POLITECNICO DI MILANO

Department of Electronic, Information and Biomedical Engineering

Master Degree course in Computer Science Engineering

Project of Software Engineering 2

PowerEnJoy

Design Document

(DD)

Version 1.0 (13 November 2016)

Reference professor: Prof. Elisabetta Di Nitto

Authors:

Davide Anghileri matr. 879194

Antonio Paladini matr. 879118

Academic Year 2016-2017

Contents

Revision History

Name	Date	Reason For Changes	Version
First complete version	11/12		1.0

Chapter 1

1 Introduction

1.1 Purpose

The purpose of this document (DD) is to give more technical details about our PowerEnJoy system than the RASD document.
This document is addressed to developers and aims to identify:

- The high-level architecture
- The design patterns
- The main components and their interfaces
- The Runtime behaviour

1.2 Scope

This Software Design Document is focused on the base level system and critical parts of the system. For this particular Software Design Document, the focus is placed on generation of the documents and modification of the documents.

Our project PowerEnJoy, which is a completely new mobile application, allows users to find an available electric car using the GPS position, reserve it for up to one hour and finally use it in respect of our rules.

Our application also allows maintainers to see the requests of support needed by our cars to repair it.

Furthermore, the system will also offer other services and functionalities like seeing past booking, asking for support or cancel a reservation.
Finally, another purpose of the system is to incentivize the use of electric cars and so reduce the CO_2 emissions.

1.3 Definitions, Acronyms and Abbreviations

1.3.1 Definitions

- *Car device*: a tablet installed into our cars that can show an interface to the drivers and can communicate with our system with an internet connection.
- *Thin Client*: a client with no Application logic inside.
- *Data encryption*: Sensitive information communicated over enterprise networks and the Internet can be protected by using encryption algorithms, which transform information into a form that can be deciphered only with a decryption key.
- Other definitions are in the RASD Document

1.3.2 Acronyms

- DD: Design Document
- MVC: Model-View-Controller
- OOP: Object-Oriented programming
- ER: Entity-Relationship model
- API: application programming interface
- RASD: requirements analysis and specifications document
- REST: REpresantional State Transfer
- RESTful: REST with no session

- IEEE: Institute of Electrical and Electronic Engineers
- GUI: Graphical User Interface
- DBMS: DataBase Management System
- BLL: Business Logic Layer
- DAL: Data Access Layer
- IP: Internet Protocol
- HTTP: HyperText Transfer Protocol
- JDBC: Java Database Connectivity

1.3.3 Abbreviations

- DB=database
- Info=information
- PEJ=PowerEnJoy

1.4 Reference Documents

- RASD produced before 1.0
- Assignments AA 2016-2017
- International Standard ISO/IEC/IEEE 29148 First edition 2011-12-01
- Design Document of Claudio Cardinale, Gilles Dejaegere and Massimo Dragano

1.5 Document structure

Our document is divided in the following sections:

- Introduction: in this section, we want to introduce the main concept and a first descriptions of our design document. It contains a justification of his utility and indications on which parts are covered in this document that are not covered by RASD.

- Architecture Design: this section is divided into different parts:
    1. Overview: this sections explains the division in tiers of our application.
    2. High level components and their interaction: this sections gives a global view of the components of the application and how they communicate.
    3. Component view: this sections gives a more detailed view of the components of the applications.
    4. Deploying view: this section shows the components that must be deployed to have the application running correctly.
    5. Runtime view: sequence diagrams are represented in this section to show the course of the different tasks of our application.
    6. Component interfaces: the interfaces between the components are presented in this section.
    7. Selected architectural styles and patterns: this section explains the architectural choices taken during the creation of the application.
    8. Other design decisions.

- Algorithms Design: this section describes the most important algorithms of our application. Pseudo code is used in order to hide unnecessary implementation details and to focus on the most important parts.
- User Interface Design: this section presents mock-ups and user experience.
- Requirements Traceability: this section aims to explain how the decisions taken in the RASD are linked to design elements.

Chapter 2

2 Architectural design

2.1 Overview

PowerEnJoy has a three tier architecture which separates GUI, Application logic and Data management on different physical layers.

We have shown lots of examples of how we will realize the GUI in the RASD. The client layer is thin, as in our application is required a persistent connection with the server, so it is unnecessary to put logic on the first level.

The application level involves an application server, which needs to handle a large number of simultaneous connections. For each user connected the server must accomplish his requests.

Finally, we have the Data Level, managed by a DBMS which performs all the action related to the insertion, deletion and update of data.

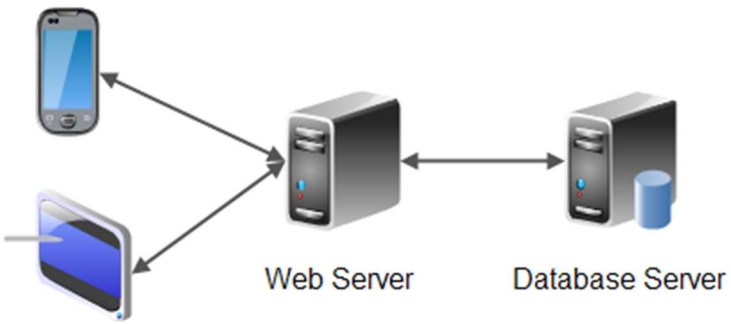

Figure 1

2.2 High level components and their interaction

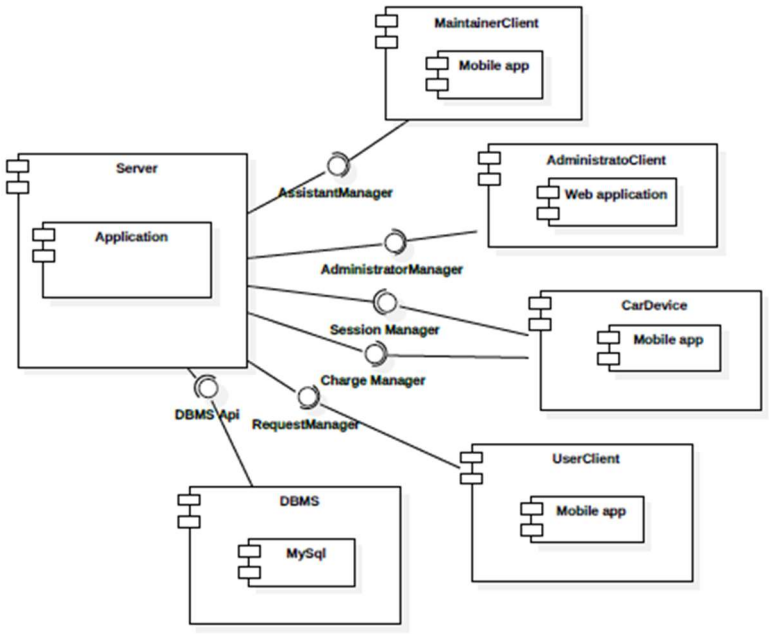

Figure 2

With the diagram in Figure 1 we want to explain the architecture used for developing our application and an overview of the entire system with the main components that we'll develop and their interactions.

The high-level components architecture is composed of six different elements types. The main element is the Server, it receives requests from clients and from the administrator client. A user client can initiate this communication from his mobile application; this communication is made in a synchronous way since the client, who initiates the communication, has to wait the answer of the central server that acknowledge him that his request has been taken into account. The Server can later send an asynchronous message to the client to notify him something. The Server communicates also with a third type of component, the car device, that can send asynchronous messages to the server to notify him about the status of the car, its position and the number of passengers. Also, the

Server can send messages to the car device to lock and unlock the car or to communicate the current charge of a trip.

Another component of our system is the administrator client that can communicate with the server in order to modify data or obtain information to monitor the system and visualize statistics.

The maintainer client communicates with the system in order to obtain a list with the current requests of assistance, with their details.

A final type of component is the Database. The server can communicate with it in order to read and write data or extract information.

2.3 Component view

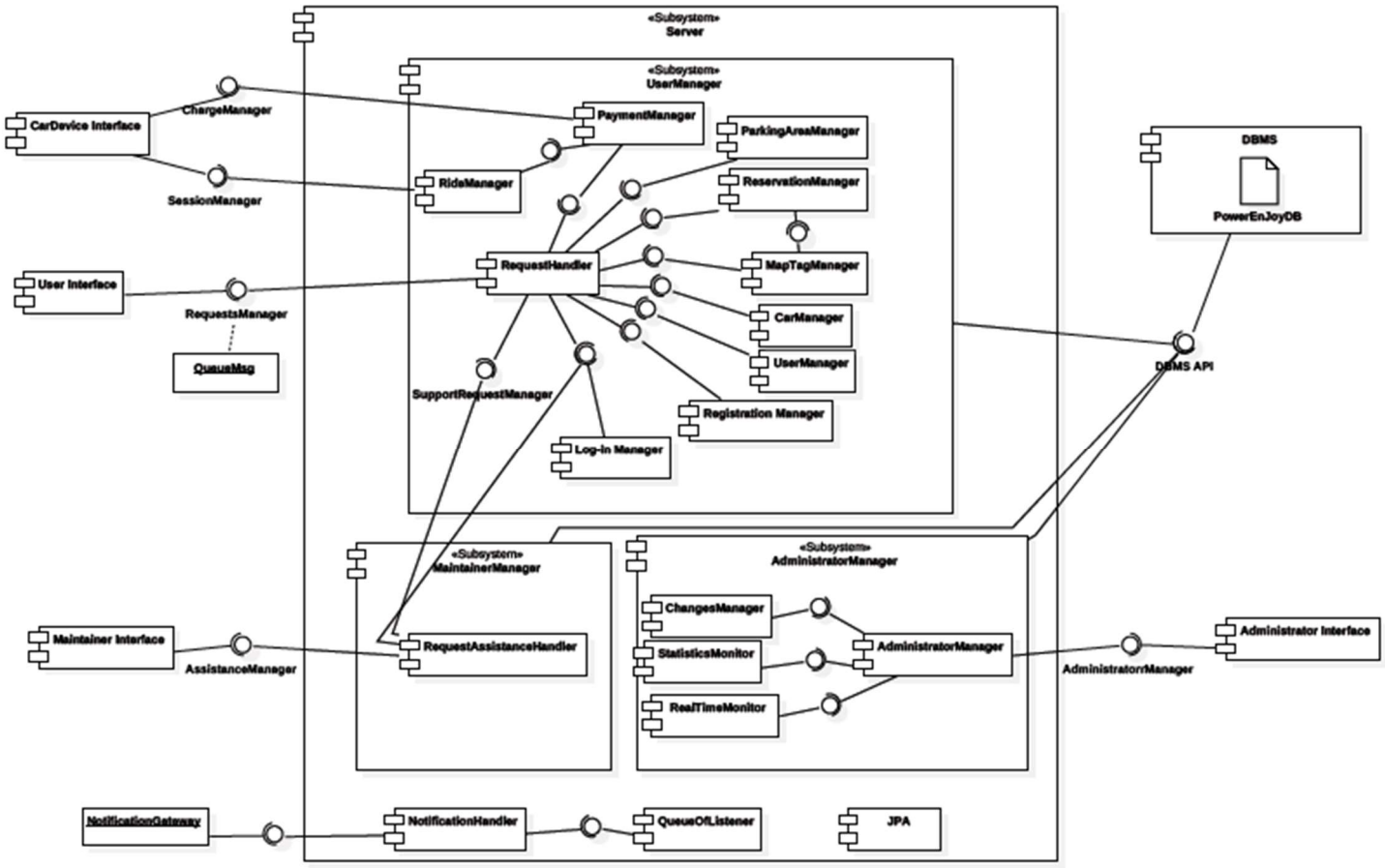

Figure 3

In the previous page there is the component diagram of our application which shows all the main functional blocks which compose PowerEnJoy. It covers all the features provided, this means that each action performed by our application is related to at least one of the components showed above.

We have three client interfaces which are the GUI for the three different types of client that we have, that is: user, maintainer, administrator. Moreover, we have a car device interface, which is the one which manages the car device and its sensors.

The **Server** component is marked with a *Subsystem* stereotype as it contains many different component and subsystem itself.

We have three different subsystems, one for each type of client.

The **User Manager** is a subsystem which manages all the possible requests and interactions with a user client application.
All users communicate with the system through a Requests Manager which puts in a queue the multiple requests made from users. A Request Handler component takes care of managing different types of requests and to delegate different jobs to the other logical components.
Now we have one component for each feature provided by the system to the user, so a Login Manager, a Registration Manager, a Car Manager and so on.
Thanks to these components a user is able to sign up to PowerEnJoy, to Login, to reserve a car, to cancel a reservation to see his past bookings and in general to perform all the other possible actions related to a user client.
Some of the component in this subsystem communicate with the Car Device through an interface which allow a two-way communication. From the Car Device to the Server in order to communicate data about a trip, and from the server to the Car Device in order to communicate the current charge of a client during a trip, or simply to lock and unlock the car.
The **Maintainer Manager** is a subsystem related to the maintainer logical area. The server provides a list of request of assistance to the maintainers which can access the information through the Assistance Manager Interface. The RequestAssistanceHandler component offer an interface to the User Manager in order to allow clients to make a request of assistance.
The **Administrator Manager** is a subsystem related to the administrator logical part of the application. Administrators communicate through an AdministratorManager interface with the server. An Administrator Manager component allows them to check the real time situation of the application, which means see all the user logged, all the parking areas, the cars and their actual position. But he can also add car to the system, as well as users or new Parking areas, he can modify prices of cars or apply discount.

An administrator can also access to statistics over the history of bookings.

A **Notification Handler** is used, combined with a Queue of Listener to keep in memory IP addresses of all the clients logged, in order to allow a persistent communication with them.

Finally, we have a **DBMS** component which manage the database of PowerEnJoy. All the logical components contained in the Server subsystem can communicate with DBMS through DBMS APIs.

2.4 Deployment view

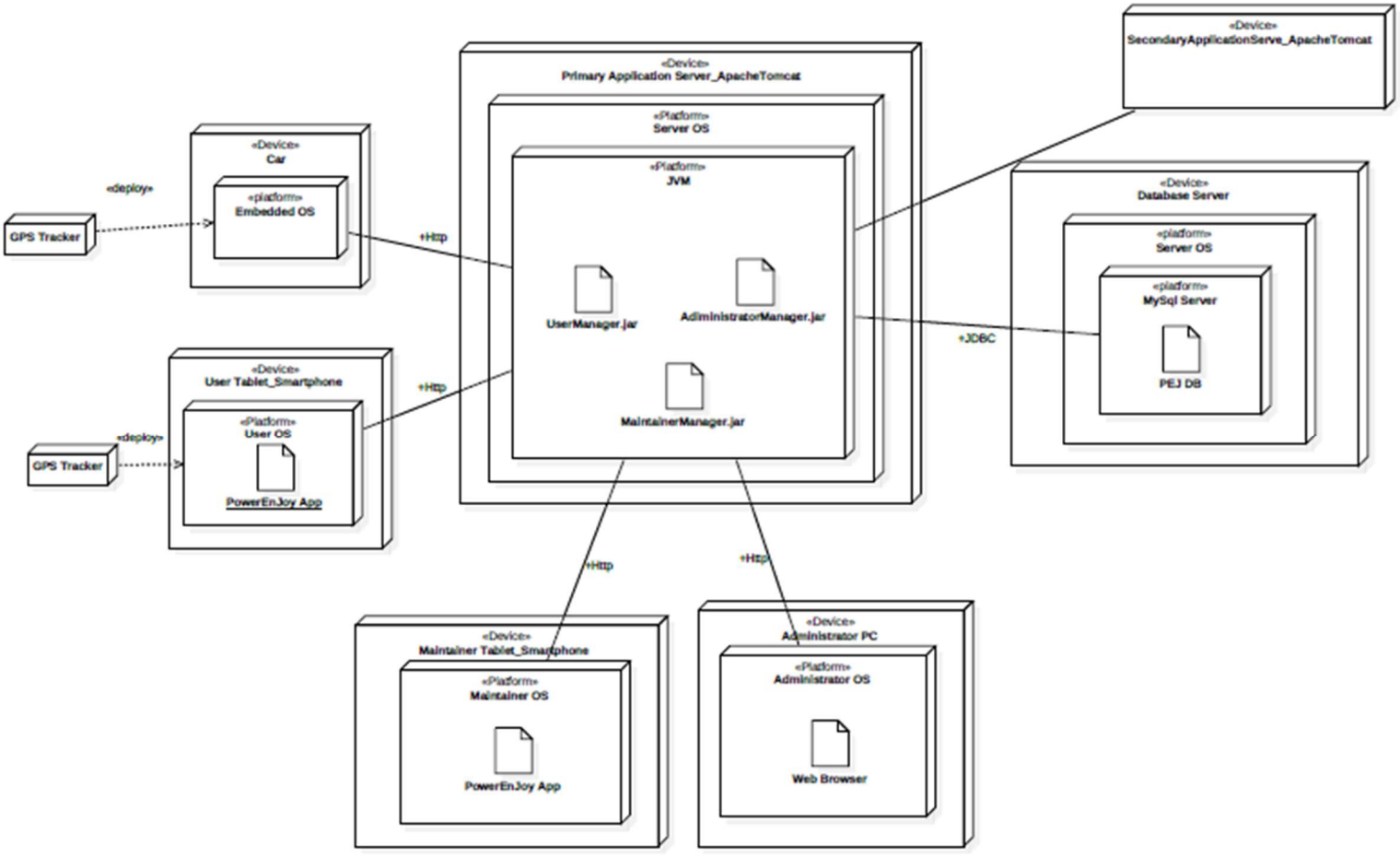

Figure 4

In *figure 4* there is a deployment view of the whole system. In the center we have the application server node, on which runs a JVM, as our application is developed in java. The three .jar files

represents the whole server application and manage the different type of clients.

Whe have also a secondary server, as specified in the RASD, which ensure us an appreccciable availability of the system.

Then we have a user Node, which is phisically represented by a tablet or a smartphone on which runs the mobile version of PowerEnJoy, which, as we said before, contains only the GUI.

The maintainer Node is similar to the user one, it contains the same application as the Maintainer area is developed within the PEJ client application.

The Car Device node represents the devices on board of each car, each of which runs an embedded operating system which allows the exchange of data with the application server.

Both Car and User node has a GPS tracker which allow to know their position.

The administrator node is quite different since the administrator area is accessible only with a browser.

Finally there is a database node, which complete the three-tier architecture, on which there is the PowerEnJoy DB. As DB server we use MySql Server.

2.5 Class Diagram

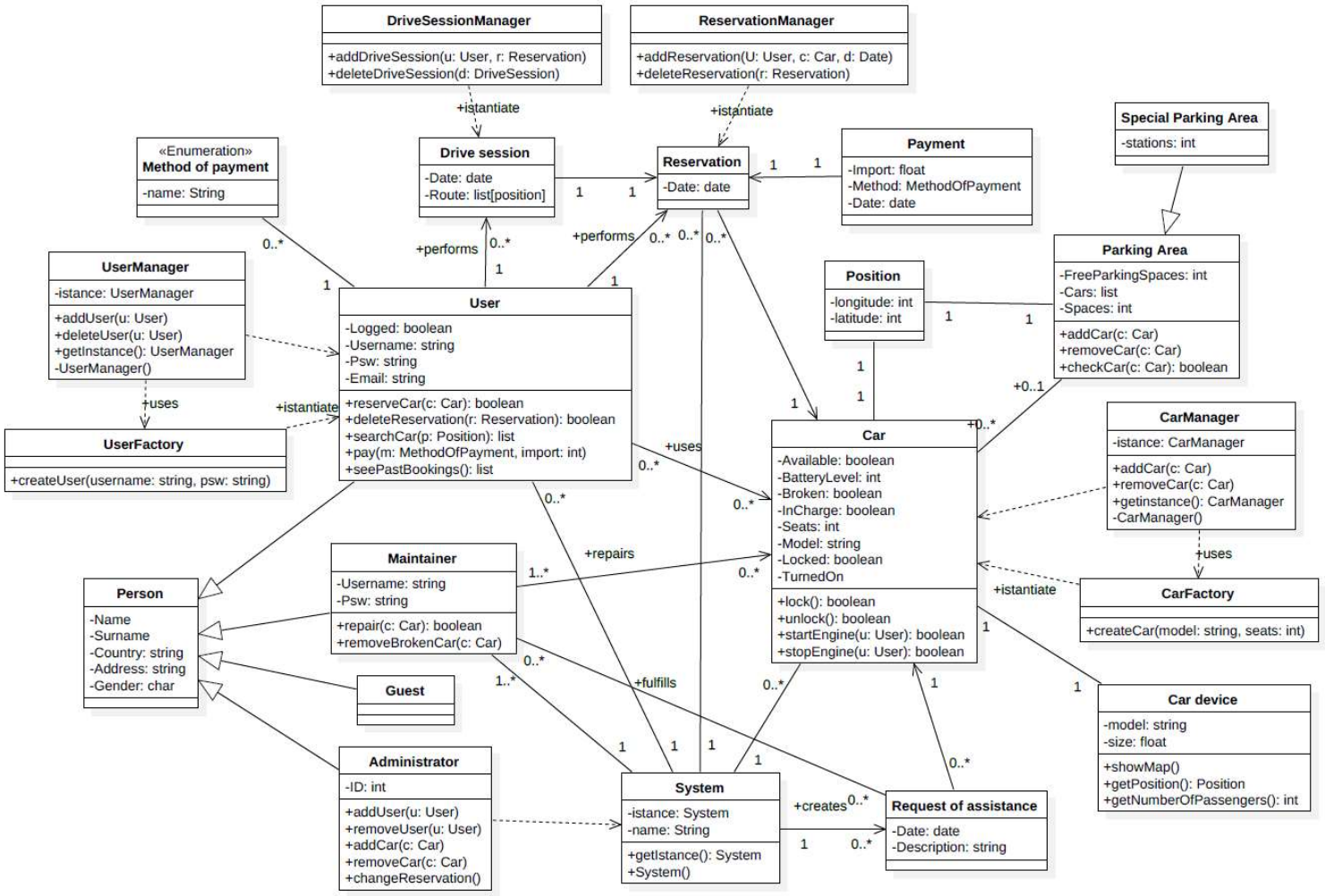

Figure 5

In *Figure 5* we have a representation of the UML **Class Diagram** that we have used to describe the static structure of the system by showing the system's classes, their attributes, operations (or methods), and the relationships among them.

As you can see in this diagram we have used the following design patterns: *Manager*, *Factory* and *Singleton*.

2.6 Runtime view

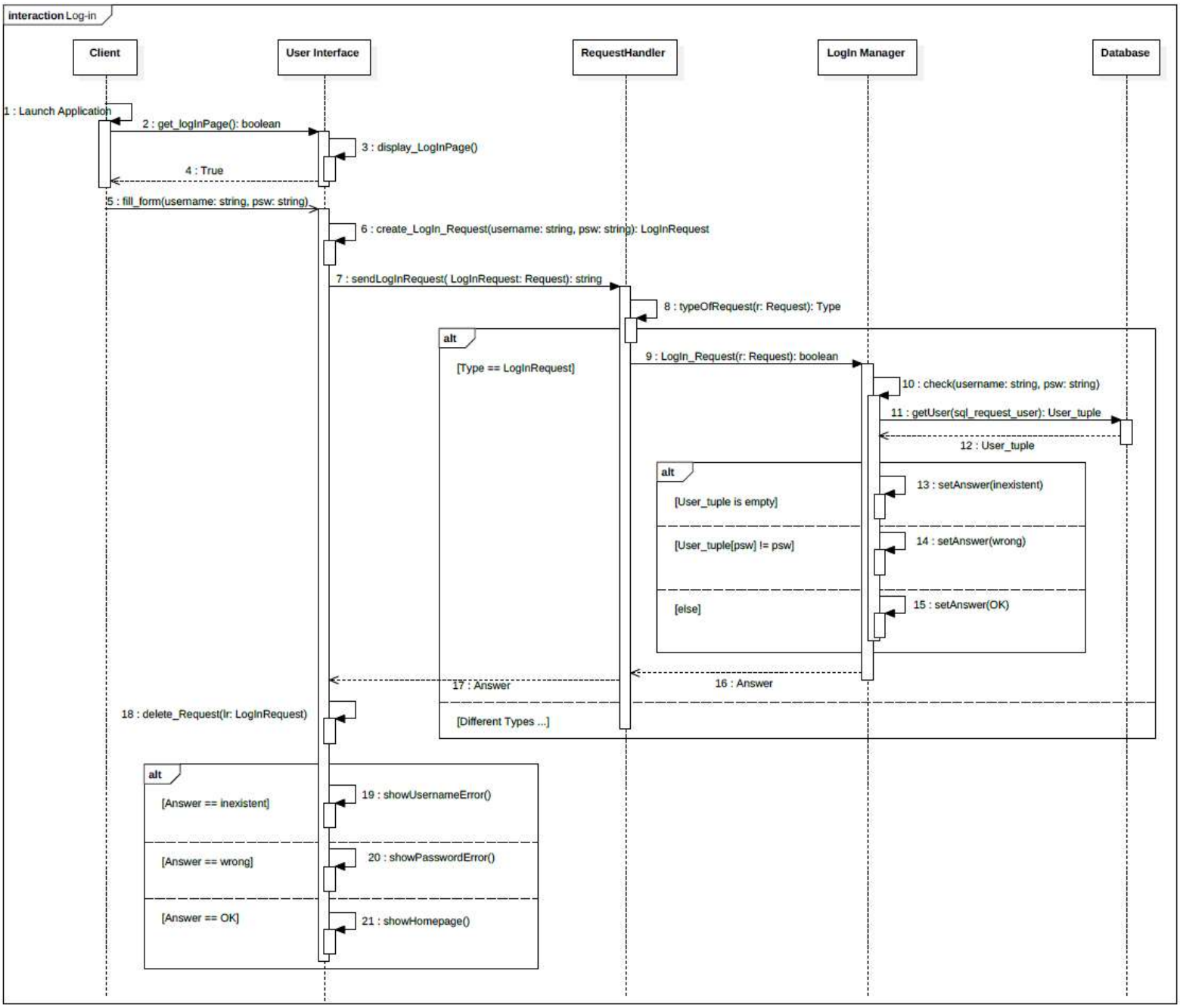

Figure 6

In *Figure 6* we have represented the Sequence Diagram for the **Log-In** of a User. The User has to launch the application and insert in the log-in Form his username and his password, then a request is send to the "responseHandler" that redirect the request to the appropriate Manager. The "LogInManager" checks if the User is present into the DB and if the password inserted is correct and then sends a response back to the "UserInterface".

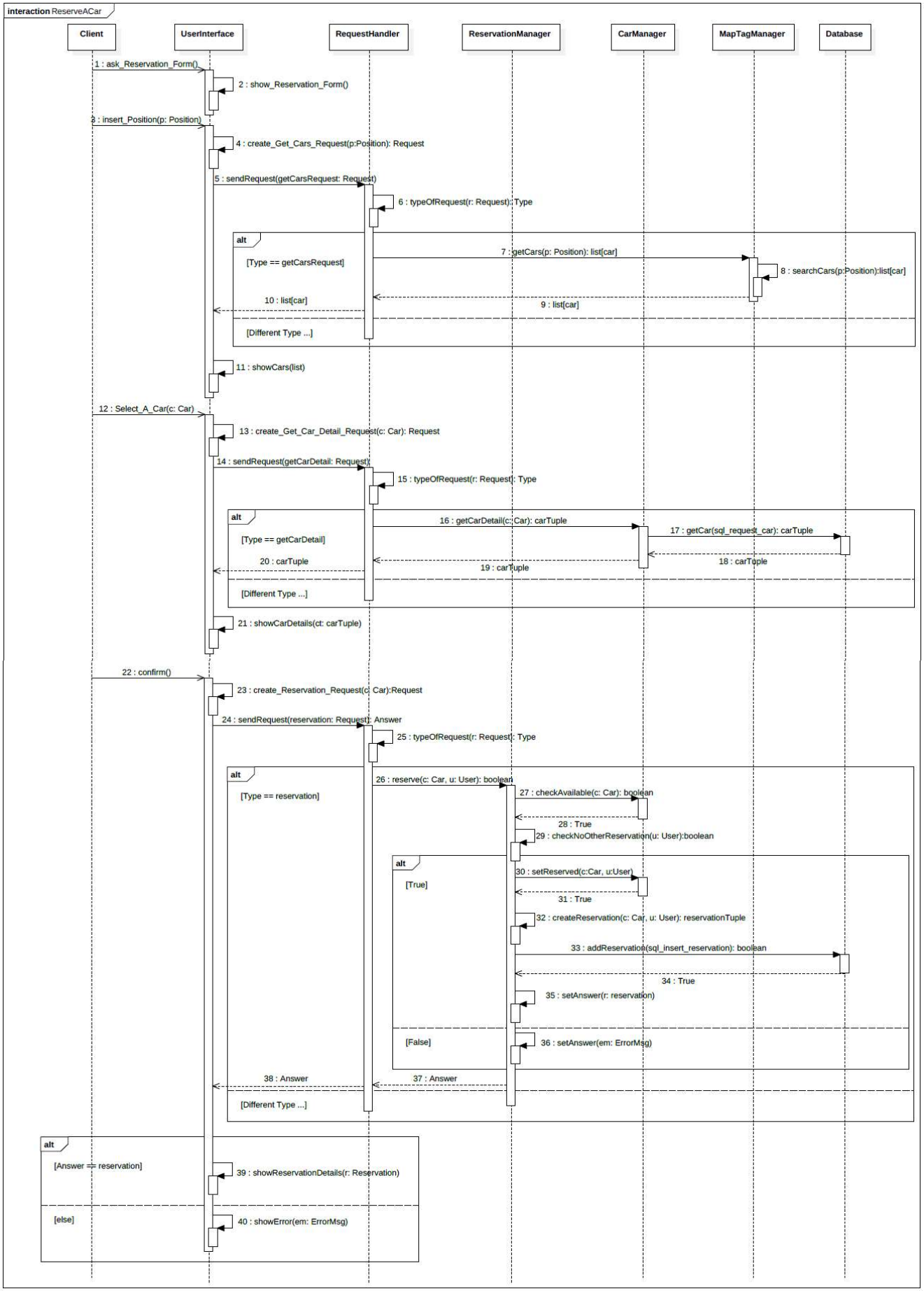
interaction ReserveACar
Client
UserInterface
RequestHandler
ReservationManager
CarManager
MapTagManager
Database
1 : ask_Reservation_Form()
2 : show_Reservation_Form()
3 : insert_Position(p: Position)
4 : create_Get_Cars_Request(p:Position): Request
5 : sendRequest(getCarsRequest: Request)
6 : typeOfRequest(r: Request): Type
alt
[Type == getCarsRequest]
7 : getCars(p: Position): list[car]
8 : searchCars(p:Position):list[car]
9 : list[car]
10 : list[car]
[Different Type ...]
11 : showCars(list)
12 : Select_A_Car(c: Car)
13 : create_Get_Car_Detail_Request(c: Car): Request
14 : sendRequest(getCarDetail: Request)
15 : typeOfRequest(r: Request): Type
alt
[Type == getCarDetail]
16 : getCarDetail(c: Car): carTuple
17 : getCar(sql_request_car): carTuple
18 : carTuple
19 : carTuple
20 : carTuple
[Different Type ...]
21 : showCarDetails(ct: carTuple)
22 : confirm()
23 : create_Reservation_Request(c: Car):Request
24 : sendRequest(reservation: Request): Answer
25 : typeOfRequest(r: Request): Type
alt
[Type == reservation]
26 : reserve(c: Car, u: User): boolean
27 : checkAvailable(c: Car): boolean
28 : True
29 : checkNoOtherReservation(u: User):boolean
alt
[True]
30 : setReserved(c:Car, u:User)
31 : True
32 : createReservation(c: Car, u: User): reservationTuple
33 : addReservation(sql_insert_reservation): boolean
34 : True
35 : setAnswer(r: reservation)
[False]
36 : setAnswer(em: ErrorMsg)
38 : Answer
37 : Answer
[Different Type ...]
alt
[Answer == reservation]
39 : showReservationDetails(r: Reservation)
[else]
40 : showError(em: ErrorMsg)

Figure 7

In *Figure 7* we have represented the interaction needed by our system when a user wants to **reserve a car**.

When a user enters in the reservation page and inserts a position, the User Interface asks at the MapTagManager which are the available cars near this position and then the ClientInterface shows them on the client mobile.

Then the user should select one car and he can receive through the CarManager and the Database all the details of the selected car.

Then the user confirms his reservation and first the ReservationManager checks that the car is still available and that the user has no other reservation active, then it create a new Reservation that is saved on the Database and finally the ReservationManager send back to the UserInterface the details about the reservation if all went right or an error message if something went wrong.

All the requests sanded by our client goes through the RequestHanlder that served them in order redirecting the requests to the right component Manager and also it is used to send back to the client the answer of the their requests.

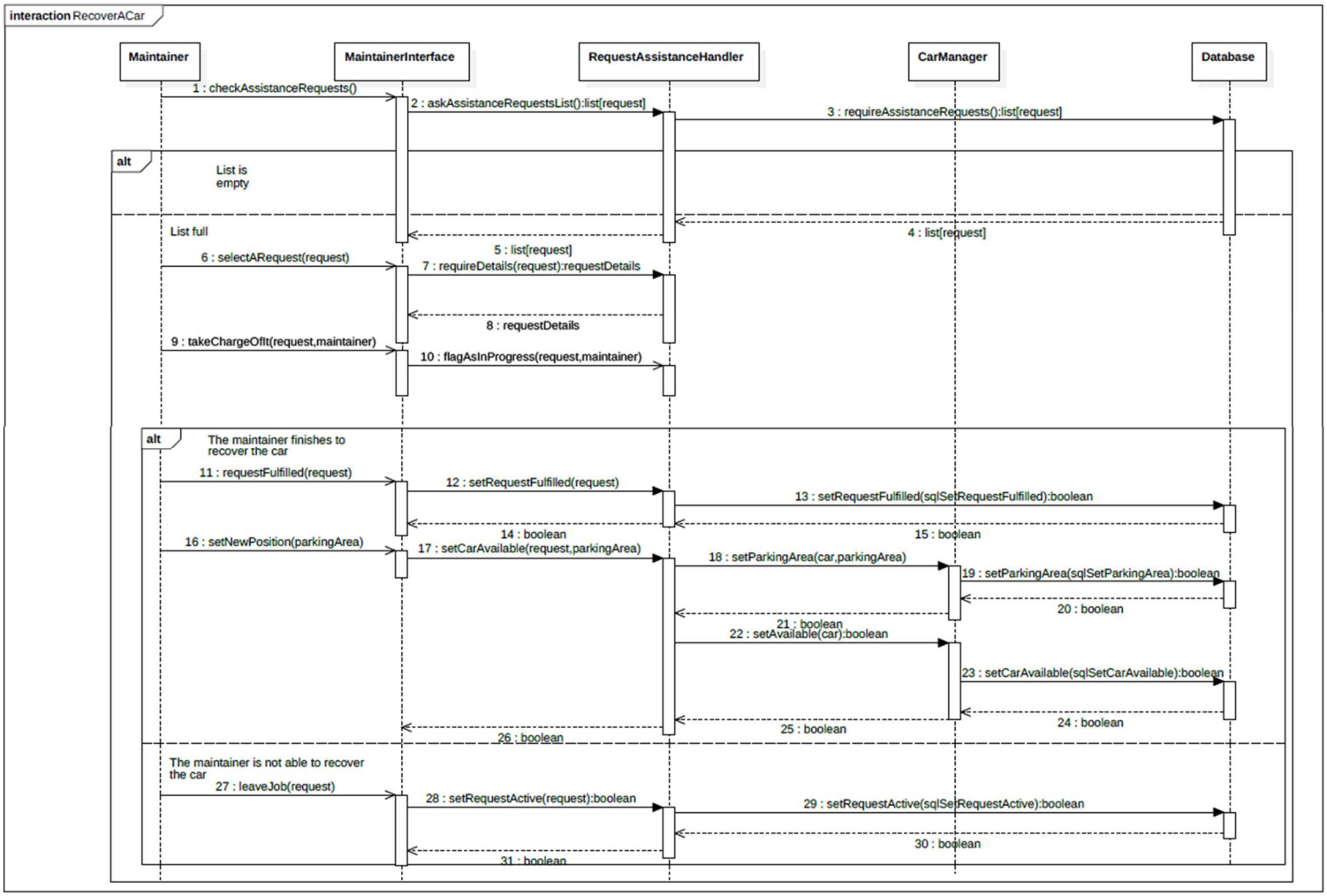

Figure 8

In *Figure 8* we show what happens when a Maintainer intervenes to **recover a car**. Supposing that the Maintainer is already logged in PEJ, he requires the list of active assistance requests.

If the list is not empty he can choose one of them and see the details about it, then he can decide to take charge of it.

A request of assistance can be represented by a simple need of moving the car from one place to another or it can mean that a car is broken and needs to be repaired. If the maintainer is not able to manage the situation he can decide to give up with the request and do something else. In this case he requires to flag the request as active again. Otherwise, once he has fulfilled the request, he asks to the system to change its state from "*in progress*" to "*fulfilled*". After this the system will set the car about which the request was done as available again, with its position updated.

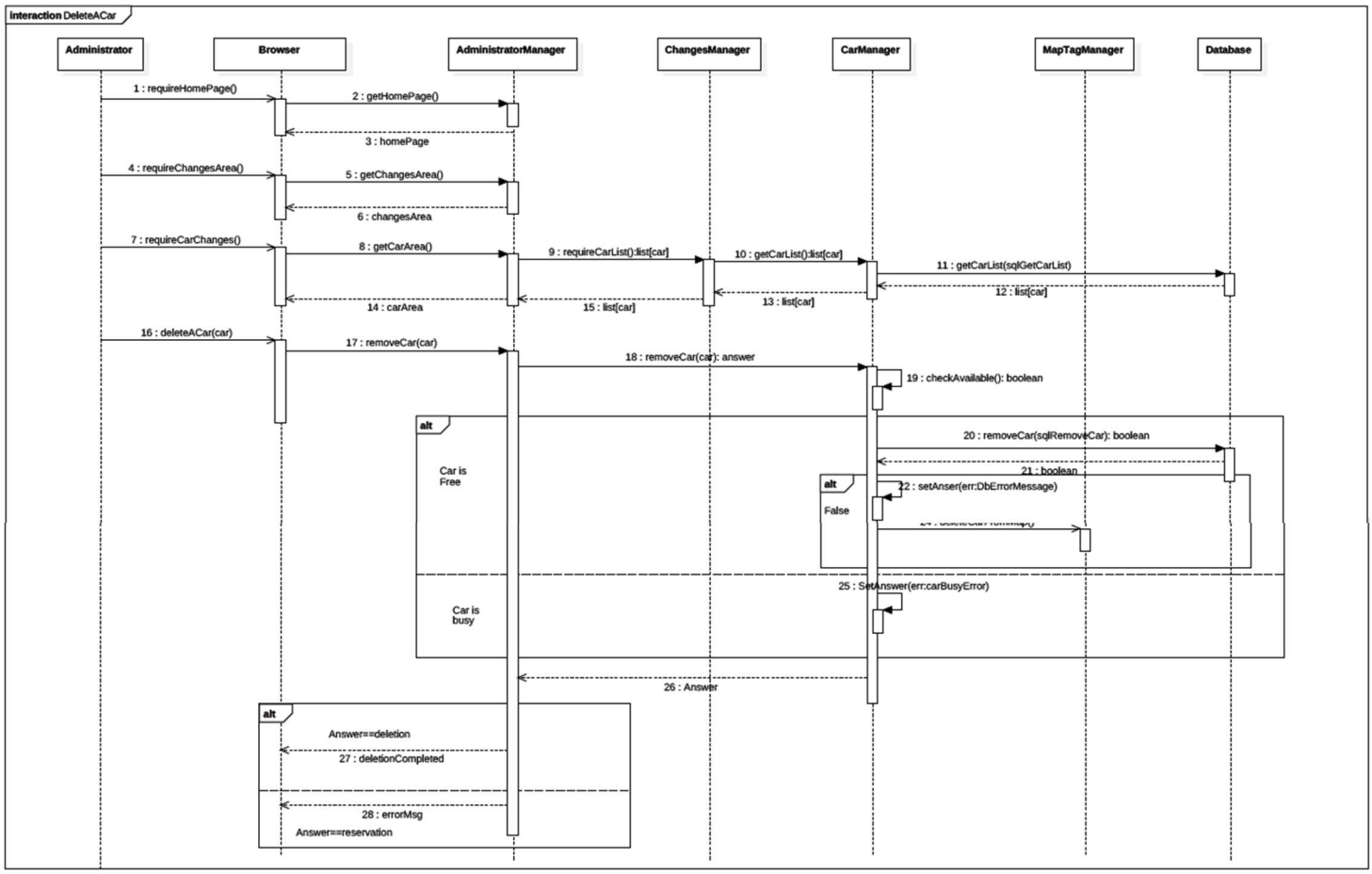

Figure 9

In *Figure 9* we see the procedure that takes place when an administrator **deletes a car** from the system. The administrator area is developed as a web application; administrators accesses to it through a web Browser. The administrator manager takes care of showing pages to the administrator.

First of all, the administrator requires the list of the cars used by PowerEnJoy, which is saved in the database.

The administrator chooses the car to be deleted, then the car manager checks if it is available, otherwise it means that someone has booked or is using it. If the car is available, it is deleted from database and from the Map which saves at runtime the position of all the cars.

2.7 Component Interfaces

Now we want to give a high-level description of the interfaces of the component present in our system[1].

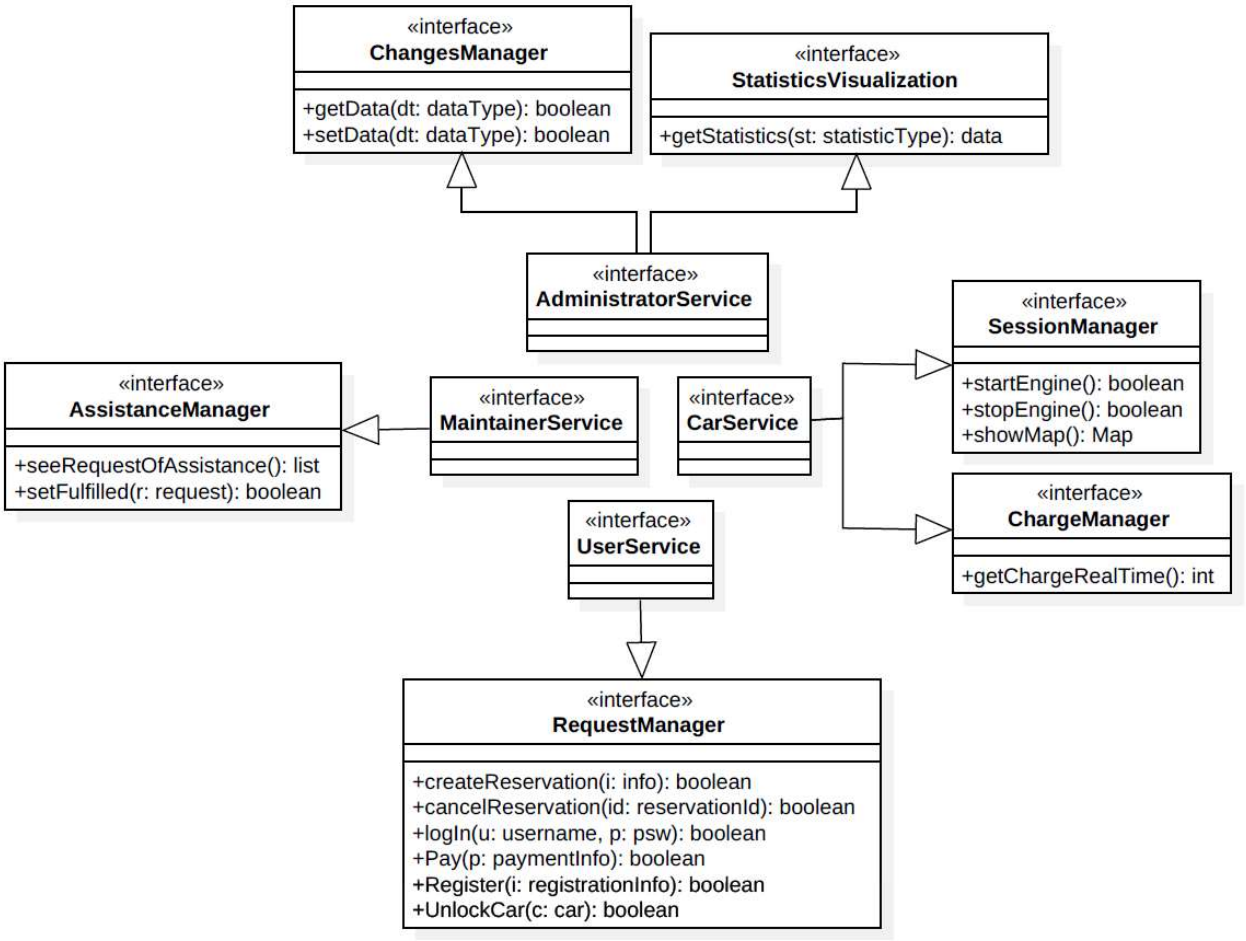

Figure 10

In the *Figure 10* we have represented the four interfaces of our system that are:

- User Service
- Maintainer Service
- Car Service
- Administrator Service

[1] The purpose of this diagram is not to provide detailed information regarding the parameters of the functions and their return values that for clarity and simplicity, in some case, they have been abbreviated using the words "info" and "type".

For the user, we have a "request manager" that takes care about all the request sanded by every user, using a queue manager to handle a large number of request. With this interface a user can sends different type of request to the server to perform different action like register, log-in, reserve a car, cancel a reservation or unlock the car.

For the car interface, we have two different managers, the first is the "session manager" that takes care about the driving session, it is used by the user to start and stop the engine of the car and he can also see the route and his position through this interface; the second interface is the "charge manager" that it's used to inform the driver about the charge of his driving session.

We also have an interface for the maintainers where they can see the details of each request of assistance and they can inform the system that they have repaired a car.

Finally, we have an interface for the administrator divided in two managers: the first takes care about the changes of data performed by the administrator and the second it's used to ask and receive statistics and data for monitoring the system by the administrator.

2.8 Selected architectural styles and patterns

2.8.1 Overall architecture

Our application is a 3-tiers architecture:

- Thin Client (GUI)
- Application logic (BLL: Business Logic Layer)
- Data management (DAL: Data Access Layer)

2.8.2 Protocols

Our tiers are connected through the network and they communicate with the following protocol.

JDBC: it's used by the application layer to communicate with the Databases. It provides methods to query and update data in the database, which is built with a relational model.
JDBC also provide some techniques for increasing the security of our system, like:

- Data Encryption: Some of the supported encryption algorithms are RC4, DES, 3DES, and AES.
- Data Integrity: guaranteed by using a message digest, with MD5 or SHA-1 hashing algorithms, and including it with each message sent across a network. This protects the communicated data from attacks, such as data modification, deleted packets, and replay attacks.
- Strong Authentication: It supports the following industry-standard authentication methods:
    - Kerberos
    - Remote Authentication Dial-In User Service (RADIUS)
    - Distributed Computing Environment (DCE)
    - Secure Sockets Layer (SSL)

Clients communicate with BLL through a **RESTFul API with JSON** which use HTTP to perform basic authentication for each request. BLL keep in memory a queue of connected clients, ensuring a persistent communication with all the clients connected. SSL is used in order to ensure the security over the exchanging of data.

2.8.3 Design patterns

Client-Server: Our application is based on a Client-Server model with 3-tiers. The clients represent the user mobile application, the maintainer mobile application, the car device application and the administrator web browser, and they are thin Client because we want to guarantee that our application runs with a good performance also on low-resources devices.
We have chosen this architecture for the following reason:

- Data consistency and synchronization: guaranteed by one single business logic application.
- We can improve the computing power, memory and storage requirements of a server to scale up with the work load.
- Improves the security between clients, that know only the server endpoint but no other clients, and they can access at only few data.
- We can serve many clients simultaneously and we can add or remove clients without stopping the application.
- We can easy administrate the system focusing on the server thanks to the fact that the clients are thin.

Adapter: Adapters are used in our mobile application to adapt the Driver interface to the RESTful API one.

Observer: We decided to use this design pattern because it works well with Symfony and because it's used to automatically notifies clients of any state and to implement distributed event handling systems.

Factory: We use a factory pattern in order to manage the creation of objects like cars, users and parking areas.

Manager: We use a manager pattern, combined with the factory, to handle multiple entities of the same type. An administrator is a manager, as he can modify or delete objects like cars and users.

Singleton: All managers and the system class are singleton, so there is only a unique instance of them at each time of the execution of the program.

MVC: We have used a Model-View-Controller in our application as shown in *Figure 11* which consists of separating the software components that implement the model of the business functionality (model), the components that implement the presentation logic (view) and those that control these features use (controller).
Also, we decided to use a Symfony framework based on PHP for the Application Server because it guarantee maintainability, scalability, modularity and reuse of code.
Furthermore, for the Client side we decided to use JavascriptMVC that is an easy to use framework for building the client interface that guarantee a good level of maintainability.

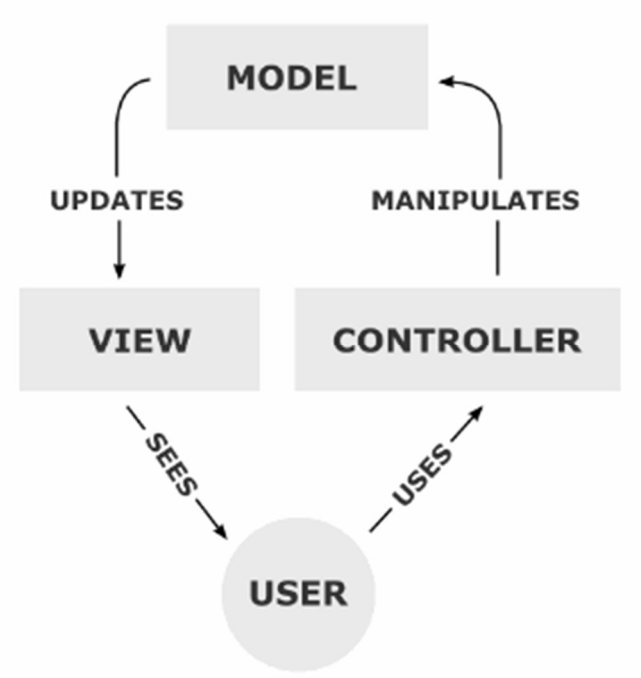

Figure 11

2.8.4 Paradigms

OOP: We decided to use an **Object-oriented programming** paradigm based on the concept of "objects", which may contain data, in the form of fields, (known as *attributes)* and code, in the form of procedures (known as *methods).*

The reason of this choice are that OOP can provide encapsulation that is a technique that encourages decoupling, and it can also provide composition and inheritance that are techniques that are techniques that lead to having a more simple code, better structured and easier to maintain and enable strong separation of concerns.

2.9 ER Model

In Figure 7 we have represented the Entity-Relationship Model of our Database.

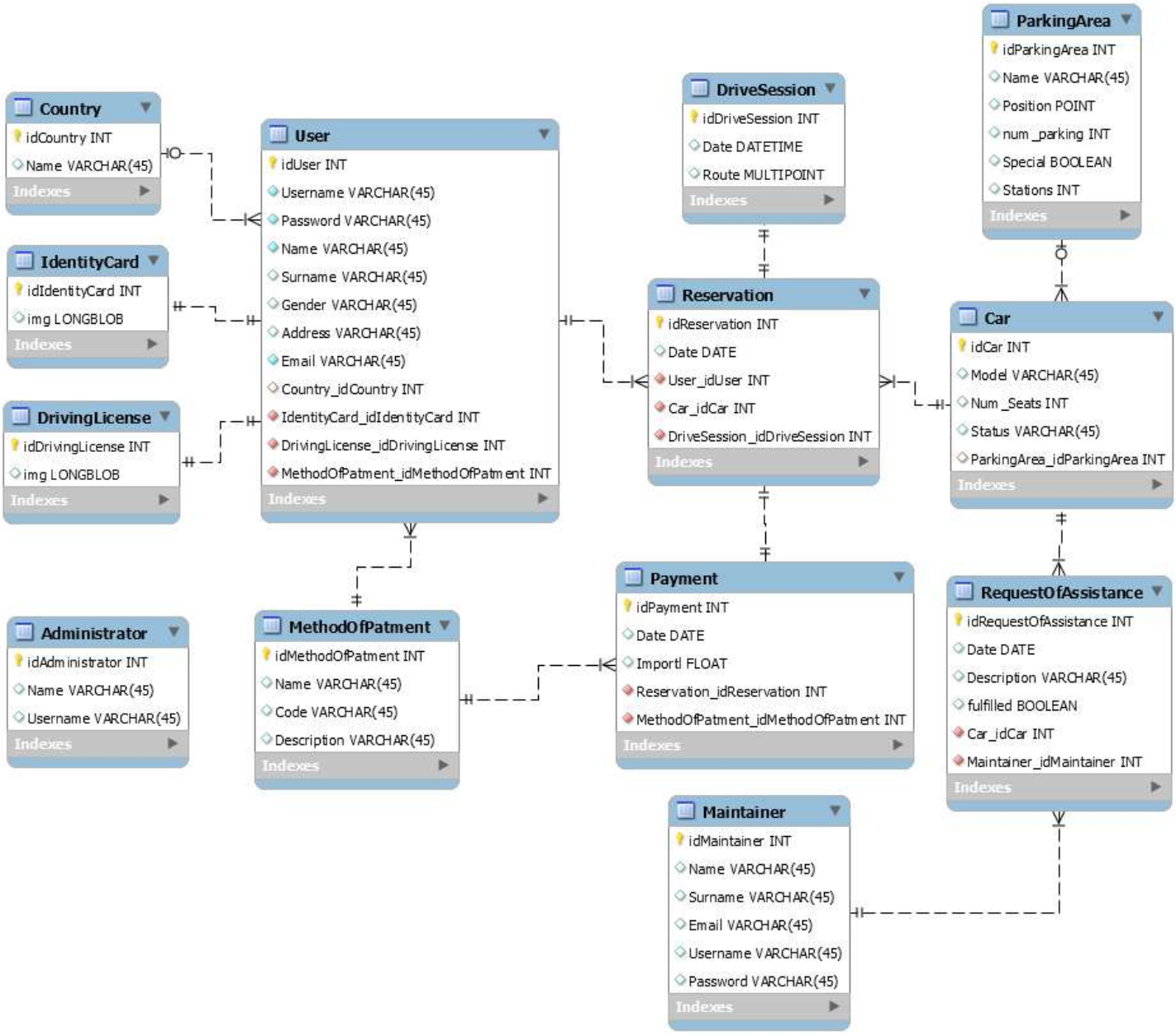

Figure 12

2.10 Other design decision

For the maps both on the UserInterface and on the CarDeviceInterface we decided to use the Google Maps Android API and Google Maps Directions API in order to create and customize the map and calculate directions between locations.
For more information, see the Google Website (https://developers.google.com/maps/documentation/android-api/)

3 Algorithm design

In this paragraph, we want to show the more significant algorithm of our system that is the function that search all the available cars near a specified position.

We decided to implement the search in this way because our function has a complexity of: $\Theta(\log(N)) + c$

and we want a function that is quite fast to ensure good performances and to satisfy usability requirements.

```
public class Car {
    private String name;
    private String model;
    private int num_seats;
    private String status;
    private Point position;
    private boolean available;
    private int batteryLevel;

    public Car(String name, String model, int num_seats,Point p, boolean a, int batteryLevel){
        this.name= name;
        this.model=model;
        this.num_seats=num_seats;
        this.status="available";
        this.position=p;
        this.available=a;
        this.batteryLevel=batteryLevel;
    }
```

Figure 13

The *Figure 13* shows the attribute and the constructor of the Class "Car" that we used for the "search Cars" algorithm.

A position is a Point represented by 2 attributes: the first represents the latitude and goes from 0 (90° S) to 180 (90° N), the second represents the longitude and goes from 0 (180° E) to 360 (180° W).

```
public class MapTagManager {

    private List<Car> cars = new ArrayList<Car>();

    public MapTagManager() {

    public List<Car> getCars() {

    public void setCars(List<Car> cars) {

    // search all the cars within the "range" from the specified position
    public List<Car> searchCars(Point position, int range) {

    // return any of the cars within the "range" from the specified position
    private int binarySearch(List<Car> cars, double p, int range) {

    private boolean nearMetersM(Point p1, Point p2, int range) {

    private double distanceFrom0(Point p) {
```

Figure 14

The *Figure 14* shows the Class "MapTagManager" that takes care about the cars of our system and their status and position.
This class takes a list of all the cars handled by our system ordered by their distance from the point (0.0), thanks to the code in Figure 11.

```
// Ordering Cars by their distance to the point (0.0)
Collections.sort(cars, new Comparator<Car>() {

    public int compare(Car c1, Car c2) {
        return Integer.compare((int) distanceFrom0(c1.getPosition()), (int) distanceFrom0(c2.getPosition()));
    }
});
```

Figure 15

In the *Figure 15* we wrote the algorithm that searches all the cars available in our system that are near a specified position within a specified range.
This function uses two other functions that are represented in *Figure 16 and 17*.

```
// return any of the cars within the "range" from the specified position
private int binarySearch(List<Car> cars, double p, int range) {
    int i = 0;
    int j = cars.size() - 1;
    int middle = 0;
    while (i <= j) {
        middle = (int) ((i + j) / 2);
        // distance from 0.0 of car with index "middle"
        double distanceFrom0 = distanceFrom0(cars.get(middle).getPosition());
        // if distanceFrom0 is greater than the searched position plus the "range"
        if (p + range < distanceFrom0) {
            j = middle - 1;
        } else if (p - range > distanceFrom0) {
            i = middle + 1;
        }
        // is a car with position between p-range and p+range
        else {
            return middle;
        }
    }
    return -1;
}
```

Figure 16

```
// return any of the cars within the "range" from the specified position
private int binarySearch(List<Car> cars, double p, int range) {
    int i = 0;
    int j = cars.size() - 1;
    int middle = 0;
    while (i <= j) {
        middle = (int) ((i + j) / 2);
        // distance from 0.0 of car with index "middle"
        double distanceFrom0 = distanceFrom0(cars.get(middle).getPosition());
        // if distanceFrom0 is greater than the searched position plus the "range"
        if (p + range < distanceFrom0) {
            j = middle - 1;
        } else if (p - range > distanceFrom0) {
            i = middle + 1;
        }
        // is a car with position between p-range and p+range
        else {
            return middle;
        }
    }
    return -1;
}

private boolean nearMetersM(Point p1, Point p2, int range) {
    if (Math.sqrt(Math.pow(p2.getX() - p1.getX(), 2) + Math.pow(p2.getY() - p1.getY(), 2)) < range) {
        return true;
    }
    return false;
}

private double distanceFrom0(Point p) {
    return Math.sqrt(Math.pow(p.getX(), 2) + Math.pow(p.getY(), 2));
}
```

Figure 17

The searchCars algorithm takes a position and a range within which searching the available cars and it works in the following way:

- Calculate the distance from point (0.0) of the Position.
- Search with the BinarySearch function any of the cars that distances from the distance of the specified position more or less than "range" (any of the cars in the blue or yellow semicircle).
- If no car is found return -1
- Else, return the index of this car (Binary Searched) into the list of cars of the MapTagManager that are ordered by their distance from the point (0.0).
- From this point search all the cars that have a greater index (that distances from (0.0) more than the distance of the Binary Searched car) as long as the distance is lower than specified position plus the range (all the cars in the blue semicircle), and save only the cars that are near the specified Position (cars in the red circle).
- Repeat the same procedure for the cars that distance less than the specified position minus the range (all the cars in the yellow semicircle) and save only the cars that are near the specified Position (cars in the red circle).
- Return the saved cars.

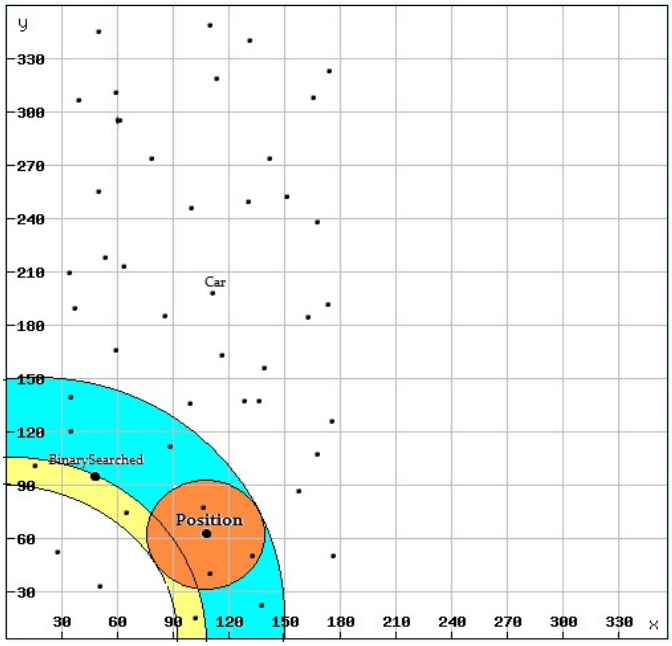

Figure 18

4 User interface design

4.1 Mock-ups

We have shown most of mock-ups in the RASD at the beginning of chapter 3.

4.2 User Experience Diagrams

Here we are going to give a more specific version of the structure of the application as it is experienced by the user.

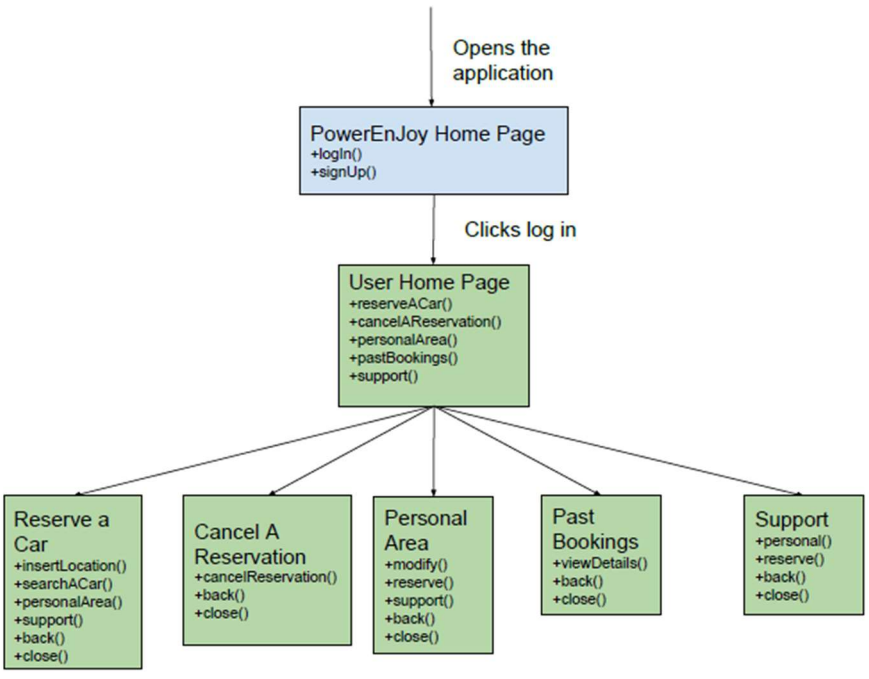

Figure 19

The diagram visible in *Figure 19* shows the structure of the application as visible by a user who log in into it. Follows a specific view of the reservation area.

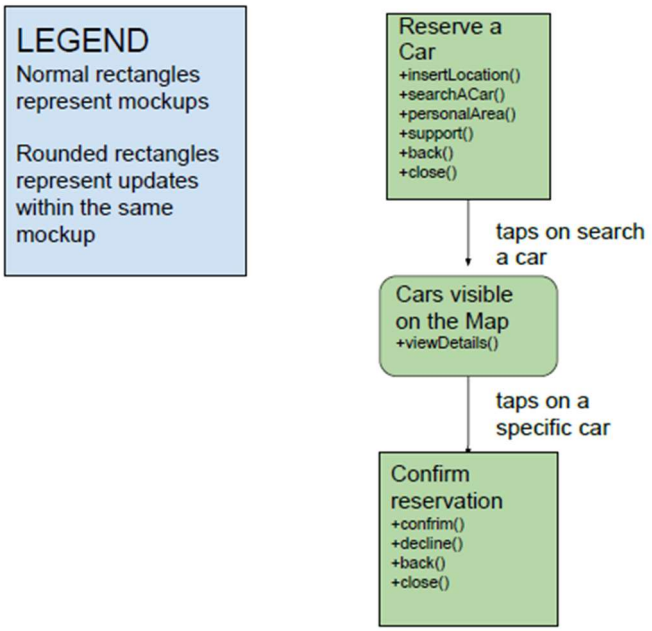

Figure 20

In *Figure 20* the diagram shows what happens when a user goes in the reservation area in order to reserve a car. What could be not so clear until this point is that for users who have already done a reservation this area is different. In particular, in this case, the reservation area shows an interface which allows to lock and unlock the car once the user is close enough to it.

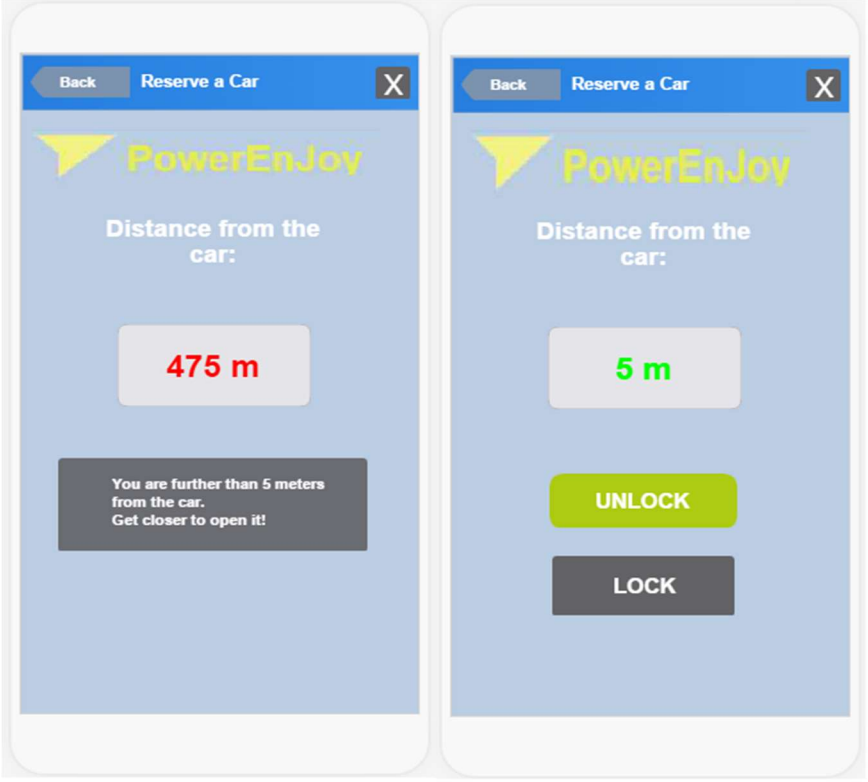

Figure 21

5 Requirement traceability

We have made this Design document with the purpose of fulfil all the Goals and Requirements need by our system that are specified in the RASD Document.
Here we present a list of this Goals and the component that we identified to satisfy them.

- ✓ G1: Allow users to enter in our cars and use them
  - RequestHandler
  - UserInterface Component
  - ResrvationManager
  - RideManager
- ✓ G2: The system gives information to the users about the position of available electric cars
  - RequestHandler
  - UserInterface Component
  - ResrvationManager
  - MapTagManager
  - NotificationHandler
  - Database
- ✓ G3: Allow users to reserve an available car
  - RequestHandler
  - UserInterface Component
  - ResrvationManager
  - CarManager
- ✓ G4: Guests must be able to register and get a private access-password
  - RequestHandler
  - UserInterface Component
  - RegistrationManager
  - Database

- ✓ G5: Allow users to be recognised by login procedure
  - RequestHandler
  - UserInterface Component
  - LogInManager
  - Database
- ✓ G6: The system allows users to delete a reservation they have already done
  - RequestHandler
  - ReservationManager
  - Database
- ✓ G7: The system keeps trace of the past charges
  - RideManager
  - ReservationManager
  - PaymentManager
- ✓ G8: Make users aware of position and conditions of parking areas (weather full or not)
  - RequestHandler
  - UserInterface Component
  - ParkingAreaManager
  - Database
- ✓ G9: System allows stops during a drive session
  - CarDeviceInterface Component
  - RideManager
- ✓ G10: The system manages car accidents
  - RequestAssistanceHandler
  - RequestHandler
  - CarManager
  - Database
- ✓ G11: The system ends travel sessions and charges the users
  - UserInterface Component
  - RideManager
  - PaymentManager

- ✓ G12: PowerEnJoy incentivizes the virtuous behaviours of the users
  - PaymentManager
  - UserManager
  - NotificationHanlder
- ✓ G13: The maintainer takes care about physical assistance
  - MaintainerInterface Component
  - RequestAssistanceHandler
  - RequestHandler
  - CarManager
  - Database
- ✓ G14: The Administrator takes care about the application
  - ChangesManager
  - RealTimeMonitor
  - StatisticsMonitor
  - AdministratorManager
  - AdministratorInterface Component
  - Database

6 References

6.1 Used Tools

To draw up this document we've used the following tools:

- Microsoft Office World 2016: to redact and format this document
- StarUML: to create Component diagram, Deployment diagram, sequence diagrams, Class diagram and Component Interface diagram.
- Paint: to create and adapt the images of this document
- GitHub: to save and control the version of the document
- MySQL Workbench 6.3 CE: to create the ER model
- Google Drawing: to create the user experience diagram

7 Hours of work

Date	Antonio's hours	Davide's hours
2016/11/18	2h	2h
2016/11/19	1h	--
2016/11/23	--	3h
2016/11/25	2h	--
2016/11/26	2h	2h
2016/11/29	4h	4h
2016/12/02	--	1h
2016/11/03	--	1.30h
2016/12/04	3.30h	--
2016/12/06	6h	6h
2016/12/08	2h	2h
2016/12/09	3.30h	3.30h
2016/12/10	4h	--
2016/12/11	1h	1h
Total	**31h**	**27h**