Software Engineering 2: "PowerEnJoy"
Design Document (V. 1.1)

Authors:
Sergio CAPRARA
Soheil GHANBARI
Erica TINTI

Milan, Italy
11/12/2016

Table of contents

1. Introduction

1.1 Revision History

Version	Date	Authors	Description
1.0	11/12/2016	S. Caprara, S. Ghanbari, E. Tinti	First release
1.1	07/02/2017	S. Caprara, E. Tinti	Corrections and updates

1.2 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 the main algorithms referred to the key functions of the system.

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

1.3 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.

1.4 Glossary and Acronyms

- **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.
- **GEB:** Green e-Box.

1.5 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
- RASD_PowerEnjoy_Caprara_Ghanbari_Tinti_v1.1.pdf
- Paper on the green move project.pdf

- Second paper on the green move project.pdf

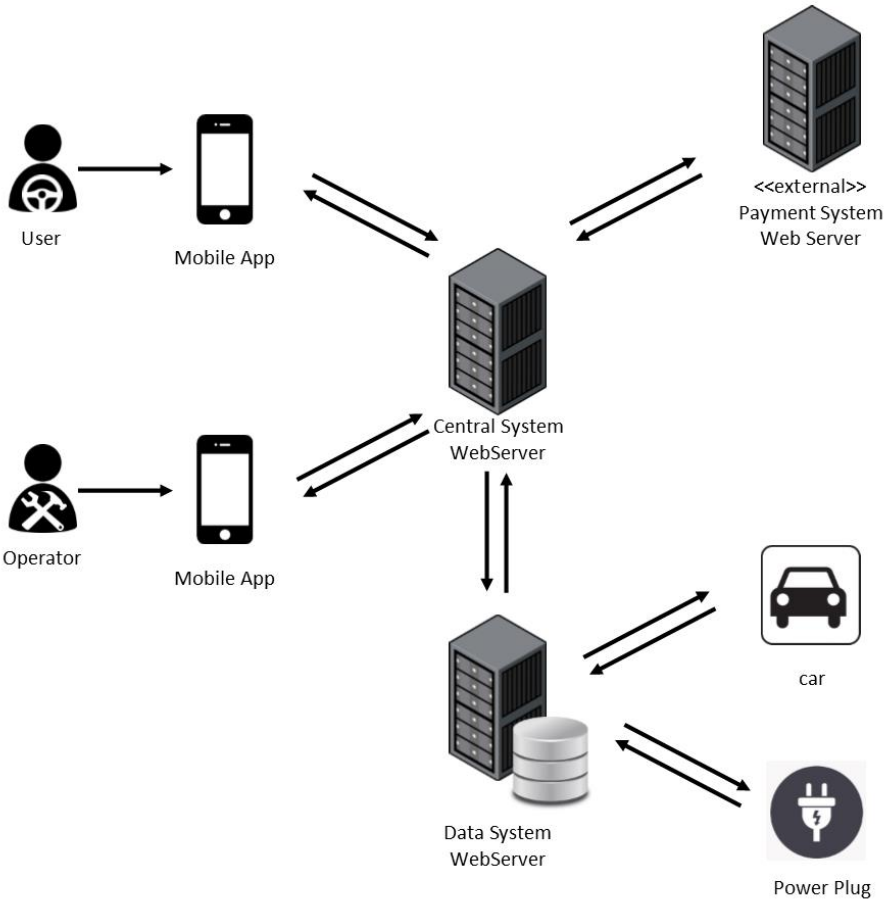
1.6 Document Structure

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.

2. Architectural Design

2.1 Overview

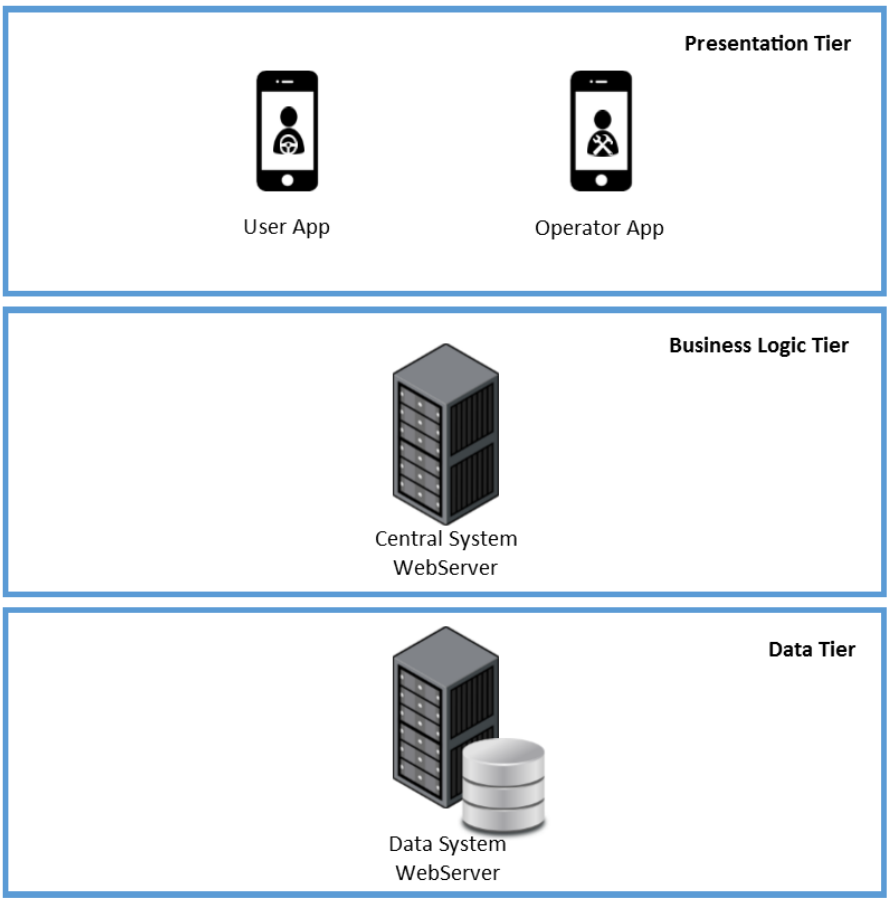


The core of the system is identified by the Central System that contains the main logic of the application and that communicates with the user devices, the Data System, and the external Payment System.

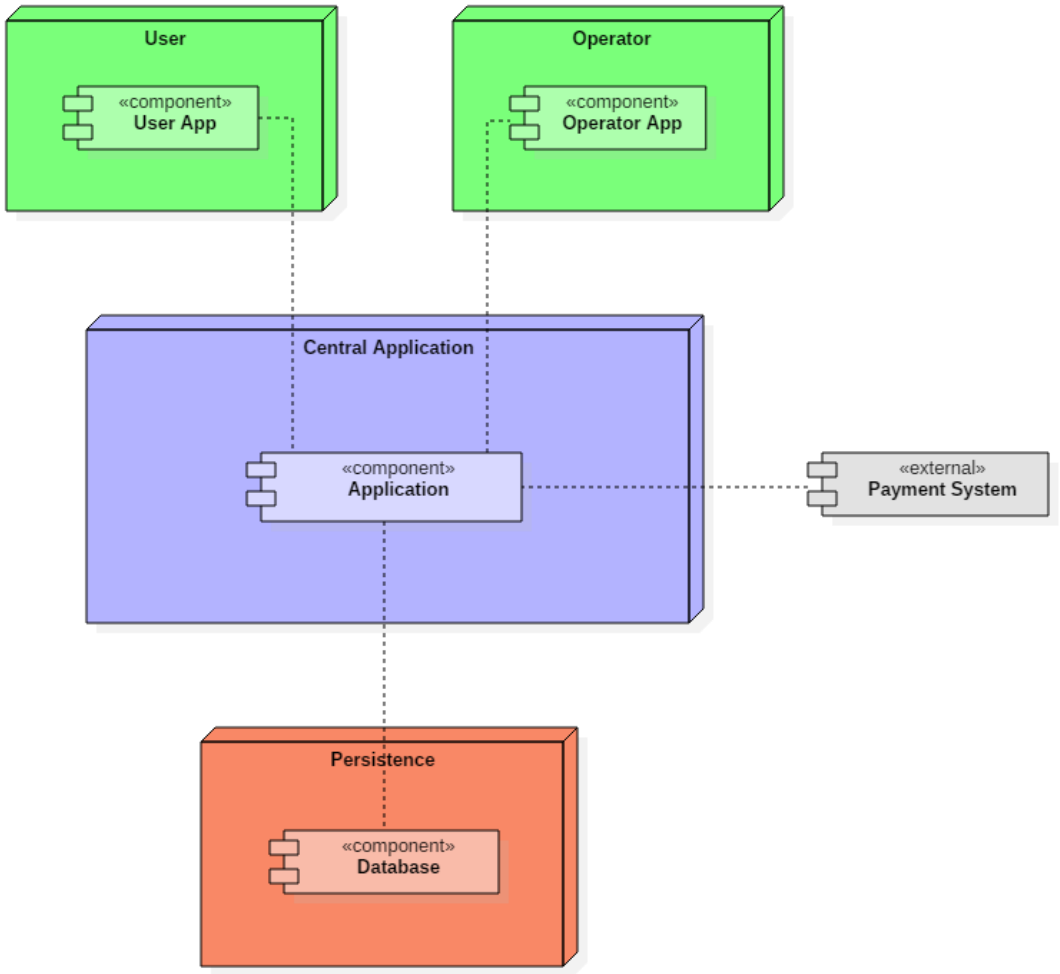
As shown in the figure, a GUI is provided to users and operators through their respective mobile applications.

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 or get their status.

The PowerEnJoy system can be organised in a three-tier architecture, as shown in the following figure.



2.2 High level components and their interaction



The main components of our system are:

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

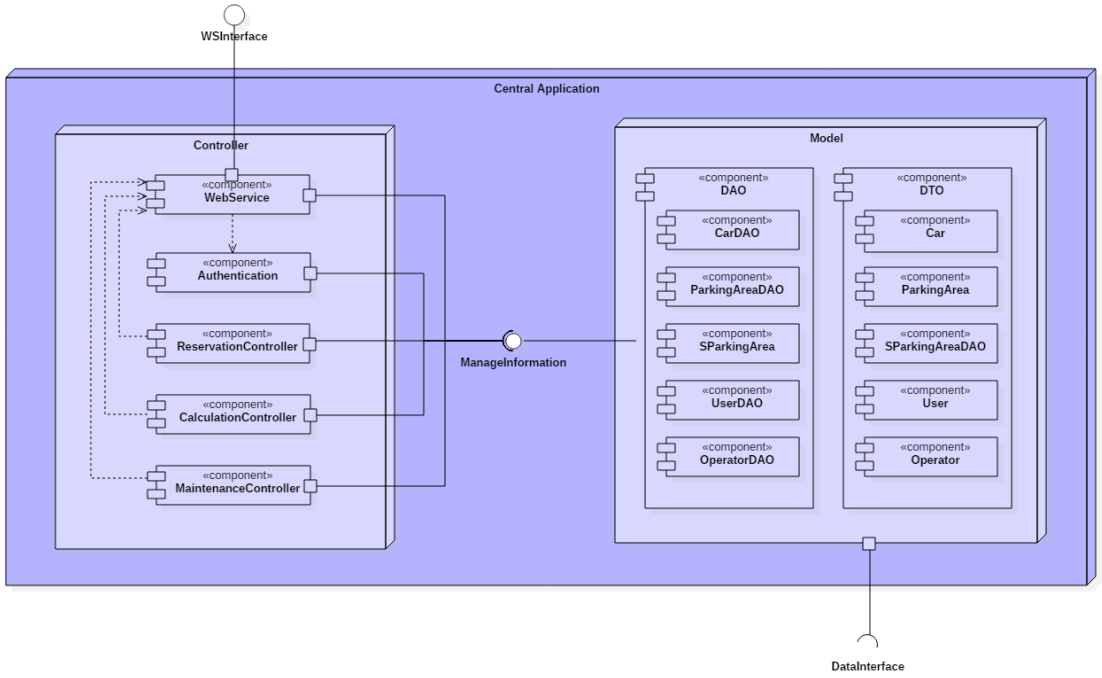
Note that, because of their different roles, the user and the operator have two different applications installed on their mobile phones.

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.

2.3 Component view

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

2.3.1 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 Maintenance Controller, used to manage the cars that are being repaired.

The ManageInformation interface provides the set of methods to let the controller communicate with the model.

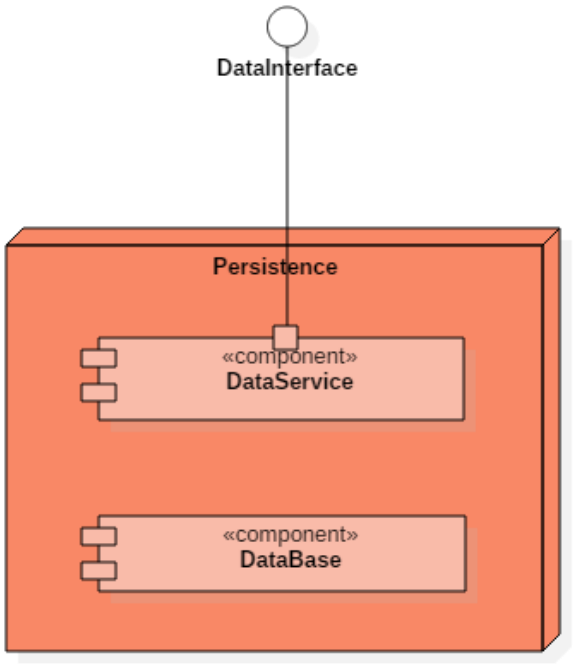
The model offers a direct way to contact the database and contains all the DAO and DTO components. Through all of them, the controller can communicate with the database in a roundabout way.

The DAOs exchange messages directly with the Database, while the DataInterface lets the model communicate with the DataService.

Database and DataService are both contained in the persistence node.

A DAO exists for all the elements that need to access the database or the DataService.

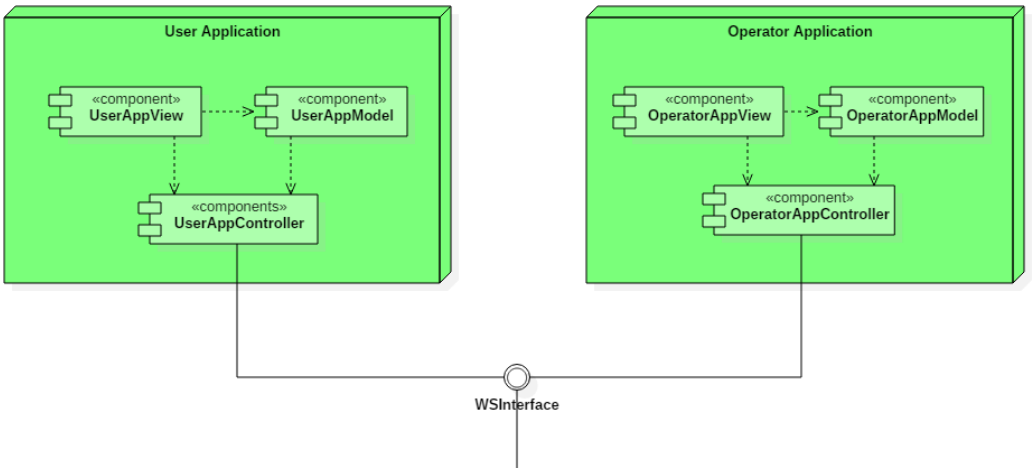
2.3.2 Persistence



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

The DataService provides a way to interact with the system installed on the physical cars and with power plugs. The purpose is to get current information about both and, for power plugs, also to set their status.

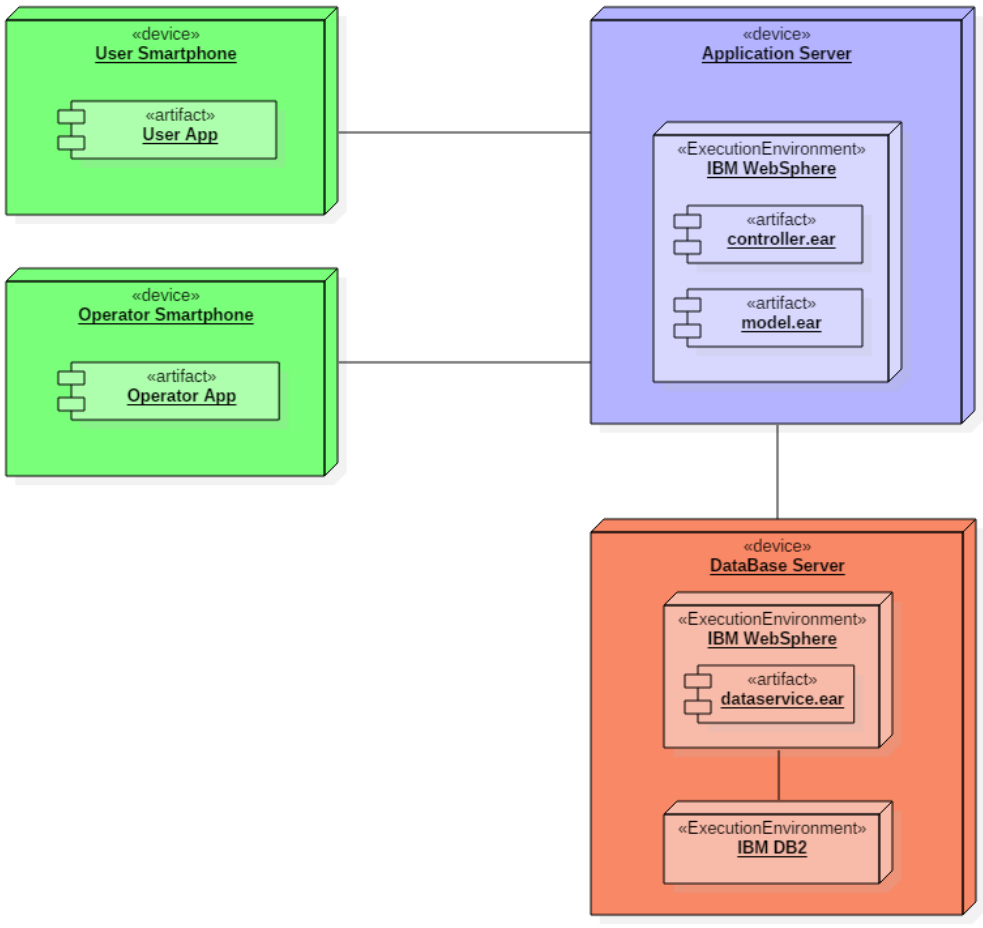
2.3.3 Applications



The user and the operator have two different applications to interface with the system. All their requests are sent to the Web Service of the controller through the WSInterface.

The applications are based on the MVC pattern, where the controller has no business logic, but only receives the data from the central application and manages the views.

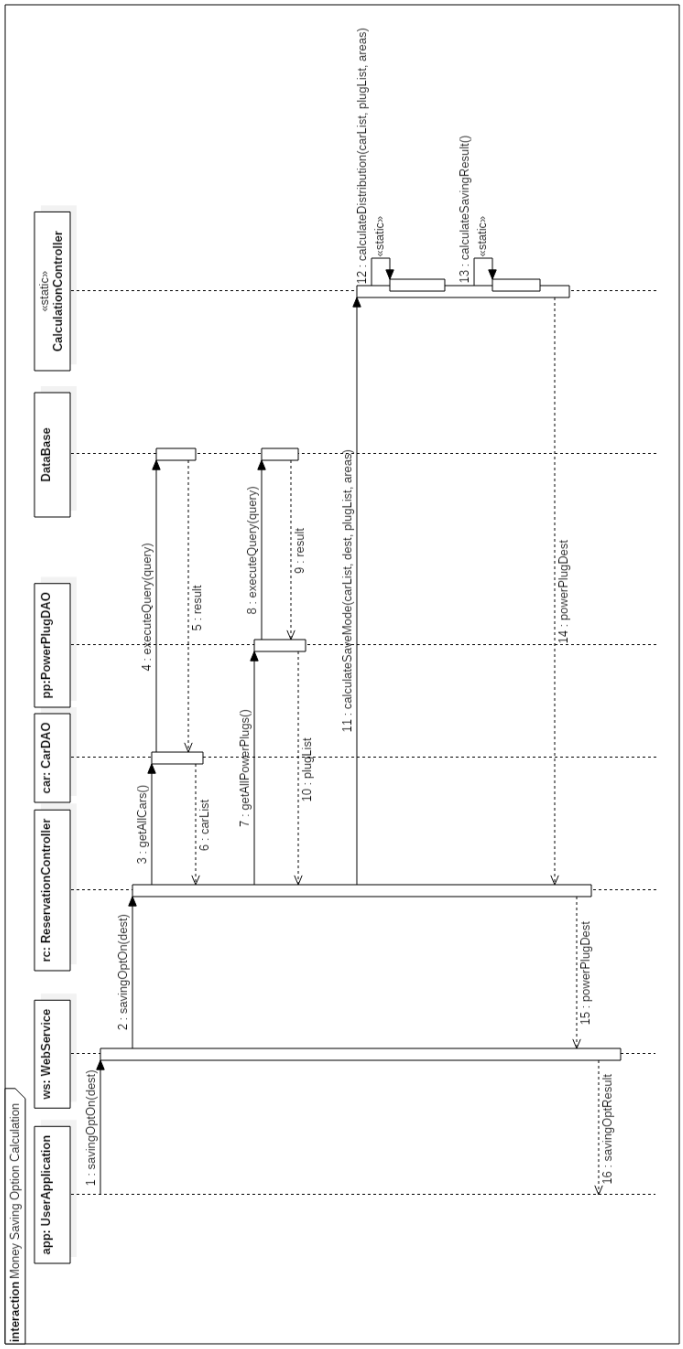
2.4 Deployment view



The figure shows the deployment view of our system.

2.5 Runtime view

2.5.1 Money Saving Option

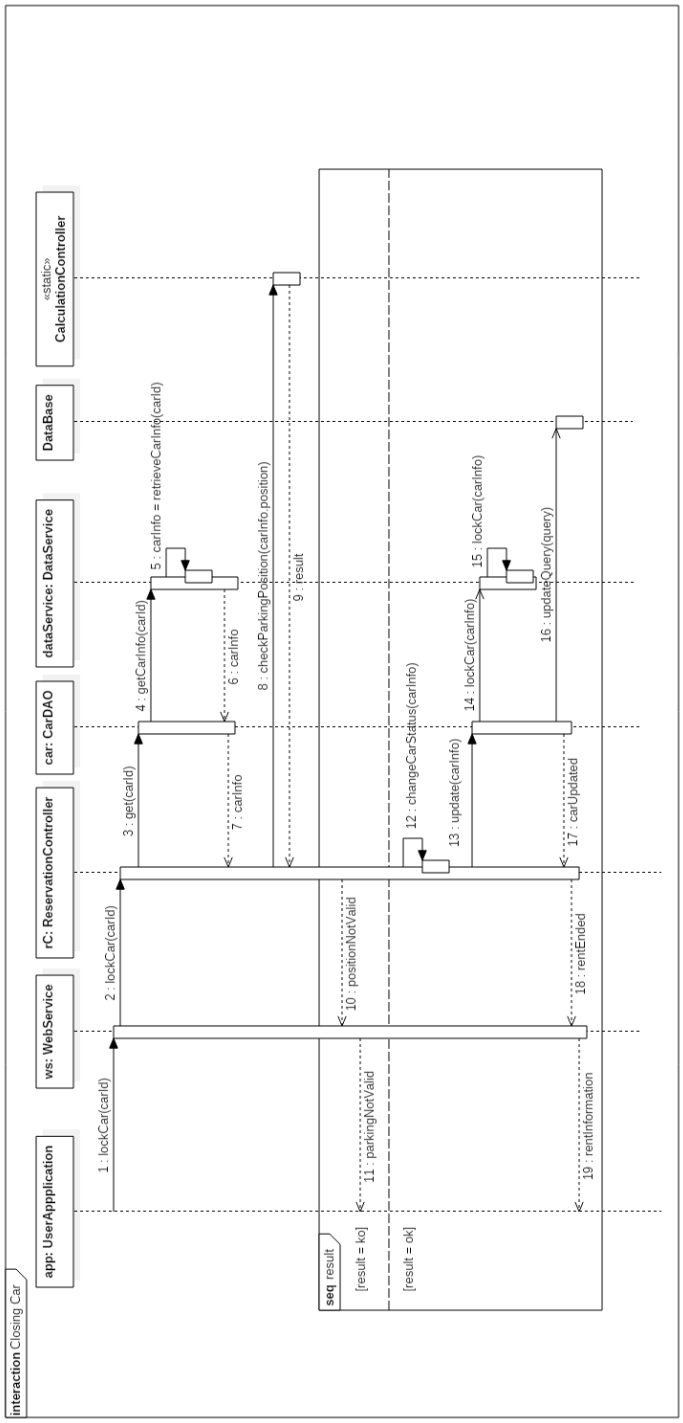


The money saving option can be enabled by the user after accessing the car. The request is sent to the Web Service, that dispatches it to the Reservation Controller. This one gets the list of all the cars, from a query executed by the CarDAO component. It also gets from the DataService the information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

2.5.2 User car lock request



The user locks the car using his mobile application. The request is sent to the Web Service that asks the Reservation Controller to perform the action.

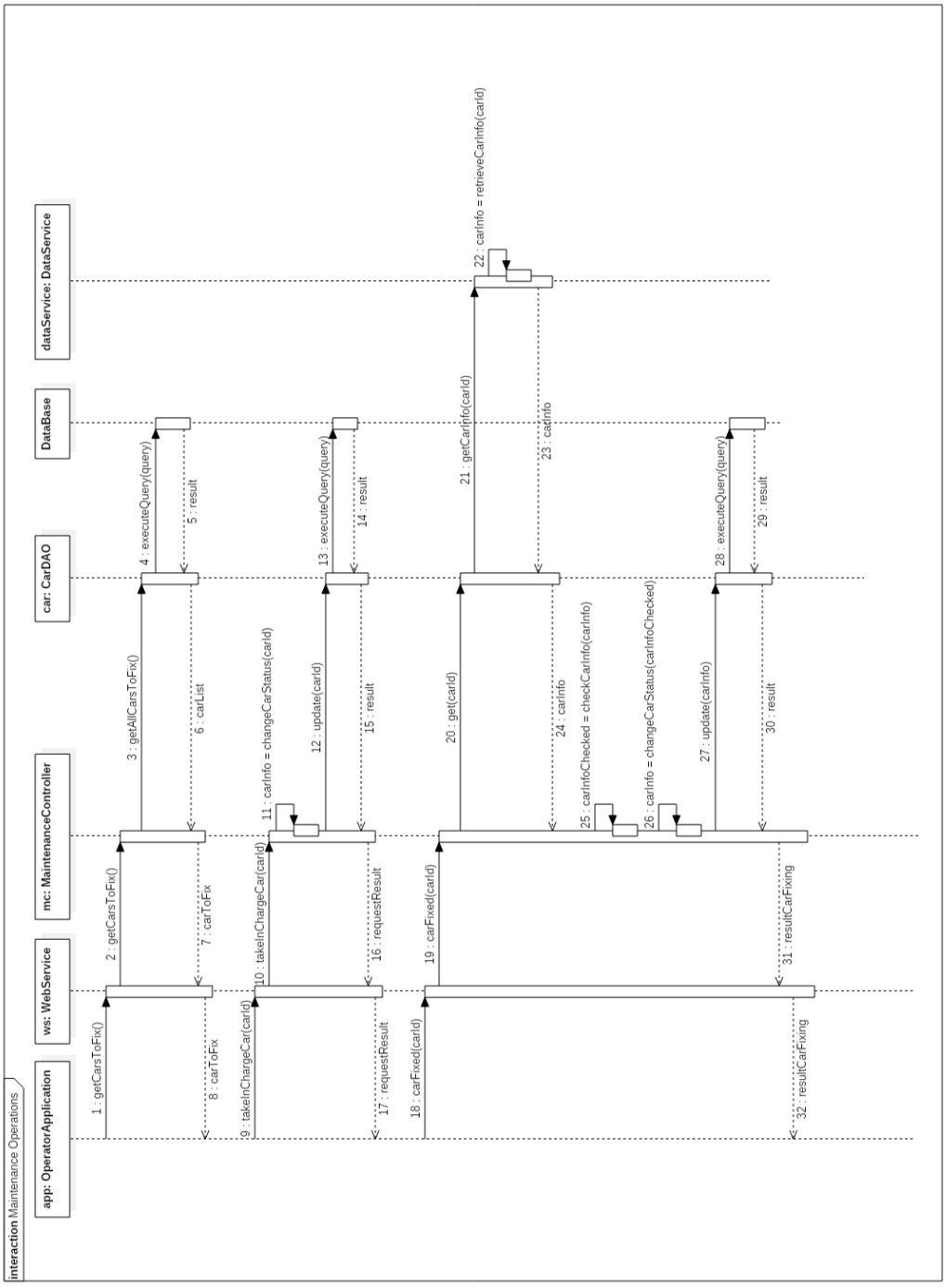
Before performing the lock, the Reservation controller sends a request to the CarDAO. It requests the position of the car to the DataService, that sends the information as a response.

Once the information obtained, the Calculation Controller receives the request to validate the parking and the response is sent back to the Reservation Controller.

Two situations are possible:

- The parking is not valid: the car is not locked and the UserApplication receives this 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.

2.5.3 Operator does the maintenance



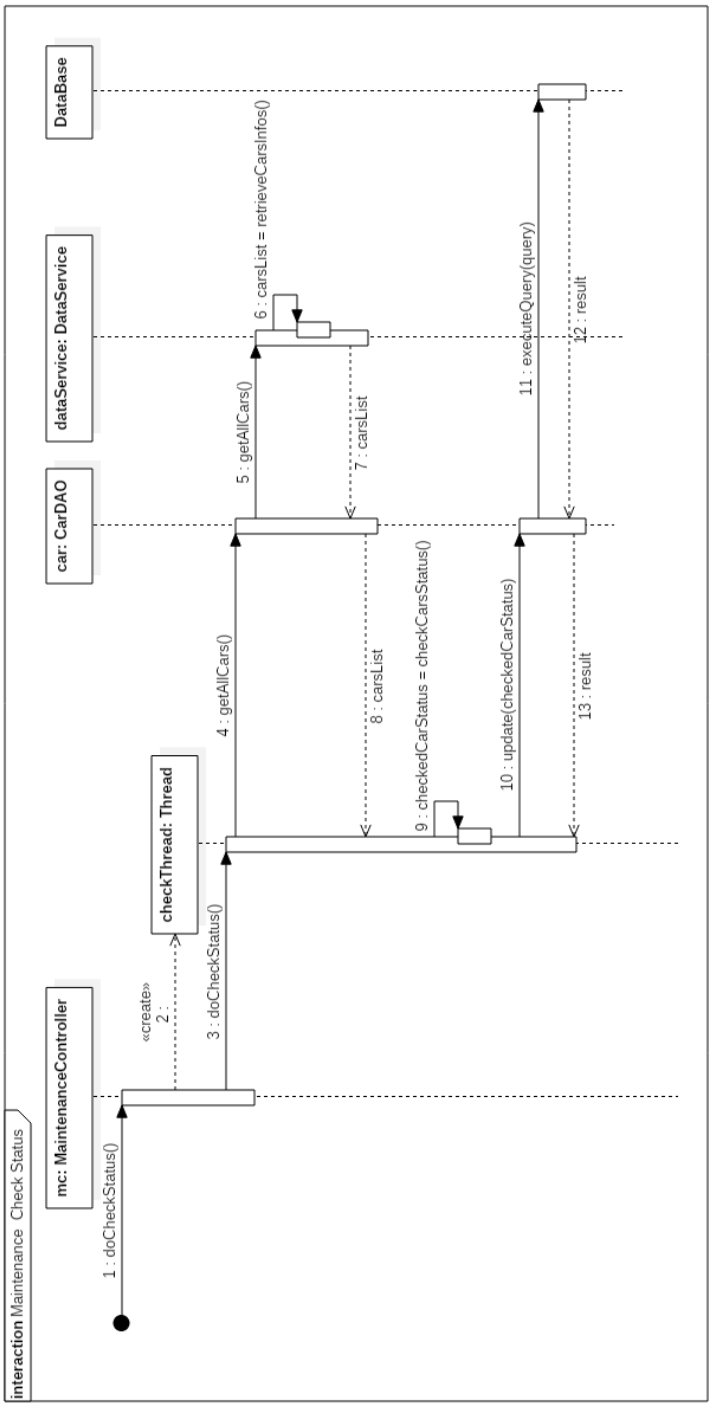
The figure shows the sequence of operations done for the maintenance of a car. The operator, through the mobile application, asks the list of cars that need to be fixed.

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, setting the “under maintenance” status, 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.

2.5.4 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 checks performed are about the Battery Level for plugged cars removing them from maintenance when the level is high enought, and moving them to maintenance when faults are found. 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, a feedback is sent back to the thread.

2.6 Component interfaces

The interfaces we use are described in the following paragraphs. These play an important role because they grant the communication between the components of our system.

2.6.1 WSInterface

This interface shows the methods through which the user and operator applications can send requests to the WebService that dispatches them to the other controllers. In the following table, we give a non-exhaustive list of methods to be defined.

Method	Description
signIn(user)	Used for user subscription, the parameter represents the user object containing all the information, such as first name, last name, ID card, Driving Licence, … The result of the call is the correct subscription or the error on some provided information.
logIn(user, pass)	Method provided to let users or operators access to their respective applications. As a result, if the data is valid, the user controller receives an acknowledgment. Otherwise, an error string is sent.

logOut(user)	Used to exit from user account. This only leads to a valid response.
reserveCar(user, carId)	Method used by the user to reserve a car using the mobile application. The input parameters are the user and the carId; the first one is used to track the user for the incoming request and the second one to refer to the car and access its details. The method will send a valid response when the car is free and in a usable status. Otherwise, a fail message will be sent.
savingOptOn(dest)	Used when the user has opened the car. This method has the destination as an input, because the logic behind needs to find a suitable charging station. The result of this call is the suggested parking for the car, when the system finds an available power plug.
finishRental(user, car)	Method performed when the user finishes using the car. The parameters are the user and the car. The second one is necessary to validate the parking and to check the status of the car. The first one contains information about the rental, such as the time, to give as a response the acknowledgment and the final amount charged to the user. The method may give a fail message when the parking is not validated or in case of payment error.
getCarsToFix()	This method is reserved to operators and returns the list of the cars that need maintenance.
takeInCharge(op, car)	Used to manage the maintenance by one of the operators. The first parameter is the operator, provides the necessary details about him; the second one is the car, and contains its id. The result may be the acknowledgment or the error (when the car has already been taken during the request time, for example).
carFixed(op, carId)	The method is useful to inform the system that the car is fixed. This contains the information

	about the operator and the carId, to check if the new status of the car is valid and to get all information about it. This method also recalls the methods necessary to update the status of the car on the database. The only response is the acknowledgment of the request.

2.6.2 ManageInformation

This interface provides the methods to be implemented in the DAO components. Default methods for the DAOs are the following:

Method	Description
add(obj)	Used for INSERT operations. The response is the result of the execution on the database.
get(id)	This method is used for SELECT queries. It provides the information of the requested element. The result contains object information.
getAll()	The method provides the list of all objects related to the related DAO, contained on the database.
update(obj)	Method used for UPDATE operations. The method gets the object to update with all its information, other than its identifier on the database. The response is the result of the update.
delete(id)	Used to DELETE from the database the object specified as a parameter, when the parameter is specified. Otherwise, this corresponds to a DELETE ALL on the database. As a response, the method gives the correct deletion or an error message.

The additional methods

- **getAllCars()**
- **getCarInfo(carId)**
- **getAllPowerPlugs()**
- **getPowerPlug(plugId)**
- **setPowerPlug(plugId)**

are necessary for the CarDAO and the PowerPlugDAO. Their only function is to call the respective methods provided by the DataInterface, that are described in the next paragraph.

2.6.3 DataInterface

The methods described in the following table are the ones used to send requests to the DataService, that communicates with the physical cars and power plugs, as previously specified.

Method	Description
getAllCars()	The method provides the list of all the cars.
getCarInfo(carId)	The input of this method is the carId. This is used to get the details of the specified car. The DataService obtains all the information and sends them in the response.
getAllPowerPlugs()	Method used to get the list of all the power plugs.
getPowerPlug(plugId)	This method is used to get the status of the power plug, whose id is specified as a parameter.
setPowerPlug(plugId)	This method is called when the application needs to set the status of the power plug. It refers to the case in which the money saving option is enabled. When correctly reserved, the response is the acknowledgment of the request. Otherwise, a fail message is sent.

2.7 Selected architectural styles and patterns

2.7.1 Architectural styles

Our System will implement a Client-Server architecture, where the Client is represented by the users and the operators' mobile applications, and the Server is represented by the Central Application and the Persistent component (**$2.1**). This allows us to have all the logic of the application and the data centralized and under control and to have a Thin-Client whose only task is to make requests to the server and to show the response in a proper way.

The communication between the mobile applications and the WebService is done through JSON.

The Server-side application will use Java EE 7 as a programming language.

The Central Application will run on IBM WebSphere Application Server (Liberty profile) that is a flexible application server that fully supports Java EE 7 and that also allows the integration with Open Source software.

IBM WebSphere Application Server (Liberty profile) will also run on the Database Server, together with an IBM DB2 DBMS.

As described in the component view, the controller has access to data through DAO Components. Each DAO component directly connects to the Database by JDBC.

All the cars are equipped with a Green e-Box (GEB), a device that provides a hardware/software interface to easily interact with the car. See **$1.4** for details on this.

2.7.2 Design Patterns

Model-View-Controller (MVC)

In our system, we use the Model-View-Controller pattern that allows us to separate the data contained in the model from the view, that shows data and takes inputs, and from the controller, that owns the logic and manages data and views.

In our case, the View is represented by the mobile applications and the Model is composed of the DAO and DTO components. We have also defined a Controller component that interacts with the Model and communicates with the View by getting and sending information.

DAO

The DAO pattern is used to separate the data access mechanism from the application logic. The main element to implement this pattern is the Data Access Object (DAO), that is an object or an interface that provides access to a database or any other data source, and the Data Transfer Object (DTO), that is used by the DAO to contain data.

In our project, we use DAO Pattern to hide all the details of data storage from the rest of the application. The DAO provides an interface, not only to access the DataBase, but also to access the physical cars and power plugs' information through the DataInterface.

Using this pattern, it will be easier in future to change the DBMS, the structure of the Database or the DataService, without changing the Controller component.

3. Algorithm Design

3.1 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).

3.2 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;
}
```

3.3 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) //weighted average;
  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;
      }
    }
  return null;
}
```

3.4 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;
}
```

3.5 Calculate the final amount to charge

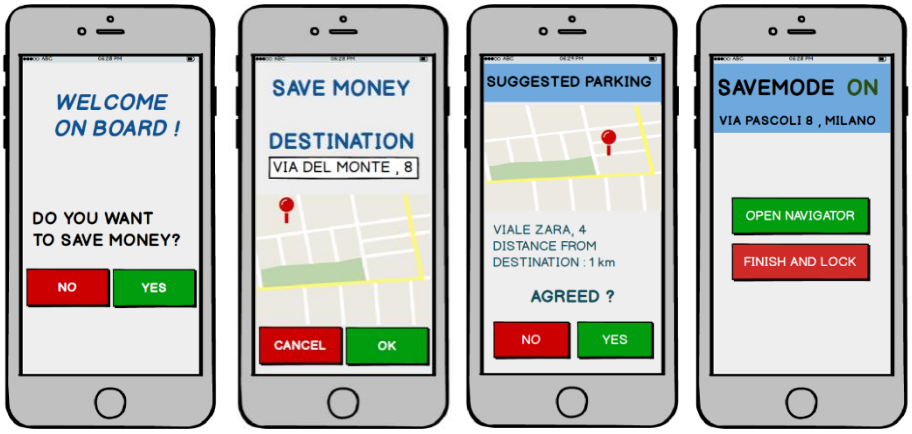
The following function is used to calculate the final amount of the ride. It considers all the possible discounts and fees, as described on the RASD.

```
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;
}
```

4. User Interface Design

4.1 Overview

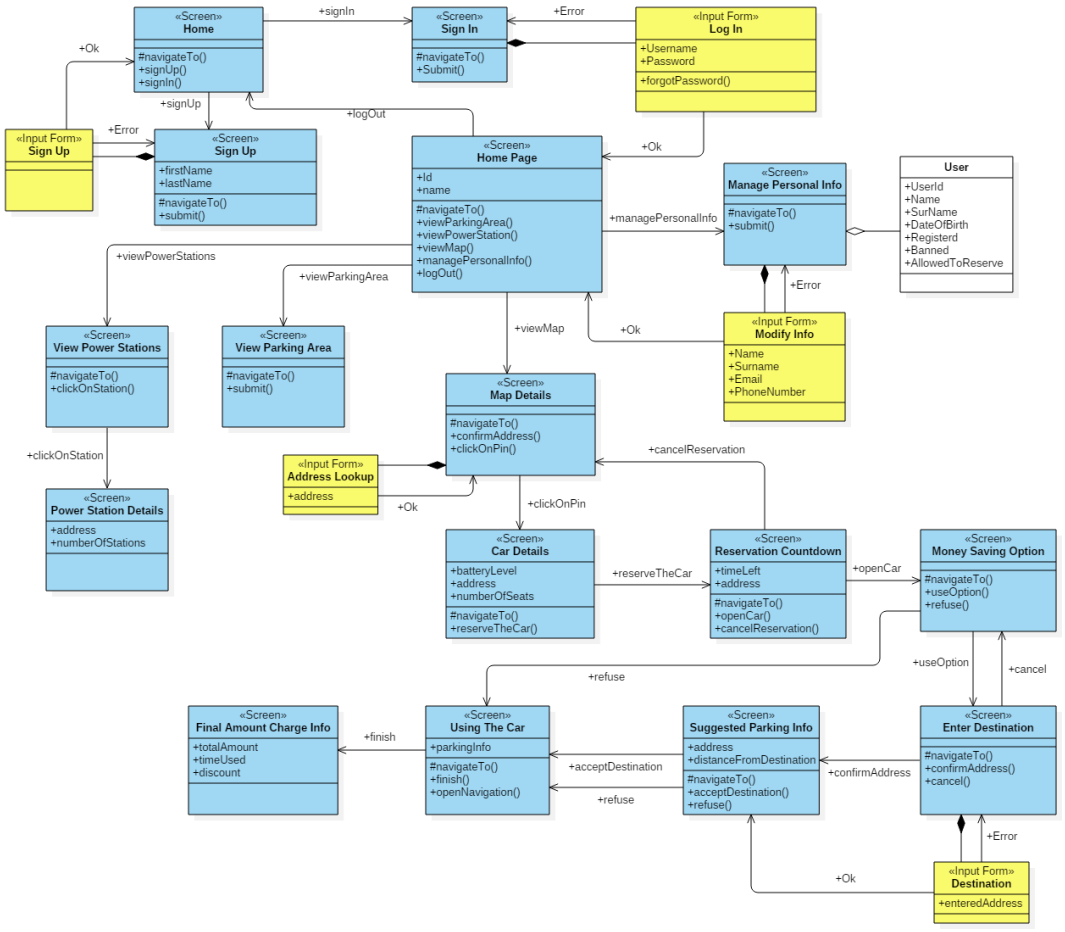
The interface of the mobile applications has already been presented on the RASD, but we wish to add some other screens that we decided to add.



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.

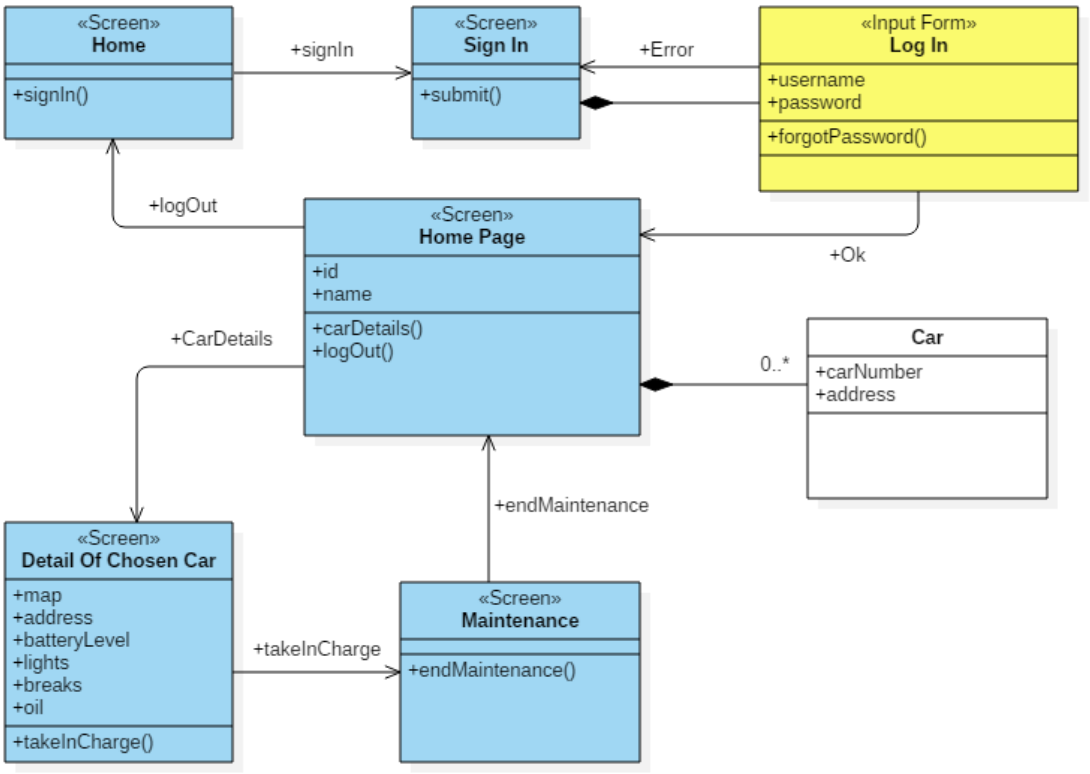
4.2 UX Diagrams

User diagram



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

Operator diagram



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

5. 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.

- UserAppView
- UserAppController
- WebService
- Authentication component
- User DAO and DTO
- Database

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

- UserAppView
- UserAppController
- WebService
- Authentication component
- User DAO and DTO
- Database

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

- UserAppController
- WebService
- ReservationController
- Car DAO and DTO
- Database
- UserAppView

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

- UserAppController
- WebService

- ReservationController
- Car DAO and DTO
- Database
- UserAppView

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

- UserAppView
- UserAppController
- WebService
- ReservationController
- Car DAO and DTO
- Database

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

- UserAppView
- UserAppController
- WebService
- ReservationController
- Car DAO and DTO
- DataService
- Database

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

- UserAppView
- UserAppController
- WebService
- ReservationController
- Car DAO and DTO
- Database

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

- UserAppView
- UserAppController
- WebService
- User DAO and DTO
- Database

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

- UserAppView
- UserAppController
- WebService
- ReservationController
- CalculationController

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

- UserAppView
- UserAppController

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

- UserAppView
- UserAppController
- WebService
- Parking Area DAO and DTO

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

- UserAppView
- UserAppController
- WebService
- Special Parking Area DAO and DTO

[G13] User should be blocked if there is a problem in the payment.

- WebService

- ReservationController
- User DAO and DTO

[G14] The user should be banned if the payment is refused during a 30 days period.

- WebService
- ReservationController
- User DAO and DTO

Operator goals

[G15] Operators should be allowed to log in to the system.

- OperatorAppView
- OperatorAppController
- WebService
- Authentication component
- Operator DAO and DTO
- Database

[G16] Operators should be allowed to see the list of cars that need maintenance and their details.

- OperatorAppView
- OperatorAppController
- WebService
- MaintenanceController
- Car DAO and DTO
- Database

[G17] Operators should be allowed to notify the take in charge of a car and the end of the maintenance.

- OperatorAppView
- OperatorAppController
- WebService
- MaintenanceController

- Car DAO and DTO
- DataService
- Database

6. References

Used tools

- Microsoft Word
- StarUML

Hours of work

For the document, each one of us has worked around 32 hours.