Automotive Driver Assistant

Developer Manual

Gabriel Bulai
Enrique Cordero
Kristiyan Dimitrov
Hampus Gunnrup
Nicholas Otieno

Table of Contents

Introduction to the project

Automotive Driver Assistant is an app used by drivers. The app guides the driver to a safer and more economical drive, by using signals from the vehicle and evaluating them. The evaluation presents a set of scores for speed, braking, distraction level and fuel rate. Each score has a range of 0 – 100, with 100 being perfect. Our goal is to increase road safety and reduce CO2 emissions, by making an app that motivates this behavior in the driver.

Design Decisions

Android is very special in the way it works. It makes a very specific distinction between the UI layer and the application layer. Java is used to program the logic behind a view, which is programmed in XML. This type of architecture is very similar to MVC and this is why the team chose to apply this type of structure to the project. The project was also adapted to use UML layers. This is why the model section of MVC is divided between foundation and model. Some basic data structures are considered part of the model and some things like comparators are part of the foundation. It is divided in this way because of their functionality and whether or not they are complex data structures.

The team has also decided and tried to have a central controller, which handles most of the activities to reduce class dependencies. Future addition to the project should try to adapt a similar structure to be cohesive with the rest of the application.

Set up

To set up a development environment for the ADA app, you will have to go through four steps.

Install Android studio

1. Make sure java 1.8(JDK) or later is installed.
2. Go to https://developer.android.com/sdk/index.html and download the installer.

3. For Windows: Run the installer, and follow the instructions.
   For Linux: Unzip the files, open the file install-Linux-tar.txt and follow the instructions.

Clone the repository from Github

1. When the installation is complete, press Checkout project from version Control.
2. Follow the instructions, and make sure to select SE-M-Project-2015.
3. When finished, the project files may present errors.
   If this happens, go into File>Open, and select the project.
   This will open a new window, and the errors should go away.
4. Android studio may log a message in the console, asking to install the SDK. If so, follow the instructions.
5. When all of the above is finished, click the "Sync Project with Gradle Files" button in the action bar.

Run the app

1. Go to Tools>Android>SDK Manager.
2. Select Android 5.0.1 (API 21) or later.
3. Press install and follow the instructions.
4. Press the play icon in the action bar.
5. Select Launch emulator, and choose the preferred device.
6. Press Ok.

Connect with AGA signals

1. Go to https://developer.lindholmen.se/redmine/projects/aga/wiki/Release_Notes_AGA_1_1.

For windows:

2. Download the simulator for windows.
3. Unzip the files.
4. Open Simulator.exe.
5. Open cmd and run the commands:

i. [your sdk path]\platform-tools\adb.exe forward tcp:8251 tcp:8251
ii. [your sdk path]\platform-tools\adb.exe forward tcp:9898 tcp:9898
iii. [your sdk path]\platform-tools\adb.exe forward tcp:9899 tcp:9899

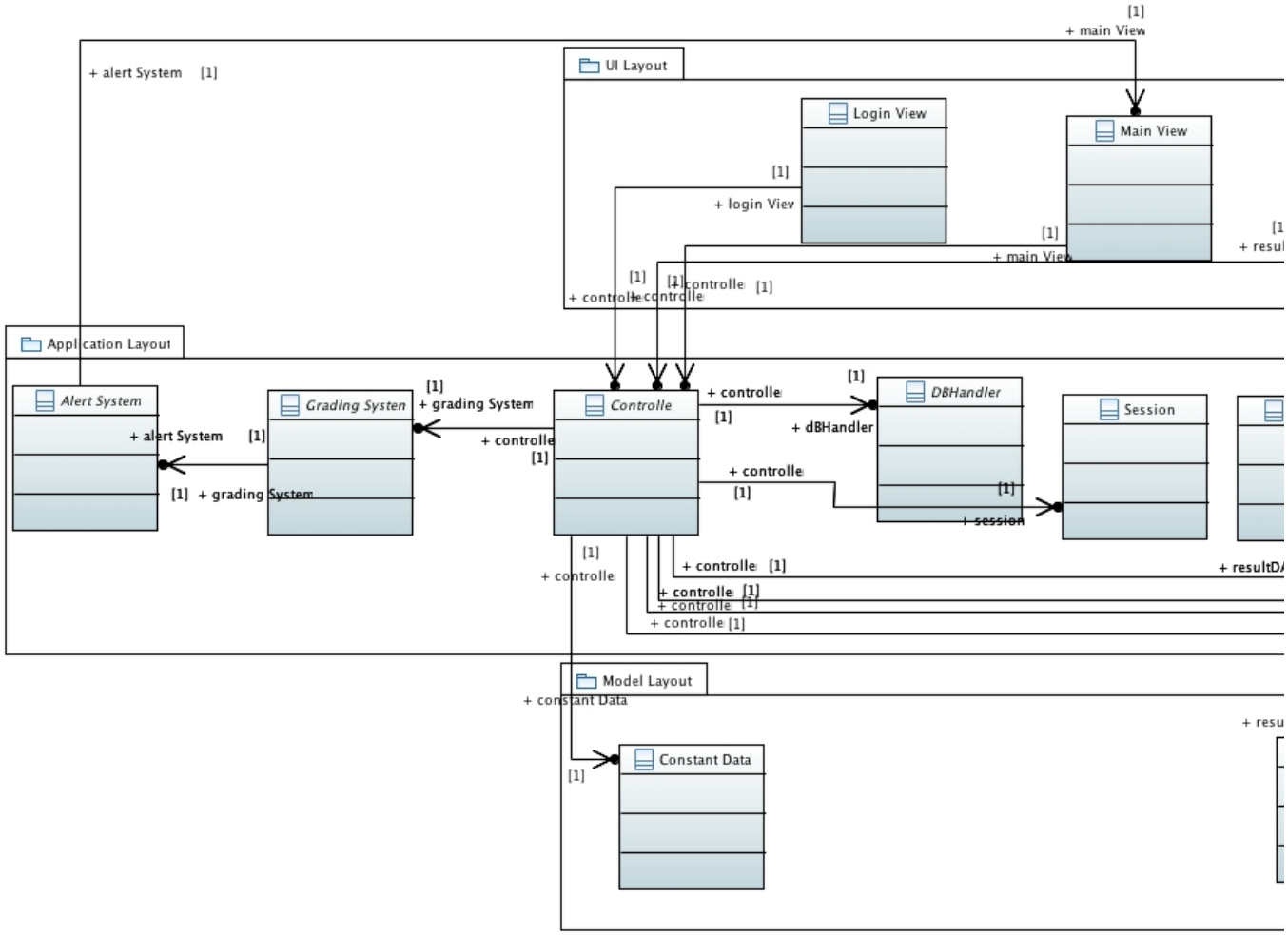
6. Repeat steps 4 and 5 every time you run the app.

For linux:

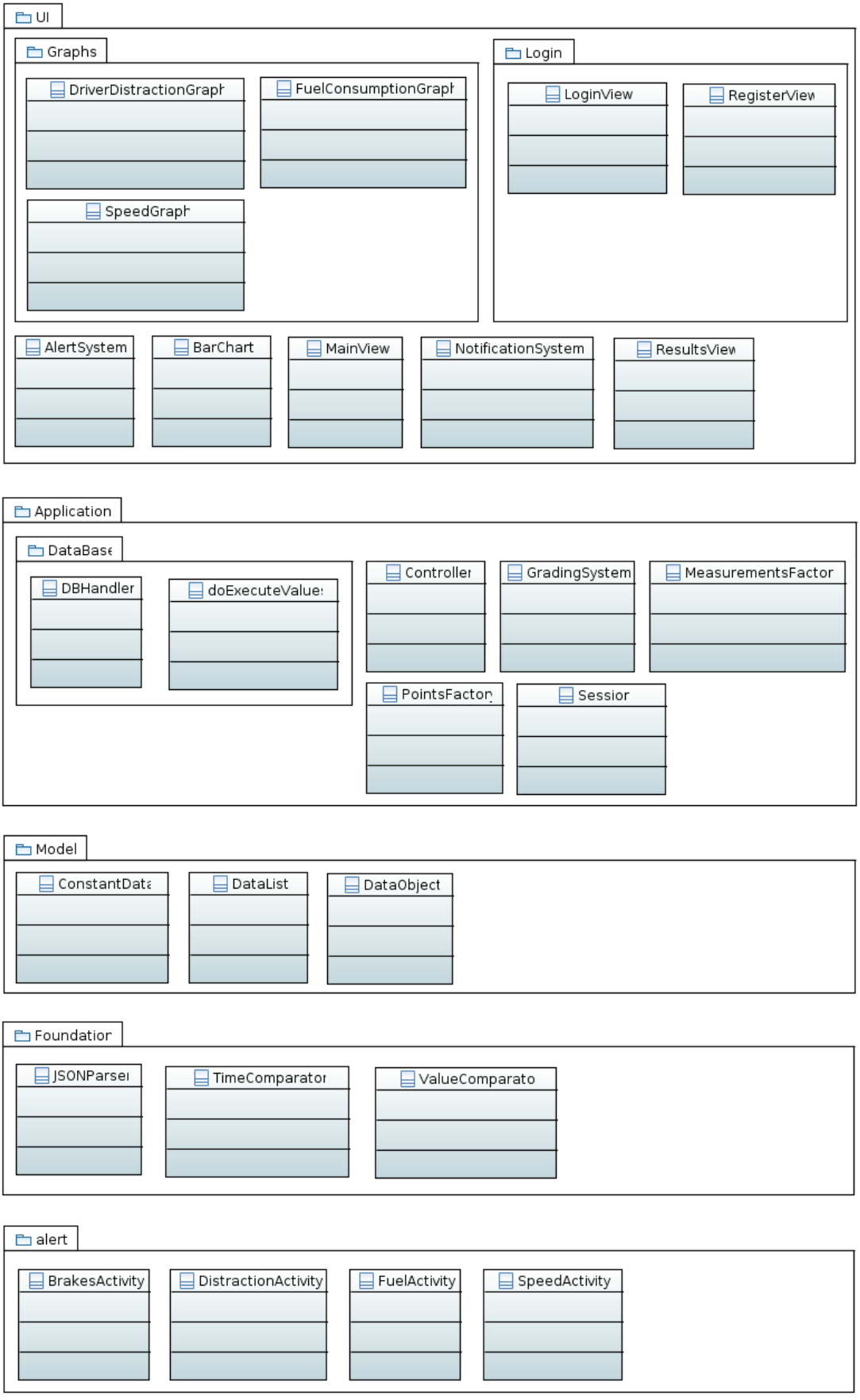
2. Download the simulator for linux.
3. Unzip the files.
4. Open simulator-fx.
5. Open a terminal and run the commands:
   i. ./[your sdk path]/platform-tools/adb forward tcp:8251 tcp:8251
   ii. ./[your sdk path]/platform-tools/adb forward tcp:9898 tcp:9898
   iii. ./[your sdk path]/platform-tools/adb forward tcp:9899 tcp:9899

UML

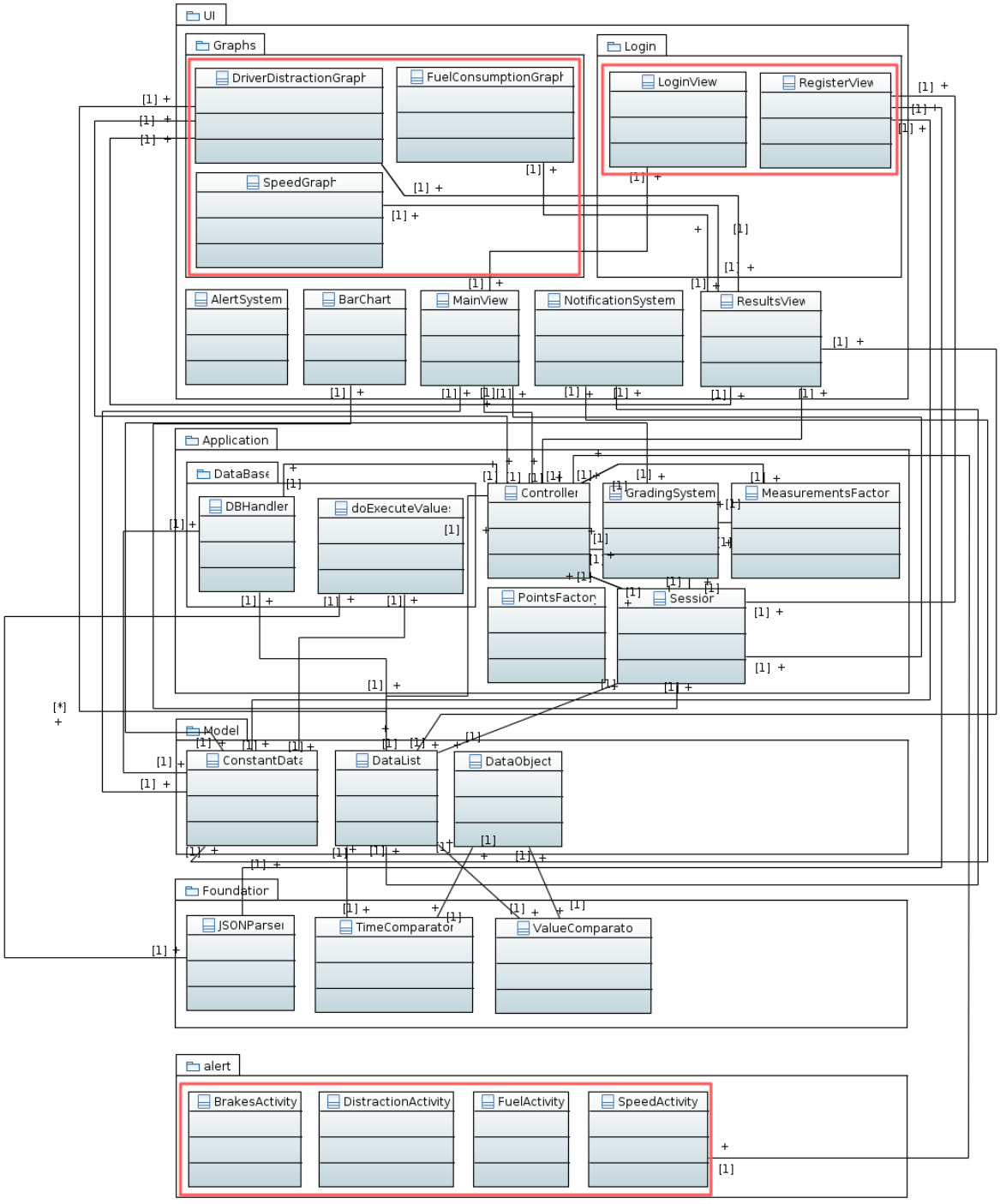
This is the original design of the application during the alpha version.



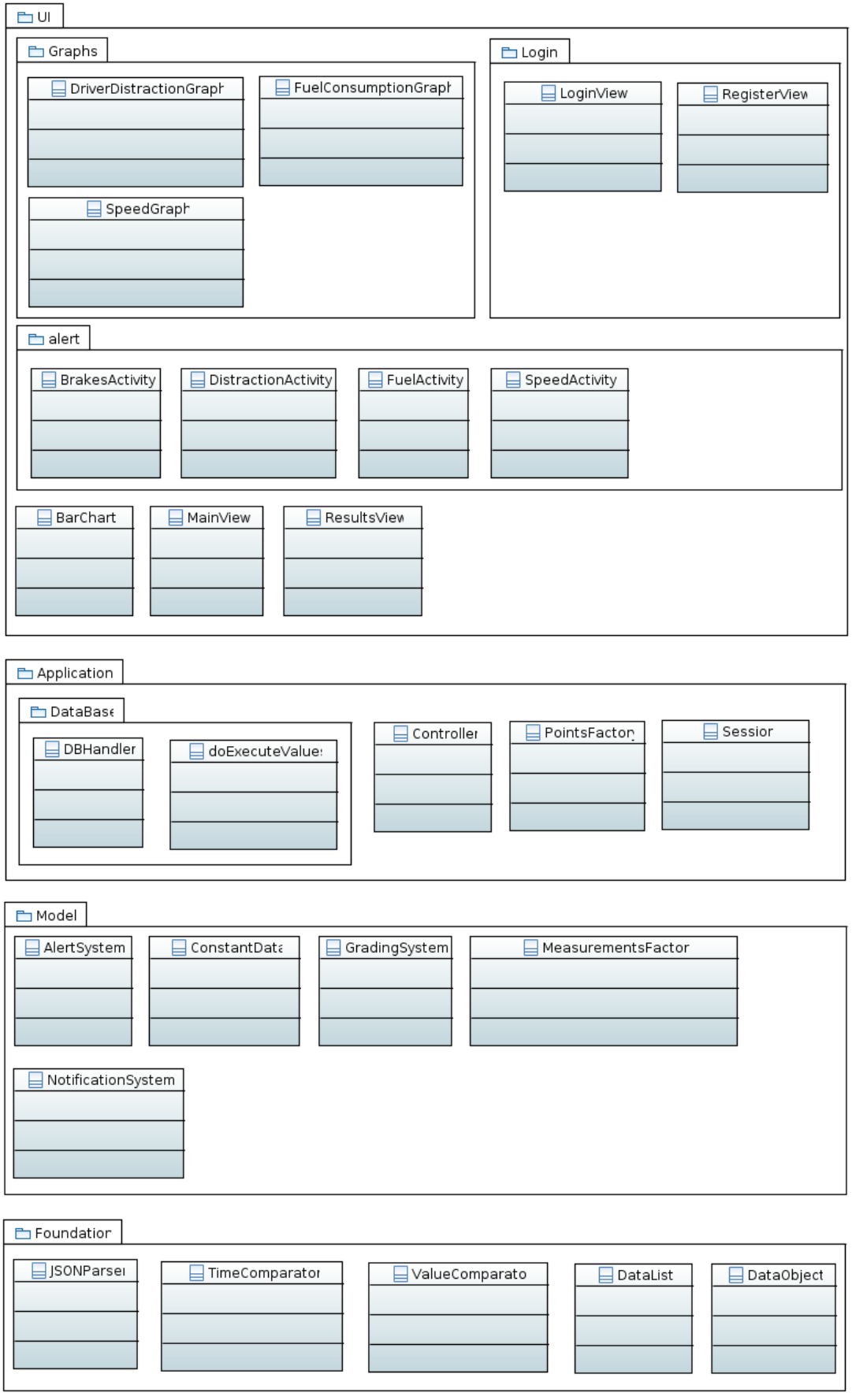
This is the design for the Beta Presentation. It shows the different layers in the project.



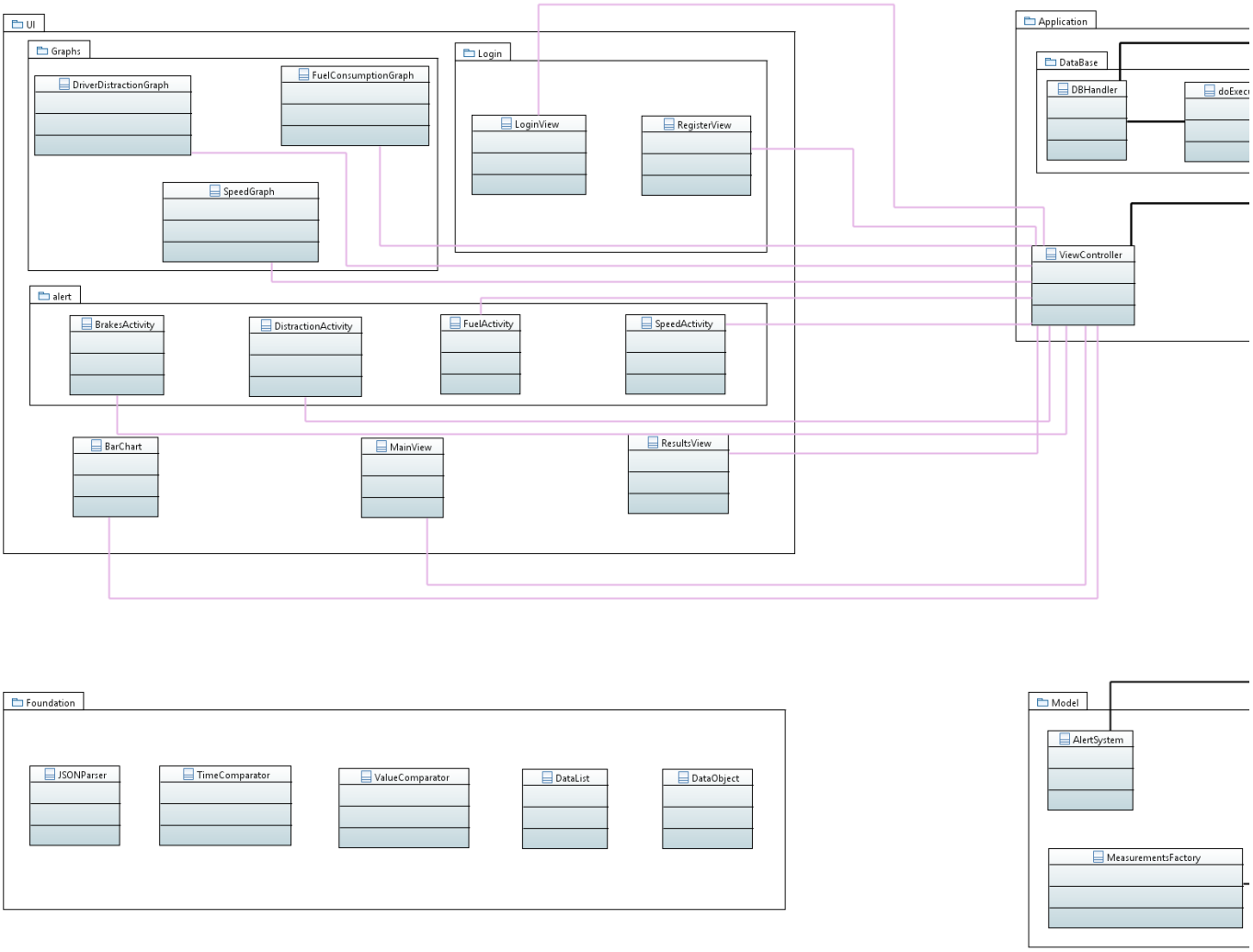
This is the design for the Beta Presentation. It shows the different relations between the classes.



This is the design that was used for the final sprint. It shows the different layers of the project.



This is the design that was used for the final sprint. It shows the different dependencies between the classes.



Architecture and code structure

Database

The database is setup on MySQL language. It is conformed of eleven tables. Eight of these tables are directly linked to the four main measurements (speed, brake, fuel consumption, and driver distraction), each measurement having a table for points and a table for the actual value of the measurement. These needed to be separate tables since the measuring is done individually per type of measurement and therefore they could be measured at different times. The other three tables are a user table where their username, password and salt are stored, a friends table where friends are stored to each user, and a final measurements table where the final measurements for a drive are stored for each user. The database is currently setup on a web service website however an SQL file will be annexed to the code to be able to set up the database in whichever server is chosen (including local host). The most important thing is to check the Back End to check the appropriate file (config.inc.php) and change where the host is located. For more specific or individual information on the database please open the database and check the inner structure.

Back End

The back ends takes care of all the functionality involving the database. It receives a request from a program in the form of a post. The post shall have an action to be performed and a given set of variables depending on the action requested. Please check each individual method to make sure you are supplying the right information regarding each action chosen.

The main file in the back end is the index, which is the contact to either the application or the webpage. Here the action is read and processed and the right database files are called to execute the action requested.

Index

The index is the main contact point with the front end. The post request is received here and processed accordingly with the action request inside the post. There is a check for all possible actions. If the action requested does not match any of the possible actions then an error message is returned. The possible actions are:

- login: Login choice. Reads the username and the password and calls the *login* method.
- register: Register choice. Reads the username and the password and calls the *register* method.
- fblogin: Facebook login choice. Reads the username and calls the *fblogin* method.
- setMeasurements: Set measurements choice. Reads the username. Decodes the JSON array for the list. Calls the *setMeasurements* method.
- setPoints: Set points choice. Reads the username. Decodes the JSON array for the list. Calls the *setPoints* method.
- getMeasurements: Get measurements choice. Reads the username. Calls the *getMeasurements* method.

- getFilteredMeasurements: Get filtered measurements choice. Reads the username, start and stop. Calls the *getFilteredMeasurements* method.
- getPoints: Get points choice. Reads the username and calls the *getPoints* method.
- getFilteredPoints: Get filtered points choice. Reads the username, start and stop. Calls the *getFilteredPoints* method.
- setFriend: Set friend choice. Reads the username and friend to be added to that user. Calls the *friendSetExecuter*.
- removeFriend: Remove friend choice. Reads the username and the friend to be removed from that user. Calls the *friendRemoveExecuter*.
- setFinalScore: Set final score choice. Reads the username along with his scores. Reads the speed, brake, fuel and distraction scores along with the measuredAt value. Calls the *finalMetricsSetExecuter*.
- getFinalScore: Get Final Score choice. Reads the username and calls the *getUserFinalMetrics* method.
- getFilteredFinalScore: Get filtered final score choice. Reads the username, the start and the stop and calls the *getUserFilteredFinalMetrics* method.
- getAllScores: Get all score choice. Calls the *getAllScores* method.
- getAllFilteredScores: Get all filtered scores choice. Calls the *getFilteredScoresAllUsers* method.
- getFriendsScores: Get friends scores. Reads the username and call the *getFriendsScores* method.
- getFriendsFilteredScores: Get friends filtered scores. Reads the username, start and stop and calls the *getFriendsFilteredScores* method.
- getAllFriends: Get all friends choice. Reads the username and calls the *getAllFriends* method.

Config

The config file declares the way the back end will connect to the database. All the important information like host, username, password and database name will be declared here. There is also a declaration of all the characteristics of the connection. For example it is declared that it will be with UTF-8 format and that the type of connection is through PHP using the PDO library. It configures the type of exception the database can throw when the connection encounters a problem and how the return type will be concerning the way the information is formatted. Currently it is set so that the PDO library will connect to the database and return database rows from your database using an associative array. This means the array will have string indexes, where the string value represents the name of the column in the database.

Currently the Config file is used in the index as required once. This way the $db variable will be initialized once and used as a parameter to call different functions.

Database Files

The database files contain all the files necessary to support the possible actions in the index. The idea with having these files in a separate package is to be able to make this package a private package. The front end does not need these files, therefore there should be a different depth to them.

Login

The login file contains the *login* method. Here the variables are received to prepare the query along with its query parameters. The password is checked along with its salt to see that it matches the hashed password stored in the database. The method will return a standard response with a success value and a message related.

This method makes use of an auxiliary method in the *functions.php* file called *checkhashSSHA*.

FBLogin

The fblogin file contains the *fblogin* method. Because the user would have already logged in through Facebook and his credentials checked by a third party then it is unnecessary to make a whole new process to log in. Instead it uses the regular *login* method and the *register* method as auxiliary functions. First it checks if the user exists by trying to login. If not possible (meaning the user does not exist) then it proceeds to register the user instead. The method returns a standard response with a success value and a message.

Register

The register file contains the register method. This method receives the database instance, the username and the password to register. First it checks that both the username and password contain a value. Then it checks whether or not the username already exists. If the username already exists then it returns a response with a success value of 0 and a message that says that the user already exists. If the user does not exist in the database then it proceeds to create the query with the query parameters. It adds the user to the database and returns a message in standard form with a success value and a message.

User Functions

The functions file contains methods, which can be defined as helper methods. These methods provide support for the *login* and *register* methods. The methods inside the file are *hashSSHA*, *checkhashSSHA* and *userExists*. The first two methods hash the password, and check a password against its hash and salt. The third method just checks if a user already exists.

Friends

The friends file contains two basic methods for managing friends and a database method, which could be considered a helper method. There is an add friend method called *friendSetExecuter* and a remove

friends method called *friendRemoveExecuter*. These two methods basically prepare the SQL query and the parameters that go along with it. Once they are ready they call the helper method called *friendsExecuter* to connect to the database and execute the SQL query along with its query parameters.

Setters

Setters are functions, which take information and store it in the database. Like most other files in the back end setters are divided into three types. There are points setters and measurements setters, which prepare the query along with the query parameters. They both use a third type of setter, which is the metrics setter. This file is the one in charge of communicating with the database and proceeds to run the query along with its query parameters.

SetPoints

The setPoints file contains one main method along with four helper methods. The *setPoints* method receives a list of all the points to be stored in the database. It iterates through the list and divides the measurements in each sub-array into the type of measurement they are: speed, brake, driver distraction and fuel economy. Once it has the singular values it can then proceed to call each of the helper methods, which are called *setBrakePoints*, *setSpeedPoints*, *setFuelPoints* and *setDistractionPoints*. These methods construct the appropriate SQL query along with its query parameters. Once done each method will call the *metricsSetExecuter* to store the information in the database.

SetMeasurements

The setMeasurements file contains one main method along with four helper methods. The *setMeasurements* method receives a list of all the points to be stored in the database. It iterates through the list and divides the measurements in each sub-array into the type of measurement they are: speed, brake, driver distraction and fuel

economy. Once it has the singular values it can then proceed to call each of the helper methods, which are called *setBrakeMeasurements*, *setSpeedMeasurements*, *setFuelMeasurements* and *setDistractionMeasurements*. These methods construct the appropriate SQL query along with its query parameters. Once done each method will call the *metricsSetExecuter* to store the information in the database.

SetMetrics

The setMetrics file contains three basic methods. Like most other classes it is compiled of two main methods and a helper method. The two main methods are called *metricsSetExecuter* and *finalMetricsSetExecuter*. The two methods basically build an SQL query along with its appropriate parameters and call the final method called *executeSet*, which can be described as a helper method. It connects to the database and executes the set of the values received through the query along with the parameters. The *finalMetricsSetExecuter* executes the saving of parameters coming as final metrics (one value of each of the four measurements). The *metricsSetExecuter* is used both by points and by measurements and therefore it is a bit more generic in the fact that it uses a table name and a column name and therefore can be reused by several values.

Getters

Getters are functions which get information from the database upon request and return such information in a given data structure. Because of the back end being on a web service then the chosen data structure is a JSON array with a defined structure depending on the type of information requested. The JSON array however will always contain a success value of 0 or 1 depending on whether or not the get was successful and a message, which explains the situation (also successful or unsuccessful). All get functions will return a third value but this one depends on the type of get action requested by the user. Often it will be

values of the four measurements, however lists are also a possibility in case of several values requested. These values will often be complied in an object of JSON array and can even be an array of arrays. These results will often be found in the subsection called "posts". Refer to each particular method to clarify what it will return.

GetPoints

The getPoints file is consisted of two main methods and eight helper methods. The two main methods are *getPoints* and *getFilteredPoints*. These two methods take a username and return the values for all four measurements and compile them in a list. They return the appropriate response along with the list. Each method has four helper methods. The helper methods are *getSpeedPoints*, *getBrakePoints*, *getDistractionPoints*, *getFuelPoints*, *getFilteredSpeedPoints*, *getFilteredBrakePoints*, *getFilteredDistractionPoints*, and *getFilteredFuelPoints*. Clearly four of them are related to the *getPoints* method and the other four contain the word filtered so they help the *getFilteredPoints* method. All of the helper functions proceed to call either the metrics *metricsExecuter* or the *filteredMetricsExecuter* accordingly.

GetMeasurements

The getMeasurements file is consisted of two main methods and eight helper methods. The two main methods are *getMeasurements* and *getFilteredMeasurements*. These two methods take a username and return the values for all four measurements and compile them in a list. They return the appropriate response along with the list. Each method has four helper methods. The helper methods are *getSpeedMeasurements*, *getBrakeMeasurements*, *getDistractionMeasurements*, *getFuelMeasurements*, *getFilteredSpeedMeasurements*, *getFilteredBrakeMeasurements*, *getFilteredDistractionMeasurements*, and *getFilteredFuelMeasurements*. Clearly four of them are related to the *getMeasurements* method and the other four contain the word filtered so they help the

getFilteredMeasurements method. All of the helper functions proceed to call either the metrics *metricsExecuter* or the *filteredMetricsExecuter* accordingly.

GetMetrics

The getMetrics file contains three basic methods. Like most other classes it is compiled of two main methods and a helper method. The two main methods are called *metricsExecuter* and *filteredMetricsExecuter*. The two methods basically build an SQL query along with its appropriate parameters and call the final method called *executeGet*, which can be described as a helper method. It connects to the database and executes the set of the values received through the query along with the parameters. It then receives the data from the SQL database and parses through it to convert it to an array with the appropriate structure. The main methods return the response from the *executeGet* in the standard format of success, message and a list.

FinalMetrics

The finalMetrics file contains four type of methods. As all the others it contains a method that is in charge of executing the SQL query along with its query parameters. This method is called *executeGetFromDB*.

The second type of method is the parsing methods. These two are in charge of getting the data in SQL form and parsing through it to return an array in JSON form. Here we find two methods called *parseFinalMetrics* and *parseScores*. They will return different lists since one of the methods is uses to parse through only the four last values measured and the other method will return all the last values in the database.

The third type of method is the helper methods, which are used as support methods for the rest of the functions in the file. Because different methods need to know which friends are related to a user then one of

the helper methods get a list of the friends of a specific user, this method is called *getUserFriends*. The other helper method is called *getAllUsers* and this one get a list of all the users in the database. This is to be able to iterate through the users and get all their final measurements.

The last type of methods are the main methods. These are the methods called from the index. These methods will call a sequence of other methods to accomplish the answer needed. They will always return a response in standard form of success, message and a third value usually in the form of a list.

Front end

The front end consists of all the java classes, layouts, drawables and other resources used closely with the application. To understand the structure of the application, knowledge about how android handles packaging is necessary. This section briefly explains the structure of the front end, and what each part does.

Layers

The java part of the front end, which handles the application logic, is structured in a set of layers. Each layer has a certain amount of application dependency and should be handled with this in mind. The root folder for the layers is, app/src/main/java/group8/com/application/.

UI

UI is the layer where all the classes that are closely communicating with the interface, resides. These classes are highly application dependent, meaning that reuse of these classes is extremely low. If a class which communicates with an interface(a layout for instance) or is used closely by the user, this class should be put in the UI layer.

Application

The Application layer mainly consists of the application logic. All the classes that evaluate signals from AGA, handles database connections, keeps track of the current session, alert the driver, etc, is put in the Application layer. The classes in this layer is also application dependent. They are, however, reusable and can for the most part be used with a new interface. If a class is handling any logic, and is not communicating directly with the user, it should be put in the Application layer.

Model

The Model layer is used for independent data such as constants and certain data types. This data is highly reusable, but can still be specific for the application.

Foundation

This layer is similar to the Model layer. It contains classes with small logical parts that tend to be reused in several places.

Layout

The layout folder is used to store all of the xml layouts used in the application. Most of them connect to one or several java classes in the UI layer. The root folder for the layouts is, app/src/main/res/layout/.

Drawable

There are several drawable folders. One for general use, which is plainly called "drawable". This folder contains all of the reusable drawable backgrounds. It also contains some images that only have one size. The other drawable folders are divided into five folders, with each folder being dedicated for a certain dpi, staring with medium and ending with xxxhigh. These folders are used to put various images such as icons and logos. There should be five sizes of each picture, one in each of the five drawable folders. This makes the app more scalable.

Values

The value folder is used to store constant data, used by other xml files, such as layouts and drawable backgrounds. The main files that are used are colors.xml, which contains hexadecimal colors stored with names, dimens.xml, which contains commonly used dimensions and strings.xml, which stores all the text, used within other xml files. Before using colors, strings or dimensions in any xml file, these value files should be checked to see if there is anything in them that might match the wanted values. Furthermore, if new values are used, they should be stored in these files with appropriate names.

Menu

The menu folder contains custom menus. When creating a view that will use an inflatable menu, that menu should be made as an xml file(look at androids documentation) and put in the menu folder.

Raw

In the raw folder, there are currently five sound files. This folder can be used for many different files, and androids explanation is to put "Arbitrary files to save in their raw form.".