Politecnico di Milano
MSc Computer Science and Engineering

Software Engineering 2
Academic Year 2016-2017

Design Document

PowerEnJoy

Authors:

Melloni Giulio 876279
Renzi Marco 878269
Testa Filippo 875456

Reference Professor:
Mottola Luca

Release Date: December 11th, 2016
Version 1.0

Table of Contents

List of Figures

1 | Introduction

1.1 Purpose

Analyzing the system-to-be in a more detailed way, it is possible to better define the functionalities that the system should provide to the users and the employees.
The user has the possibility to register to get a personal account on *PowerEnJoy*, which is by the way mandatory to access to all the functionalities provided. A logged-in user can search for an available car within a certain range from his current position or from a given address and, in case, he can pick up the car. Finally, he has the possibility to reserve a chosen car (he can also delete the reservation subsequently if the car won't be used: in this way he is free from fees).
Moreover, to promote a more eco-friendly behaviour of the users, the system applies special discounts or extra charges on the cost of the rides according to some different conditions that will be described in the Non Functional Requirements section.
With regards to the company employees, the system will let them use special functions to retrieve specific information about the cars (e.g.: battery level, internal working status) and their positions. In this way, they can move cars from safe areas to charging areas and vice versa if needed.

1.2 Scope

The system-to-be will interact with three main world entities: *People*, *Physical platform* and *External software services*. *People* entity includes the clients and all those people who have a relation with the system, such as the employees. For what concerns the clients, the interaction with the system consists in both the functionalities that the system provides and the events that are triggered by the users. Log-in, reservations of cars, rides, notifications, parking are the main shared phenomena between the clients and the system.
Employees collaborate with the system in terms of managing the cars: the search of cars, the notification of their states, the parking are the core shared phenomena between the employees and the system.
Physical platform consists of the physical infrastructure of the world with which the system exchanges information. It includes the cars, the *Safe Areas* and *Charging Areas*. Unlocking, locking the cars and routes management are the main phenomena that are shared with the system.
Safe Areas and *Charging Areas* come into account when considering the search for the availability of parking lots or of power plugs.
Finally, *External software services* are the software tools that cooperate with the system-to-be. They can be considered external systems that already exist in the environment. The main ones are the payment system and GPS location system. The first one interacts with

PowerEnJoy in the reservation and in the payment phases while GPS system helps with all those actions that require information about position (search for a car, notification, parking, reservation).

1.3 Definitions, Acronyms and Abbreviations

PowerEnJoy is the name of the system that has to be developed.

System sometimes called also *system-to-be*, represents the application that will be described and implemented. In particular, its structure and implementation will be explained in the following documents. People that will use the car-sharing service will interact with it, via some interfaces, in order to complete some operations (e.g.: reservation and renting).

Renting it is the act of picking-up an available car and of starting to drive.

Ride the event of picking-up a car, driving through the city and parking it. Every Ride is associated to a single user and to a single car.

Reservation it is the action of booking an available car.

Car a car is an electrical vehicle that will be used by a registered user.

Not Registered User indicates a person who hasn't registered to the system yet; for this reason he can't access to any of the offered function. The only possible action that he can carry out is the registration to get a personal account.

Registered User interacts with the system to use the sharing service. He has an account (which contains personal information, driving license number and payment data) that must be used to access to the application in order to exploit all the functionalities.

Employee it's a person who works for the company, whose main task is to plug into the power grid those cars that haven't been plugged in by the users. He is also in charge of taking care of the status of the cars and of moving the vehicles from a safe area to a charging area and vice versa if needed.

Safe Area indicates a set of parking lots where the users have to leave the car at the end of the rent; the set of the Safe Areas is pre-defined by the system management. These areas are spread all over the city.

Plug defines the electrical component that physically connects the car to the power grid.

Charging Area is a special *Safe Area* that also provides a certain number of plugs that connect the cars to the power grid in order to recharge the battery.

Registration the procedure that an unregistered user has to perform to become a registered user. At the end, the unregistered user will have an account. To complete this operation three different types of data are required: personal information, driving license number and payment info.

Search this functionality lets the registered user search for available cars within a certain range from his/her current position or from a specified address.

RASD is the acronym of *Requirements Analysis and Specification Document*

DD is the acronym of *Design Document*

1.4 Reference Documents

During the writing of this document, the following resources have been taken into account:

- Specification Document:
  - *Assignments+AA+2016-2017.pdf*
  - *RASD.pdf*
- Example document:
  - *Sample Design Deliverable Discussed on Nov. 2.pdf*
- Papers on *Green Move Project*

1.5 Document Structure

The first chapter recalls the purpose and the scope of the system that has to be developed and in particular of the current document, which focuses on the architecture of *PowerEnJoy*.
In the second chapter, which is the core of the *Design Document*, the overall architecture will be presented at various levels of abstraction. After that a brief explanation of the architectural styles and patterns used is provided.
The third chapter covers the algorithmic part of the system; some interesting algorithms that will be useful later on in the real implementation are described in pseudocode.
The fourth chapter contains the mockups of both the webapp user interfaces and the adhoc touch screen embedded into the cars. Along with them, there is a small description.
The fifth chapter reports the mapping of the requirements, identified in the *RASD*, with respect to the architecture's components.
The sixth and seventh chapters simply refers to the time effort spent in writing the *DD* and the tools used to do it.

2 | Architectural Design

2.1 Overview

The system-to-be, *PowerEnJoy*, will adopt a three-tier architecture, which means that the application will be designed by taking into account three physical levels as shown in the following figure:

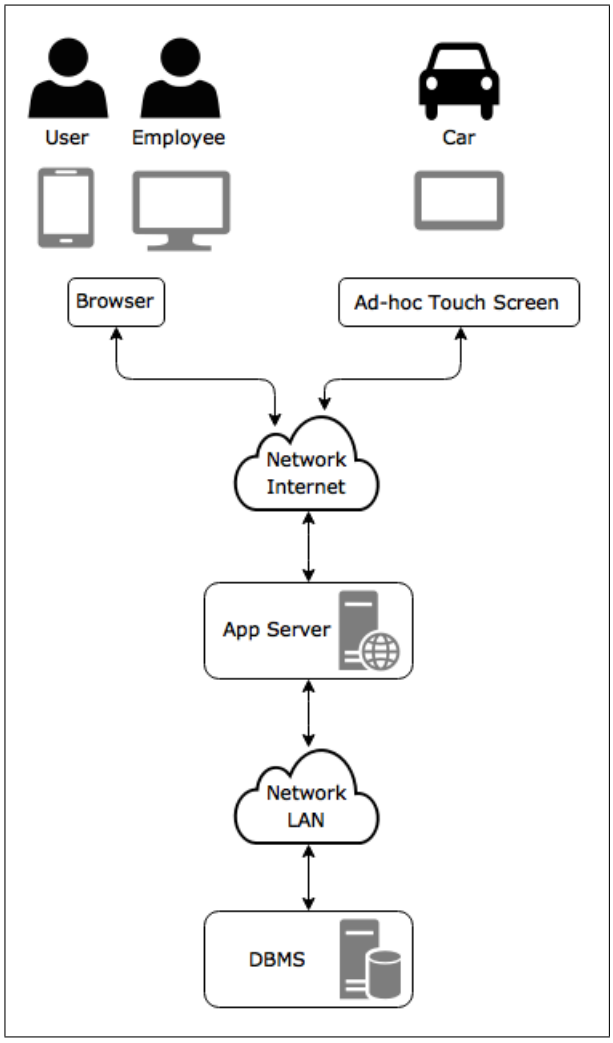

Figure 2.1: High Level Architecture

- **Client Tier** includes all the devices (PCs, smartphones) and software with which

the people (*Users* and *Employees*) and the cars will be able to interact with the system.

- **Application Tier** represents the set of machines (and software) where the core of the application will be run on.
- **Data Tier** is the physical layer in charge of storing the data that are necessary to the system.

Each of these tiers will run a precise piece of software and following sections will go into the details in terms of software components, but it is useful to have already at this point a glimpse on the logical distribution of the application modules.
The figure suggests that *Client Tier* will be organized quite differently between the PCs, smartphones and the Car sides. While PCs in the Client Tier will only host the Presentation layer, in the Car there will be both the GUI and a small part of application logic. This is necessary to elaborate some information on the car, such as the interaction of the car with other components within the car itself (i.e. sensors) or out of it (i.e. the charging plug).
For both the PCs and the Car, however, the business logic of the application is on an Application server, in the Web Tier. Finally, the persistent data are stored in the Data Tier.

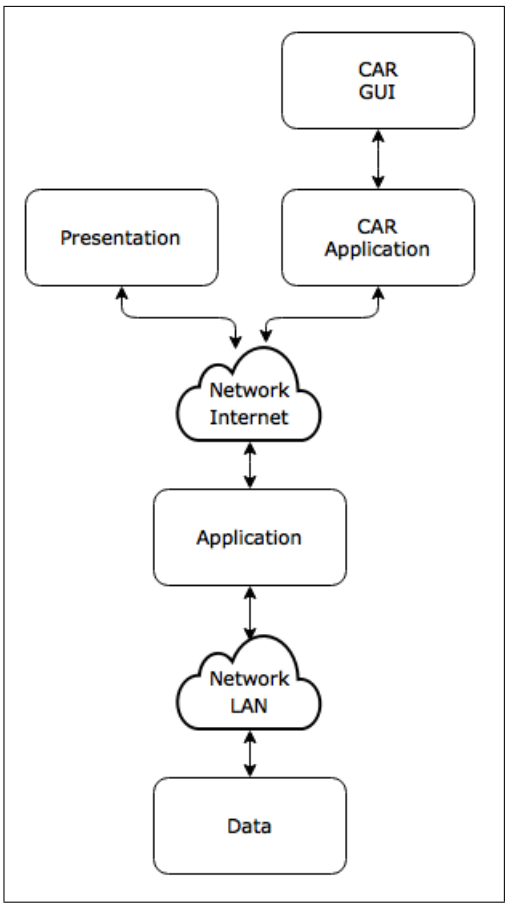

Figure 2.2: Overview Architecture

2.2 Component View

This section focuses on the internal structure of the system-to-be in terms of the sub-components that it will be composed of and their relations.
Before going into details, it is useful to recall from the Overview section that the deployment of the application is slightly different between the PC/Mobile and the Car platforms. This is reflected in the following component diagram in which the *Device* (i.e. Mobile and PC) and the *Car* nodes are shaped in different ways, hosting different components and different interfaces with the server.
One one hand, the *Devices* interact with the application through *WebAppAPI*, on the other hand, the *Car* exposes its proper *CarAPI* and deals with the *ServerAPI* to share information with the application running on the server.
With regard to the Webapp, there are the following components to take into account:

ViewRender is the component in charge of managing the viewable part of the application. It has two main roles: firstly, to fetch the user inputs coming from the Device browser (through the *WebAppAPI*) and to pass them to the *Dispatcher*, and secondly to ship the Web pages that it receives from the other components to the *Device* browser.

Dispatcher is the item responsible for the sorting of the incoming requests. It scans the type of request that it receives from the *ViewRender* and selects the suitable component that will take care of it.

ReservationManager is the module designed to manage the reservation requests. It elaborates the data it gets from the *Dispatcher* and it produces an output in terms of a Web page that it sends to the *ViewRender*. In doing that, it has to communicate with the *Model* component to get information from the Database and eventually to update it. It also exchanges information with the *ServerCommunicationManager* to notify the selected car of the reservation request.

RegistrationManager handles the registration requests. It is connected to the *Model* in order to update the data on the Database. It also produces an appropriate Web page and sends it to the *ViewRender*.

StateManager is the component that takes care of updating the cars state. Thus, it is linked with the *Model* and with the *ServerCommunicationManager*. As the previous Manager components, it produces an Web page to notify the success of the operation.

LogInManager deals with the log in requests. It updates the Database thanks to the *Model* component. It produces a Web page.

MapController is the component that manages the map that the user and the system can deal with. It makes use of an external *MapService* thanks to the *MapAPI* and it retrieves data (such as the location of the cars or the Safe areas and Charging areas) from the database and updates the *Model* too. It exchanges information with the *ReservationManager* too.

RideManager is in charge of the rides management. It has both to retrieve data from the car (thanks to the *ServerCommunicationManager* and the *Dispatcher*) and to ship information to it. To do so, it also needs to get info from the *MapController*.
It is linked to the *RideCostCalculator* and *PaymentManager* components that offers

to it the obvious services. Finally, this component access the Database through the *Model*.

RideCostCalculator is the specific component that compute the cost of a ride. To perform this operation it needs info from the *RideManager*.

PaymentManager handles the communication with the external *Payment System*. To do so, it uses the *PaymentAPI*.

ServerCommunicationManager is the specific component of the Webapp that manages the communication with the car. It receives info from the car thanks to the *CarCommunicationManager* through the *ServerAPI* and sends them to the *Dispatcher* to be elaborated. It also delivers info (coming from the *ReservationManager*, the *StateManager* and the *RideManager*) to the car exploiting the *CarAPI*

Model is the component that is responsible for the data management within the app. It is linked to all the Manager components and it is the bridge between the server on which the app is running and the Database server. A suitable interface is provided by the DBMS, *DBMSAPI*.

With regard to the Car, the following components are designed:

CarCommunicationManager handles the data that the car receives from the Webapp and those data that it has do send back. It is linked to the *CentralUnit*.

CentralUnit manages the info that come from all the sensors within the car and those that it receives from the server. As a consequence, it communicates with the *ScreenManger* to update the data to be displayed on the screen.

ScreenManager is the component that organize the data received from the *CentralUnit* and display them on the screen panel. It also gets the user input and delivers it to the *CentralUnit*.

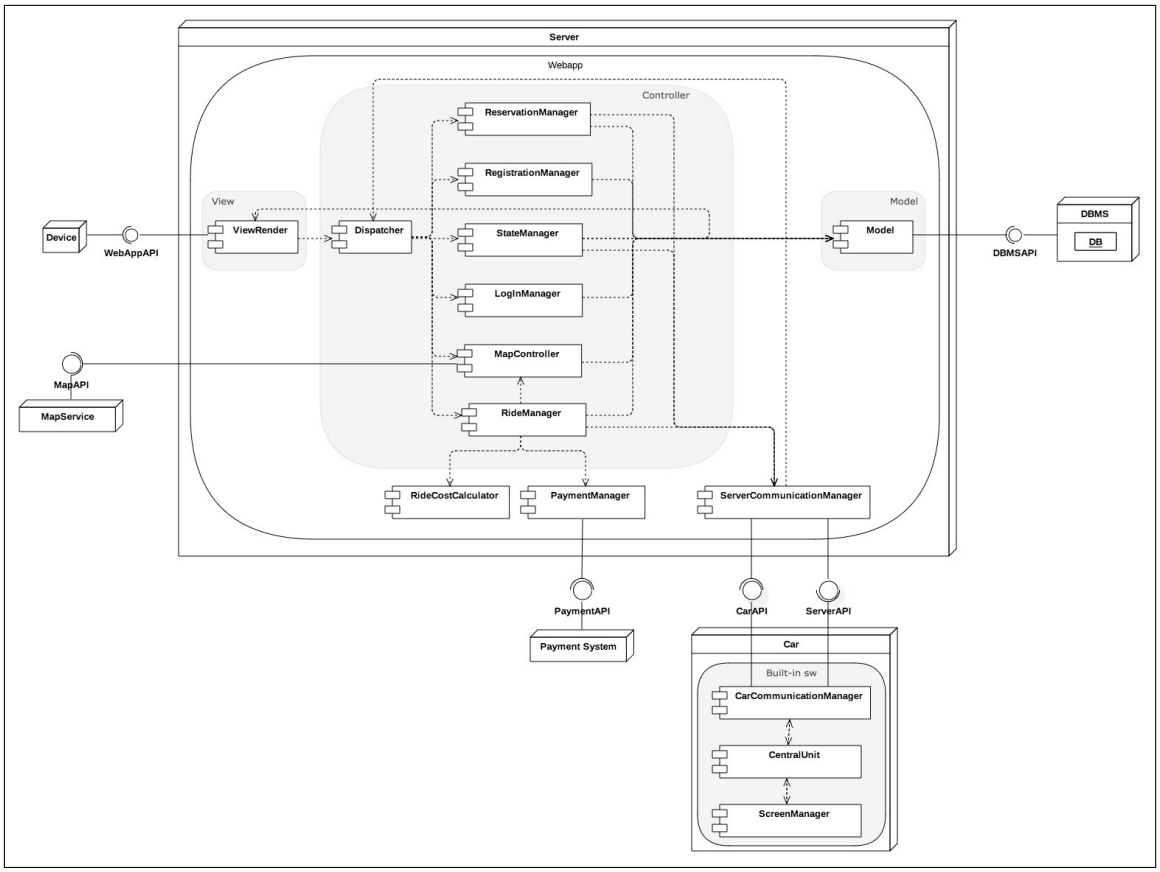

Figure 2.3: Component Diagram

2.3 Deployment View

As already seen in the Overview section, the system is organised in a three-tier architecture. This section explores the distribution of the application over the physical nodes. Let us introduce the following image which represents the deployment of the application:

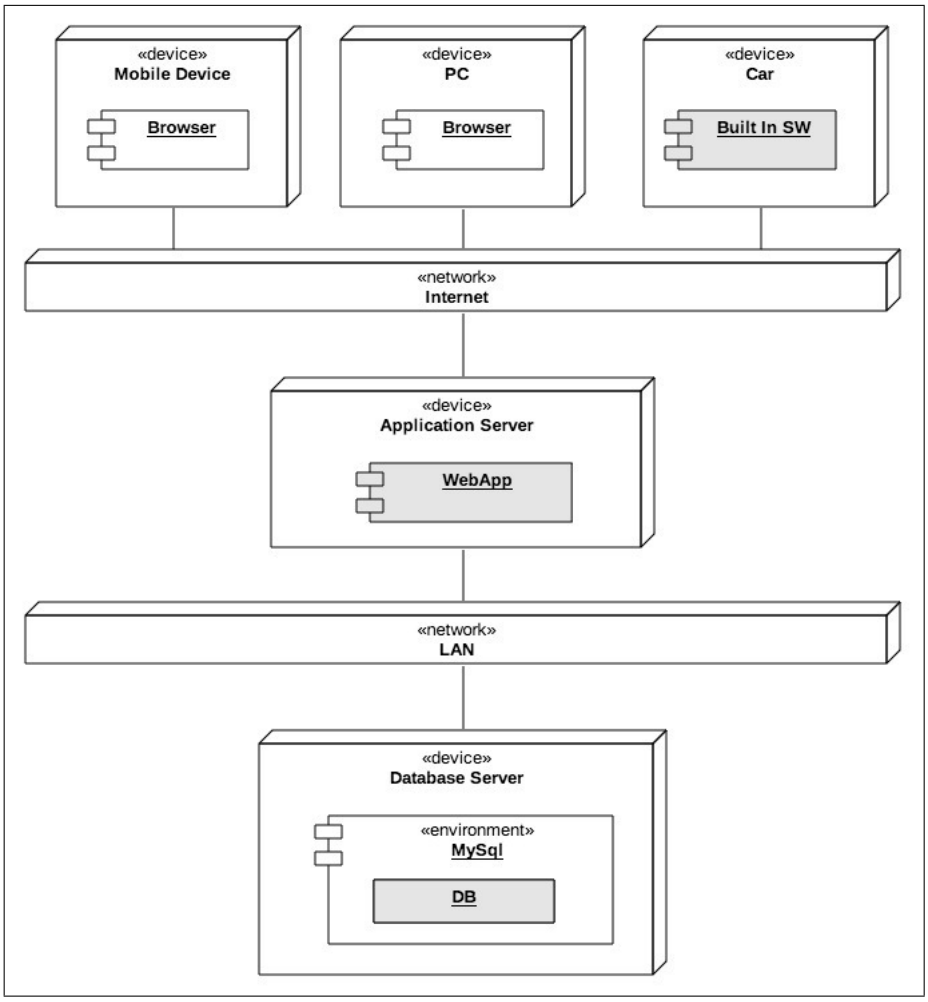

Figure 2.4: Deployment Diagram

The **Client Tier** includes three types of devices: *Mobile Device*, *PC* and *Car*. The first two ones can access the Webapp surfing the *Internet* through a browser that is already installed on the device. As already stated in the Overview, the *Cars* are treated slight differently and they have a *Built In SW* which manages the communication with the WebApp. The connection with the server passes through *Internet* anyway.
The **Application Tier** consists in an *Application Server* on which the business logic of the application is run. It is here where all the requests coming from the devices of the Client Tier are processed and as a consequence proper Web pages are created. The *Application Server* is then linked to a remote *Database Server* through a LAN network.
The **Data Tier** hosts a *Database Server*, that is a remote machine whose job is to store all the data that are necessary for the system to provide its functionalities. The *Database Server* can access the physical DB through an appropriate software, *MySql* which makes possible the exchange of data from and to the *Application Server*.

2.4 Runtime View

2.5 Component interfaces

2.6 Selected Architectural Styles and Patterns

2.7 Other Design Decisions

2.7.1 ER Diagram

An important issue in the Design Process is to select the suitable data that the system has to store in the database.This is crucial in the Design Phase because it affects the implementation of all the system functionalities, which rely on these data.The following diagram shows the pieces of information that the database server provides to the application.

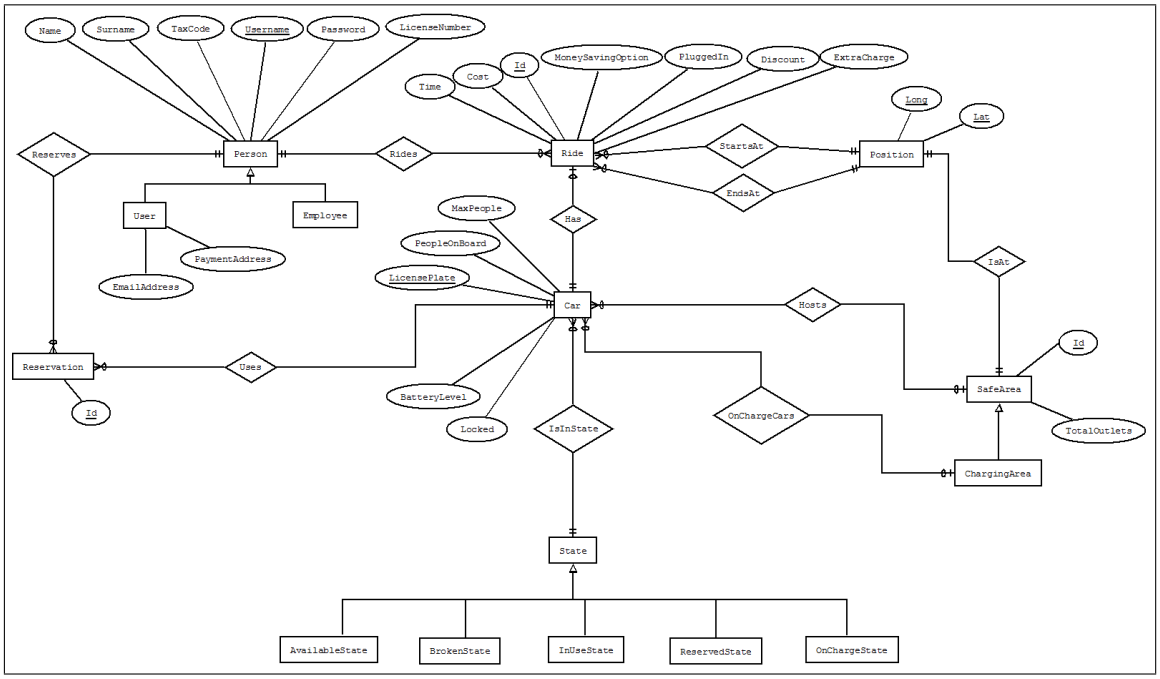

Figure 2.5: ER Diagram

The diagram suggests that the system has two types of *Person*: *User* and *Employee*. As already stated in the *RASD* this is done to distinguish the roles and permissions of the two types of actors.
The system has to store information about the *Ride* such as the time (in order to be able to compute the cost), possible discount or extra charge and the initial and final position so to keep track of the associated *Car*.
Cars are identified by their unique license plate and the system is always updated on their current state, which is one of the shown above.
The system also records the info of each *Reservation* in terms of the user and the corresponding car. This is clearly not enough to identify in a unique way a reservation, so the attribute Id is used as a key field for it.
Finally the system stores the data that are necessary to properly represent the *SafeAreas*

and the *ChargingAreas*: their position, their id, information about the number of total outlets and the cars that are parked. This piece of information is crucial for some key functionalities of the system, such as the computation of the uniform distribution of cars in the city.

3 | Algorithm Design

3.1 Discount and Penalties Calculation

```
//Discount calculator

float CAR_PENALITY := 0.3;
float CAR_CHARGE_DISCOUNT := 0.2;
float CAR_PEOPLE_DISCOUNT := 0.1;
float CAR_POSITION_DISCOUNT := 0.3;
float CAR_CHARGE_CONSTANT := 0.5;
float CAR_PEOPLE_CONSTANT := 0.3;
float MIN_BATTERY_CONSTANT := 0.2;

//Check if the user has some discounts or penalities
function float calculateDiscount (Position position, integer
    peopleNumber, float carCharge, integer price, Car car){
    if not checkPenalty()
        return price*(1-max(checkDiscount(position, car),
            checkDiscount(peopleNumber), checkDiscount(carCharge)))
            ;
    endif
    return price*(1+CAR_PENALITY);
}

//Check if the user can obtain the discount of the car left with
     more than 50% of battery
function integer checkDiscount(float carCharge){
    if carCharge ≥ CAR_CHARGE_CONSTANT
        return CAR_CHARGE_DISCOUNT;
    endif
    return 0;
}

//Check if the user can obtain the discount because he had two
    passengers
function integer checkDiscount(integer peopleNumber){
    if peopleNumber ≥ CAR_PEOPLE_CONSTANT
        return CAR_PEOPLE_DISCOUNT;
    endif
```

```
    return 0;
}

//Check if the user can obtain the discount because he had left
    the car plugged in a charging area
function integer checkDiscount(Position position, Car car){
    if chargingAreaPositions.contains(position) && car.isPlugged
        ()
        return CAR_POSITION_DISCOUNT;
    endif
    return 0;
}

//Check if the user has left the car with left of 20% of battery
     or he has left the car too far from the nearest charging
    area
function bool checkPenalty(){
    if carCharge < MIN_BATTERY_CONSTANT || tooFarFrom(position)
        return true;
    endif
    return false;
}

//Check if exists a charging area near the car
function boolean tooFarFrom(Position position){
    for each Position called chargingAreaPosition in
        chargingAreaPositions)
        if Position.distance(chargingAreaPosition, position)<
            MAX_DISTANCE
            return false;
        endif
    endfor
    return true;
}
```

3.2 Distribution

```
//Uniform distribution

float KM_INCREASE := 0.5;

//This function gives the nearest position to the place where
    you want to park
function Position giveBestPosition(Position finalPosition){
    int totalPositionCounted := 0;
    int totalCarCounted := 0;
    return checkUniformity(finalPosition, totalPositionCounted,
        totalCarCounted, KM_INCREASE).getFirst().getPosition();
}
```

```
//This method checks the uniformity finding the best place where
    the variance is high
function List<PositionMediaVariance> checkUniformity(
   finalPosition, totalPositionCounted, totalCarCounted, range){
   List<PositionMediaVariance> positionMediaVarianceList;

   if totalCarCounted == 0 && totalPositionCounted == 0
      for each Position called chargingAreaPosition in
         chargingAreaPosition)
         totalPositionCounted++;
         totalCarCounted = totalCarCounted +
            chargingAreaPosition.getCars().size();
      endfor
   endif

   for each Position called chargingAreaPosition in
      nearChargingAreaPositions(range, finalPosition){
      if chargingAreaPosition.hasFreePlug()
         positionMediaVarianceList.add(new PositionMediaVariance
            (chargingAreaPosition, totalCarCounted /
            totalPositionCounted, totalCarCounted /
            totalPositionCounted - chargingAreaPosition.getCars
            ()));
      endif
   }

   if not positionMediaVariance.isEmpty(){
      return positionMediaVarianceList.sort();
   }

   checkUniformity(finalPosition, totalPositionCounted,
      totalCarCounted, range + KM_INCREASE);
}

//This checks the nearest charging area in a certain range.
function List<Position> nearChargingAreaPositions(int range){
   List<Position> nearChargingAreaPositionsArrayList;
   for each Position called chargingAreaPosition in
      chargingAreaPositions)
      if Position.distance(chargingAreaPosition, finalPosition)
         < range
         nearChargingAreaPositionsArrayList.add(
            chargingAreaPosition);
      endif
   endfor
}

//Return if the car is plugged in a charging area
```

```
function boolean isPlugged()
```

4 | User Interface Design

Figure 4.1: Webapp Interface on Computers

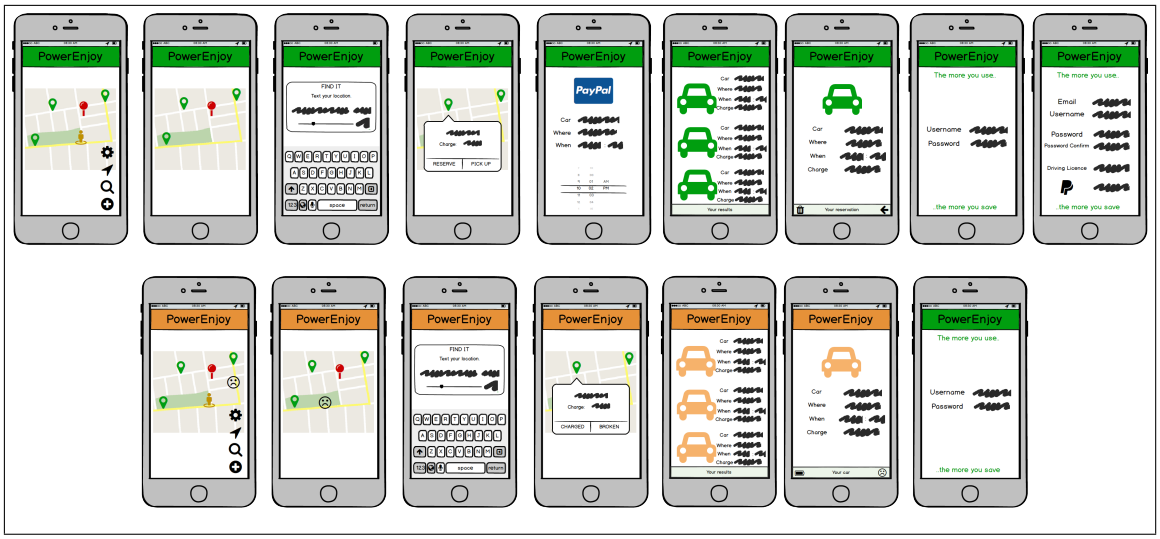

Figure 4.2: Mobile Interface on Mobile Devices

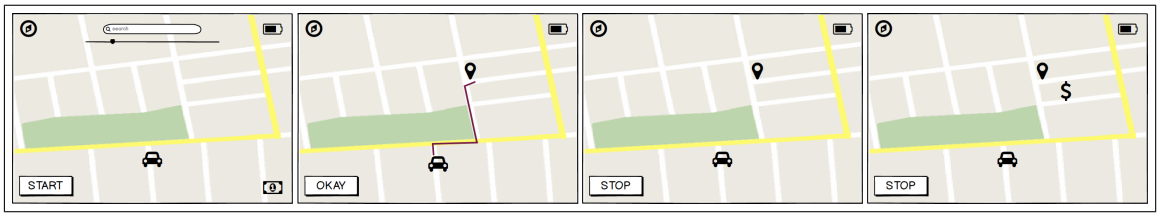

Figure 4.3: Webapp Interface

5 | Requirements Traceability

6 | Effort Spent

7 | References

7.1 Tools

During the writing of this document, the following application tools have been used:

- Star UML, for creating all types of UML models
- TeXstudio, for writing the document in LaTeX
- Balsamiq, for creating the mockups of the user interface
- draw.io, for drawing the diagrams of the architecture