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: January 15th, 2017
Version 1.1

Table of Contents

List of Figures

1 Introduction

1.1 Purpose

The current document describes the architecture of the system *PowerEnJoy*. It covers both the physical implementation with an insight on the deployment layout and the logical distribution of the software modules.
The *Design Document* takes into account all the considerations that have been made in the previous document, the *RASD*, and shows how these issues can be tackled in concrete with design choices.

1.2 Scope

The *Design Document* explores the architecture of the system-to-be by means of software design principles and known patterns that fit the given problem. Throughout the document the system will be analysed at different conceptual and granular levels with heterogeneous views and diagrams: the Component view, the Deployment view, the Runtime view with sequence diagrams, the User Interfaces and the ER Diagram. Moreover, the most significant algorithms of the application will be sketched up. The document is also a point of reference for the traceability of the requirements that have been identified in the *RASD*: a dedicated section will explain how these requirements are fulfilled with the proposed architecture elements.

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 pseudo-code.
The fourth chapter contains the mockups of both the webapp user interfaces and the ad-hoc 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:

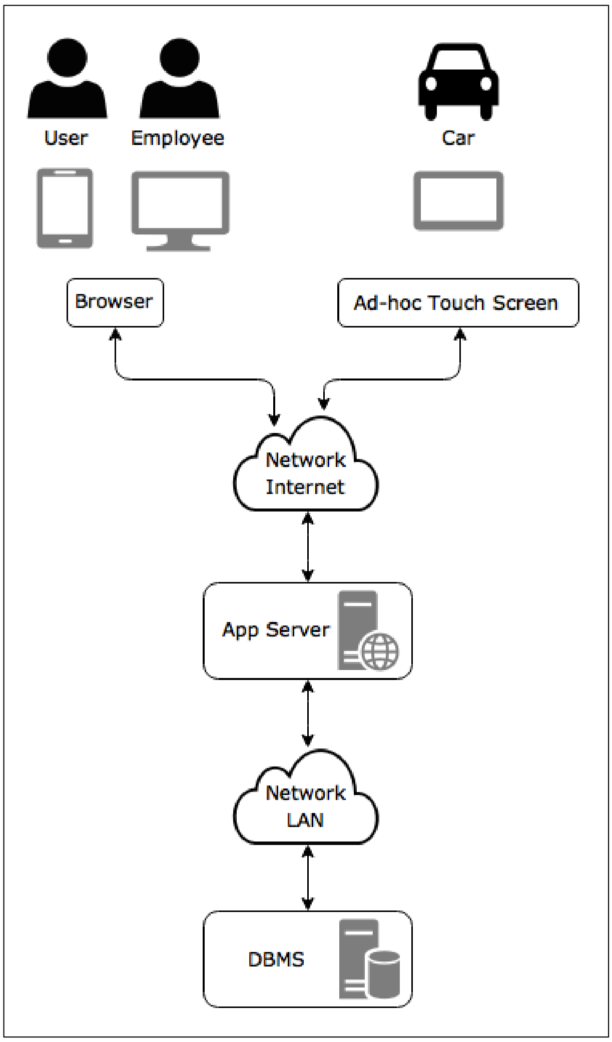

Figure 2.1: High Level Architecture

- **Client Tier** includes all the devices (PCs, smartphones and car screens) 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 the 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.

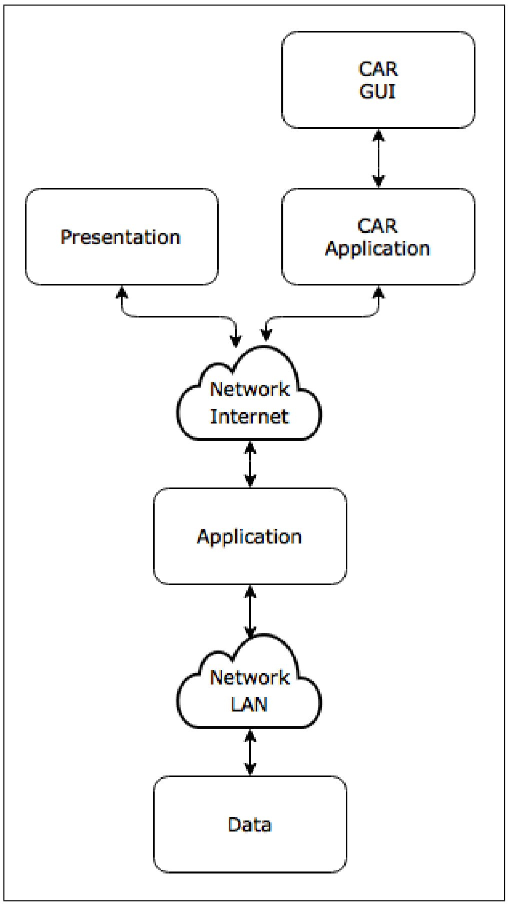

Figure 2.2: Overview Architecture

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.

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/Smartphone and the Car platforms. This is reflected in the following component diagram in which the *Device* (i.e. Smartphone 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*.

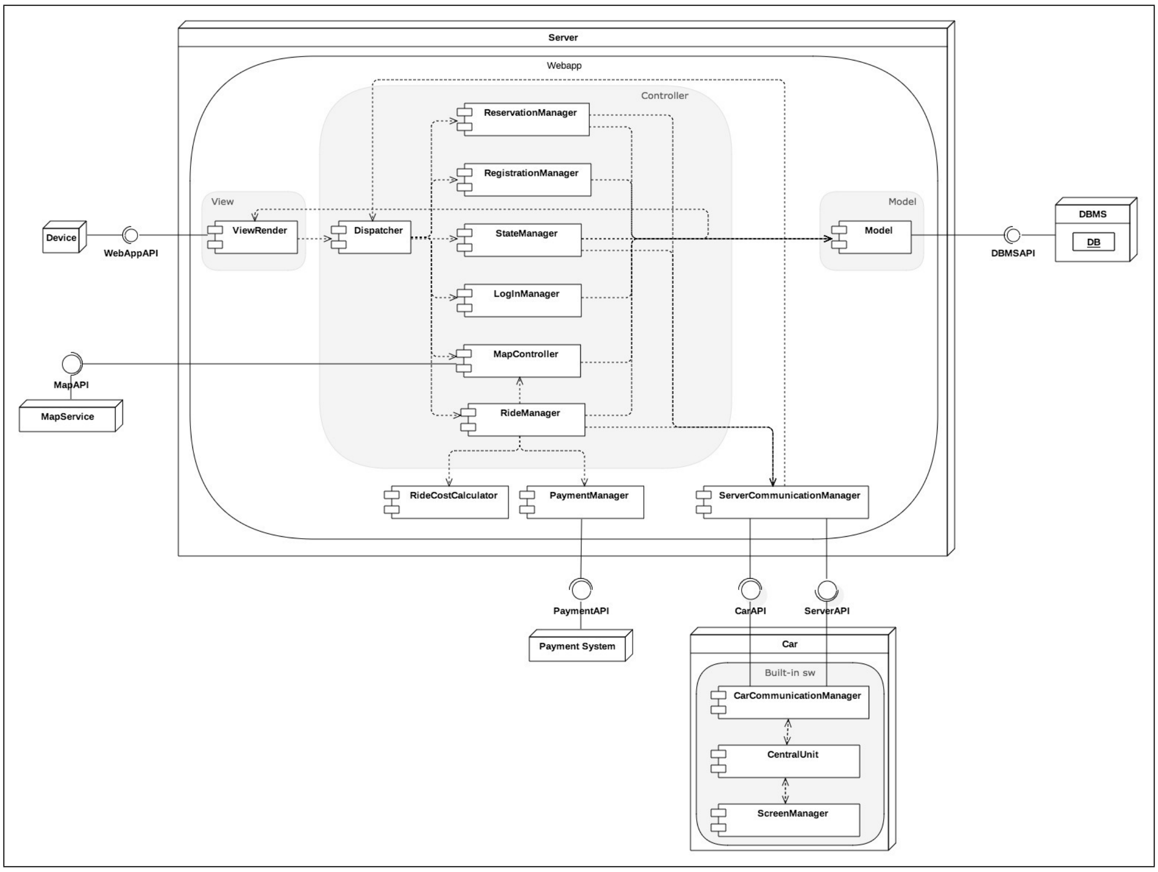

Figure 2.3: Component Diagram

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:

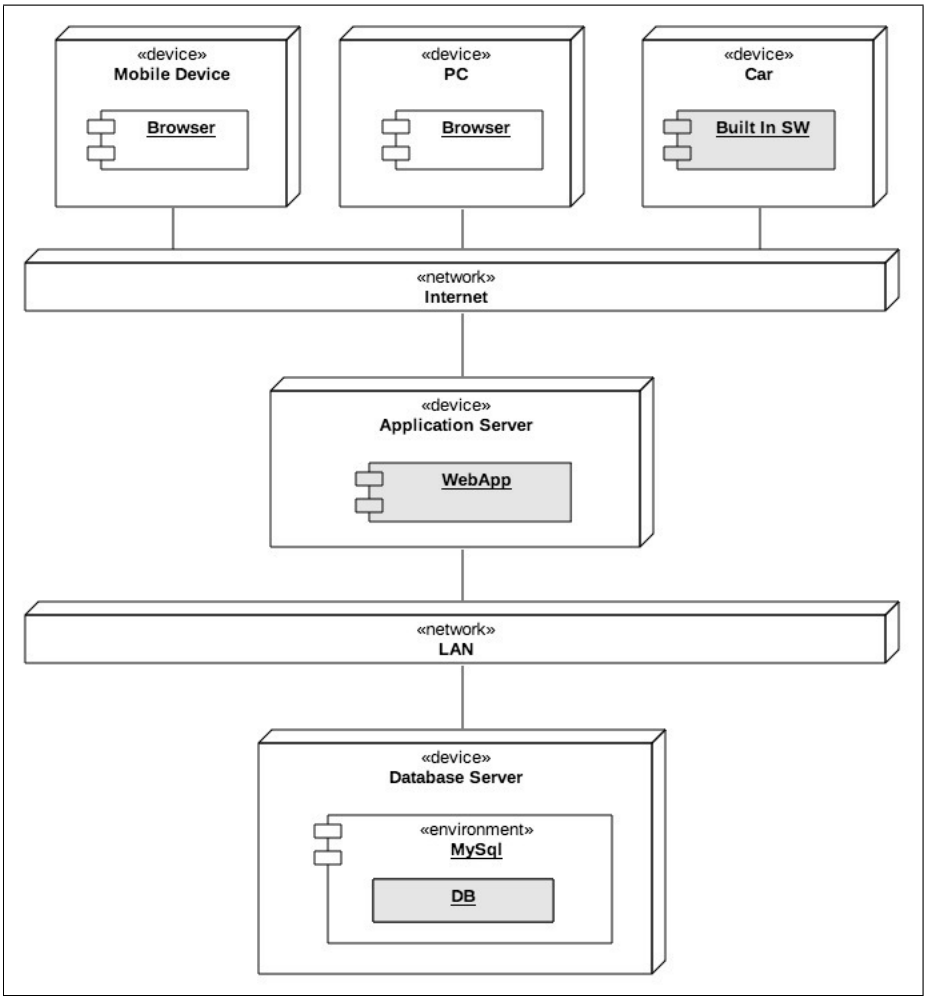

Figure 2.4: Deployment Diagram

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

In this section, some *sequence diagrams* will be provided in order to better explain how the components of the system behave and interact with each others to fulfill the key functionalities.

2.4.1 Reservation

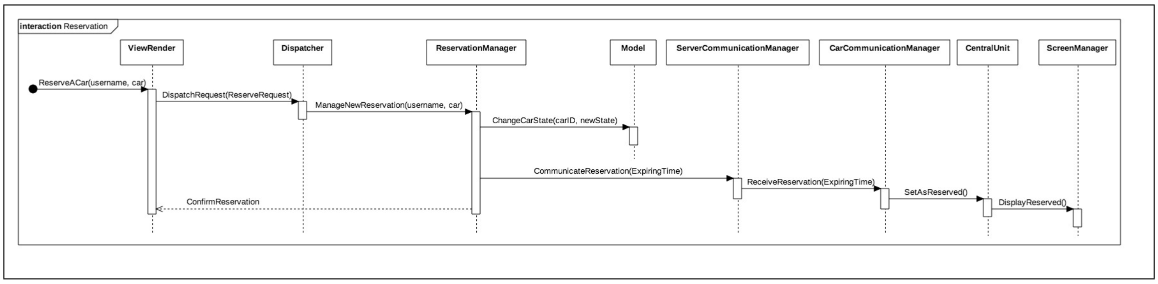

Figure 2.5: Reservation - Sequence Diagram

The flow of information starts from the user, through the ViewRender, and then request is dispatched to the ReservationManager. It is responsible for updating the model and to notify the operation also to the chosen car. Finally the user receive a confirmation for his reservation.

2.4.2 Ride Start

In order to pick-up an available car, the user has to use the Webapp to start the operation. Before unlocking the car, the system checks that the user balance is at least greater than zero. In the negative case, the user is notified about the abortion of the operation, caused by a non-positive balance. Otherwise the system change the status of the chosen car and unlocks it. Then the user can get on board, and through the touch screen can activate the *Money Saving Option* and enter his final destination. Finally he presses the *start* button. The system detects whether the user has enabled the special option and in this case it searches and notifies the suggested area where to park in order to be eligible to obtain the discount on the ride. Finally, the on board screen prompts a message stating that the user can start driving.

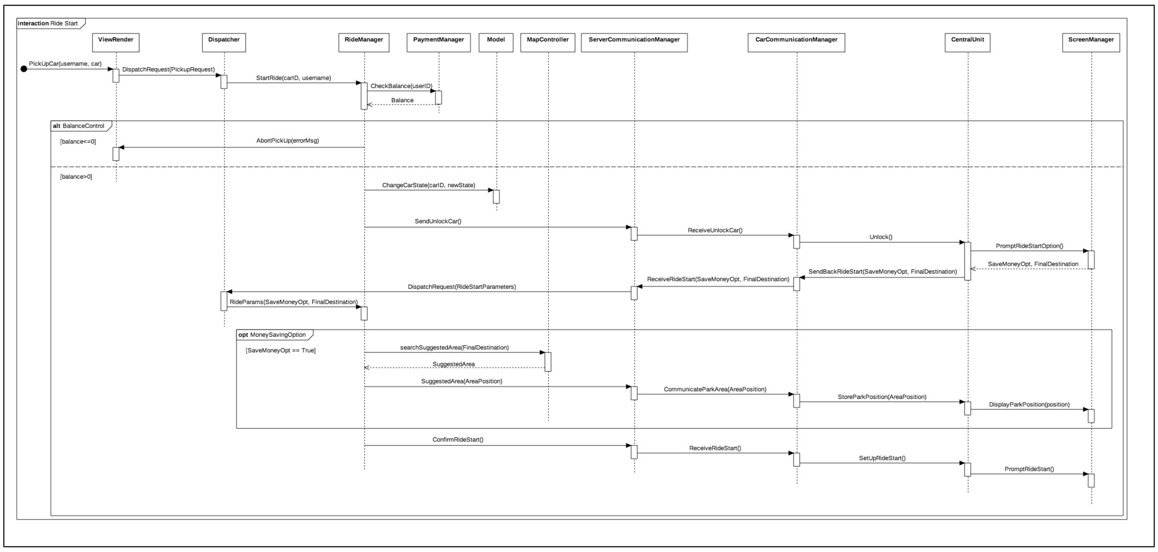

Figure 2.6: Start of the Ride - Sequence Diagram

2.4.3 Ride Stop

The stop signal of the ride is given by the user from the touch screen embedded into the car. So the car communicates with the system, sending the message that the user has ended his ride and along with it, it sends also the key parameters (i.e. final position, battery level and people on board) that will be used to calculate the discount or the penalty that has to be applied to the ride cost. Then the system computes a temporary cost of the ride that will be displayed on the car screen. The user confirms it and gets off the car. After the confirmation, the user has a limited time to plug the car into the power grid to obtain the relative discount. When the timer ends, the car sends a message to the system to communicate whether the user has plugged in the car. At this point the system calculates the real cost of the last ride, since it has all the required data, and complete the payment stage of the operation through the external system.

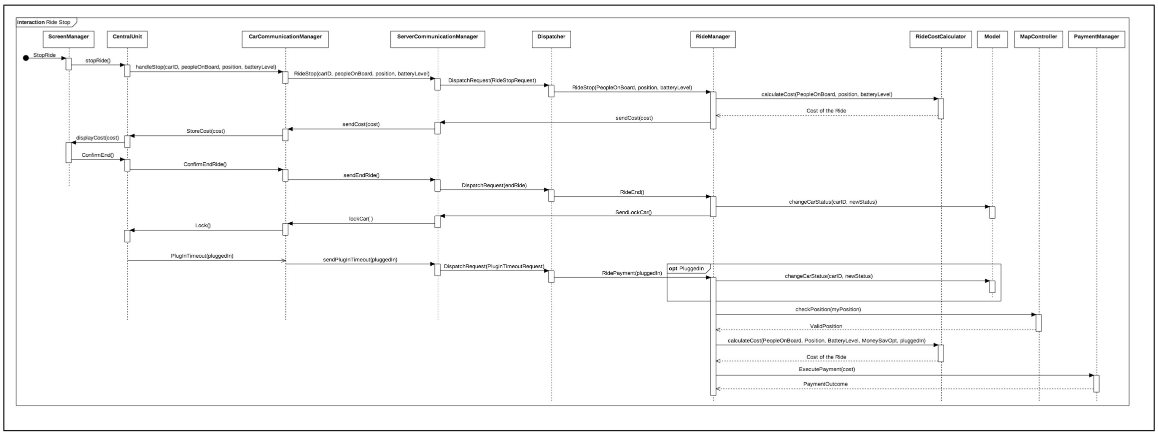

Figure 2.7: Stop of the Ride - Sequence Diagram

2.4.4 Car Plug-in

The employee of the company is responsible for plugging in all the cars that haven't been attached to the power grid from the users. Through the Webapp the employee registers the operation, then the system takes care of updating the status of the car.

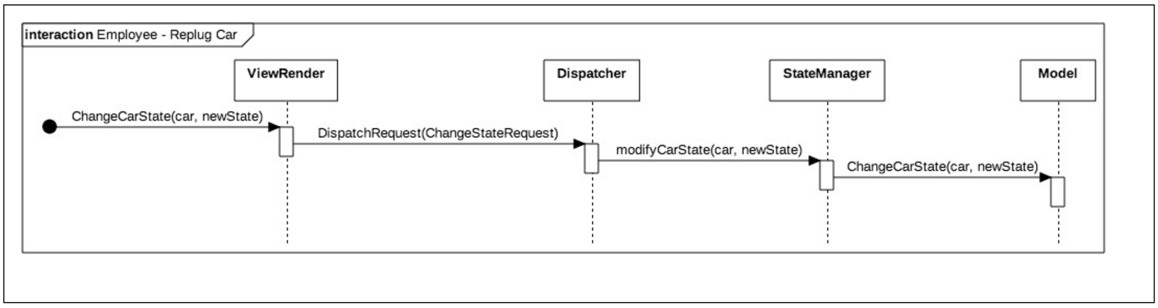

Figure 2.8: Car Plug In by Employee - Sequence Diagram

2.5 Component interfaces

Each component of the system, already identified in the *Component View*, has a specific interface that other components can use in order to require the execution of a certain operation (maybe with some input parameters) that sometimes might return a concrete result. Then, in this section, the interfaces that each component offers are briefly described along with theirs input parameters and output effects.

ViewRender

Interface Name	ReserveACar
Input	Identifier of the car, username
Output	-
Description	Through the webapp, the user can select the car to reserve and send the request to the system. Therefore the ViewRender collects the identifier of the selected car and the username and passes these data to the Dispatcher

Interface Name	PickUpCar
Input	Identifier of the car, username
Output	-
Description	When a user wants to start a new ride, using the webapp, he can simply select the available car he want to use. The ViewRender component catches his username and the unique identifier of the chosen car and, after that, calls the suitable interface of the Dispatcher passing to it the input data.

Interface Name	AbortPickUp
Input	Error message
Output	Webpage
Description	This interface refers specifically to the case in which the ride cannot be started because the user has a non-positive money balance. Hence, the output will be a webpage that reports this critical information to the user.

Interface Name	ChangeCarState
Input	Car identifier, new status of the car
Output	Webpage
Description	It permits to update the status of the indicated car with the new state. In output will be a new webpage that confirms the result of the operation and shows all the information of the car.

Dispatcher

Interface Name	DispatchRequest
Input	Specific request of operation
Output	-
Description	It processes the incoming request from the user/employee; it is responsible for passing the request to the proper component that knows how to deal with it.

ReservationManager

Interface Name	ManageNewReservation
Input	username, car identifier
Output	Webpage
Description	It handles the request for a new reservation. Using the given username and car identifier, the ReservationManager collaborates with other components of the system in order to complete the process. As an output, a webpage is produced with summary information about the reservation operation.

Interface Name	DeleteReservation
Input	username, reservation id
Output	Webpage
Description	The user has the possibility to cancel his reservation before the interval time expires. If this operation is carried out without errors, according to the service policy, the reservation is effectively deleted from the model and an output webpage will be generated with a confirmation message. On the contrary, if the reservation cannot be cancelled the webpage will simply notify an error message.

RegistrationManager

Interface Name	NewUserRegistration
Input	Name, Surname, Address, Date of Birth, email, phone number, driving license, payment information
Output	Webpage
Description	A person who wants to use *PowerEnJoy* must be registered to the service. The operation of registration of a new user, offered via this interface, requires all the personal information of the user and the payment data. The result of this operation is a webpage with a recap of the inserted information and a confirmation of the success of registration.

StateManager

Interface Name	ModifyCarState
Input	New state, car identifier
Output	Webpage
Description	This operation refers to the case in which an employee wants to change the state of a certain car; consequently, the new state and the car identifier are required as inputs. While doing this operation the Model component will be invoked.

LogInManager

Interface Name	LogInRequest
Input	Username and password of the account
Output	Webpage
Description	Used in case of login operation, it requires the username of the account and its password; if the check is successful it returns a suitable webpage according to the account type (user or employee) otherwise, in case of failure, the returned webpage contains an error message.

MapController

Interface Name	SearchSuggestedArea
Input	Final destination
Output	Suggested Area
Description	This functionality is triggered when the user selects the money saving option. The user is given the positions of the suggested areas where to park the car, which are computed starting from the final destination of the ride inserted by the user.

Interface Name	CheckPosition
Input	Position
Output	Validity flag position
Description	At the end of each ride, the system must check the final position of the car. Hence this operation checks if the position received from the car is valid, in terms of GPS coordinates; a validity flag of the given position will be produced as output.

RideManager

Interface Name	StartRide
Input	Car identifier, username
Output	-
Description	It is the first operation that is enabled when a user wants to pick up a car; then the RideManager interacts with other components to complete the request. The process begins from the webapp and, if no errors are encountered, concludes on the screen embedded into the car. Otherwise, the errors that occur during the operations are reported to the user through the webapp (via the ViewRender).

Interface Name	RideParams
Input	SaveMoneyOpt, Final Destination
Output	-
Description	This lets the RideManager component know whether the user has enabled the *MoneySavingOption* and, in case the final destination; these two parameters are fetched by the on-board screen of the car.

Interface Name	RideStop
Input	Number of people on board, final position, battery level
Output	-
Description	This is the first operation that the RideManager does when the user wants to end his ride. It takes as input the number of people on board, the final position (where the car has been parked) and the battery charge level; it asks the proper component to compute a temporary cost of the ride that is then communicated to the user through the on board screen. Note that the act of plugging the car into the power grid has not been taken into account yet because the system gives the user a small amount of time to get off the car and possibly to plug it in. Only after this time interval it will be possible to check whether the user has plugged in the car and to compute the real final cost.

Interface Name	RideEnd
Input	-
Output	-
Description	It is a signal received from the car, when the user has confirmed the end of the ride through the screen. So, the status of the car can be updated.

Interface Name	RidePayment
Input	PluggedIn
Output	-
Description	At the end of the ride, as the last operation, the system has to perform the payment of the total cost; the use of this interface triggers, inside the RideManager component, a sequence of calls to interfaces of other components which are necessary to execute the final operations of the ride. The input of this interface is a flag meaning whether the user has plugged the car into the power grid at the end of his ride.

RideCostCalculator

Interface Name	CalculateCost
Input	Number of people on board, final position of the ride, battery level, car plugged into power grid, money saving option enabled
Output	Cost of the ride (floating point number)
Description	It computes the total cost of the ride starting from the following input parameters: the number of people on board, the final position of the car, the final battery charge level and a flag that tells whether the user has plugged the car into the power grid at the end of the ride. During the computation, the service policy about discounts and extra charges presented in the *RASD* will be applied; the output of this operation is a number which represents the final cost of the ride.

PaymentManager

Interface Name	ExecutePayment
Input	Cost of the ride
Output	Payment outcome
Description	It triggers the payment operation that will rely on the external payment system. As input it takes only the computed cost of the ride and returns a successful/failure message.

ServerCommunicationManager

Interface Name	CommunicateReservation
Input	Expiring Time
Output	-
Description	It sends a properly encoded message to the dual component placed into the car; in the message the time at which the reservation will expire is stored.

Interface Name	SendUnlockCar
Input	-
Output	-
Description	It signals the car to unlock itself.

Interface Name	SendLockCar
Input	-
Output	-
Description	It signals the car to lock itself.

Interface Name	ReceiveRideStart
Input	Money Saving Option enabled, final destination
Output	-
Description	It receives as inputs a flag about the state of the money saving option (enabled or not) and the final destination of the ride inserted by the user. These information will then be passed to the Dispatcher through the appropriate interface.

Interface Name	ConfirmRideStart
Input	-
Output	-
Description	It sends a message to the car to notify the start of the ride.

Interface Name	SuggestedArea
Input	Position of the parking area
Output	-
Description	It sends a message to the car with the suggested areas where the user can park in order to obtain the discount associated with the saving money option.

Interface Name	RideStop
Input	Car identifier, number of people on board, final position, battery level
Output	-
Description	It communicates that the user has ended the ride. All the information collected in the car are encapsulated in a proper request that will be handled by the Dispatcher.

Interface Name	SendEndRide
Input	-
Output	-
Description	It signals that the user has confirmed through the screen the end of the ride, after having seen the possible final cost of the ride.

Interface Name	SendPluginTimeout
Input	PluggedIn
Output	-
Description	It indicates if the user has plugged the car into the power grid.

Interface Name	SuggestedArea
Input	AreaPosition
Output	-
Description	Communicates to the car the computed positions of the areas where the user can park.

Interface Name	SendCost
Input	cost
Output	-
Description	It sends to the car the computed cost of the ride that is passed as input.

Model

Interface Name	ChangeCarState
Input	Car identifier, new state
Output	-
Description	It changes the current state of the given car into the new one.

CarCommunicationManager

Interface Name	ReceiveReservation
Input	Expiring Time
Output	-
Description	It receives the reservation and the expiring time for it.

Interface Name	ReceiveUnlockCar
Input	-
Output	-
Description	The car receives the signal of unlock from the system. This operation is not carried out by this component anyway, but the request is forwarded to the CentralUnit.

Interface Name	SendBackRideStart
Input	Save Money Option, final destination
Output	-
Description	It sends back to the system the preferences indicated by the user through the on board screen; in particular, the preferences cover the possibility that the user has enabled the money saving option (via the first parameter, which is a flag) and the final destination of the ride.

Interface Name	CommunicateParkArea
Input	Area position
Output	-
Description	It communicates to the car the predefined position of the area where the user can park.

Interface Name	ReceiveRideStart
Input	-
Output	-
Description	The system has successfully completed all the checks and the tasks about the start of the ride; this will communicate the CentralUnit that it must prepare the car to start.

Interface Name	HandleStop
Input	Car identifier, number of people on board, final position, battery level
Output	-
Description	It receives from the CentralUnit all the data the have to be passed the to system in order to properly manage the end of the ride.

Interface Name	ConfirmEndRide
Input	-
Output	-
Description	The user, through the on board screen, has confirmed the end of the ride and he is getting out of the car.

Interface Name	SendCost
Input	Cost of the ride
Output	-
Description	It stores the computed cost of the ride, received as input, into the CentralUnit.

Interface Name	LockCar
Input	-
Output	-
Description	It signals that the car must be locked; this request will be passed to the CentralUnit.

Interface Name	PlugInTimeout
Input	PluggedIn
Output	-
Description	It is triggered when the time that the user has to get out of the car and to plug it into the power grid has expired. The CentralUnit communicates to this component through this interface to let it know whether the car has been plugged in by the user in time. This operation is asynchronous.

CentralUnit

Interface Name	SetAsReserved
Input	-
Output	-
Description	It stores into the car memory the reserved state.

Interface Name	Unlock
Input	-
Output	-
Description	It unlocks the car so the user can get on.

Interface Name	StoreParkPosition
Input	Area position
Output	-
Description	It stores into the car memory the position of the area where the user can park; this position will be displayed by the ScreenManager.

Interface Name	SetUpRideStart
Input	-
Output	-
Description	It resets the internal memory of the car in order to correctly manage the new ride; in particular this involves the resetting of the counter of the time spent on driving from the user.

Interface Name	StopRide
Input	-
Output	-
Description	After the user has selected the end of the ride through the touch screen, the CentralUnit collects from the internal memory and sensors all the data that are necessary to compute the cost of the ride.

Interface Name	StoreCost
Input	Cost
Output	-
Description	It stores the given cost of the current ride into the memory.

Interface Name	ConfirmEnd
Input	-
Output	-
Description	The user has confirmed the end of the ride and this must be communicated to the system in order to start all the final operations on the ride.

Interface Name	Lock
Input	-
Output	-
Description	It locks the car

ScreenManager

Interface Name	DisplayReserved
Input	-
Output	-
Description	It shows on the screen that the car has been reserved.

Interface Name	PromptRideStartOption
Input	-
Output	-
Description	It activates on the screen the initial options for the ride; this includes the possibility to enable the money saving option and a special field where to insert the final destination. A start button is then displayed.

Interface Name	DisplayParkPosition
Input	position
Output	-
Description	It displays on the screen the position on the map where the user can park to obtain special discount.

Interface Name	PromptRideStart
Input	-
Output	-
Description	Notifies the user that the system and the car are ready to start the ride.

Interface Name	StopRide
Input	-
Output	-
Description	At the end of the ride, the user can press the stop button to communicate to the system that he has finished his ride.

Interface Name	DisplayCost
Input	Cost
Output	-
Description	It shows on the screen the temporary cost of the current ride(given as input).

2.6 Selected Architectural Styles and Patterns

In this section the design principles and choices that have been adopted in the development of the system will be faced.
First of all, the type of the application. *PowerEnJoy* will be developed as a Webapp in order to make it independent from the particular underlying Operating System. Thus, the software portability increases and the programmers' work overhead decreases.
The choice of a Webapp leads to the implementation of a standard multi-tier architecture and this is why the proposed one is a three-tier structure. The presence of heterogeneous nodes on the *Client Tier* (PCs/Smartphones and Cars) with different roles has to be taken into account in the design phase. The solution is achieved with a proper and different software layers organisation: PCs/Smartphones run only the Presentation layer through the browser, while the Cars are equipped with a Built In SW which is in charge of both the GUI and part of the application logic. The system, however, is able to interact with

both of these clients thanks to proper APIs.
With regard to software design principles, there can be found, at this stage, at least two suitable design patterns: the *MVC* and *State* patterns.
The *MVC Pattern* is recommended for identifying the logical functionalities of the software modules (thus, increasing the cohesion) and to make it easier to apply changes to them in the future (thus, increasing the maintainability).
The *State Pattern* is suitable for treating the car states. With this solution, in each state, only a defined set of operations can be applied to a car. This makes the problem of managing a car simpler and avoids undesired behaviours.

2.7 Other Design Decisions

2.7.1 Technological Choices

The implementation of the system will be carried out using:

- the *JavaEE* standard, specifically for realizing the web-application and all the other related components that form the back-end part, because it lets to manage quite easily a lot of connections (from users and cars), better satisfy both functional and non functional requirements and, lastly, it enables the development team to focus mainly on the application logic thus abstracting from low level details
- a combination of *HTML, CSS* and *Javascript* for the front-end part of the system, which is the one that the users see and interact with, for the reason that these three different technologies let create both functional and fancy web pages and do not add any high performance specific requirement

2.7.2 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.

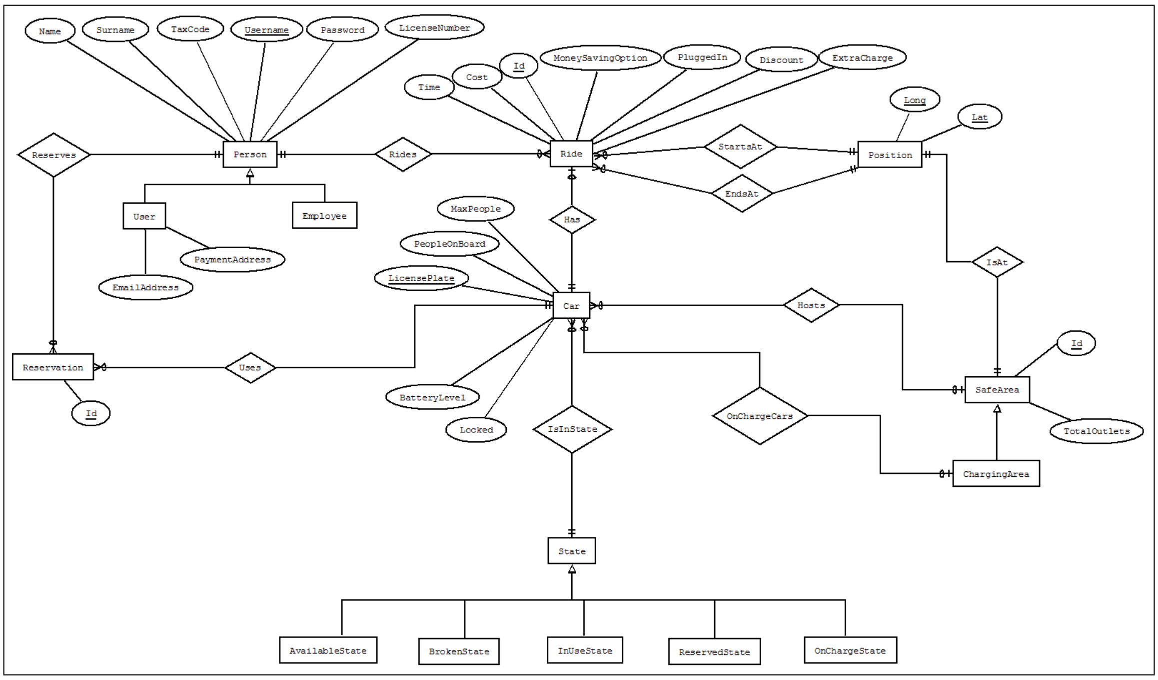

Figure 2.9: ER Diagram

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_PENALTY := 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 penalties
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 Cars Uniform 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

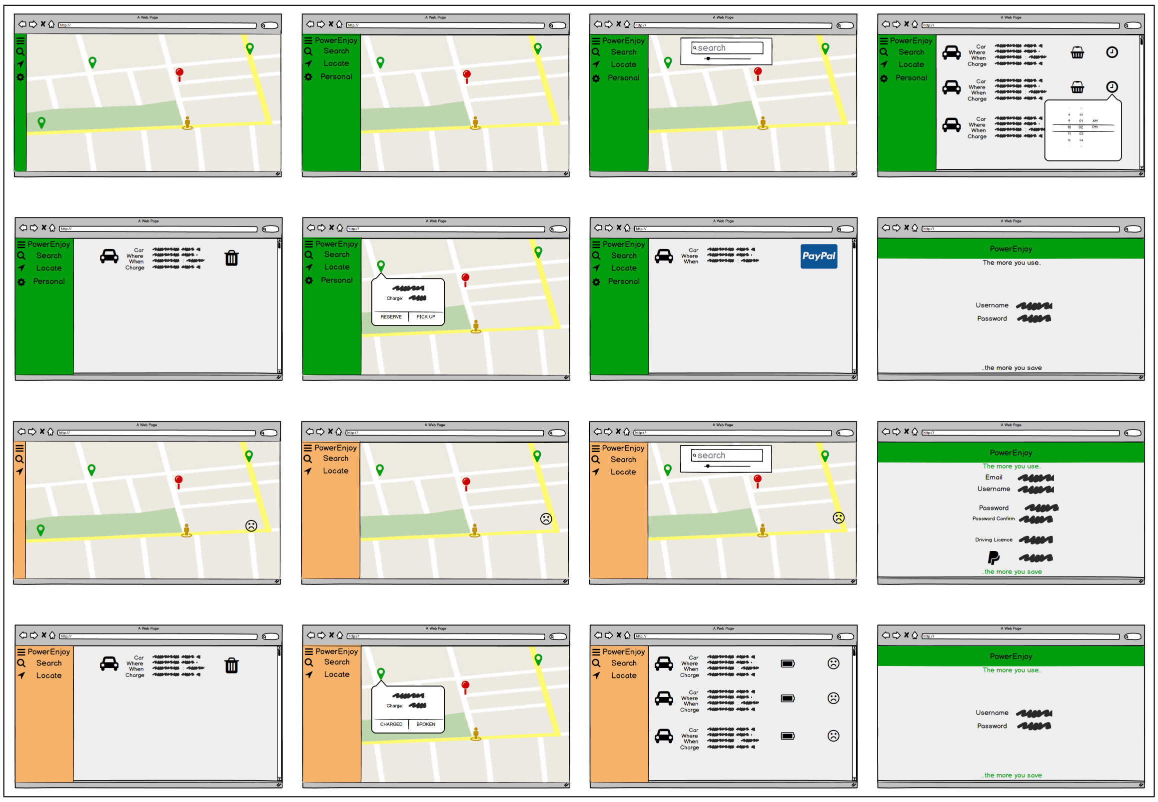

Figure 4.1: Webapp Interface on Personal Computers

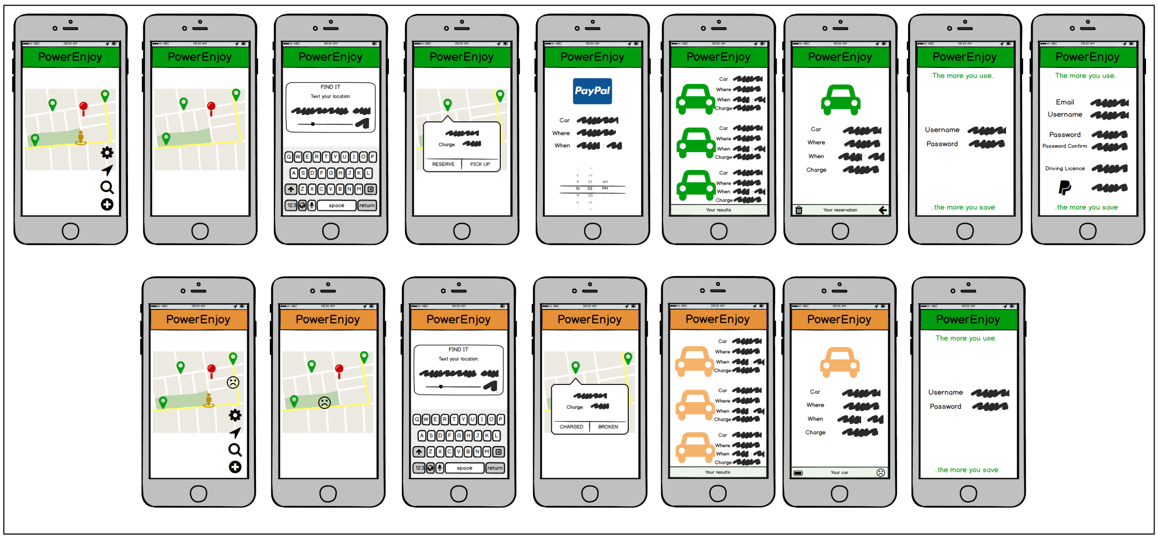

Figure 4.2: Mobile Interface on Mobile Devices

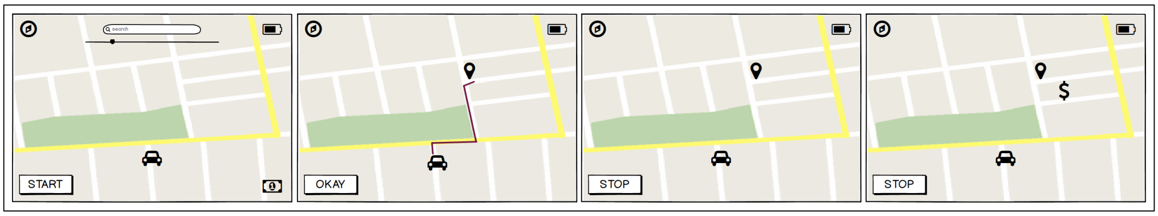

Figure 4.3: On-Board Car Screen Interface

5 Requirements Traceability

This section shows how both Functional and Non-Functional Requirements, already listed in the *RASD*, are mapped with respect to the components that the system is made of.

5.1 Functional Requirements

[R1] The system checks whether the user entered all the required data.

- ViewRender

[R2] The system checks whether the current user is already registered or not.

- ViewRender
- RegistrationManager
- Model

[R3] The systems checks that there is only one user with that name.

- ViewRender
- RegistrationManager
- Model

[R4] The system generates a password for the access and sends it back to the new user.

- ViewRender
- RegistrationManager

[R5] The system checks the input data and verifies whether the account really exists.

- ViewRender
- LogInManager
- Model

[R6] The system confirms the access.

- ViewRender
- LogInManager

[R7] The system verifies whether the given location exists.

- RideManager

- MapController

[R8] The system shows the available cars within the range of distance specified by the user from the given location.

- ViewRender
- MapController

[R9] The system obtains the user position via GPS.

- ViewRender
- Dispatcher

[R10] The system shows the available cars within the range of distance specified by the user from his current position.

- ViewRender
- Dispatcher
- MapController
- Model

[R11] The system obtains the user position via GPS and checks he is nearby.

- ViewRender
- Dispatcher
- MapController
- Model

[R12] The system verifies the car is available.

- Model

[R13] The system unlocks the car.

- RideManager
- Model
- ServerCommunicationManager
- CarCommunicationManager
- CentralUnit

[R14] The system shows those cars that are in the area specified by the user.

- ViewRender
- Dispatcher
- MapController
- Model

[R15] The system verifies that the user balance is not negative.

- RideManager

- PaymentManager

[R16] The system flags the car as Reserved.

- ReservationManger
- StateManager
- Model

[R17] The system checks that the user has one active reservation.

- ReservationManager
- Model

[R18] The system lets the user delete his active reservation.

- ReservationManager
- Model

[R19] The system checks the position of the car at the end of the ride.

- CentralUnit
- CarCommunicationManager
- ServerCommunicationManager
- Dispatcher
- RideManager
- MapController

[R20] The system checks the battery charge level at the end of the ride.

- CentralUnit
- CarCommunicationManager
- ServerCommunicationManager
- Dispatcher
- RideManager

[R21] The system checks whether the car is plugged into the power grid by the user at the end of the ride.

- CentralUnit
- CarCommunicationManager
- ServerCommunicationManager
- Dispatcher
- RideManager

[R22] The system checks the number of people on board.

- CentralUnit
- CarCommunicationManager
- ServerCommunicationManager

- Dispatcher
- RideManager

[R23] The system computes the discount or extra charge.

- RideManager
- RideCostCalculator

[R24] The system computes the total cost of the ride.

- RideManager
- RideCostCalculator

[R25] The system shows the cost on the car screen.

- RideManager
- ServerCommunicationManager
- CarCommunicationManager
- CentralUnit
- ScreenManager

[R26] The system requests the payment to the external system.

- RideManager
- PaymentManager

[R27] The system locks the car.

- RideManager
- ServerCommunicationManager
- CarCommunicationManager
- CentralUnit

[R28] The system confirms the access in administrator mode.

- ViewRender
- Dispatcher
- LogInManager
- Model

[R29] The system shows the cars within the range of distance specified by the employee from the given location.

- ViewRender
- Dispatcher
- MapController
- Model

[R30] The system shows the current state of the car.

- ViewRender
- StateManager
- Model

[R31] The system updates the state of the car with the new value.

- ViewRender
- Dispatcher
- StateManager
- Model

5.2 Non Functional Requirements

1. In order to distinguish between the users and the employees, the system provides two different modes of execution of the application: user mode and administrator mode. This is also graphically represented by a slightly different interface of the app.
    - ViewRender
    - LogInManager
    - Model
2. The reservation has an upper limit on time: the system can reserve a car for up to one hour before the ride. After that time expires with no ride, the system charges 1 EUR fee to the user.
    - ReservationManager
    - StateManager
    - Model
    - PaymentManager
3. The system provides discounts or extra charges on the last ride with the policy specified in the *RASD*.
    - RideManager
    - RideCostCalculator

6 Changelog

- 1.0
  - Initial release of the document
- 1.1
  - Added section *2.7.1 Technological Choices*, in which are listed the different technologies and standards that will be used in the implementation phase.

7 | Effort Spent

In order to complete this document, each author worked for 25 hours.

8 | References

8.1 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