.  2.1097631  Accuracy: 1092.0
Time duration: 5 minutes
Number of stops: 4
Time stopped: 00:00:24.962
Average speed while running: 17.0 km/h

Speed record:
02/10/2013  19:47:04:309    7
02/10/2013  19:47:06:313    8
02/10/2013  19:47:06:614    9
02/10/2013  19:47:06:807   10
02/10/2013  19:47:07:362    9
02/10/2013  19:47:09:807   10
02/10/2013  19:47:11:321   11
02/10/2013  19:47:12:386   12
02/10/2013  19:47:12:534   13
02/10/2013  19:47:12:842   14
02/10/2013  19:47:13:290   15
02/10/2013  19:47:13:895   16
02/10/2013  19:47:14:314   17
02/10/2013  19:47:17:892   18
02/10/2013  19:47:18:310   19
02/10/2013  19:47:18:892   20
02/10/2013  19:47:19:949   21
02/10/2013  19:47:25:830   22
02/10/2013  19:47:28:398   23
...

The email with the congestion detection notification contains some general statistics about the event, that are the ones that are taken into account to decide if there has been congestion or not, and also the speed record for the whole period that has been
analyzed to detect the traffic jam. This is done in order to enable the receiver to process the data by they own. In the former example there is not the complete speed record because it is quite long.

In figure 6.14 there is the piece of speed graph that correspond to the traffic detection from the previous example. In this picture we can approximately see the statistics that are sent by email, for example the number of stops, because we can see in the speed graph that the vehicle stops four times as it is stated in the email.

![Speed Graph](image1)

**Figure 6.14:** A piece of a speed graph when a congestion event took place.

### 6.7 Retrieve and take profit of the weather forecast

The method to retrieve the weather forecast from the AEMET is explained in section 5.6.4. This functionality has been proved in different locations and the localization and forecast retrieving is done successfully. If there are problems related to the Internet access, the application will just leave a message saying that the forecast is not available. The weather forecast will be seen in the driving panel while during the trip and will look like the example in figure 6.15.

![Weather Forecast](image2)

**Figure 6.15:** A capture of the weather forecast that appears on the driving panel.
By analyzing the weather forecast and comparing it with the other data that is measured from the application, it can detect some risky situations and events. For example, if there is a high wind speed forecast and the vehicle speed is very high, the driver will receive an alert like the one in figure 6.16.

![Figure 6.16: A capture of a weather alert that is thrown when a high vehicle speed and a high wind speed is expected.](image)

### 6.8 Environmental efficiency summary

This functionality is implemented in the `EnvironmentActivity`, that is explained in section 5.5.4. A good point of this report is the layout organization, that allows to add different screens with different graphs and statistics easily. This makes the report scalable.

A problem that has been encountered is on generating the graphs, specially in the case of calculating the graph that shows the fuel consumption over the speed. The problem consists of a high processing time. This high delay is caused when retrieving the data from the database. It is mainly seen when the data is correlated between different database tables.

Another drawback is the reliability of the results that are shown in this report. As it is stated in former sections, the fuel consumption value is just an approximation, that means that the fuel consumption data that is displayed on the graphs and the statistics may not be the real one. This fact causes some strange behaviors on the resulting graph.
6.9 Extending vehicle life summary

In this case, the functionality is implemented in the \textit{VehicleActivity}. The layout is also an advantage as the environmental efficiency summary. Another similarity is the high processing time. Even it is carried with a progress dialog, it lasts quite a long time. This is caused by the same reason as in the environmental efficiency summary.

The most important advantage of this functionality is that it is a very interesting feature that is not much frequent in other applications. Mainly, it can be very useful for vehicle owners.

6.10 Driving safety summary

This functionality is mainly based on the events that are thrown during the trip. It is implemented in the \textit{SafetyActivity}. To test this feature, we have generated some events and we have seen that they appear on the event evolution graph and on the event statistics. This report is the simplest in terms of algorithm complexity as it just requires to look up the event table from the database (figure 6.10) and show the recorded data.

We have also checked that the peak events correspond with the recorded data in the same way that it is done in section 6.4, where the alert events are compared with the recorded data. We have seen that the event data corresponds to the stored data and storing the events in this way is much more efficient than computing the events data by analyzing the continuous data tables.
Chapter 7

Future work

There are several issues to take into account as a next step for the application. As this application has been developed basically to proof several capabilities of using a smartphone with a vehicle, the next step will be to define the specific purpose of the application. The application may vary if it will be released, for example, by a car manufacturer, car rental service or insurance company.

It can be very interesting to know specific information from the car manufacturers to be able to adapt the application for specific vehicles. If we know the features and the expected behavior of the vehicle, the evaluation of the driving skills could be more precise and reliable. Information from vehicle and road experts would also be very useful in order to contribute on the driving evaluation mechanisms of the application.

From the car manufacturers point of view, the fact of using the OBD interface may not be the best solution. In the case the application is distributed by the car manufacturer, they may be more interested in using a proprietary protocol with an interface implemented in the car designed specially for this purpose. This solution could provide some advantages.

It would be more user friendly, as no additional OBD device is needed. It also might be more safe, because the manufacturer may only give access to certain parameters. From the manufacturer point of view, it would be good as it could only give access to the manufacturers application.

A standardized solution for the connectivity between the car and the smartphone is the MirrorLink [41]. The MirrorLink is a device interoperability standard that offers integration between a smartphone and a car’s infotainment system. For example, users could
communicate with the smartphone from the steering wheel or from the infotainment screen.

For high quality car manufacturers a possible solution could be to implement, for example, a built in Android device in the vehicle. This could increase the price of the vehicle, but it would give the possibility of using an application as the one proposed in the project without the need of a smartphone.

In the case of congestion and accident detection, there should be an agreed mechanism between the application developer and the road authorities and emergency services. This mechanism should define a communication protocol to send the information of the recorded circumstances. Related with the congestion detected, it could be also interesting to know from the road authorities the criteria to determine if there is a traffic jam or not depending on the road.

Related with the recorded data storage, it could be a good idea to use some external server to store the trip information. The application is intended to record all the user trips. As the smartphone storage may be limited, it is a good idea to store the data into an external server. This mechanism also allows to have the information in a safe place and the users may have the ability to compare their driving performance with other users that use the same vehicle. By reporting the recorded data to an external server may be also interesting for vehicle manufacturers and for vehicle owners that are not the usual drivers to let them know how the driver is using the vehicle.

It would also be good for a real release of the application to arrange the user interface in order to improve the user experience. From the engineering point of view it is not much important, but it is for the end user satisfaction.

Another application issue that may be improved is the database structure. By testing the application it is seen that the computing time when consulting several database tables is extremely high. This fact is due to that each parameter is stored in a different database table. Even it may not be efficient in terms of storage, it could be computationally more efficient to use a single table for several parameters, for example storing in the same table the vehicle speed, the engine speed, and the fuel consumption. It has the advantage that the parameters would be in the same row at the same table and it will not be required to retrieve and compute the data from different database tables.
It is explained in former chapters that the fuel flow value is not much accurate. In the case of releasing an application for real users, this value should be reliable. Some modern cars are implementing the commands to retrieve the fuel flow through the OBD, but, as this application may use a proprietary interface from the car manufacturer, the value should be provided by that interface. However, this is an issue that must be arranged in a real utilization.

In the state of the art (Chapter 2), there are explained some proposals from other authors that are doing research in some fields that this application involves, for example the fuel efficiency or the driving safety. There are proposals of algorithms that evaluate the driving skills of the driver by using a smartphone. These algorithms are not used in our application due to their complexity, otherwise, in a final application, it could be very promising to implement them, as the final results could be much better.

There are several issues related with the event detection and reporting that could be improved. In the case of accident detection and reporting, it could be interesting to send more information about the accident event. The idea could be to send an information message after sending an emergency message with, for example, the speed and accelerations data stored in the database just from some time before the event happened. In order to enhance the reliability, it could be a good idea to have access to the eCall [18] system of the car. The GPS system may be more reliable in the eCall system than using the smartphone. The accident detection will be also more accurate in the eCall system.

Related with the normal alerts that are thrown during the trip, it could be very promising to keep adjusting the thresholds to throw an alert depending on other parameters. For example, in the case of the alerts generated by the accelerations, it could be interesting to vary the thresholds according to the weather conditions or the road type. A specific case is that the risk of accident for the same lateral acceleration value may not be the same if the road is dry than if the road is wet.

Another improvement that could be related with the risky event detection is the use of the gyroscope in detecting aggressive and risky turns. By using the gyroscope, the application could correlate the curve angle with the lateral accelerations and determine more accurately the aggressiveness of the driving pattern.
In this chapter we have explained several issues that can be improved from the proposed application, but what is important to note is that the most of them are not weaknesses of the proposal, they are potentialities that make the proposed scenario a startup for a lot of functionalities and implementations.
Chapter 8

Conclusions

The aim of this project is to prove that the use of the smartphone with the vehicle has a very promising future. It is seen that there are a lot of functionalities that can be implemented in this scenario. On the development of this project, it has been proved that there are a lot of features that may be very attractive for different types of companies and users.

For car manufacturers, the use of the smartphone with the vehicle is a very interesting topic as they can provide a lot of functionalities using the user smartphone instead of implementing extra on-board devices. Using the user smartphone allows cheap and upgradeable systems that will keep improving the driving experience without changing or modifying the car.

This approach can also be very interesting for the situations where the driver is not the owner of the vehicle. This could be the case of public transportation, garbage collection or car rental companies. This type of companies are usually worried about how the drivers use the vehicles. The proposed scheme allows to provide information about how the drivers are taking care of those vehicles.

From the users' point of view, the proposed scenario can also be attractive. The users may be interested in saving fuel, because, at the end, it is a way to save money while they are also friendly with the environment. As the drivers may also be the owners of the vehicles, it is interesting for them to take care of the vehicle. And the safety functionalities are very important for a vehicle user too.
It seems that, in the future, this type of applications will use additional communication interfaces instead of the OBD. However, the possibility of using the OBD allows the user to have very innovative features without the need of using a new car. This solution may not be much promising for the car manufacturers, but yes for the users.

Authorities may also be interested in the use of the proposal described in this project. The functionality of avoiding risky situations, the detection and location of traffic jams or the accident detection and reporting are features that can improve the traffic management and the performance of the emergency services.

A weakness of the concept of using a smartphone with the vehicle is the difficulty of implementing a VANET (Vehicular Ad-hoc Network). This is because the communication interfaces that are implemented in the smartphones are not designed to create ad-hoc networks between them. This fact could be solved by implementing this kind of communication interfaces in the vehicles, for example by using the IEEE 802.11p. However, it is a promising field that should be considered for a near future.

Focusing on the actual application that has been developed in this project, we have seen that there are some features that can be improved, but the purpose of the project has been achieved, as we have seen a wide variety of functionalities that the system can perform. Some of the features are related with fields that are under research, that means that there are several research groups and companies that are working in them. This is the case, for example, of the environmental efficiency, the driving safety monitoring or the accident detection and reporting. A quite interesting approach is the functionality of generating a report about the driving skills according to extending the vehicle life. This topic has not been much developed and it seems that it can be very useful. In fact, as it is mentioned above, it is a very interesting feature for the scenarios where the driver is not the vehicle owner.

However, it is seen that what is relevant from the application is not just each feature that is implemented, the most important idea that we can extract from it is the feasibility of integrating all this features in a single application. The easiness of installing a smartphone application and using it with the vehicle resulting in having a considerable amount of promising features to improve the driving experience seems to be a part of a near future. At least, in this project, the use of smartphones for smart driving is a concept that has been proved.
Bibliography and References


