MOBILE MAPPING CAR

SW Architecture & Design

Supervisor: MSc. Torben Gregersen

Team: AAD1

Editors:

- Guillaume Trousseau
- Jaime Escuer
- Iñaki Abadia
- Roger Prat

AARHUS
UNIVERSITY

Applied App Development
(ITIPRJ)

Content

1 Introduction

1.1 Purpose

The purpose of this document is to describe the software implementation of our project. This document will provide the necessary information in order to understand the behavior of the system or any required further development.

1.2 Document Structure and Reading instructions

In this section you'll find the different views we've created. Our SW Architecture is based on the N+1 model. This model is intended for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views" [IEEE Software 12 (6), pp. 42-50]. The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers.

Our project requires 4+1 views, as follows:

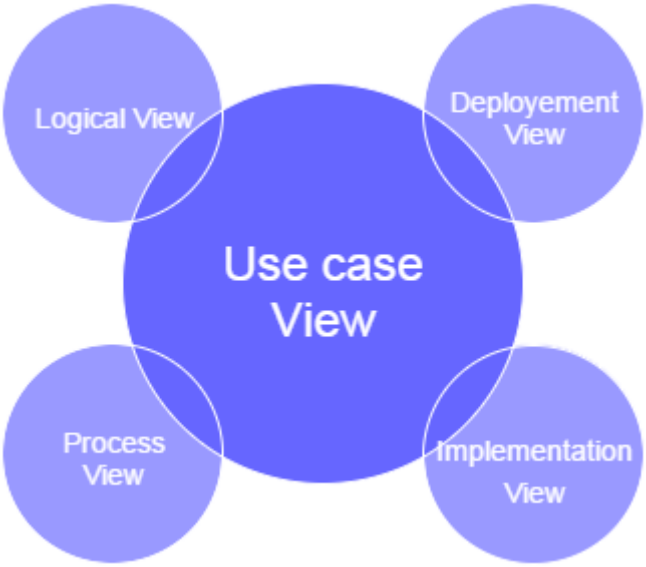

Figure 1: 4+1 model view

1.2.1 Use case view

The Use case view contain every information regarding the system usage. It contains one or more scenario for each use case, explaining how the system works and how it should be used.

1.2.2 Logical view

The logical view consist of an explanation of the internal operation of the system. For each use case, a detailed description of the classes involved is given. Sequence diagrams are used to describe the flow in the system and static class diagram to explained interactions between classes, services and the database.

1.2.3 Process view

The process view describes the various processes / threads in the system, what the interaction between the threads are, which and when processes are running in parallel with each other.

1.2.4 Deployment view

The deployment view, sometimes called physical view, describes how physical components and software components are implemented. Additionally, describes the communication protocols used.

1.2.5 Implementation view

The implementation view describe software components and file structure in the system. Include the package diagram.

2 Logical View

2.1 Main Smartphone's perspective

2.1.1 Static Class Diagrams

2.1.1.1 Graphical User Interface

Here, we are explaining how the Graphical User Interface (GUI) is working and what are the interaction between the different components. The GUI consist of the StartScreen, ConnectScreen, AboutActivity and MapList. There is no communication between StartScreen and the LiveVideoScreen and AboutActivity. The ConnectScreen send data to the StartScreen. The MapList activity consists of three fragments each one fulfilling a different task. The communication between fragments are done using three interfaces, implemented by MapList. In order to fulfill its task, MapListFragment is accessing the database using the MapDataSource class.

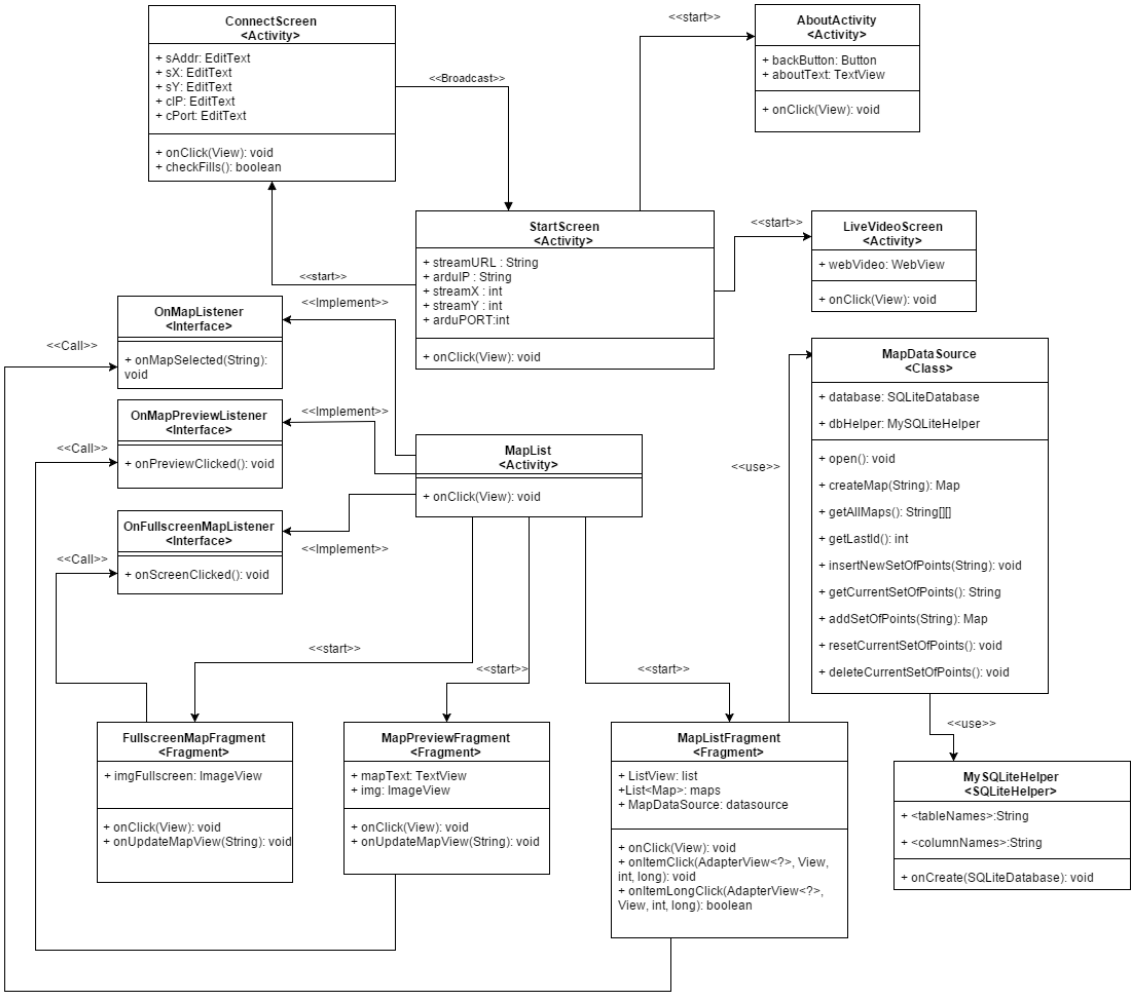

Figure 2: GUI - Static class diagram

2.1.1.2 Drawing task

There is an explanation of how the system collects data and uses them to create a map. The *LiveVideoScreen* activity starts the *ArduinoConnection* service which launch *ConnectTask* to create a connection between the microcontroller and the smartphone. Instruction such as "recording" are sent by the *ArduinoConnection* service using *SendTask*. Once the data retrieved from the microcontroller and stored in the database the *DrawingService* service is launched. *DrawingService* access the database using the *MapDataSource* class to get the current set of measures then it draw the map and store it in the database still using *MapDataSource*.

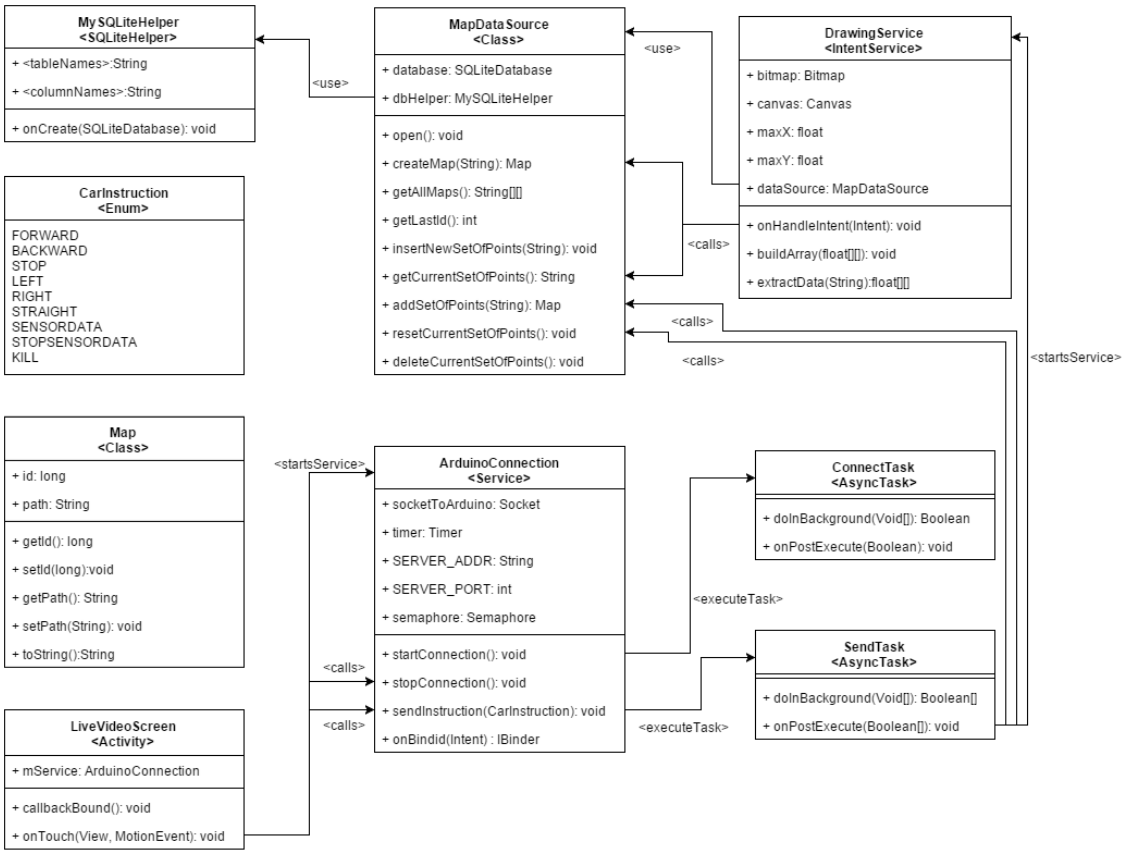

***Figure 3*: Drawing Task - Static class diagram**

2.1.2 Use cases - Sequence Diagrams

2.1.2.1 Use case 1.1: Input connection data

This use case describes how the user can input the necessary data for initializing the connection with the rest of the system.

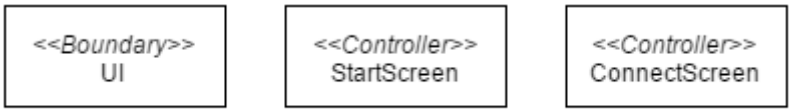

Figure 4: use case 1.1 - Involved classes

UI:

The boundary-class UI is a class containing the graphic elements of the App and define the appearance of the App to the user. The system is implemented in the Android framework so this class is a gathering of XML file defining the UI for each controller.

StartScreen:

The StartScreen activity is the main screen of the app where the user can choose to click on “connect” to switch to the ConnectScreen activity and “Start recording” to switch to the LiveVideoScreen activity.

ConnectScreen:

The ConnectScreen activity includes text input where the user can enter the IP and the port of the car’s microcontroller and live-video’s smartphone. Once it is done, the user can click on “Save & Connect” then “Back” to go back to the StartScreen.

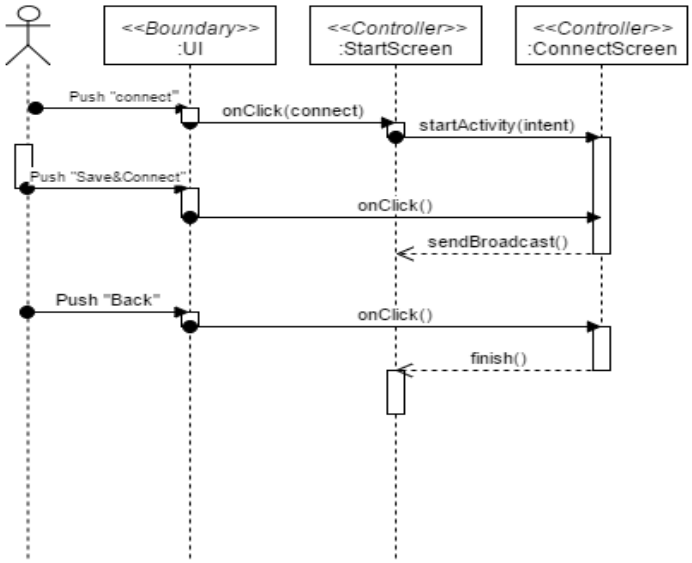

Figure 5: Use case 1.1 - Sequence Diagram

2.1.2.2 *Use case 1.2: Connect to µController*

This use case describes how the user can connect his smartphone with the car's system.

Figure 6: Use case 1.2 - Involved classes

UI:

The boundary-class UI is a class containing the graphic elements of the App and define the appearance of the App to the user. The system is implemented in the Android framework so this class is a gathering of XML file defining the UI for each controller.

LiveVideoScreen:

At launch, the LiveVideoScreen activity will start the ArduinoConnection service.

ArduinoConnection:

The boundary-class ArduinoConnection is an interface allowing communication between the app and the microcontroller. The ArduinoConnection service will try one started to connect with the microcontroller using the information entered in the ConnectScreen.

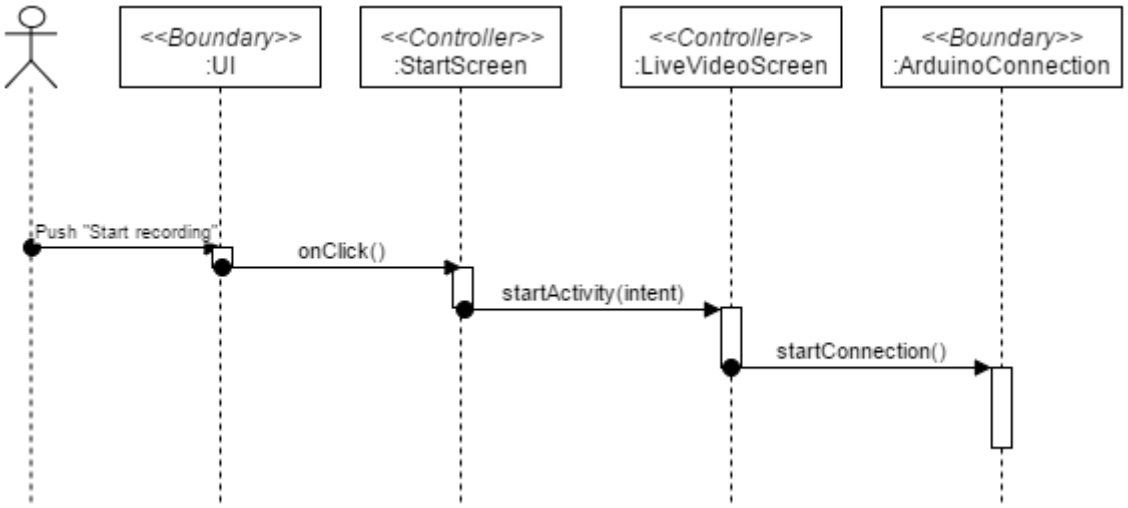

Figure 7: Use case 1.2 - Sequence Diagram

2.1.2.3 Use case 2: Stream video

This use case describes how the smartphone on the car streams live video to the user's smartphone. The smartphone mounted on the car records video through his camera and streams it in MJPEG format. The video is shown on the main screen of the user's smartphone.

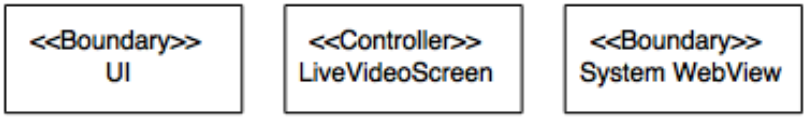

Figure 8: Use case 2 - Involved classes

UI:

The boundary-class UI is a class containing the graphic elements of the App and define the appearance of the App to the user. The system is implemented in the Android framework so this class is a gathering of XML file defining the UI for each controller.

LiveVideoScreen:

The LiveVideoScreen includes four buttons corresponding to the four directions. When they are pressed the information is relayed to the ArduinoConnection service.

System WebView:

The System WebView is a browser bundled inside of our app. It just loads a web page and shows it in the UI

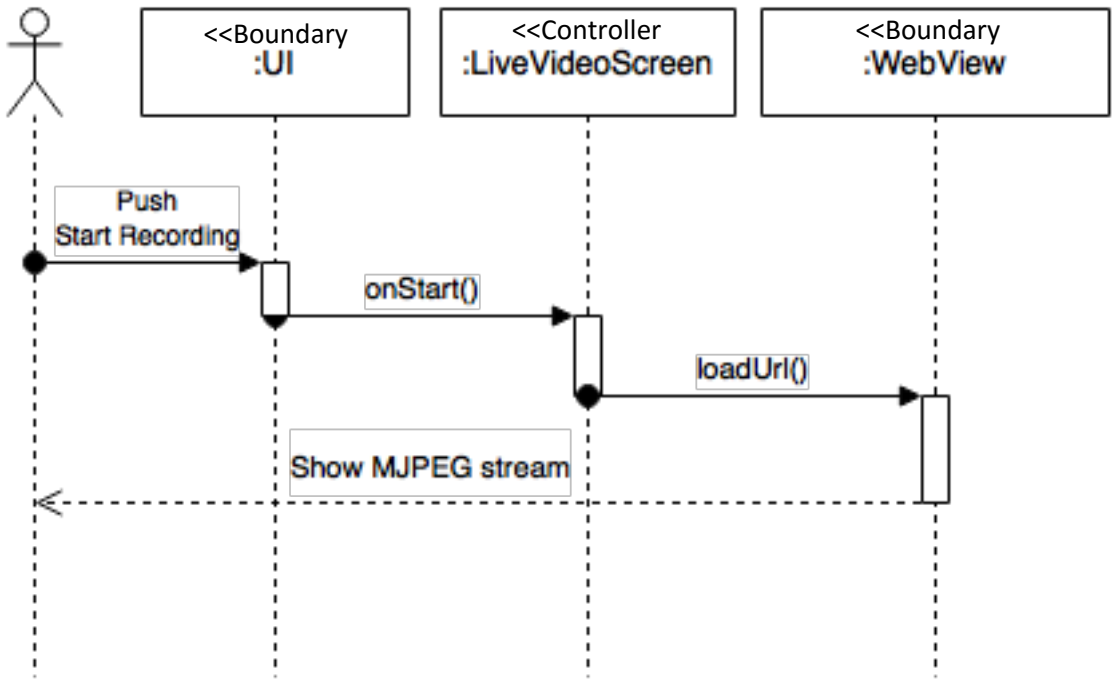

Figure 9: Use case 2 -Sequence Diagram

2.1.2.4 Use case 3: Navigate the car

This use case describes how the user moves the car with his smartphone. The user can move the car forward and backward or make it turn. Only one speed is available.

Figure 10: Use case 3 - Involved classes

UI:

The boundary-class UI is a class containing the graphic elements of the App and define the appearance of the App to the user. The system is implemented in the Android framework so this class is a gathering of XML file defining the UI for each controller.

LiveVideoScreen:

The LiveVideoScreen includes four buttons corresponding to the four directions. When they are pressed the information is relayed to the ArduinoConnection service.

ArduinoConnection:

The boundary-class ArduinoConnection is an interface allowing communication between the app and the microcontroller. When the ArduinoConnection service is notified that a button has been pressed, it tries to send a message to the microcontroller resulting in the steering of the car.

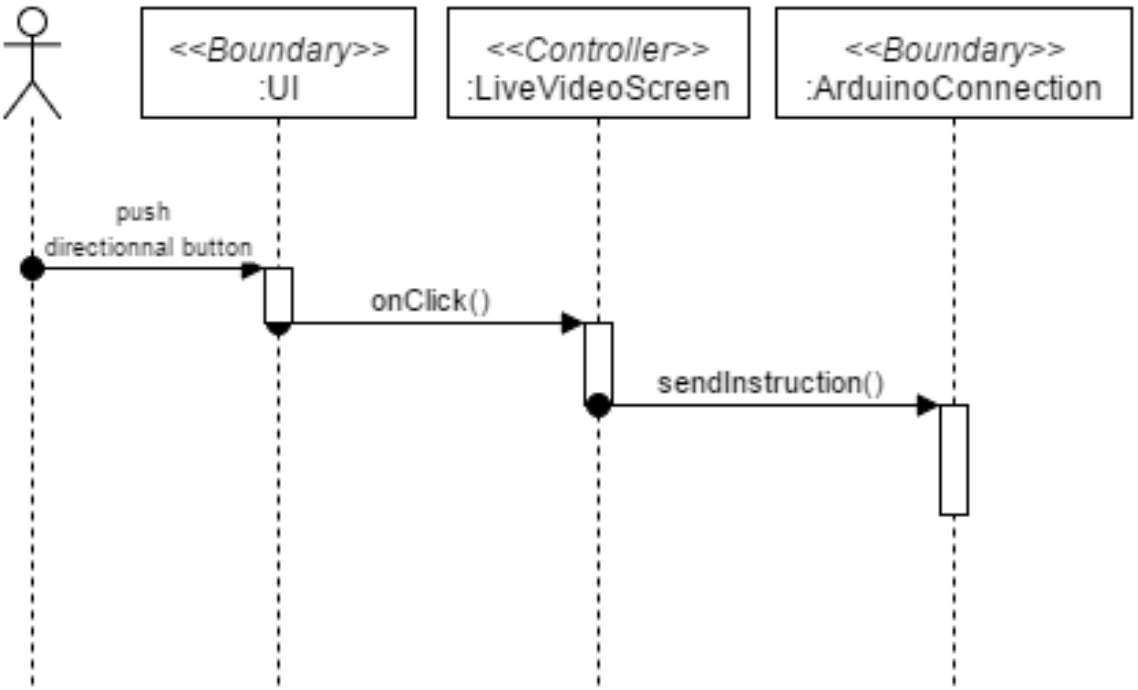

Figure 11: Use case 3 - Sequence Diagram

2.1.2.5 Use case 4: Drawing a map

This use case describes how the user can launch the process of collecting data then transforming those data into a 2D map. The user can press the "REC" button for the Microcontroller to starts sending points.

Figure 12: Use case 4 - Involved classes

UI:

The boundary-class UI is a class containing the graphic elements of the App and define the appearance of the App to the user. The system is implemented in the Android framework so this class is a gathering of XML file defining the UI for each controller.

LiveVideoScreen:

The LiveVideoScreen activity includes a "REC" button, when pressed the button color will change and a notification will be send to the ArduinoConnection service. When the user wishes to stop recording he just need to press the button again which will return to his previous state and inform the ArduinoConnection service.

ArduinoConnection:

The boundary-class ArduinoConnection is an interface allowing communication between the app and the microcontroller. When the ArduinoConnection service is notified that the "REC" button has been pressed, it launches a timer that will ask for data from the microcontroller at a regular interval. These data are stored in the database using the MapDataSource class.

When it received the information that the recording must be stopped from the LiveVideoScreen activity, the service stops the timer and launches the DrawingService.

MapDataSource:

The MapDataSource is a DAO containing the method to interact with the SQLite database. When first instantiated, it create the database using MySQLiteHelper.

MySQLiteHelper:

The MySQLiteHelper is the SQLiteHelper and contain the information needed to create the database, updating it or deleting it.

DrawingService:

The DrawingService is a service in charge of drawing the map from the points stored in the database by the ArduinoConnection service. When launched it retrieved the currently stored points in the database then, using these, it draw the map and create a PNG file which is saved in the picture directory of the smartphone's SD card. The file path of the PNG is saved in the database.

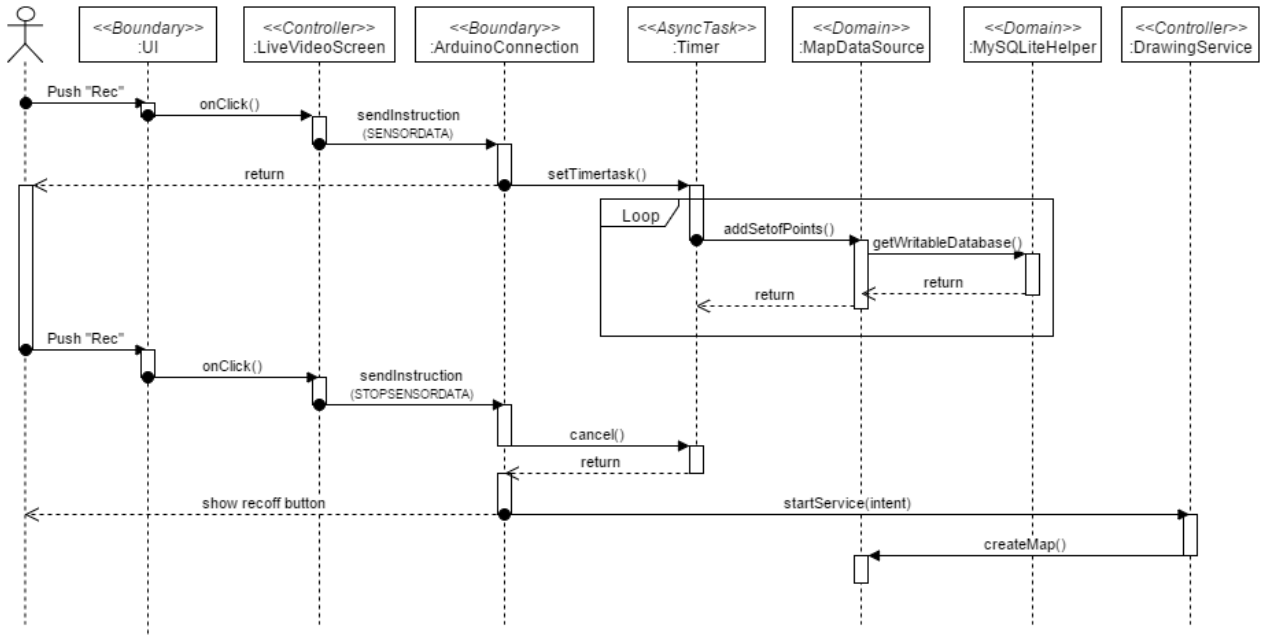

Figure 13: Use case 4 - Sequence Diagram

2.1.2.6 Use case 5: Manage recorded maps.

The use case describes the user's access to the list of saved maps. From the "control screen" and from the "start screen" the user can access the map list. The maps list consists of a list of the maps created by the user's smartphone from the car's results. The interface is separated in two parts: the list and a preview of the currently selected map.

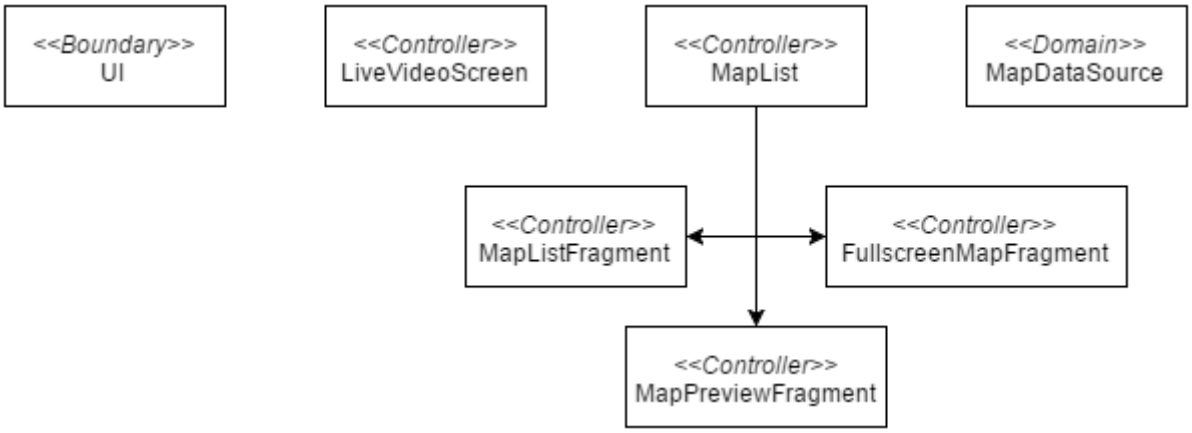

Figure 14: Use case 5 - Involved classes

UI:

The boundary-class UI is a class containing the graphic elements of the App and define the appearance of the App to the user. The system is implemented in the Android framework so this class is a gathering of XML file defining the UI for each controller.

StartScreen:

On the StartScreen, the user can press the "Map list" button to launch the MapList activity.

LiveVideoScreen:

Alternatively, the StartScreen activity includes a button which launches the MapList activity when pressed.

MapList:

MapList is the first activity launched when pressing on the "Map list" button. This activity is hosting three fragment used for managing the map: MapListFragment, MapPreviewFragment and FullscreenFragment.

MapListFragment:

The MapListFragment fragment is in charge of retrieving all the maps name stored in the database and displaying them as a list.

MapPreviewFragment:

The MapPreviewFragment fragment is in charge of displaying the currently selected map in the list. When the user selects one of the maps, the map name is send from MapListFragment to MapPreviewFragment by the MapList activity. MapPreviewFragment can then display a preview of that map.

FullscreenMapFragment:

When the MapPreviewFragment loads a selected map, the same map is loaded in the FullscreenMapFragment. The user can then switch between the two views by clicking on the preview or the full screen map.

MapDataSource:

The MapDataSource is a DAO containing the method to interact with the SQLite database. When first instantiated, it create the database using MySQLiteHelper.

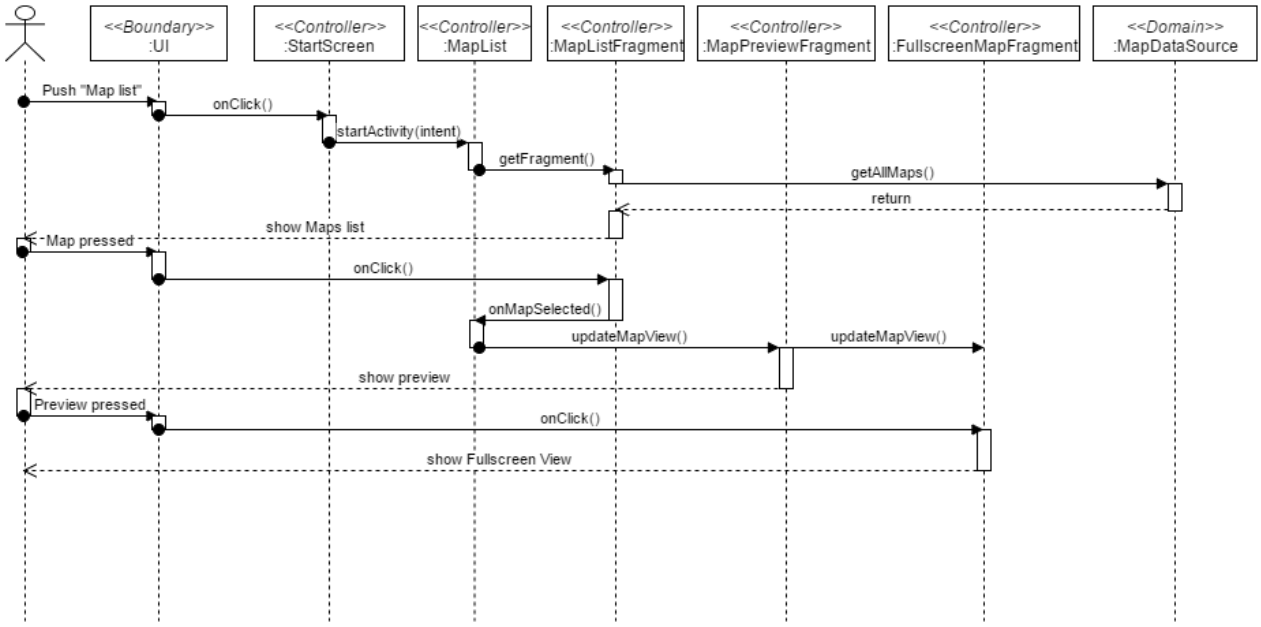

Figure 15: Use case 5 - Sequence Diagram

2.1.2.7 Use case 6: Delete map

This use case describes how the user can delete a map from the maps list. When a map is long-pressed, an Alert Dialog pops up letting the user delete the map or cancel the action.

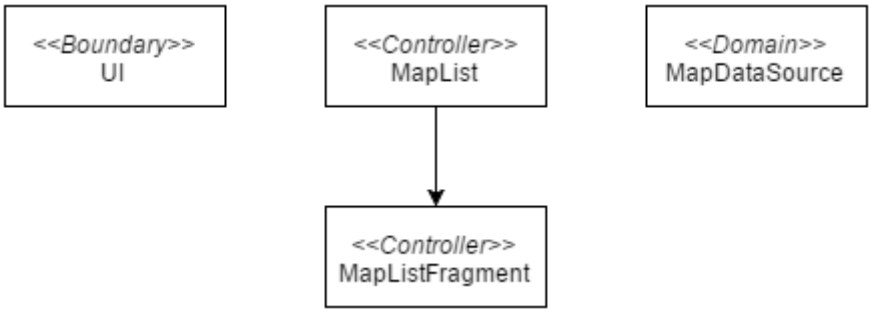

Figure 16: Use case 6 - Involved classes

UI:

The boundary-class UI is a class containing the graphic elements of the App and define the appearance of the App to the user. The system is implemented in the Android framework so this class is a gathering of XML file defining the UI for each controller.

MapList:

MapList is the first activity launched when pressing on the "Map list" button. This activity is hosting three fragment used for managing the map: MapListFragment, MapPreviewFragment and FullscreenFragment.

MapListFragment:

The MapListFragment fragment is in charge of retrieving all the maps name stored in the database and displaying them as a list. If the user presses one of the maps for a few second a pop-up will appear asking if the user wants to delete this map: pressing "yes" will delete the map while pressing "no" will close the pop-up.

MapDataSource:

The MapDataSource is a DAO containing the method to interact with the SQLite database. When first instantiated, it create the database using MySQLiteHelper.

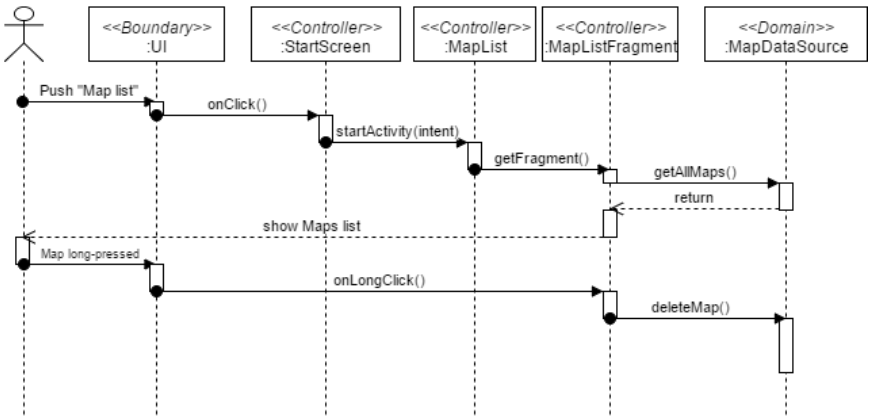

Figure 17: Use case 6 - Sequence Diagram

2.2 Car's perspective

Since the software for the microcontroller was developed in a Non-object oriented programming language, it's a bit tricky to create class diagrams for it. The following figures are an approximation of the real software.

2.2.1 Static class diagram

Here are shown the elements that the microcontroller uses. We have modeled as classes the Servo, Sensors, Car and Networking operations.

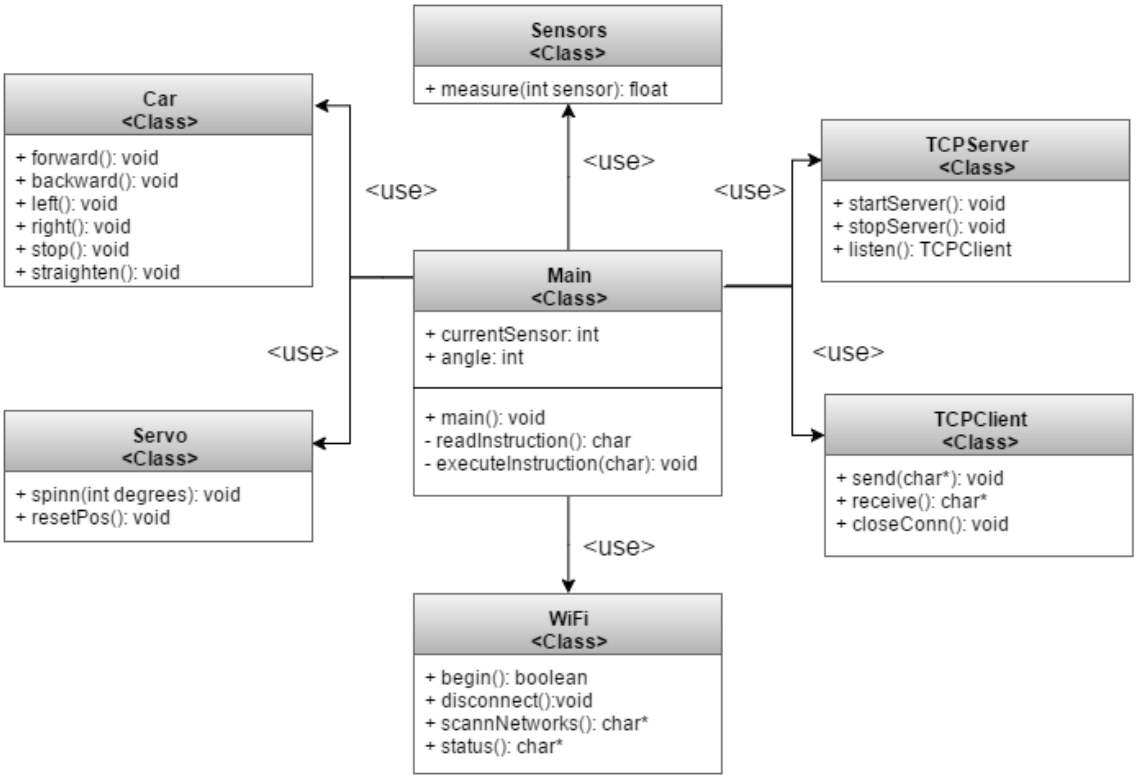

Figure 18: Car's static class diagram

2.2.2 Sequence diagrams

2.2.2.1 Use case 1.2: Connect to µController

Initialize connection to the specified network, once connected the TCP Server starts listening for the client. This sequence would be prior to the Smartphone's use case start.

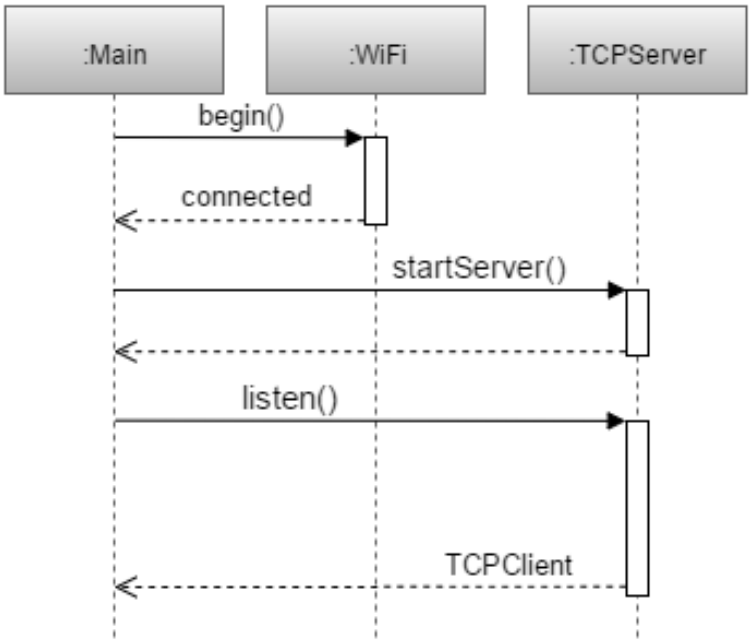

Figure 19: Sequence diagram - Use case 1.2 (Car's perspective)

2.2.2.2 Use case 3: Navigate the car

The car receives the instruction from the Smartphone, executes it and awaits for the next one. This sequence would be executed right after the Smartphone's use case end.

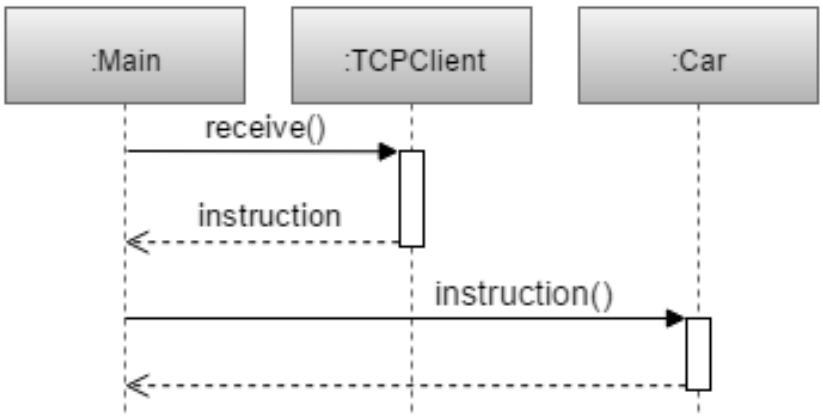

Figure 20: Sequence diagram - Use case 3 (Car's perspective)

2.2.2.3 Use case 4: Drawing a map

The Smartphone is ready to perform 9 times this same operation, to cover all 360º. The Arduino takes 4 measures, one for each sensor and builds the String sensorData to be sent to the Android. This sequence would be performed during the data retrieval loop at the Smartphone's use case.

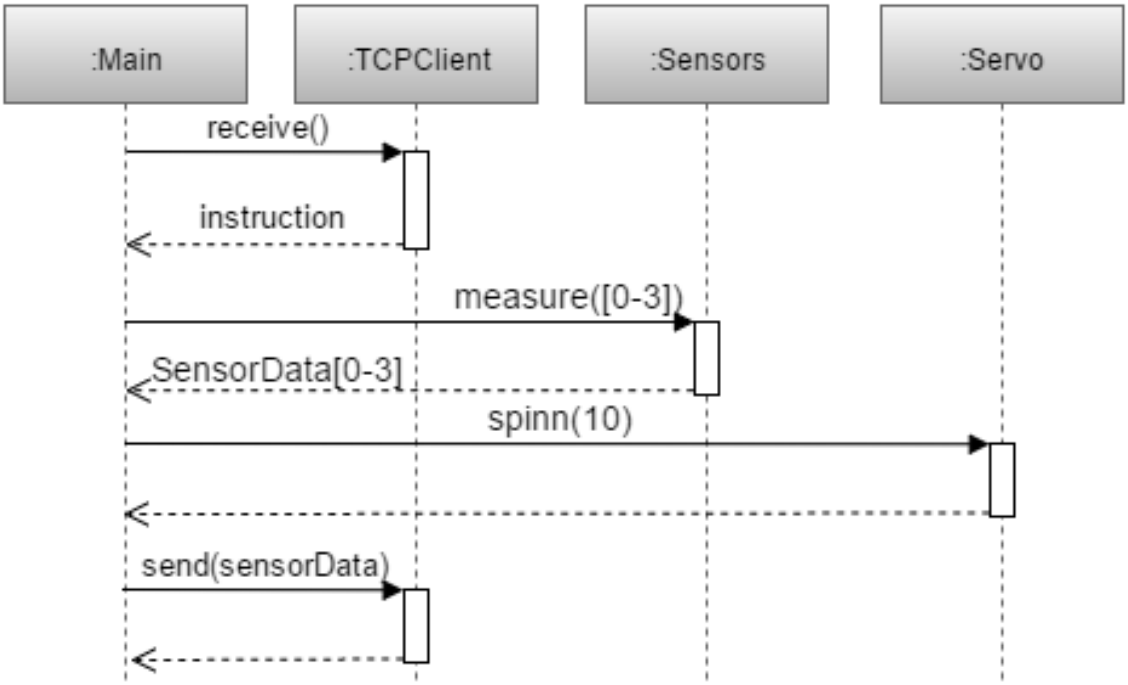

Figure 21: Sequence diagram - Use case 4 (Car's perspective)

3 Process View

In this section we'll describe the processes and therefore, its threads, of the different components of our system.

3.1 Main smartphone

Out smartphone is running one main process, that'd be the App. Inside the app we have several threads running:

- The UI Thread (Or main thread)
- Threads we launch to perform network operations (Mainly Async Tasks)
- Android System WebView (MJPEG steam)

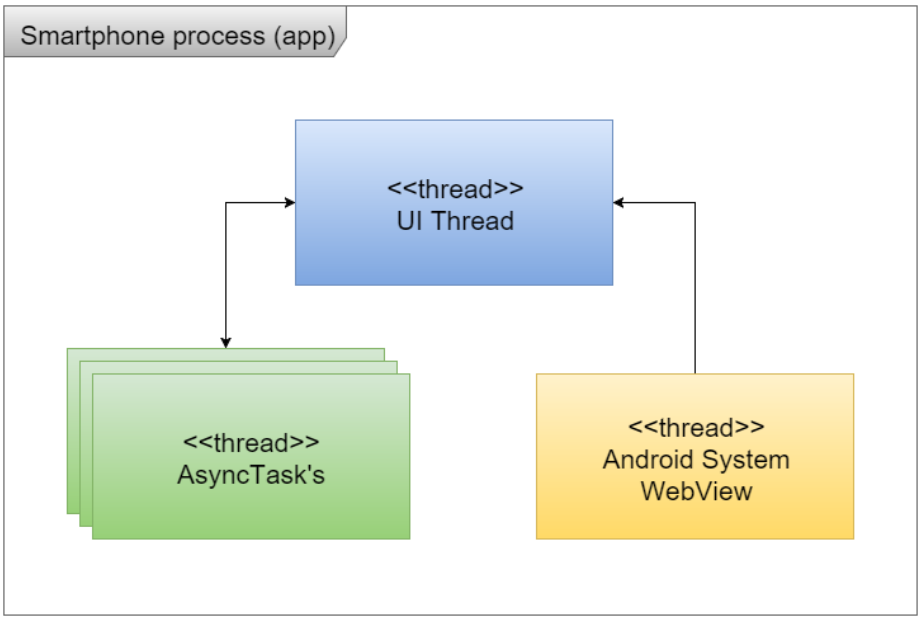

Figure 22: Smartphone's threads

The communication between the Android System WebView and the UI Thread consists of a continuous update of the MJPEG stream served by the secondary smartphone/IP Cam.

The UI Thread launches the Async Tasks to connect to the microcontroller, send commands and retrieve data from it.

3.2 Car

The car is composed by two CPU's, the microcontroller and the secondary smartphone/IP Cam. The microcontroller is the ATMega2560, it doesn't support multithreading so there'll be only one thread. Regarding the streaming device (secondary smartphone/IP Cam), we'll assume it only has one thread since we do not control its functionality.

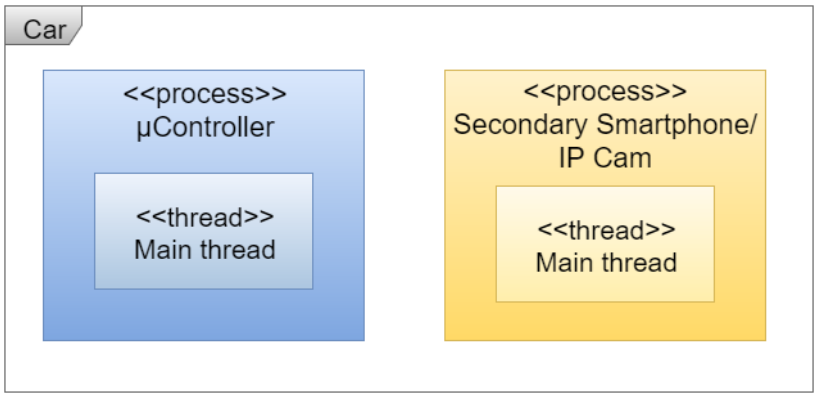

Figure 23: Car's processes and threads

3.3 Process interaction

The following figure shows how the threads communicate with the rest of the devices on the system.

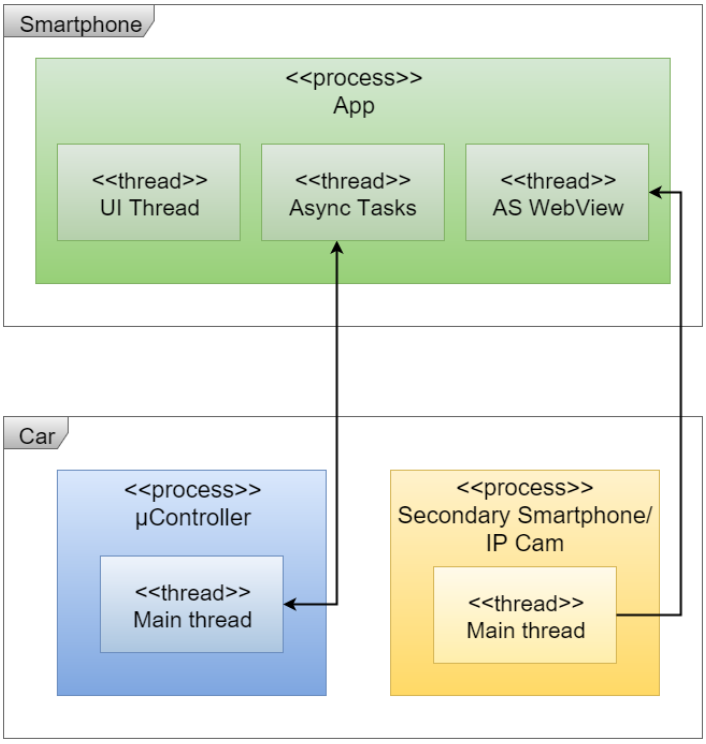

Figure 24: Thread and Process interaction

The App and the µController communicate via TCP sockets, they use them to send commands to the car and sensors data to the Smartphone. The Android System WebView retrieves the MJPEG stream that the Secondary Smartphone/IP Cam serves over HTTP.

4 Deployment View

In this section we'll be explaining how the different parts of the system are implemented on different platforms. Also, communication and protocols will be explained in detail.

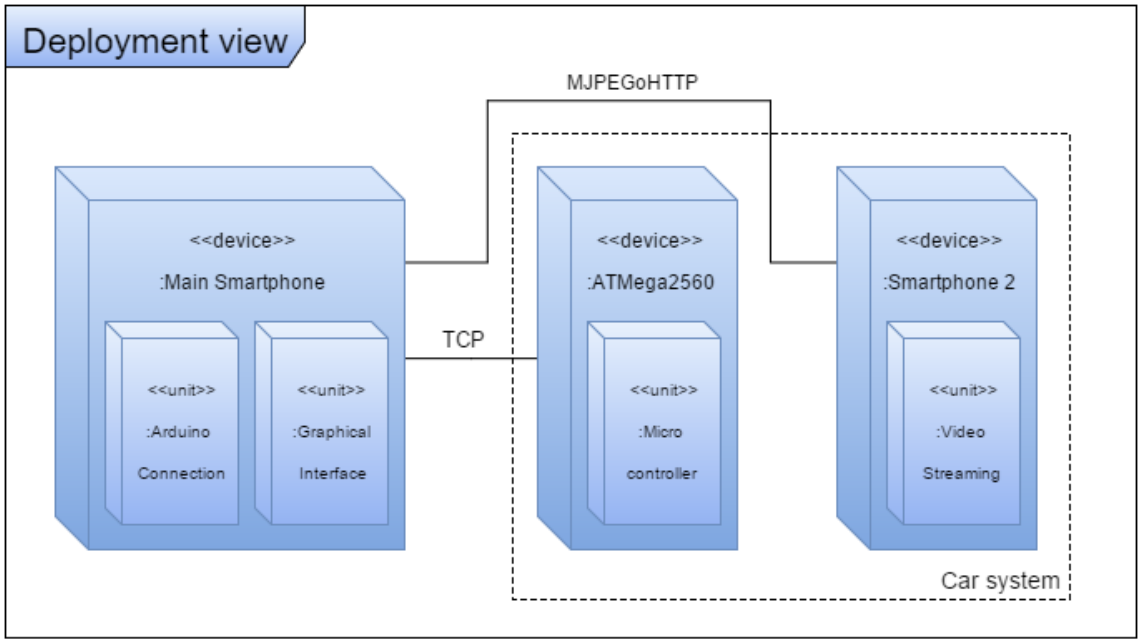

Figure 25: Deployment view

4.1 Node description

4.1.1 Main Smartphone

This device is the one that the user handles to interact with the system. We have two units in it: "Arduino Connection" in charge of dealing with the Arduino (ATMega2560) and "Graphical Interface" which between other tasks handles the video stream.

4.1.2 ATMega2560

The microcontroller in the Arduino, this is the main part of our car's system. We have attached a Wireless LAN board to connect via TCP to the Main Smartphone

4.1.3 Smartphone 2

This device is used solely to server an MJPEG stream that the Main Smartphone will use. Connected via WIFI to the same network as the Main Smartphone

4.2 Protocol description

Here we'll explain the protocol followed between the Main Smartphone and the microcontroller.

Once the smartphone and the microcontroller are connected via TCP sockets the microcontroller is continuously listening for commands. We have steering commands (Forward, backward, left and right), data commands (new set of points, stop sending) and the kill connection command.

When the user starts recording, the Smartphone will be ready to ask for 9 sets of points to the microcontroller, unless the user decides to abort the operation (In which case the map will be plotted only with the points received to that moment). When the Smartphone asks for the new set of points, then expects a set of points (Obvious) and sends the next command which can be either next set of points or stop sending.

4.3 Data format description

Each of the point sets are sent in Strings in the following format:

{dist1-ang1/dist2-ang2/dist3-ang3/dist4-ang4}

Where:

- float distN: Distance measured by the Nth sensor
- integer angN: Angle of the Nth sensor in that measure

5 Implementation View

5.1 IDE Setup

In order to deploy all the software to both the Android Smartphone and the microcontroller, we first have to setup the workspace.

5.1.1 Android

We used a physical device for testing and debugging. This implies we had to install the ADB drivers for such device, universal drivers can be found at developer.android.com.

Figure 26: Google USB Driver download

Once installed, we created a new project in Android Studio targeting SKD 23 and with minimum SDK 19.

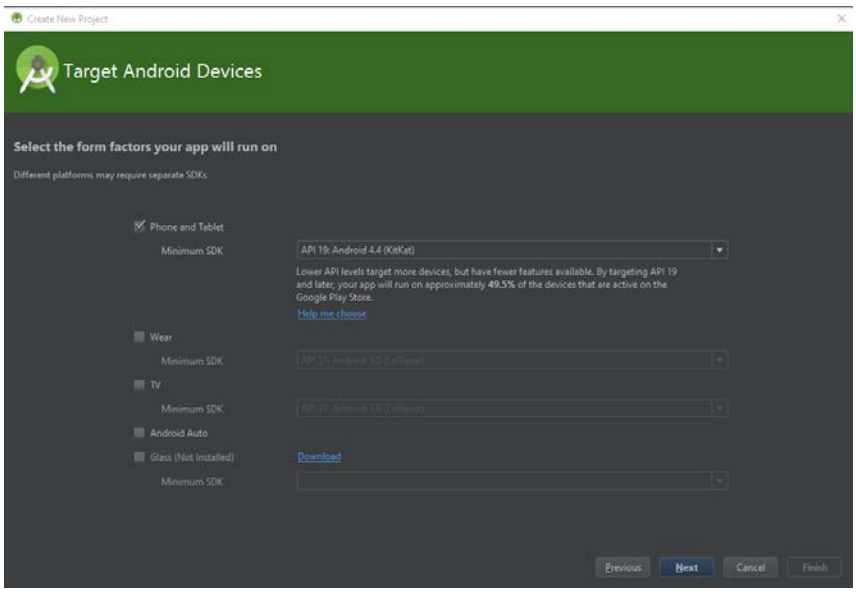

Figure 27: Android Studio Project configuration

5.1.2 Arduino

We had to install the Arduino IDE. A good alternative to this is using Atmel Studio 7, but this requires some further configuration and the official Arduino IDE was enough for our purpose.

Figure 28: Arduino IDE download

The only configuration needed with the Arduino IDE was choosing the right board and chip (ATMega2560).

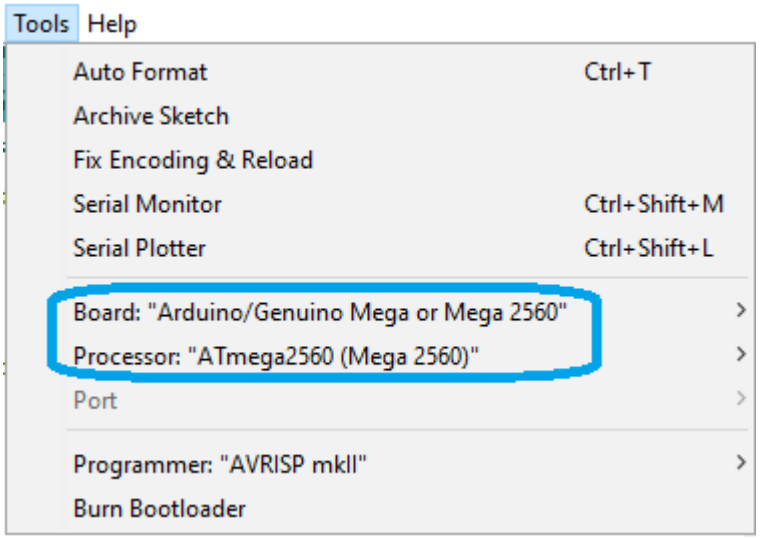

Figure 29: Arduino IDE configuration

5.2 Files organization

On the android, the application is divided into two packages:

- The “classes” package, containing all the activities, services, and database related classes.
- The “resource” folder containing the file related to the layout of the different views in the application including all pictures and sounds.

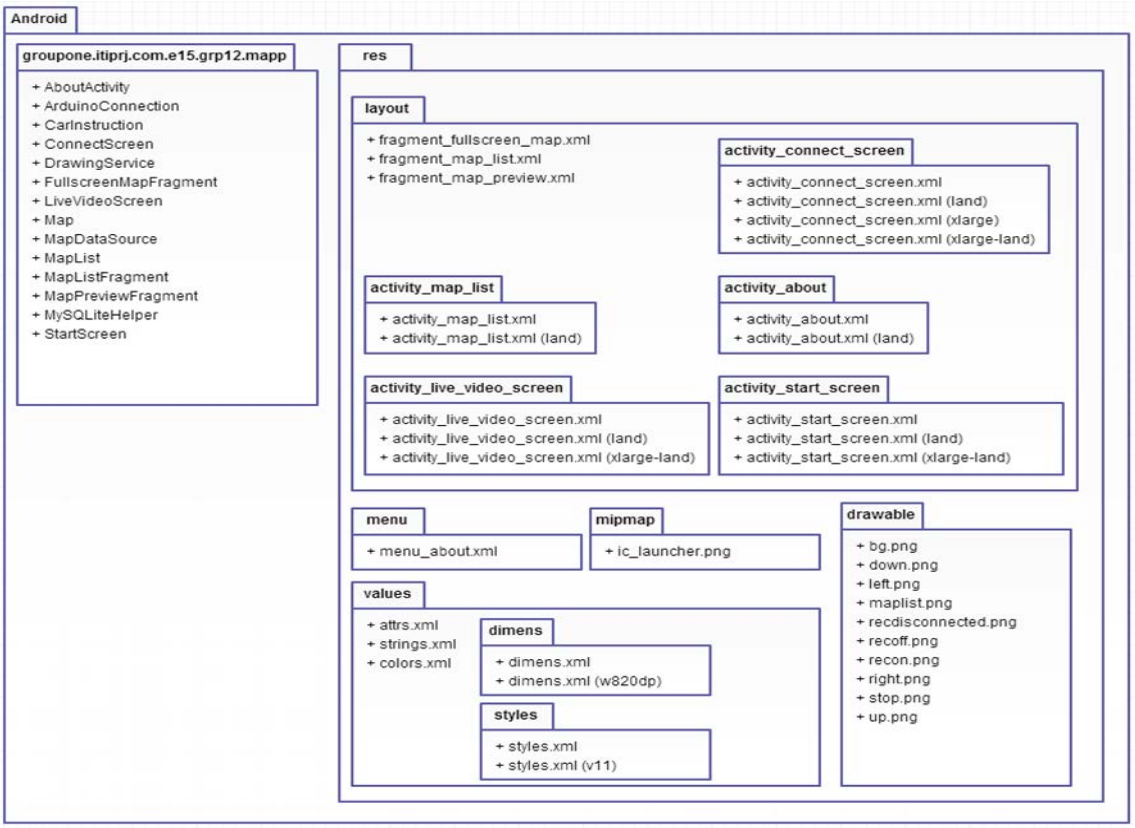

Figure 30: Package diagram

5.3 Coordinate calculation

The sensors used by the system can measure the distance between them and an obstacle in front of them, but for the DrawingService to make it into a map a conversion of the distance into x and y coordinates is needed. To make this conversion we use the different angles of the sensors. There’re four sensors, each one at a 90° angle with the ones next to it.

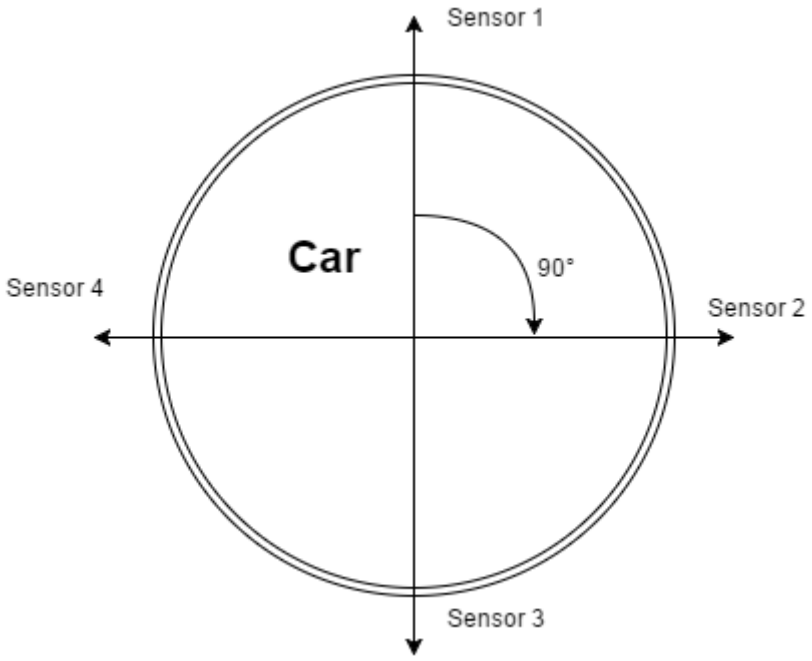

Figure 31: Sensors placement

The data sent by the microcontroller consist of a string with multiple pairs {Distance measured, Angle}. We choose the sensor 1, the one at the front of the car, as the angle 0°. When the servomotor make the platform turn we just add 10° to each sensors angle.

This data is retrieved by the DrawingService service which parses the string into a 2D array. For each {Distance Measured, Angle} the service calculates the corresponding x and y coordinate using the following formulas:

$$\sin(B) = \frac{opposite}{hypotenuse} \qquad \cos(B) = \frac{adjacent}{hypotenuse}$$

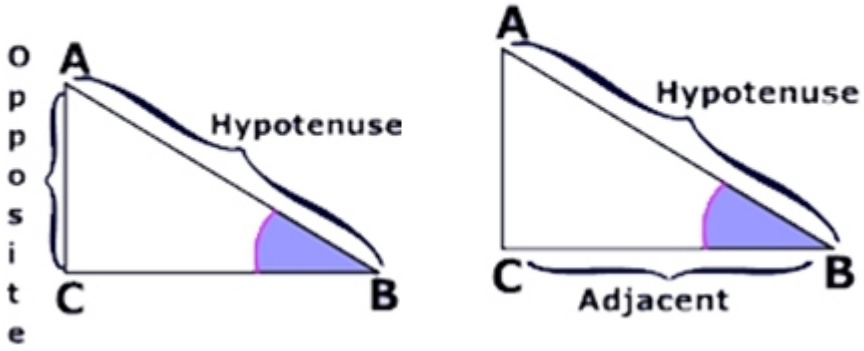

Figure 32: Trigonometry

In our system we have the hypotenuse (the distance measured) and its angle (The sensor's angles at the moment of the measurement) so we just apply the formula, our x and y coordinates being the opposite and adjacent sides.

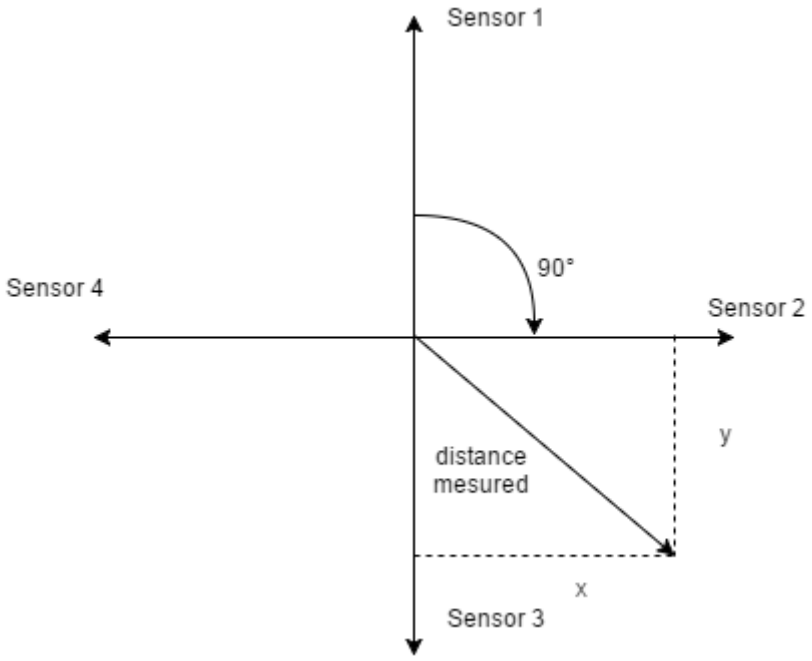

Figure 33: Coordinate calculation

5.4 Sensor measurement

The following code is the one we used to deal with the sensors. It triggers the signal and waits for it to come back while counting how much time does this take. Then converts this time to distance in cm and logs it.

```
//we activate the Trigger signal
  digitalWrite(Trigger, LOW);
  delayMicroseconds(2);
  digitalWrite(Trigger, HIGH);
  delayMicroseconds(10);
  digitalWrite(Trigger, LOW);

//We obtain the duration, using "pulseIn()" function
  duration = pulseIn(ECHO, HIGH);

//Calculate the distance (in cm) based on the speed of sound.
  distance = duration/58.31;

  Serial.write("Sensor:\r\n");
  Serial.println(distance);

//Delay before next reading.
  delay(400);
```

The function "*pulseIn*()" reads a pulse (either HIGH or LOW) on a pin. For example, if value is HIGH, *pulseIn*() waits for the pin to go HIGH, starts timing, then waits for the pin to go LOW and stops timing. Returns the length of the pulse in microseconds or 0 if no complete pulse was received within the timeout.

5.5 Servo operation

We have used an Arduino library to handle the servo (Servo.h). The procedure to use it goes as follows.

We first create a Servo variable *myservo.*

```
#include <Servo.h>

////////////  Variable for servo  ////////////////
Servo myservo; //Variable of my servo
int pos=0; //variable where we put the position
```

Then we set the pin we'll be using to operate it and set it to the initial position (0º).

```
/* SERVO */
myservo.attach(13);
myservo.write(pos);
```