

**Software Engineering 2: “PowerEnJoy”**

**Design Document (V. 1.0)**

Authors:

Sergio CAPRARA

Soheil GHANBARI

Erica TINTI

Milan, Italy

11/12/2016

**Table of contents**

[1. Introduction 3](#_Toc469163098)

[1.1 Purpose 3](#_Toc469163099)

[1.2 Scope 3](#_Toc469163100)

[1.3 Glossary 3](#_Toc469163101)

[1.4 Reference Documents 4](#_Toc469163102)

[1.5 Document Structure 4](#_Toc469163103)

[2. Architectural Design 6](#_Toc469163104)

[2.1 Overview 6](#_Toc469163105)

[2.2 High level components and their interaction 7](#_Toc469163106)

[2.3 Component view 8](#_Toc469163107)

[2.3.1 Central Application 8](#_Toc469163108)

[2.3.2 Persistence 9](#_Toc469163109)

[2.3.3 Applications 10](#_Toc469163110)

[2.4 Deployment view 10](#_Toc469163111)

[2.5 Runtime view 12](#_Toc469163112)

[2.5.1 Money Saving Option 12](#_Toc469163113)

[2.5.2 User car lock request 14](#_Toc469163114)

[2.5.3 Operator does the maintenance 16](#_Toc469163115)

[2.5.4 Checking status of the cars 18](#_Toc469163116)

[2.6 Component interfaces 19](#_Toc469163117)

[2.7 Selected architectural styles and patterns 19](#_Toc469163118)

[2.8 Other design decisions 19](#_Toc469163119)

[3. Algorithm Design 20](#_Toc469163120)

[3.1 List of available cars 20](#_Toc469163121)

[3.2 Check distance and Unlock the car 20](#_Toc469163122)

[3.3 Money saving option 21](#_Toc469163123)

[3.4 Validate the parking 22](#_Toc469163124)

[3.5 Calculate the final amount to charge 23](#_Toc469163125)

[4. User Interface Design 24](#_Toc469163126)

[4.1 Overview 24](#_Toc469163127)

[4.2 UX Diagrams 24](#_Toc469163128)

[5. Requirements Traceability 27](#_Toc469163129)

[6. References 31](#_Toc469163130)

# Introduction

## Purpose

The purpose of this document is to provide more details than the RASD, concerning the PowerEnJoy system.

It contains the architecture, the main components, diagrams shoving the user experience and the interaction between components and has also some algorithms referred to the key functions of the system.

The document is addressed to programmers and aims to be a helpful guide for the development of the system.

## Scope

PowerEnjoy is a car-sharing service to which the user can register and access using a mobile application. Driving license, ID card and payment details must be provided to be able to reserve and use cars.

The user informs the system when he gets close to the car he reserved or when he parks it. Discounts or extra charges may be calculated after the usage.

Another type of user, the operator, is charged of doing maintenance on cars that have problems.

Both users and operators interact with the main system sending requests, while the system has access to databases and an external payment system to perform needed actions and to get information concerning users, cars, or power stations where cars can be recharged.

A more detailed description of the system is contained in the RASD.

## Glossary

* **User:** the person registered to the system and allowed to access to its functions.
* **Operator:** a person with technical skills, that fixes car issues.
* **App:** short term used to define a mobile application.
* **Power Plug:** a column with one or more electricity socket where it is possible to charge the car.
* **Safe Area** (or Parking Area): a parking area with parking shared with all the other divers and not especially reserved to PowerEnjoy.
* **Special Parking Area** (or Power Station): a parking area reserved exclusively to PowerEnjoy cars where, for each parking space there is a Power Plug where it is possible to charge a car.
* **Car:** PowerEnjoy car.
* **Reservation:** the relation between a user and a car, that allows the user to start using the car. The reservation guarantees that no one else can reserve and use the reserved car till the end of the rental.
* **DB:** database, the collection of system data.
* **GUI:** Graphic User Interface, the interface that allows the user to interact with the system.
* **RASD:** Requirements Analysis and Specifications Document.
* **DAO:** Data Access Object.
* **DTO:** Data Transfer Object.
* **MVC:** Model-View-Controller, the pattern used for the development.
* **UX:** user experience.

## Reference Documents

The documents used as a reference to provide the design document are:

* Assignments AA 2016-2017.pdf
* Sample Design Deliverable Discussed on Nov. 2.pdf
* IEEE Standard for IT – System Design – Software Design Description
* Structure of the design document

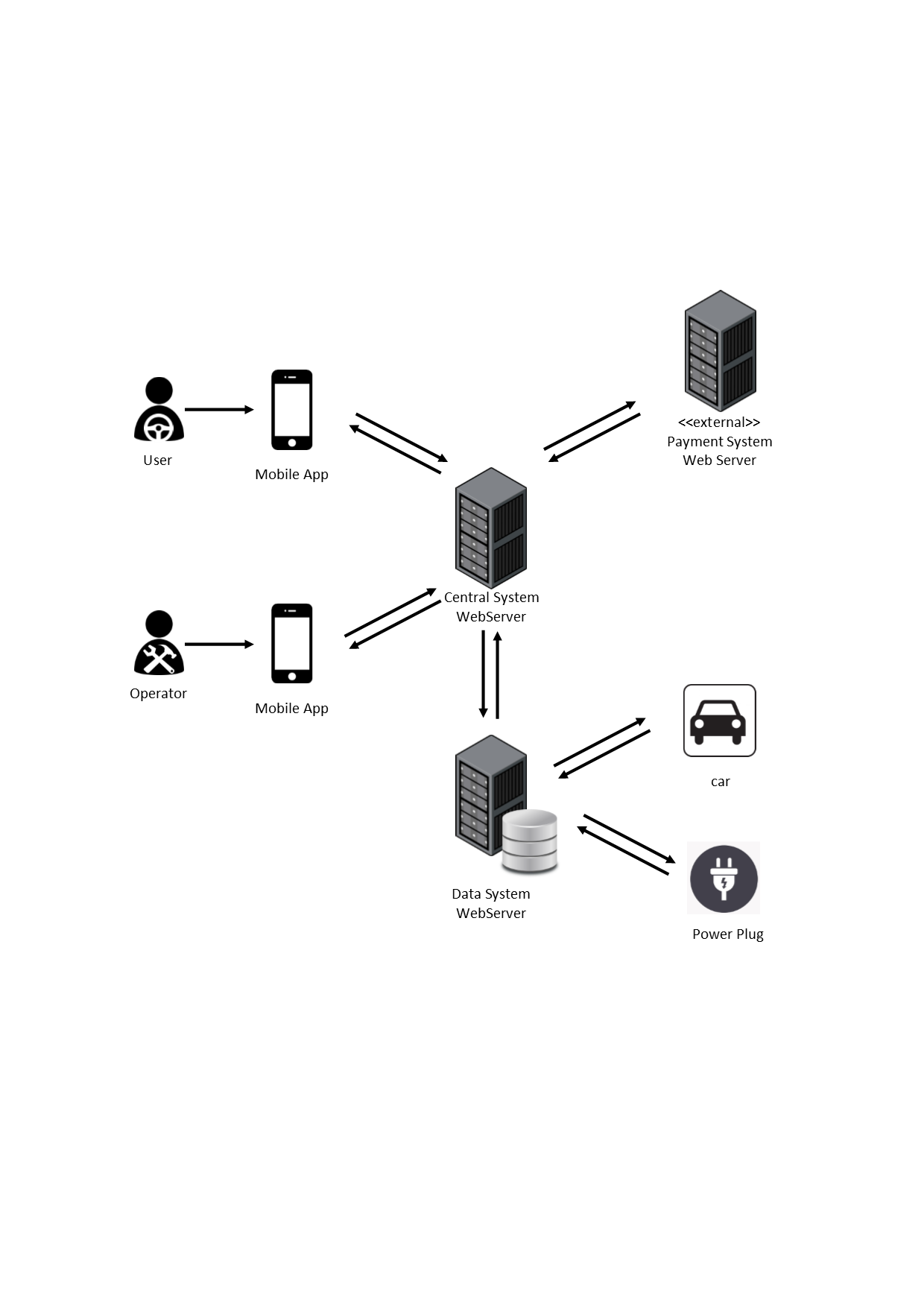
## Document Structure

The document is divided into the following chapters:

* **Introduction:** in this part, we introduce the design document, by defining the purpose and the scope of this document. A set of definitions is presented, to explain abbreviations and terms used in the document.
* **Architectural Design:** this section contains the detailed description of the architecture to be used for the deployment of the system. It contains the view of all the components with their description. Also, sequence diagrams are presented to show the interaction and the use of the various components.
* **Algorithm Design:** in this section, we describe the most relevant algorithms of the application, using pseudo code, to explicitly show critical points and provide a way to provide a valid solution.
* **User Interface Design:** this part of the document contains additional mock-ups for the user and operator applications interfaces. User experience diagrams are also provided, in order to show the relations between the screens of the app and the sequence of screen navigation.
* **Requirement Traceability:** this section explains which is the link between the goals specified in the RASD and the components defined in the Design Document.

# Architectural Design

## Overview



The PowerEnJoy system can be organised in a three-tier architecture.

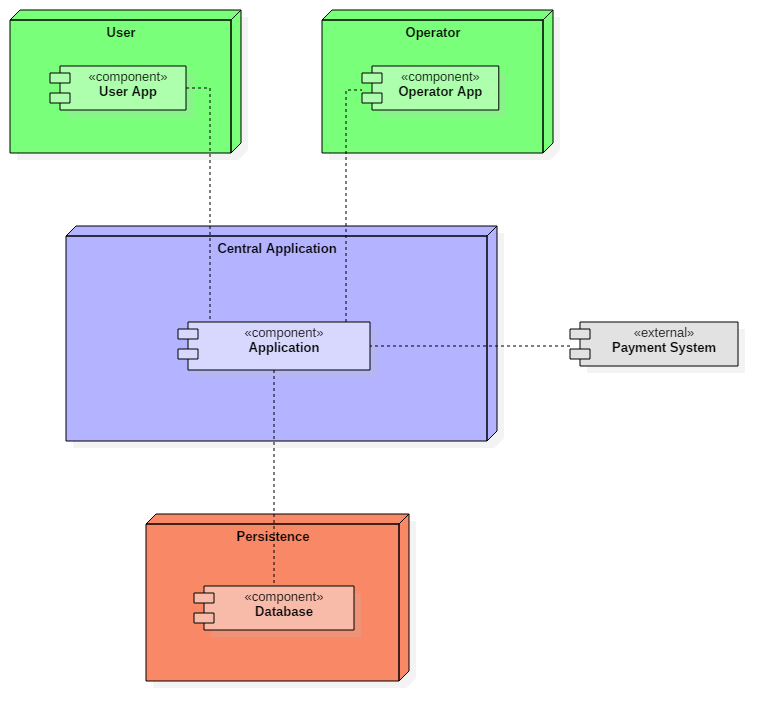
As shown in the figure, a GUI is provided to users and operators through their respective mobile applications.

The core of the system is identified by the Central System that contains the main logic of the application.

The Data System collects the data of all the cars, users and operators into its database and can interact with cars and power plugs to change their state or get their status.

The system interacts also with an external system that performs user payments.

## High level components and their interaction



The components of our system are:

* the User Application;
* the Operator Application;
* the Central Application, containing the main system logic;
* the Database.

The main application interacts with the database to get all information concerning users, operators, cars, parking areas and special parking areas. Most of the interactions concern the request or the update of car information, such as its position before reserving it or after parking it.

The database contains all the necessary details but gets updated by the system too, when the user finishes using the car.

The user and the operator have a mobile application installed on their mobile phones to interact with the system, for example, for reserving cars or for releasing them. The response of the system depends on the information provided by the database.

Note that, because of their different roles, the user and the operator have two different applications installed on their mobile phones.

The external payment system is another important component our system interacts with, to which our application sends requests for user payments. The response can affect user status (active/blocked/banned) that would be updated, in case of changes, on the database that contains user's information.

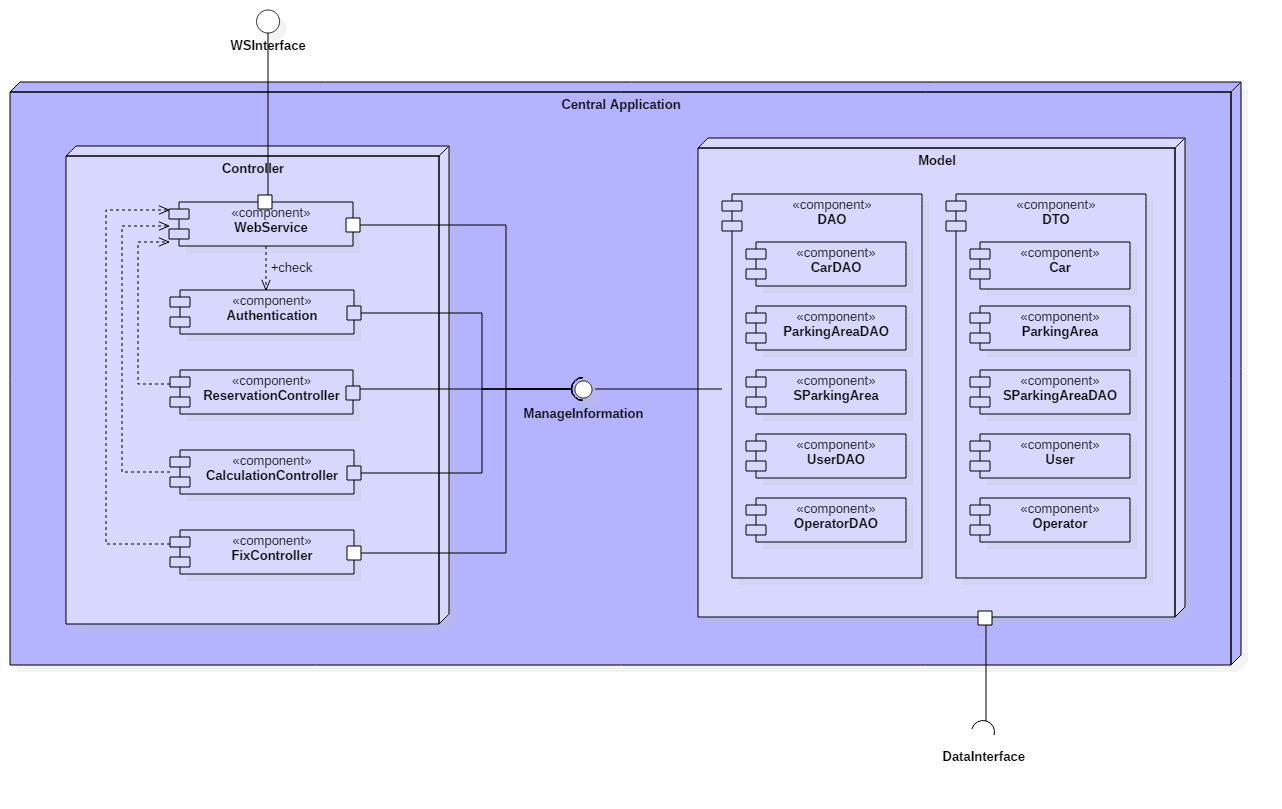
## Component view

In the following paragraphs, we will discuss in detail the components presented in the high-level components view.

## Central Application

The controller has components specialised in different operations. These are:

* the Web Service, that acts as a dispatcher for the incoming requests from user applications, checks the correct authentication and gets the results from the other controllers;
* the Authentication, which is used for the validation of the user login information;
* the Reservation Controller, that handles the requests for car reservation, providing the result to the web service, and manages all the other operations related to reservations;
* the Calculation Controller, called when needed to do all the calculations;
* the Fix Controller, used to manage the cars that are being repaired.



The interface Manage Information offers the set of methods to let the controller communicate with the model.

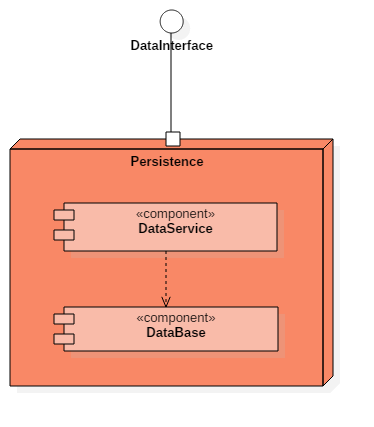
The model offers a direct way to contact the database and contains all the DAO and DTO components. Through all of them, the controller is able to communicate with the database in a roundabout way.

A DAO exists for car, parking area, special parking area, user, and operator.

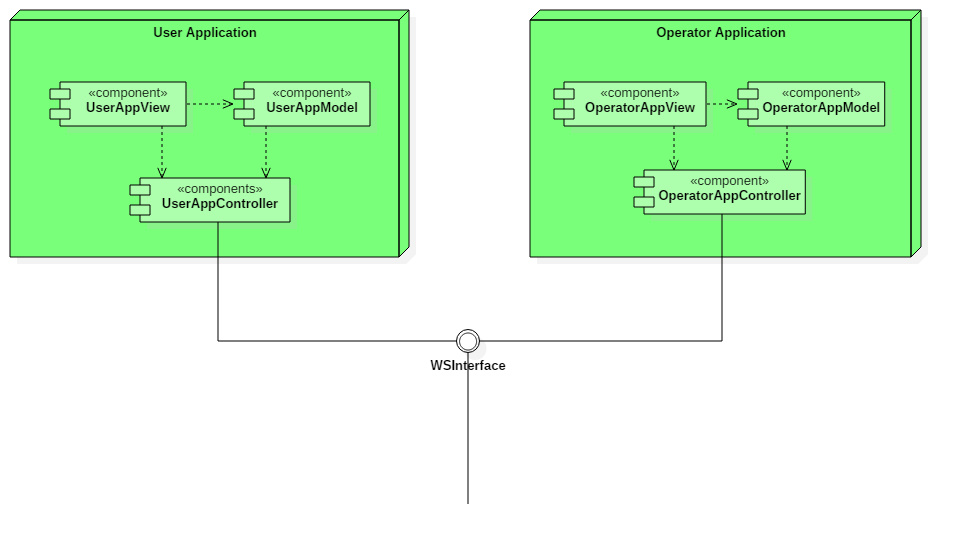
## Persistence

The persistence contains the database, where all the data of the system is stored.

The component used to interface with the DB is the Data Service; this one, makes the interaction with the system installed on the cars possible. The purpose is to get current information and provide them to the system, so that it is possible to update them after.



## Applications

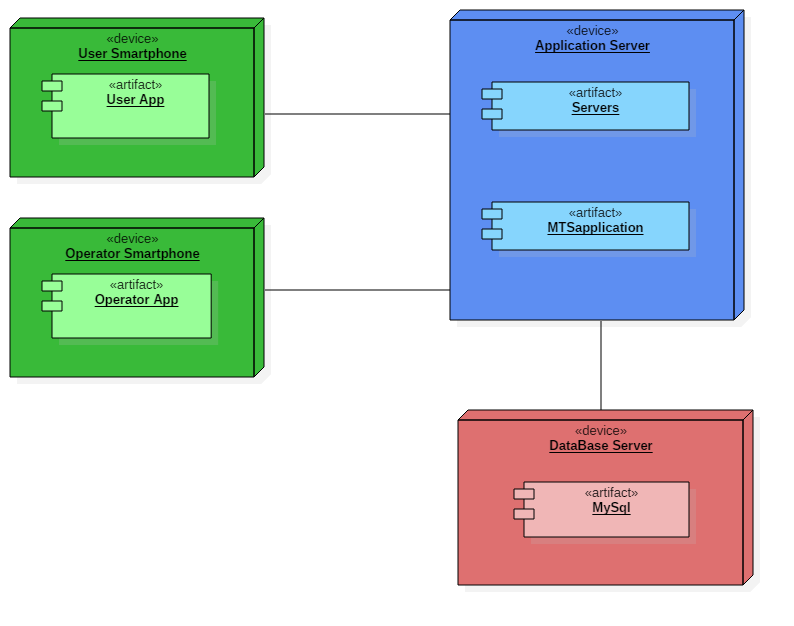


As already explained, the user and the operator have two different interfaces to the system. All their requests are passed to the Web Service of the controller through the WSInterface.

The applications are based on the MVC pattern.

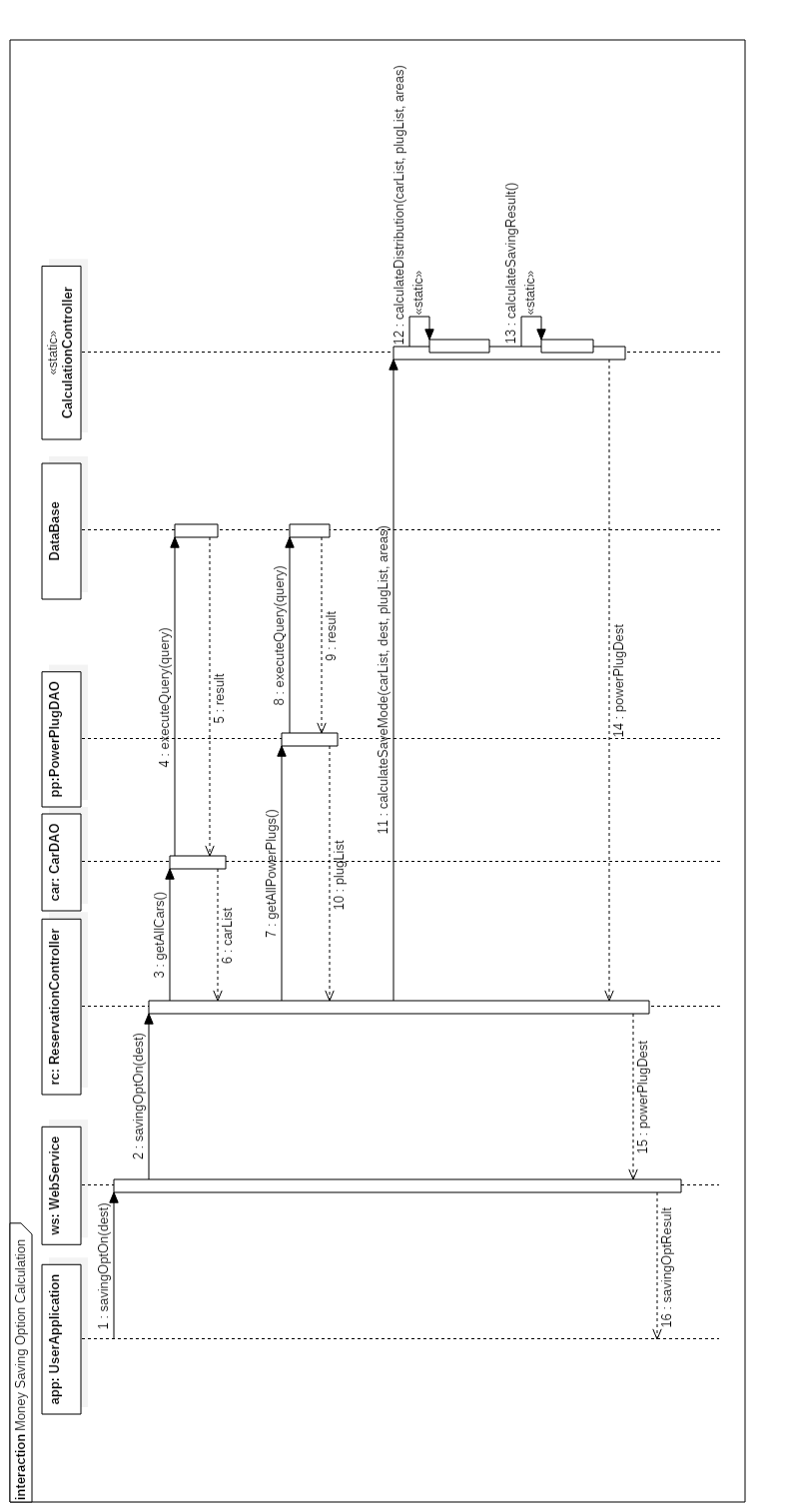
## Deployment view

The following figure shows the deployment view of our system.



## Runtime view

## Money Saving Option

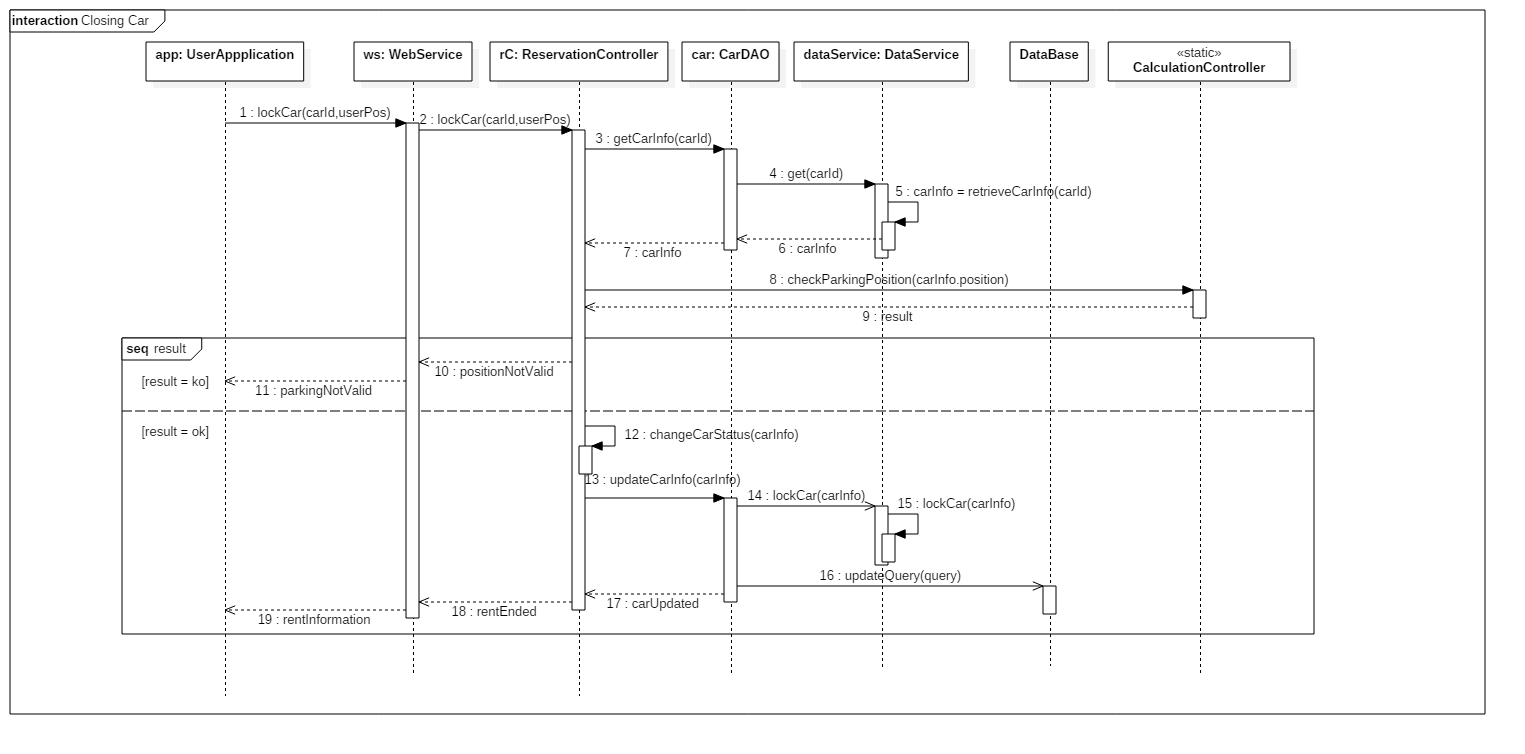


The money saving option can be enabled by the user after accessing the car. The request is sent to the Web Service, that dispatches it to the Reservation Controller. This one gets the list of all the cars, from a query executed by the CarDAO component. It also gets from the DB the information of all the power plugs, to view their availability.

After this, the Reservation Controller sends a request to the Calculation Controller, that calculates the distribution of cars and determines the destination power plug.

The result is given to the Web Service, that shows it to the user through the Application.

## User car lock request



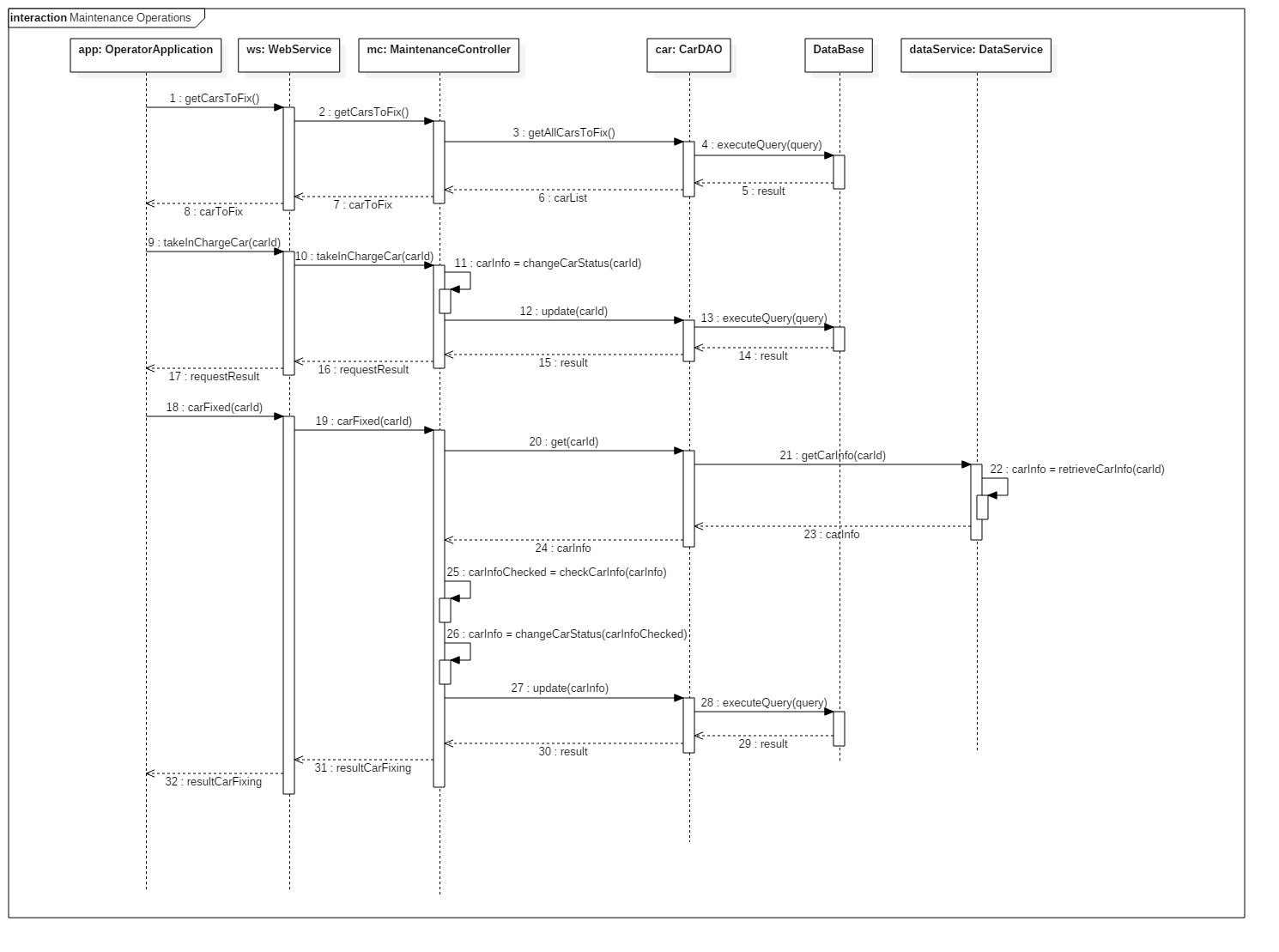
The user locks the car using his mobile application. Once the request sent, the Web Service catches it and asks the Reservation Controller to lock the car.

This action goes to the CarDAO, that gets status of the car by asking the Data Service; this one provides all the details so that the Reservation Controller can let the Calculation Controller validate the parking.

Two situations are possible:

* The parking is not valid: the car is not locked and the UserApplication receives the information
* The parking is valid: the Reservation Controller updates the status of the car and requires the CarDAO to lock the car. The request is sent to the DataService and then the CarDAO updates the information on the database. The rent ends.

## Operator does the maintenance



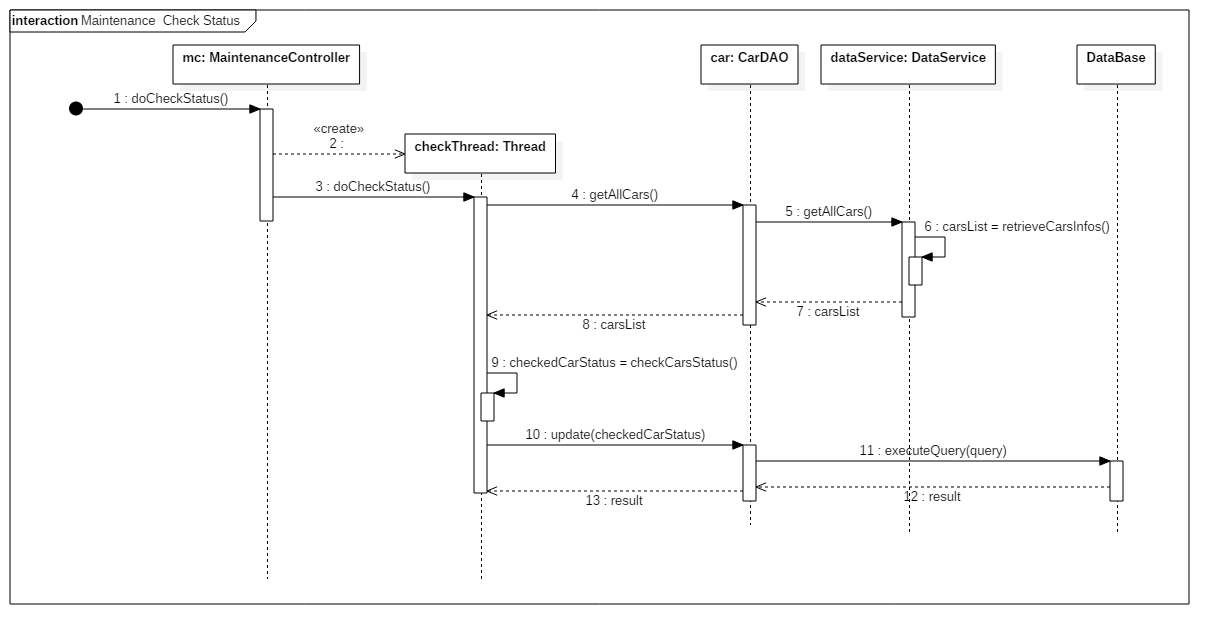
The figure shows the sequence of operations done for the maintenance of a car. The operator, through the mobile application, asks the list of cars that need to be fixed.

The WebService asks the MaintenanceController, that sends the request to the CarDAO. This one obtains this information from the Database.

The list is given back to the OperatorApplication, so that the operator can choose the car to fix. The WebService contacts the MaintenanceController, that updates car info and contacts the CarDAO.

Once the maintenance is finished, the app contacts the MaintenanceController through the WebService. It asks the DataService to check car info and changes the status of the car. After this operation, the Database get an update, with the new information of the car. The result obtained is given to the WebService, that sends it to the OperatorApplication.

## Checking status of the cars



This sequence diagram shows the operation done by the MaintenaceController to automatically check the status of the car and update the information on the Database.

The controller generates a thread that asks the CarDAO to have a check on all the cars; the DataService provides the required information and the checkThread sends an update request to the CarDAO, that contacts the Database.

At the end of the operation, the result is sent back to the thread.

## Component interfaces

Data system

DAO

WS

## Selected architectural styles and patterns

MVC:

Factory:

DAO:

Client server

## Other design decisions

# Algorithm Design

## List of available cars

When the user has accessed the app and has logged in, he is automatically redirected to the map view, displaying the available cars. The function that is used to prepare the map view of the area around the user is implemented as follows:

**function retrieveCarList(position, dist)** {

List carPins = new List();

float maxLat = position.getLatitude() + kmToDegrees(dist);

float minLat = position.getLatitude() - kmToDegrees(dist);

float maxLon = position.getLongitude() + kmToDegrees(dist);

float minLon = position.getLongitude() - kmToDegrees(dist);

array[] cars = System.getAvailableCars(

minLat, maxLat, minLon, maxLon);

foreach (car in cars) {

float lat = car.getLatitude();

float lon = car.getLongitude();

carPins.add(lat, lon);

}

return map;

}

The area to be shown on the map is supposed to cover around 2km from the current position of the user (or the address he enters).

The showPins() method is used to make the available cars visible on the GUI.

## Check distance and Unlock the car

The system uses a function to unlock the reserved car to the user. This only happens when the distance between the user and the car is under a certain interval.

**function unlockTheCar(user, car)** {

distance = checkDistanceBetween(user, car);

if (distance < 10) {

car.unlock();

return true;

} else {

return false;

}

}

A general function is defined and used to calculate the distance between two objects, e.g. for checking the distance between the car and the closest power station after parking other than for calculating distance between user and car.

**function checkDistanceBetween(elem1, elem2)** {

float long = degreesToKm(

abs(elem1.getLongitude() - elem2.getLongitude()));

float lat = degreesToKm(

abs(elem1.getLatitude() - elem2.getLatitude()));

distance = sqrt(long^2+lat^2); // Euclidean distance

return distance;

}

## Money saving option

This function is used to help the system find a free power plug to the user when he accepts to enable the Money saving option. The system must consider the destination address and the distribution of cars in the city to keep it balanced to provide the address of the chosen power station in which the user can park.

The city is divided into areas of the same size so the system can calculate the average of cars inside each of them, sort them by distance from the destination and then check power stations contained in each one of them till It finds the closest one having a free slot.

**function findFreeSlotForMoneySavingOption(user, addr)** {

Slot slot = null;

List areas = System.getParkingAreas();

Map carNumber = DB.countCarsInAreas();

float average = avg(carNumber);

List sortedAreas = areas.sortByDistance(

addr.getLatitude(), addr.getLongitude());

foreach (area in sortedAreas) {

List ps = area.getPowerStations();

foreach (station in ps) {

if (carNumber.get(area) < average

and station.getFreeSlots().length > 0) {

slot = station.reserveFreeSlot();

}

}

return slot;

}

## Validate the parking

When the user parks the car, the system needs to check if the car has been parked inside a valid parking area.

The following function does this check by calling the *isInside()* function which objective is to ensure that the coordinates of the car are contained in the area represented by the coordinates of the parking areas.

The function covers the case in which the system has more than one safe parking area.

**function validateParking(car)** {

float longitude = car.getLongitude();

float latitude = car.getLatitude();

List parkingAreas = System.getParkingAreas();

foreach(parkingArea in parkingAreas) {

bool canPark = isInside(parkingArea, carPosition)

if (canPark == true) {

return true;

}

}

return false;

}

## Calculate the final amount to charge

The following function is used to calculate the final amount of the ride. It considers all the possible discounts and fees, as described on the RASD.

**function calculateFinalAmount(car)** {

float timeUsed = car.getTripLength();

float passengerDiscount = 0,

batteryDiscount = 0,

plugDiscount = 0,

lowBatteryFee = 0;

float total = timeUsed \* System.pricePerMinute;

if (car.getPassengersNum() > 1) {

passengerDiscount = total \* 0.1;

}

if (car.getBatteryLevel() > 50) {

batteryDiscount = total \* 0.2;

}

if (car.isCharging() == true) {

plugDiscount = total \* 0.3;

}

if (car.getBatteryLevel() < 20

and checkDistanceBetween(findClosestStation

(car.getPosition()), car) > 3) {

lowBatteryFee = total \* 0.3;

}

float finalCharge = total – passengerDiscount +

– batteryDiscount – plugDiscount + lowBatteryFee;

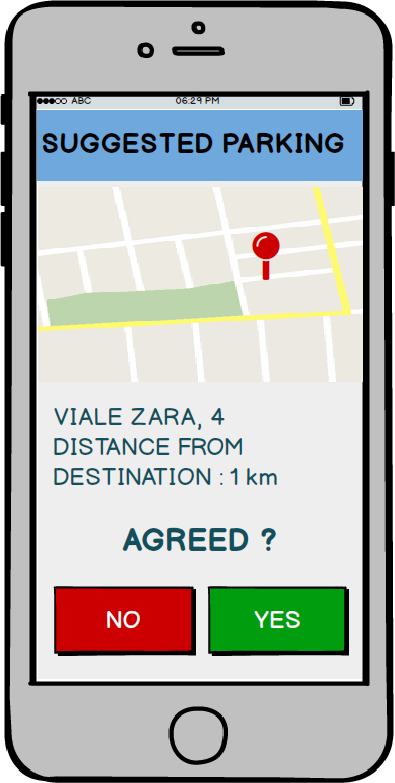
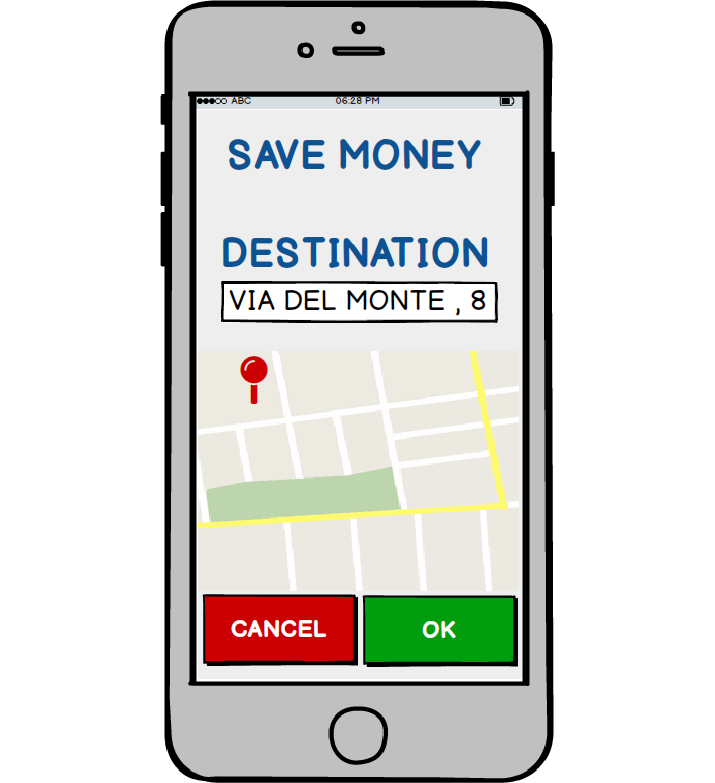
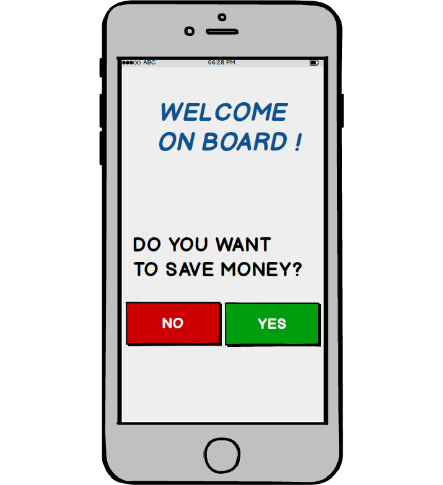
return finalCharge;

}

# User Interface Design

## Overview

The interface of the mobile applications has already been presented on the RASD, but we wish to add some other screens that we decided to add.



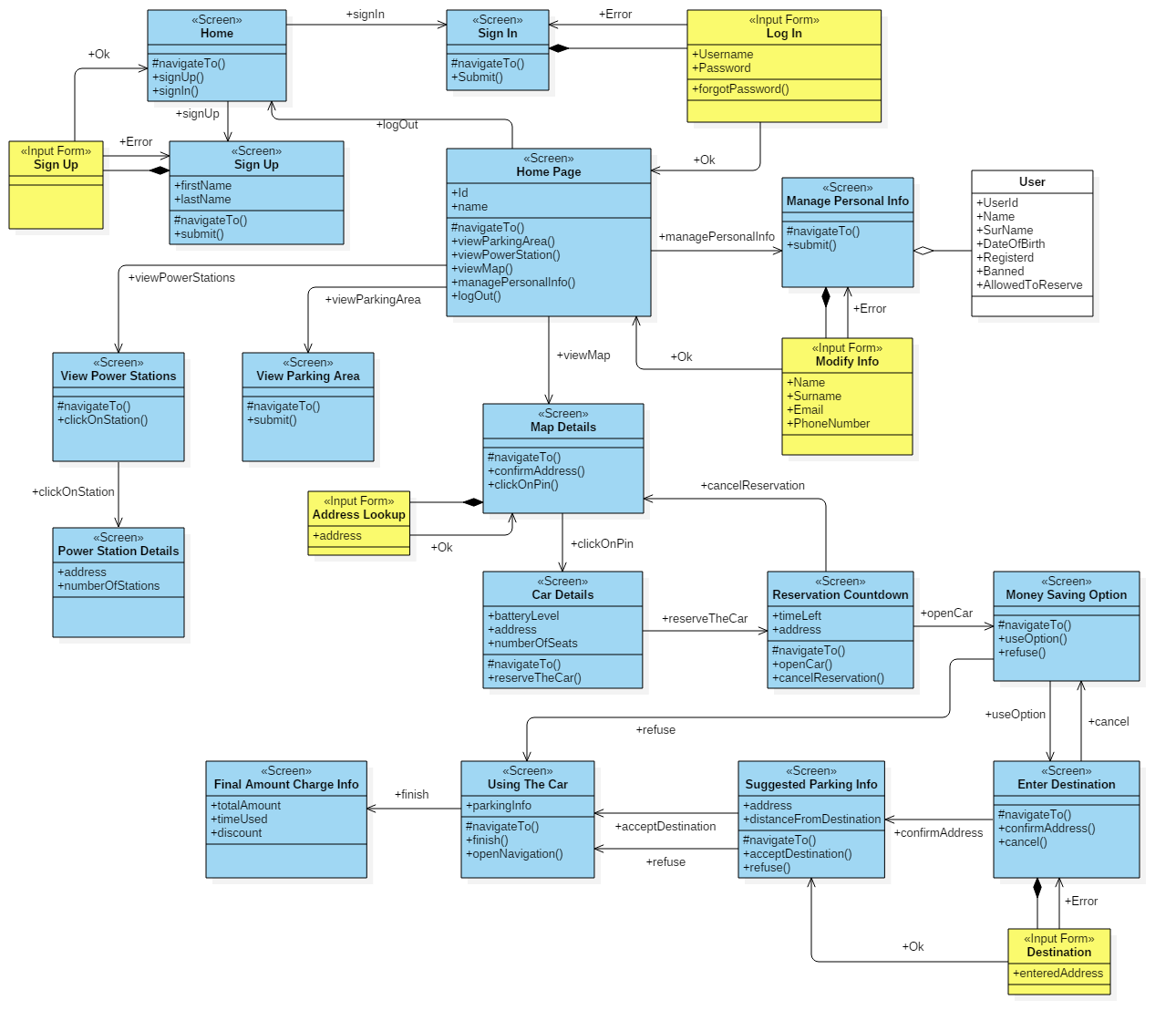
The pictures shown refer to the money saving option that can be enabled by the user once he opens the car. In this case, the destination is required, so that the system can calculate the closest free power plug, considering the distribution of cars in the city. The user will be able to accept or refuse the suggested parking and in case he accepts, the system will reserve the free power plug.

We decided to keep two different applications, one especially for the user and the other one for the operator, so that we can manage them separately and because we allow user registration but for the operators we assumed that they are already known to the system.

## UX Diagrams

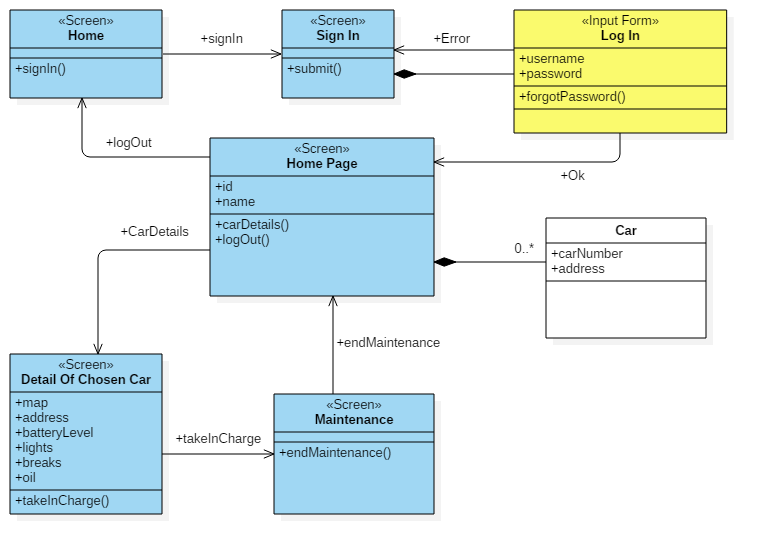
**User diagram**

The following diagram shows how user actions are performed and the sequence of the navigation between the screens.



**Operator diagram**

The following diagram shows the result of actions performed by the operator on the user interface.



# Requirements Traceability

Here we present the components that are involved in the fulfilment of the goals presented in the RASD.

**User goals**

[G1] The user should be able to register to the system.

* The UserAppController
* The WebService
* The Authentication component
* The user DAO and DTO
* The Database

[G2] Already registered users should be allowed to log in.

* The UserAppView
* The UserAppController
* The WebService
* The Authentication component
* The user DAO and DTO
* The Database

[G3] The user should be able to find available cars around him.

* The UserAppController
* The WebService
* The car DAO and DTO
* The Database
* The UserAppView

[G4] The user should see the battery level of a car before making a reservation.

* The UserAppController
* The WebService
* The car DAO and DTO
* The Database
* The UserAppView

[G5] The user should be able to reserve a car.

* The UserAppView
* The UserAppController
* The WebService
* The ReservationController

[G6] The user should be granted the access to the reserved car once he reaches it.

* The UserAppView
* The UserAppController
* The WebService
* The ReservationController

[G7] The user should be allowed to cancel a reservation.

* The UserAppView
* The UserAppController
* The WebService
* The ReservationController

[G8] The user should be able to access profile and payment method and make changes.

* The UserAppView
* The UserAppController
* The WebService
* The user DAO and DTO
* The Database

[G9] The user should be informed of the amount he has been charged of.

* The UserAppView
* The UserAppController
* The WebService
* The ReservationController
* The Calculationcontroller
* The car DAO and DTO

[G10] After the rental, the user should be able to inform the system that he is leaving the car.

* The UserAppView
* The UserAppController

[G11] The user should be able to see all the parking areas.

* The UserAppView
* The UserAppController
* The WebService
* The parking area DAO and DTO

[G12] The user should be able to see all the special parking areas.

* The UserAppView
* The UserAppController
* The WebService
* The special parking area DAO and DTO

**Operator goals**

[G13] Operators should be allowed to log in to the system.

* The OperatorAppView
* The OperatorAppController
* The WebService
* The Authentication component
* The operator DAO and DTO
* The Database

[G14] Operators should be allowed to see the list of cars that need maintenance and their details.

* The OperatorAppView
* The OperatorAppController
* The WebService
* The car DAO and DTO
* The Database

[G15] Operators should be allowed to notify the take in charge of a car and the end of the maintenance.

* The OperatorAppView
* The OperatorAppController
* The WebService
* The car DAO and DTO
* The MaintenanceController

# References