POLITECNICO
MILANO 1863

AA 2016-2017

Software Engineering 2 project: PowerEnJoy
Design Document (DD)

Matteo Franchi - 807046
Luca Guida - 878001
Alessandro Lavelli - 875788

Document version 1.2

Contents

1 Introduction

1.1 Revision History

- Version 1.0 (11 Dec 2016)
- Version 1.1 (13 Jan 2017)
  - added further details about Telematic Control Unit and Car Controller in *Physical architecture diagram*, *High level component diagram, Car services component diagram, 2.8 Other design decisions;*
  - added *ManageRental* interface to *Rental Module*;
  - some *Control* objects in BCE diagrams were renamed in order to make more evident the underlying mapping between *Control* elements and components of the system.
- Version 1.2 (22 Jan 2017)
  - replaced *JSP* with *Java servlet* in paragraph *Recommended implementation* (section *2.4*) for better compatibility with *Java Persistence API*

1.2 Purpose

The purpose of this document is to provide an overall guidance to the architecture of the software product and therefore it is primarily addressed to the software development team.

1.3 Scope

The product aims to function as the digital backbone of the services provided by the Power Enjoy car-sharing company, which will include:

- Search and reservation of available electric cars
- Browse of previous rent sessions
- Unlock/Lock of rented cars during the ride
- Safe and simple payment options
- 24/7 customer support availability

1.4 Definitions, acronyms, abbreviations

- **AJP**: Apache JServ Protocol
- **API**: Application Programming Interface
- **BCE**: Business Controller Entity

- **DD**: Design Document
- **JDBC**: Java DataBase Connectivity
- **JPA**: Java Persistence API
- **JSP**: Java Server Pages
- **RASD**: Requirements Analysis and Specifications Document
- **RMI**: Remote Method Invocation
- **UX**: User Experience

1.5 Document conventions

In component diagrams of section *2.3 Component View* the dependency relation between components is specified using two different graphical notations in order to make the schemes more readable: the first one directly connects the required interface with the provided one using a solid line, while the second one connects components via a dashed arrow. The two notations are equivalent.

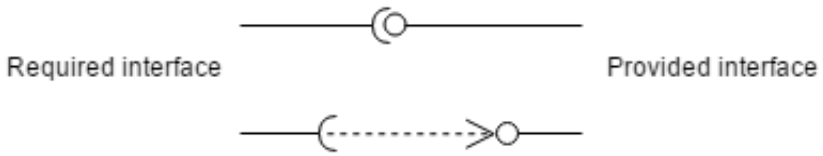

Figure 1: Dependency relation

1.6 Reference documents

- *Project goal, schedule, and rules (Assignments AA 2016-2017.pdf)*
- *Requirements Analysis and Specification Document*, version 1.2
- *Design Patterns* by Jason McDonald
  dzone.com/refcardz/design-patterns
- *Haversine formula*
  - *rosettacode.org* (www.rosettacode.org/wiki/Haversine_formula)
  - *ryanduell.com* (www.ryanduell.com/2012/12/determining-the-distance-between-two-geographic-points)
- *Google Maps Distance Matrix API* documentation
  developers.google.com/maps/documentation/distance-matrix/start

- *Java client library for Google Maps API Web Services* documentation
  github.com/googlemaps/google-maps-services-java
- *UML component and deployment diagram*
  uml-diagrams.org

1.7 Document structure

- **1 Introduction**

  This section introduces the design document. It explains the utility of the project, text conventions and the framework of the document.

- **2 Architectural design**

  This section illustrates the main components of the system and the relationships between them, providing information about their operating principles and deployment. This section will also focus on the main architectural styles and patterns adopted in the design of the system.

- **3 Algorithm design**

  This section describes the most critical parts of the platform in terms of algorithms. Java-like code is used to present the most important algorithmic aspects, which will be described without the least relevant implementation details.

- **4 User Interface design**

  This section presents mockups and further details about the User Interface using UX and BCE diagrams.

- **5 Requirements traceability**

  This section associates the decisions taken in the RASD with the ones taken in this DD.

2 Architectural design

2.1 Overview

This chapter illustrates the system architecture from both the physical and the logical standpoint:

- section *2.2 High level components and their interactions* provides details about physical components of the system;
- section *2.3 Component view* gives a global view of the logical components of the application and how they communicate;
- section *2.4 Deployment view* describes the tier division of the system and shows the components that must be deployed in order to run the applications properly;
- section *2.5 Runtime view* provides a series of examples of how components actually communicate with each other with the use of Sequence Diagrams;
- section *2.6 Component interfaces* focuses on the interfaces between the modules of the system;
- section *2.7 Selected architectural styles and patterns* explains the architectural choices taken during the design of the system in terms of patterns and paradigms;
- section *2.8 Other design decisions* introduces other relevant design choices which were taken while designing the platform.

2.2 High level components and their interactions

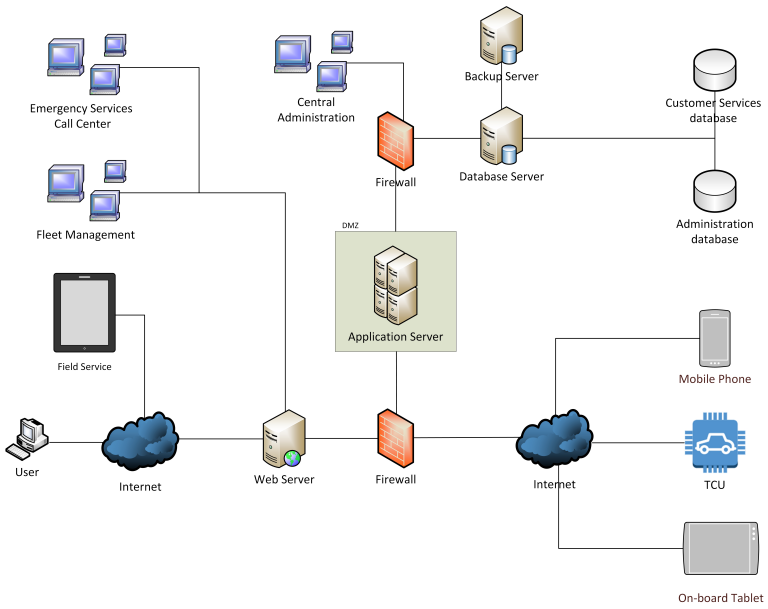

Figure 2: Physical architecture diagram

The general architecture of the system can be streamlined into the 3 logic layers: Presentation, Application and Data. The presentation layer can be divided into the client tier and the web tier. While employees and web browser users can respectively access the system's functionalities and the official company website through the Web Server, customers logged into the mobile application can communicate with the Application Server (which is located in a "demilitarized zone") directly. On top of that, the Database Server periodically stores critical data in the Backup Server. Keep in mind that both the Central Administration and the Administration Database are implemented through a third-party ERP solution, therefore no specifics will be provided for those in the following pages.

2.3 Component view

High level component diagram

The following diagrams show the main components of the system and the interfaces through which they interact.

- The client side is made of three components which refer to the employee web services, the customer mobile application, customer web services and the on-board tablet application.
- The server side has several components: in order to get more comprehensibility they are gathered into four subsystems.
  - The *employee web services* component will support the fleet management performed by the employees;
  - the *customer web services* will provide access to static contents of the website such as rules, tutorials and other information about the car sharing service.
    This part of the system will not be further analyzed in other sections of the document;
  - the *customer mobile services* provides the interfaces to rent a car, to manage personal profiles and to leave a feedback;
  - finally, the *car services* component contains the interfaces used by the tablet application on the car and its Telematic Control Unit.

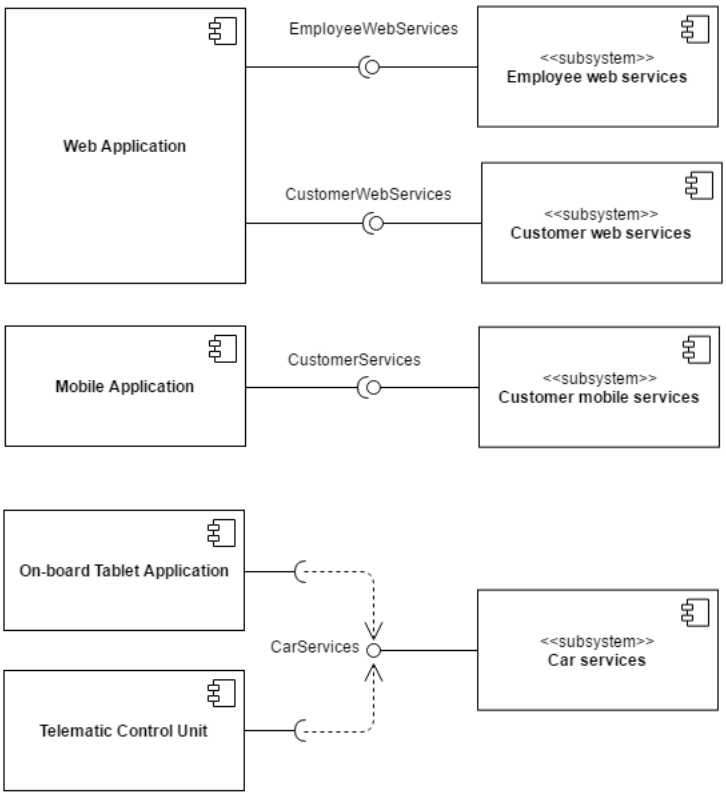

Figure 3: High level component diagram

Customer mobile services projection

Customer mobile services subsystem contains three components: *Feedback Module*, *Rental Module* and *Account Manger*. These components provide to the customer mobile application the following interfaces: *feedback*, *rental history*, *book session* and *profile management*. In order to fulfill their goals these components need to communicate with the DBMS, Maps API, payment gateway and SMS gateway. It also needs ManageCar interface, provided by *CarController* component, to unlock the car.

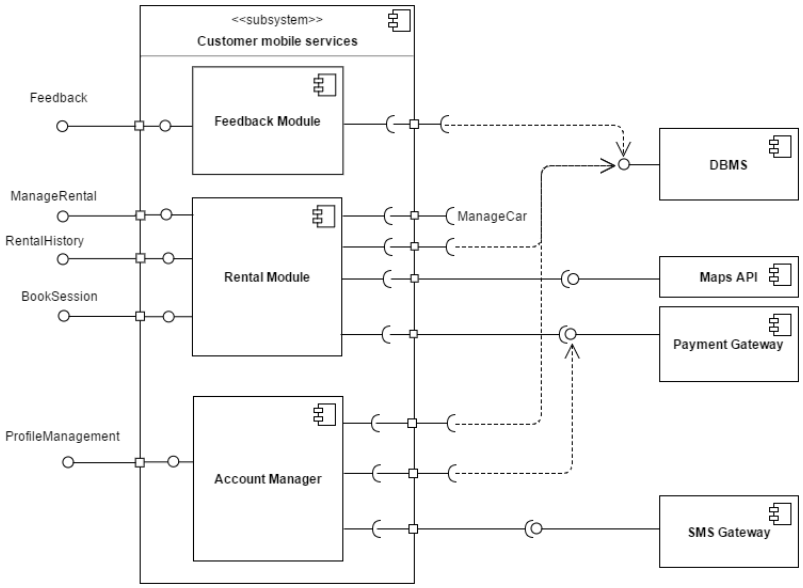

Figure 4: Customer mobile services projection

Car services projection

In the car services subsystem there are three main components: navigation controller, car controller and SOS call module. Trivially the navigation controller and the SOS Call Module provide, respectively, the interfaces to navigate, manage the rent session and perform a SOS call, while the car controller achieves all the functional requirements concerning the management of the car state, lock-/unlock of the vehicle and the end of the ongoing rent session. The Emergency Service Manager component mainly consists of an external VoIP API used by the SOS Call Module.

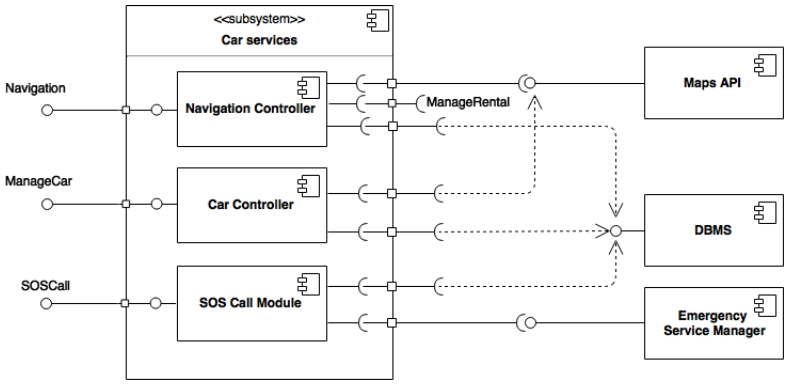

Figure 5: Car services projection

Employee web services projection

This subsystem communicates with the web application used by the employees for the fleet management and for incoming SOS calls. The fleet management employee needs the interface ManageCar in order to update the state of the car and unlock it remotely. Moreover, field agents use this subsystem through the web browser of their devices in order to perform maintenance activities on the fleet and other tasks.

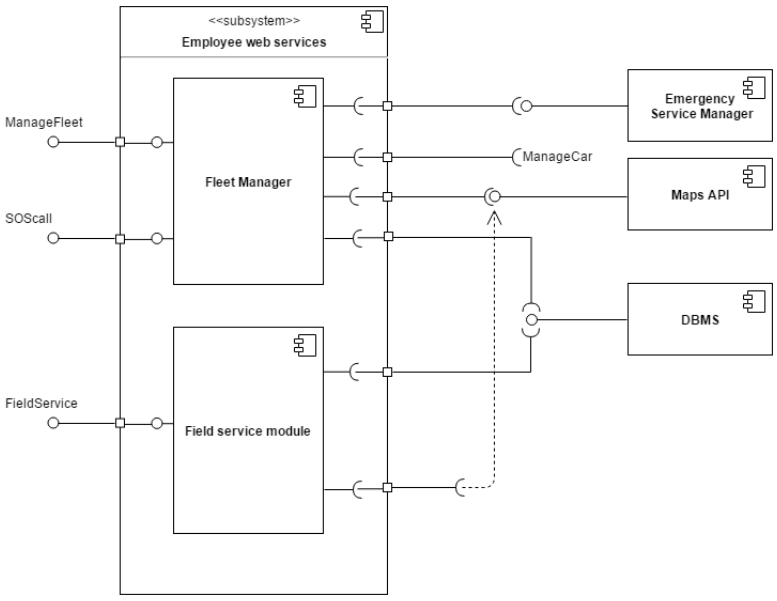

Figure 6: Employee web services projection

Entity–relationship diagram

The following diagram provides a graphical representation of the conceptual model of the database.

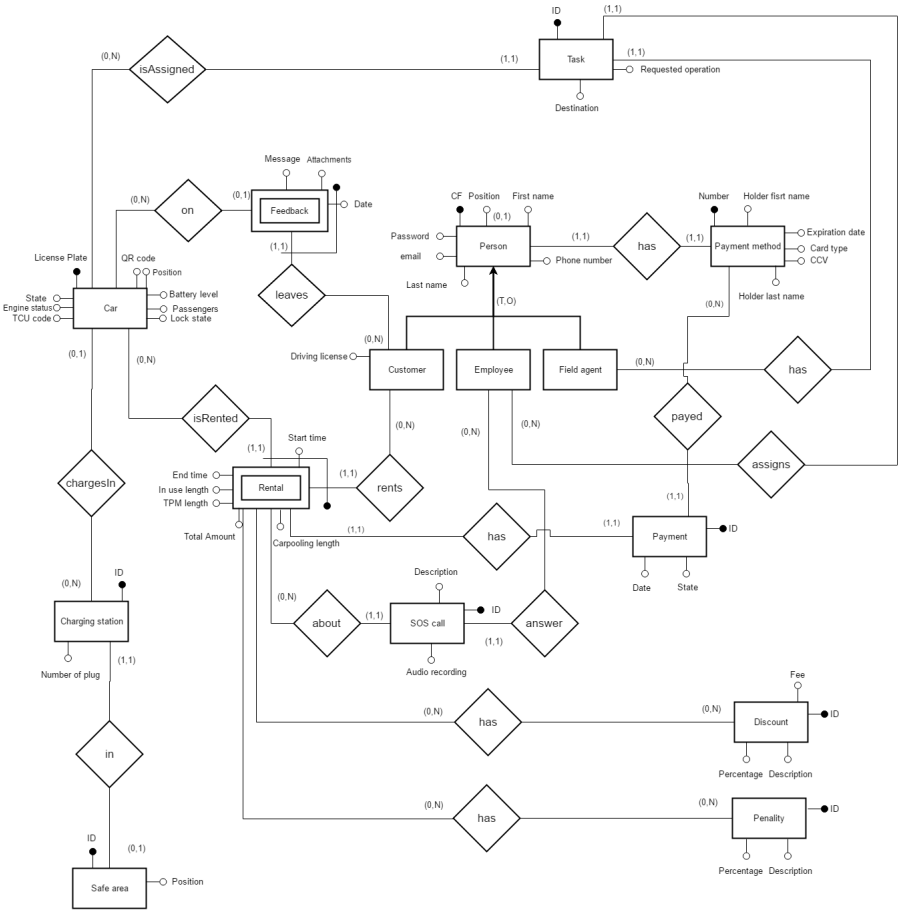

Figure 7: Entity–relationship diagram

2.4 Deployment view

The system architecture is divided into 5 tiers and it is based on JEE framework.

- The first tier is the *client tier*: it is composed by the tablet and mobile phone applications which communicate directly to the application layer using the RMI system. Moreover, the web app is used by fleet management employees and field agents to perform their tasks.
- The *web tier* (tier 2) contains the web server implemented with the Apache HTTP platform, which is composed of the *static content module*, and *mod_jk* and *mod_proxy_ balancer,* connectors which are used respectively for the connection with the servlet server and the load balancer.
- On the third tier, the *Script engine tier*, there is Tomcat, which generates the dynamic content, via Servlet or JSP, requested by the web server using the *ajp13* protocol.
- The *application tier* (tier 4) consists of *JBoss* which handles the *java beans* and all the business logic.
- The *data tier* (tier 5) is mainly composed by the Database Server. The communication between tier 3 and tier 4 is performed via *JDBC connector* which uses *jdbc protocol*.

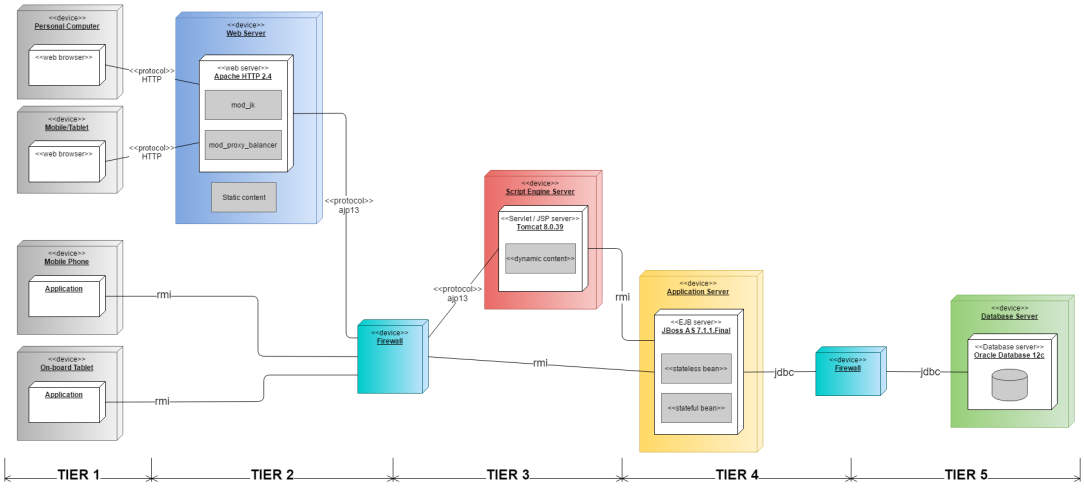

Figure 8: Deployment diagram

Recommended implementation

- **Client tier**: the customer mobile app may be implemented using a cross platform mobile development tool such as *Xamarin*, a platform which allows to develop native Android, iOS and Windows Phone apps in C#; the on-board tablet app may be implemented using either *Xamarin* or any other Android development tool.
- **Web tier**: the web pages may be implemented using HTML 5.0, Java Script and CSS.
- **Script engine tier:** the dynamic content of web pages may be generated using Java *servlets*.
- **Application tier**: The EJB application server uses *stateless java beans* and stores the data (and the state with the client) on the database using *JPA* and mapping the object with the data through *entity beans*.
- **Data tier**: the database may be implemented with *Oracle database 12c*.
- For web, script engine, application and data tier a server such as *Oracle SPARC T7* may be a good solution because it suits well with *JEE* applications and *Oracle DBMSes*.

2.5 Runtime view

The following sequence diagrams describe the interactions that happen between the main components of the product when the most common features are utilized. Beware that this is still a high-level description of the actual interactions that are going to take place, so functions and their names may be added or modified during the development process.

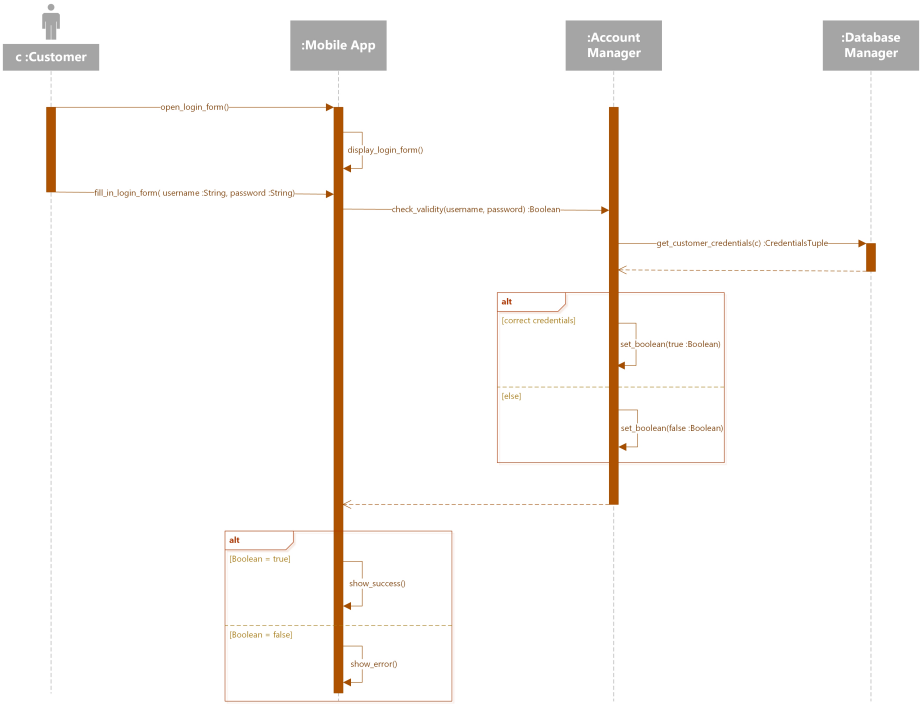

Figure 9: Customer Login

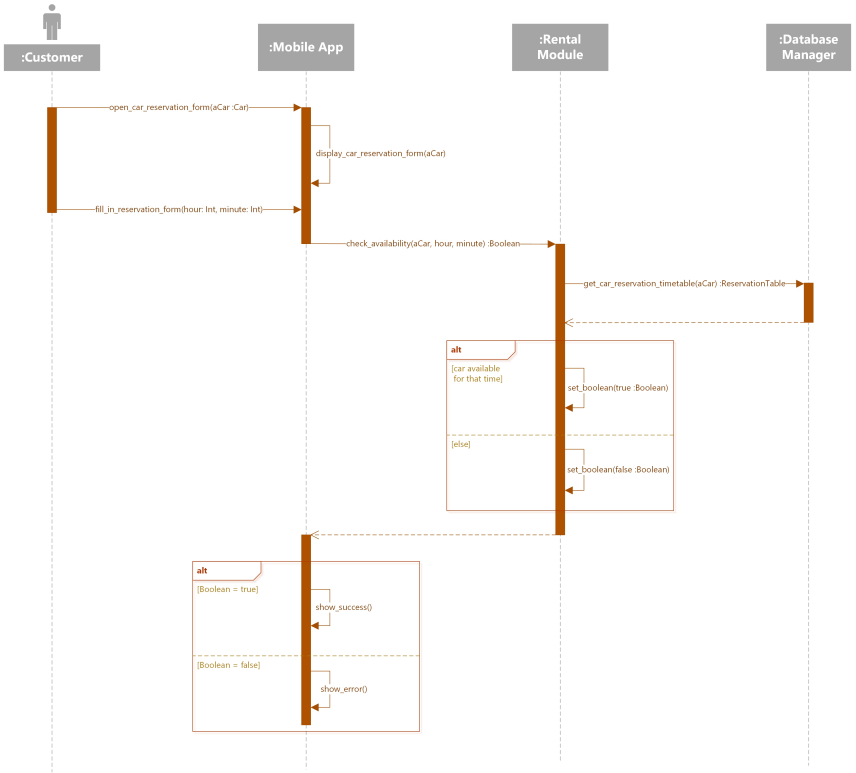

Figure 10: Car Reservation

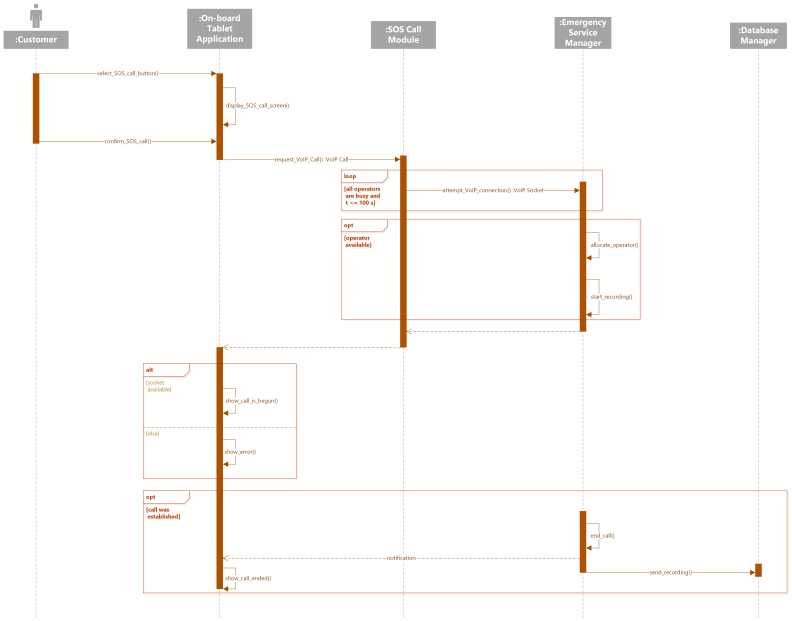

Figure 11: SOS Call (with exception)

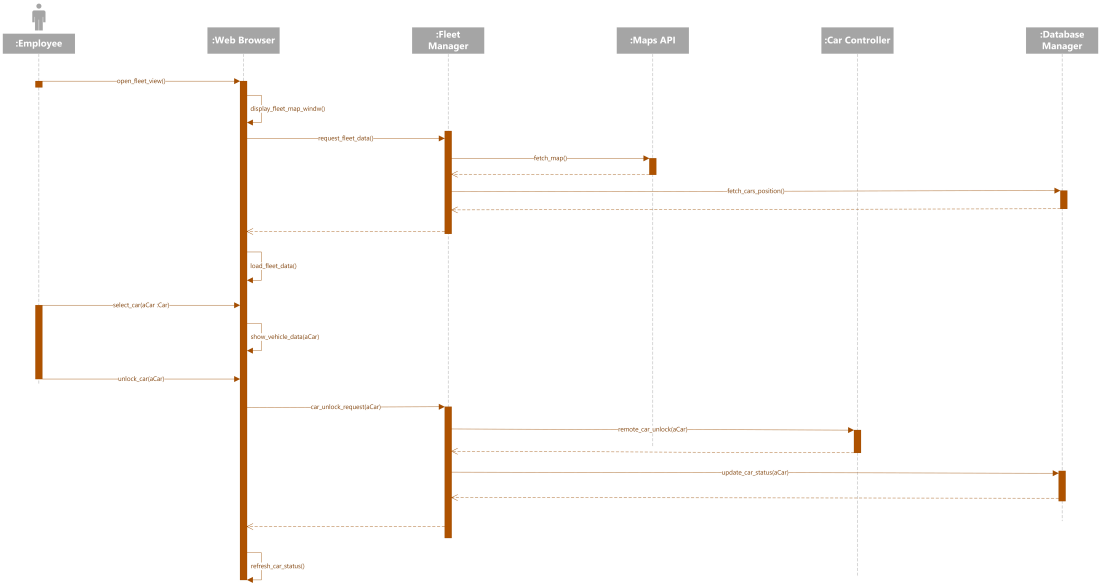

Figure 12: Fleet Management : Remote Car Unlock

2.6 Component interfaces

This diagram shows the main methods belonging to the interfaces of the components.

- The interface *CarServices* behaves as a proxy between the on-board tablet application and server services, so it gathers all the methods of *Car on-board tablet services* components extending their interfaces. In particular, there are methods for updating the state of the car, launching an SOS call and for all the other functionalities offered during the ride.
- Like CarServices, *CustomerServices* also gathers all the methods needed to provide the client with functionalities such as book a car, show the available ones, leave feedback, show his/her rental history and manage his/her profile.
- The last interface is *EmployeeWebServices* which extends the two interfaces of the *Employee web services* subsystem: *FieldService* and *ManageFleet*. These two interfaces provide the methods for managing the fleet by the office side (show the state of the entire fleet and assign a specific car to one agent) and on the field (show assigned tasks and confirm the completion of them).

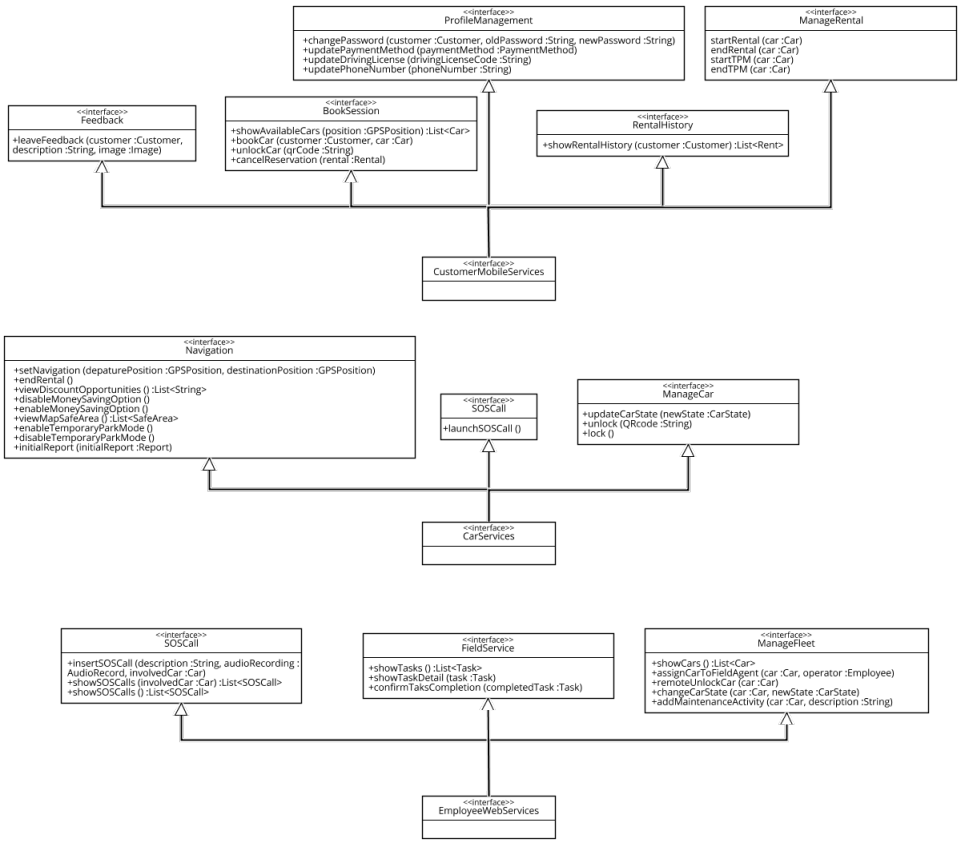

Figure 13: Component interfaces diagram

2.7 Selected architectural styles and patterns

- Selected design patterns :
  - **Proxy pattern** allows for object level access control by acting as a pass through entity or a placeholder object.
    (Used in some components of the component diagrams)
  - **Facade pattern** supplies a single interface to a set of interfaces within a system.
    (Used in some components of the component diagrams)
  - **State pattern** ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.
    (Used in the car state-chart diagram)
  - **Bridge pattern** defines an abstract object structure independently of the implementation object structure in order to limit coupling.
    (Used in the interfaces of the component diagrams)
- Recommended architectural pattern for implementation :
  - **Model-View-Controller pattern** divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. This is one of the most common and effective ways to avoid a dangerous level of coupling between the various parts of the whole system.
- Recommended design patterns for implementation :
  - **Factory pattern** exposes a method for creating objects, allowing subclasses to control the actual creation process. It is particularly useful if applied in combination with the MVC pattern.
  - **Observer pattern** lets one or more objects be notified of state changes in other objects within the system. It is practically essential for the application of the MVC pattern.
  - **Strategy pattern** defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior. This pattern, if applied correctly, guarantees a high level of polymorphism.
  - **Visitor pattern** allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure. It contributes to the overall decoupling of the system.

2.8 Other design decisions

Maps APIs usage

- **Customer mobile app**: considering that the app will be a *cross-platform native* mobile application, different APIs will be used to render maps on each operating system. This means that the interactive map of the *Search for available cars* screen will be implemented using Apple Maps API on iOS, Google Maps API on Android, Windows/Bing Maps API on Windows Phone.

- **On-board tablet app**: assuming that the app will run only on customized Android tablets, when the customer selects the turn-by-turn navigation feature an *intent* will be used to launch the official *Google Maps* native application. If the customer enables the Money Saving Option, he/she will be asked to enter his/her destination; then, if an available charging station has been found within 1 km from the target address, the customer can start turn-by-turn navigation towards the station just pressing a button: an *intent* will be used to launch the official *Google Maps* native application with the destination already set (`google.navigation` intent). Buttons for parking, ending the rent session, SOS calls or discount opportunities will be rendered over *Google Maps* using Android *Draw over other apps* permission.

- **Employee web-app**: an interactive map showing vehicles positions will be rendered using Google Maps Javascript API.

Telematic Control Unit

As already introduced in the *RASD*, the TCU is a customized micro-controller which works as a physical interface of the *Car Controller* component: it receives commands from the back-end of the platform via cellular network (lock/unlock car doors, ...) and continuously sends back to the system data from TCU sensors (GPS coordinates, GPS speed) and car built-in sensors connected to the TCU via *OBD* (On-Board Diagnostics) connector (engine status, door lock status, passengers weight sensors, ...). The TCU was designed as an independent hardware component with respect to the car on-board tablet because it has a critical role in the system: a failure of the tablet must have no impact on the the TCU functions.

3 Algorithm design

The following paragraphs present a description and simplified Java-like code of the most relevant algorithms for the *PowerEnJoy Platform*.

3.1 Available cars search algorithm

The purpose of this algorithm is to determine the subset of cars belonging to the fleet which satisfy the following conditions:

- the car status is *Available*
- the *great-circle distance* between the car and the target position is less or equal than parameter D
- the target position is defined by the GPS coordinates of the customer current location/address specified by the user

The *great-circle distance* between two coordinates (direct point-to-point aerial transit path between the two positions) may be computed with different approaches; the most common one is the so called *Haversine formula:*

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Figure 14: *Haversine formula*, where F represents latitude, l longitude, r the radius of the sphere, and d the distance between the two positions

The following algorithm is a possible implementation of the *Haversine formula*:

- Inputs:
  - *position1*: GPS coordinates of position 1
  - *position2*: GPS coordinates of position 2
- Output:
  - distance between positions 1 and 2

Algorithm 1 *Haversine formula* algorithm

```
public class Haversine {

    public static final double R = 6372.8; // In kilometers

    public static double haversine(GPSposition position1, GPSposition position2) {

        double dLat = Math.toRadians(position2.lat - position1.lat);

        double dLon = Math.toRadians(position2.lon - position1.lon);

        position1.lat = Math.toRadians(position1.lat);

        position2.lat = Math.toRadians(position2.lat);

        double a = Math.pow(Math.sin(dLat / 2),2) + Math.pow(Math.sin(dLon / 2),2) *
          Math.cos(position1.lat) * Math.cos(position2.lat);

        double c = 2 * Math.asin(Math.sqrt(a));

        return R * c;

    }
}
```

The following code is a possible implementation of the *Available cars search* algorithm:

- Inputs:
  - *cars*: list of cars
  - *targetLocation*: GPS coordinates of the target position
  - *D*: maximum distance parameter
- Output:
  - list of available cars nearby the customer

Algorithm 2 *Available cars search* algorithm

```
public ArrayList<Car> availableCarsFromLocation(ArrayList<Car> cars, GPSposition targetLocation, double D)
throws NoAvailableCarsException
{

    ArrayList<Car> availableCars = new ArrayList<Car>();

    for(Car c : cars){

        if(c instanceof AvailableCar) {
            double distance = Haversine.haversine(c.location, targetLocation);
            if (distance < D)
                availableCars.add(c);
        }

    }

    if(!availableCars.isEmpty)
        return availableCars;
    else throw new NoAvailableCarsException();

}
```

3.2 Rental fare computation algorithm

This algorithm should be used to calculate the total cost of each rent session depending on the different tariffs (which may vary during the ride itself) and behaviors followed by the customer.

- Possible tariffs :
  - Standard mode : full price per minute
  - Temporary Parking Mode : reduced price per minute
- Possible discounts : (total percentage discount cannot exceed 50%)
  - If the system detects the customer took at least two other passengers onto the car for at least 80% of the rent session time, the system applies a discount of 10% on the last ride
  - If a car is left with no more than 50% of the battery empty, the system applies a discount of 20% on the last ride
  - If a car at the end of the rental is parked in a safe area and plugged into the power grid by the customer, the system applies a discount of 30% on the last ride
  - If the customer enables the Money Saving Option, he/she can input his/her final destination and the system provides information about the charging station where to leave the car to get a discount of 20 %
- Possible penalties :
  - The system must detect the distance between the car and the nearest power grid at the end of the rental and charge 30% more on the last ride if the distance exceeds 3 km
  - The system must detect the battery level of a car at the end of the rental and charge 30% more on the last ride if the level is lower than 20%
  - If the customer doesn't turn off the Temporary Parking Mode within 20 minutes from its activation, then the system will end the rent session and charge the customer with a 20 € fee

Algorithm 3 *Rental fare computation* algorithm

```
public float calculateFare(Rental rental)
{
    int totalDiscount = 0; //in the form of percentage

    for (Discount d : rental.getDiscounts()){
        if (totalDiscount + d.getPercentage() > 50){
            totalDiscount = 50;
        }
        else totalDiscount += d.getPercentage();
    }

    int totalFee = 0;

    for (Penalty p : rental.getPenalties()){
        totalFee += p.getFee();
    }

    int totalPenalty = 0;

    for (Penalty p : rental.getPenalties()){
        totalPenalty += p.getPenalty();
    }

    float totalFare = (rental.getInUseLenght()*STANDARD_TARIFF + rental.getTPMLenght()*TPM_TARIFF)*
      (100-totalDiscount +
    totalPenalty)/100 + totalFee;
    return totalFare; //in the form of euros
}
```

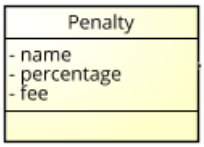

Figure 15: Penalty class

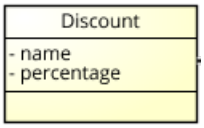

Figure 16: Discount class

3.3 Destination choice with Money Saving Option algorithm

This algorithm aims to determine the safe area which allows the customer to obtain the Money Saving Option discount.

A possible implementation of this algorithm is provided:

- Inputs:
  - *safeAreas*: list of all the safe areas
  - *customerDestination:* desired destination of the customer
  - *maxDistance*: maximum distance between *customerDestination* and available safe area
- Output:
  - *nearestSafeArea:* the nearest safe area to the desired destination

Algorithm 4 *Destination choice with Money Saving Option* algorithm

```
public SafeArea nearestSafeAreaMoneySavingOption(ArrayList<SafeArea> safeAreas,
                                                 GPSposition customerDestination, double maxDistance)
  throws  NoSafeAreasAvailableException

{
    SafeArea nearestSafeArea = null;

    for(SafeArea s : safeAreas){

      if(isCorrectCandidate(s,customerDestination,maxDistance)){

            if(nearestSafeArea == null){
                nearestSafeArea = s;
            }else{
                if(Haversine.haversine(s.getLocation(),
                                       customerDestination) < Haversine.haversine(nearestSafeArea.getLocation(),
                                                                                  customerDestination)){
                    nearestSafeArea = s;}
                }
        }
    }

    if(nearestSafeArea != null){
        return nearestSafeArea;
    }else{
        throw new NotSafeAreasAvailableException();
    }
}

//This method is used as a support for the nearestSafeAreaMoneySavingOption method
private static Boolean isCorrectCandidate(SafeArea safeArea, GPSposition customerDestination, double maxDistance)
{
    double distance = Haversine.haversine(customerDestination, safeArea.getLocation());
    return safeArea.hasChargingStation() && safeArea.getChargingStation().hasAvailablePlug() && distance <= maxDistance;
}
```

3.4 Field agent's tasks ordering algorithm

The purpose of this algorithm is to determine an ordering for the set of tasks which are assigned to a field agent.
Every time the field agent opens the *Field service* feature of the *Employee web-app,* a list of tasks is presented to him: the vehicles which need maintenance should be listed in order of increasing distance from the agent's current position.

Note that the distance which this algorithm should determine is a *driving distance*, not a *great-circle distance*: this is why an external API such as *Google Maps Distance Matrix API* should be used. As an example, a possible usage of the aforementioned API is presented: the integration is made possible by using the *Java client library for Google Maps API Web Services*, a library which brings the *Google Maps API* to server-side Java applications. The library may be easily added to any Java project via *Maven*:

```
<dependency>
        <groupId>com.google.maps</groupId>
        <artifactId>google-maps-services</artifactId>
        <version>(insert latest version)</version>
</dependency>
```

Algorithm 5 Driving distance between `origins` and `destinations` using *Java client library for Google Maps API Web Services*

```
1  public double drivingDistanceBetweenPosition1And2(GPSposition position1, GPSposition position2) {
2
3      double distance = 0;
4
5      GeoApiContext context = new GeoApiContext().setApiKey("API-KEY");
6
7      DistanceMatrixApiRequest request = DistanceMatrixApi.newRequest(context)
8
9          .origins(position1.lat + "," + position1.lon)
10
11         .destinations(position2.lat + "," + position2.lon)
12
13         .mode(TravelMode.DRIVING)
14
15     DistanceMatrix matrix = request.await();
16
17     distance = matrix.rows[0].elements[0].distance;
18
19     return distance;
20
21 }
```

The following algorithm, given all the tasks of a field agent, sorts them by increasing distance:

- Input
  - *fieldAgent* object
- Output
  - *orderedTasks* array
- Support method
  - *sortHashMapByValues*: this method, given an hash map, returns a new one which is ordered by its values

Algorithm 6 *Field agent's tasks ordering* algorithm

```
1  public ArrayList<Task> orderFieldAgentTasks(FieldAgent fieldAgent) {
2
3          GPSposition agentPosition = fieldAgent.getLocation();
4          HashMap<Task, Double> taskDistanceMap = new HashMap<>();
5
6          for(Task t : fieldAgent.getAssignedTasks()){
7
8              Double d = drivingDistanceBetweenPosition1And2(agentPosition, t.getAssignedCar().getLocation());
9
10             taskDistanceMap.put(t, d);
11
12         }
13
14         HashMap<Task,Double> orderedHashMap = sortHashMapByValues(taskDistanceMap);
15         ArrayList<Task> orderedTasks = new ArrayList<>(orderedHashMap.keySet());
16         return orderedTasks;
17
18 }
```

Advanced Algorithm: Instead of sorting the agent's tasks only considering the driving distance between the agent's position and the car location, a more advanced version of the *Field agent's tasks ordering* algorithm may compute the ordering taking into account two factors: the car position and the destination of the task. Given a task, the next task on the list should be the closest to the destination of the current task (automobile repair shop, car wash, ...), and not the nearest to the agent's current position. So this algorithm should provide the shortest itinerary for the agent in order to optimize the time and the traveled distance.

4 User Interface design

4.1 Mock-ups

As described in section *3.1 User Interfaces* of the *Requirements Analysis and Specification Document*,

- customers will use a mobile app to register and login to the platform, search for available cars, manage their reservations and personal information, leave feedbacks about the service
- customers will also use an on-board tablet app to start the rental, get driving directions and find useful information about parkings and discounts
- company employees will manage the fleet and perform maintenance activities accessing an Employee web-app via laptops, desktop computers or tablets

The following mock-ups provide an idea of how the mobile app, web-app and on-board tablet app will look like:

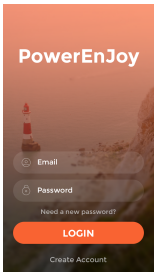

Figure 17: **Mobile app**: Customer login

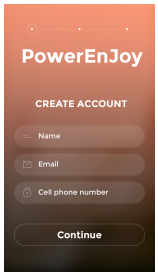

Figure 18: **Mobile app**: Create new account

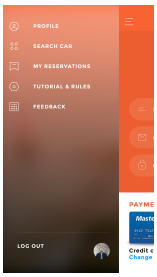

Figure 19: **Mobile app**: Navigation menu

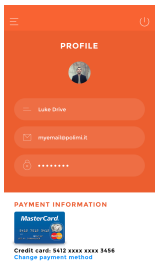

Figure 20: **Mobile app**: Customer profile

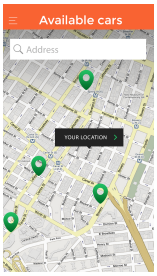

Figure 21: **Mobile app**: Search for available cars

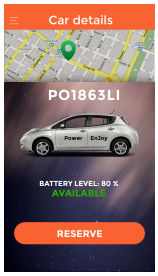

Figure 22: **Mobile app**: Vehicle details

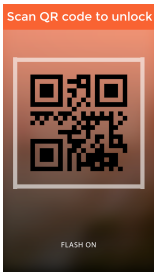

Figure 23: **Mobile app**: Vehicle unlock via QR code

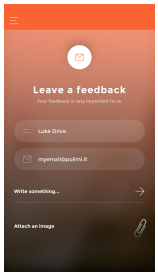

Figure 24: **Mobile app**: Feedback

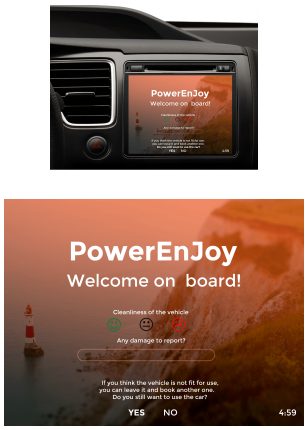

Figure 25: **On-board tablet**: Welcome screen

Figure 26: **On-board tablet**: Features

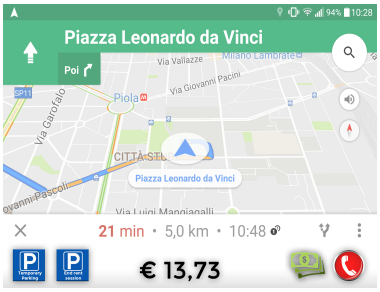

Figure 27: **On-board tablet**: Turn-by-turn navigation

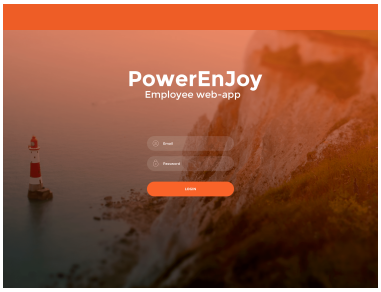

Figure 28: **Employee web-app**: login

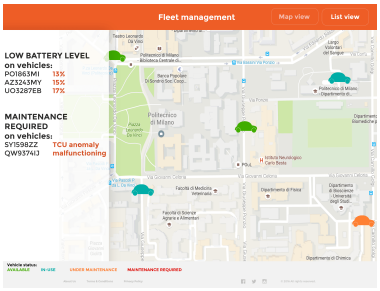

Figure 29: **Employee web-app** | **Fleet management**: Map view

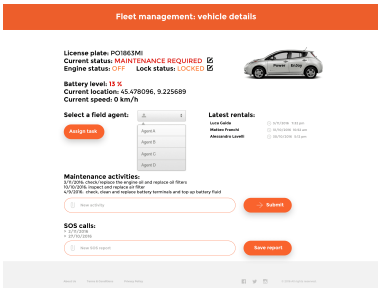

Figure 30: **Employee web-app** | **Fleet management**: Vehicle details

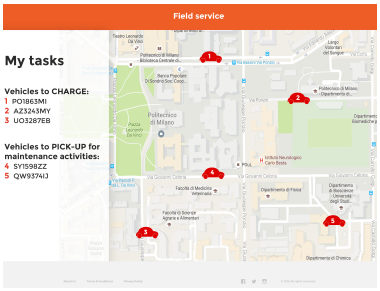

Figure 31: **Employee web-app** | **Field service**: My tasks

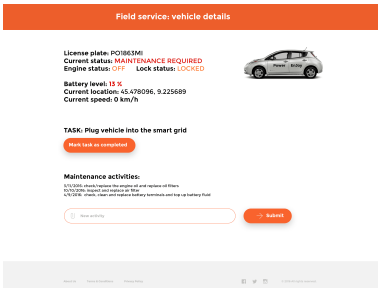

Figure 32: **Employee web-app** | **Field service**: Vehicle details

4.2 UX diagrams

The following UX diagrams provide additional information about the user interface of the customer mobile app, on-board tablet app and Employee web-app. In particular, they represent significant properties of the UI screens and they model the navigational relationships between them. Screens are modeled using *<<screen>>* stereotyped classes, while input forms and screen compartments are represented with separate *<<input form>>* and *<<screen compartment>>* stereotyped classes. The usage of the *back* button of the browser or of the mobile device is ignored in this context.

4.2.1 Customer mobile app

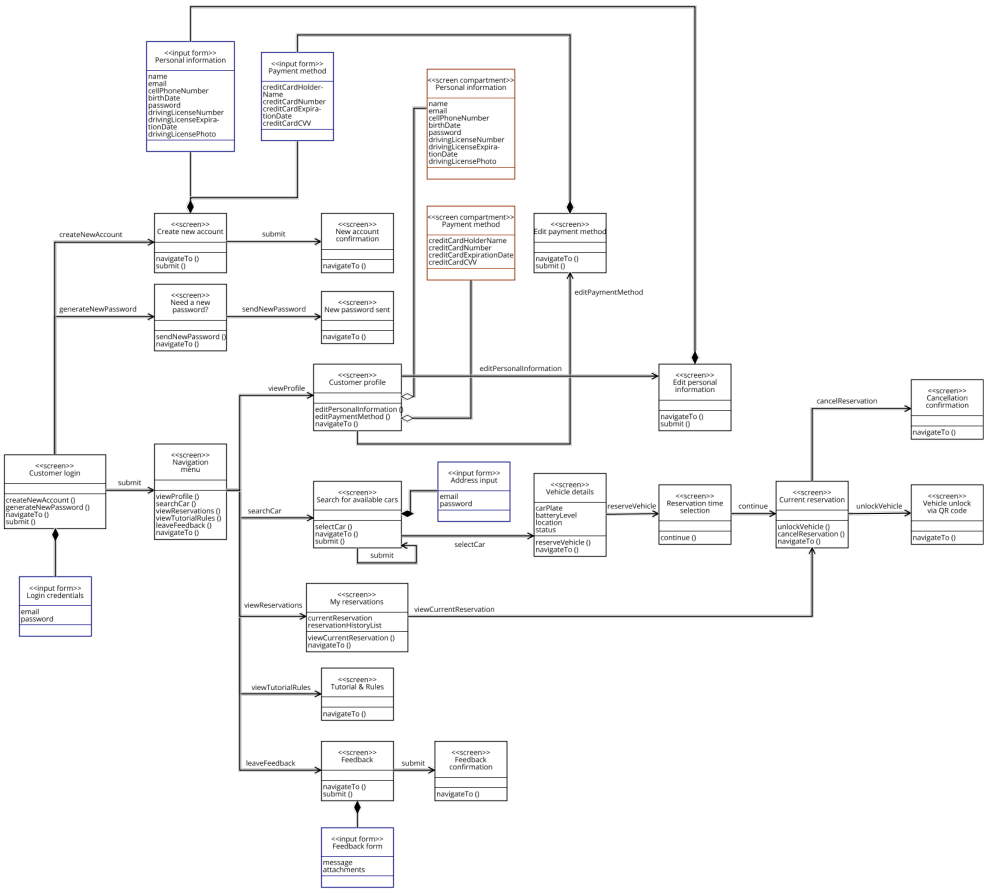

Figure 33: Customer mobile app UX diagram

4.2.2 On–board tablet app

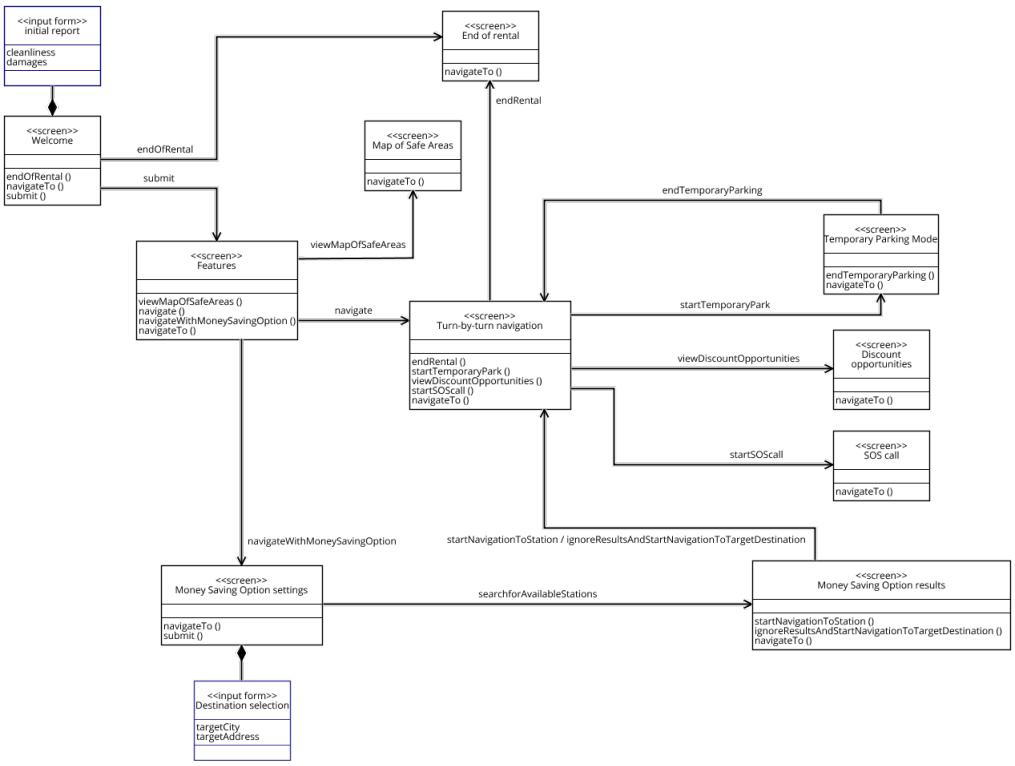

Figure 34: On–board tablet app UX diagram

4.2.3 Employee web app

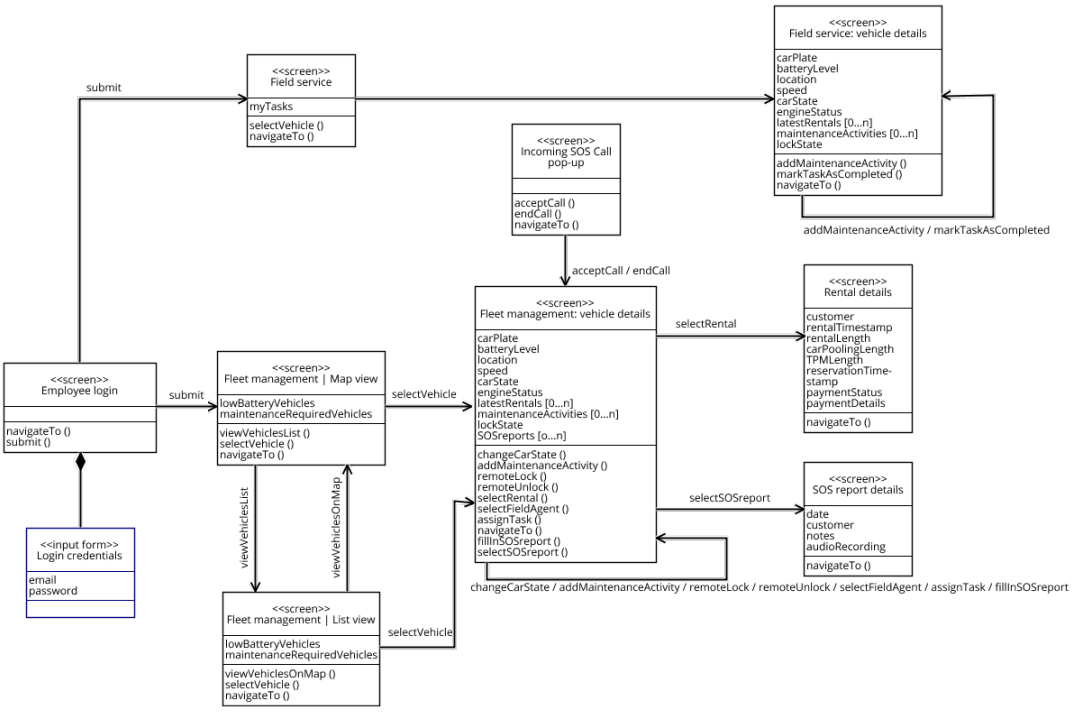

Figure 35: Employee web-app UX diagram

4.3 BCE diagrams

Assuming that a Model-View-Controller design pattern is adopted, the following diagrams present a possible representation of the system separating internal representations of information from the ways that information is presented or processed. *Boundaries* are objects, such as user interfaces, that interface with system actors; *Controls* are objects that manage the interactions between boundaries and entities implementing the application logic required to process user requests; *Entities* are used to model the access to data.

4.3.1 Customer mobile app

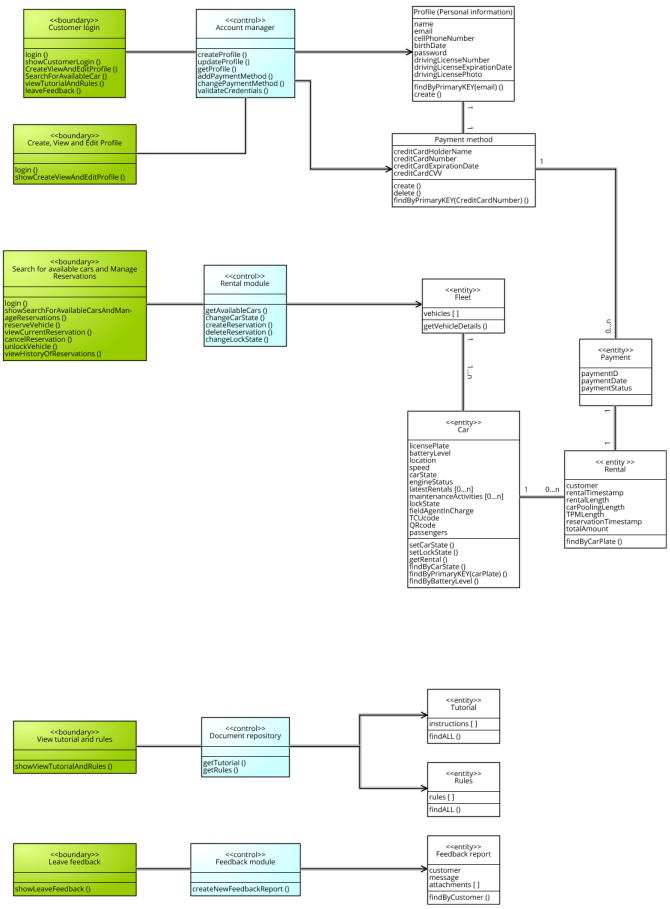

Figure 36: Customer mobile app BCE diagram

4.3.2 On–board tablet app

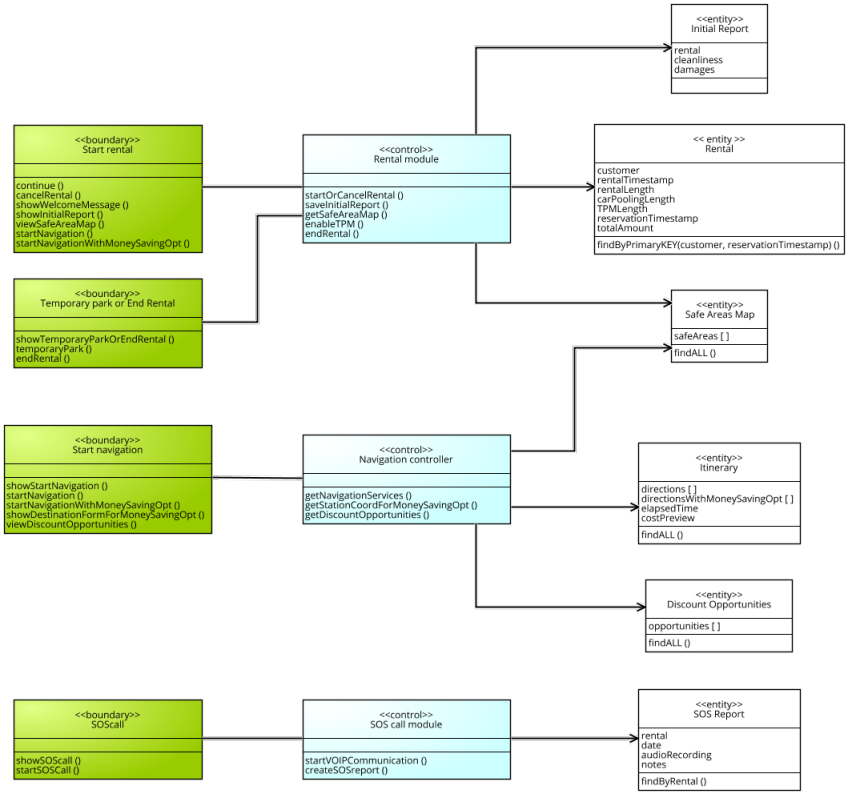

Figure 37: On–board tablet app BCE diagram

4.3.3 Employee web app

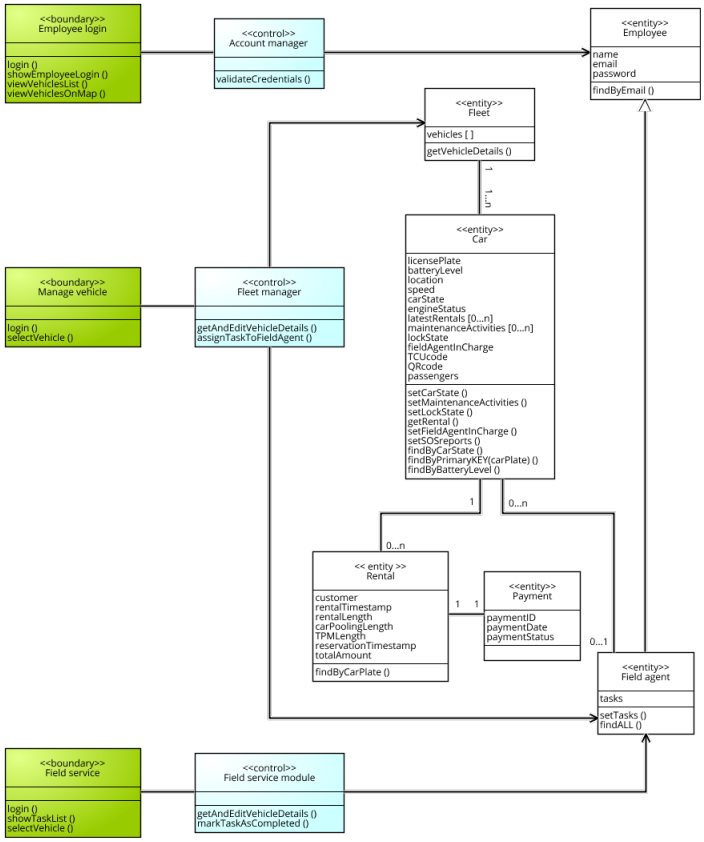

Figure 38: Employee web-app BCE diagram

5 Requirements traceability

While making the design choices presented in the this document, the main goal was to fulfill in a complete and correct way the goals specified in the *Requirements Analysis and Specification Document*.
The following list provides a mapping between goals and requirements defined in the RASD and system components illustrated in the DD.

- Goal G1 (The system should allow new users to register and login into the system in order to make use of the platform features) | Requirements R1.1-R1.6
  - Customer mobile services
    - Account manager
- Goal G2 (Customers should be able to find the location of available cars within a certain distance from their current location or from a specific address) | Requirements R2.1-R2.5
  - Customer mobile services
    - Rental module
- Goal G3 (Customers should be able to reserve a car among the available ones) | Requirements R3.1-R3.6
  - Customer mobile services
    - Rental module
- Goal G4 (The system should discourage bad behaviors of the customers through surcharges or penalties) | Requirements R4.1-R4.3
  - Customer mobile services
    - Rental module
  - Car on-board tablet services
    - Navigation controller
    - Car controller
- Goal G5 (The system should allow the customer to unlock the car he/she reserved once he/she is nearby) | Requirements R5.1-R5.5
  - Customer mobile services
    - Rental module
  - Car on-board tablet services
    - Car controller
- Goal G6 (The system should allow the customer to park for short periods of time without ending the rent session) | Requirements R6.1-R6.3

  - Customer mobile services
    - Rental module
  - Car on-board tablet services
    - Car controller

- Goal G7 (The system should end the rent session when it detects that the car was parked in a safe area and the customer got out of the car) | Requirements R7.1-R7.4
  - Car on-board tablet services
    - Car controller

- Goal G8 (The system should charge the customer proportionately to the time spent driving the vehicle and notify him of the current charges through the screen of the on-board tablet of the car) | Requirements R8.1-R8.3
  - Customer mobile services
    - Rental module
  - Car on-board tablet services
    - Car controller

- Goal G9 (The system should encourage virtuous behaviors of the customers, such as carpooling, responsible battery usage and parking, money saving option) | Requirements R9.1-R9.5
  - Car on-board tablet services
    - Navigation controller
    - Car controller

- Goal G10 (The system should support management activities by the car-sharing company employees concerning car maintenance and car accidents) | Requirements R10.1-R10.7
  - Employee web services
    - Fleet manager
    - Field service module

Appendix

Used tools

To redact the document the following tools were used:

- **LyX**: open source document processor based on top of the LaTeX typesetting system

  www.lyx.org

- **Adobe Photoshop**: raster graphics editor

  www.adobe.com/products/photoshop.html

- **Signavio**: web-based business process modeling tool

  www.signavio.com

- **Draw.io**: online diagram software

  www.draw.io

- **GitHub**: Git repository hosting service

  github.com

- **SourceTree**: Git client

  www.sourcetreeapp.com

Effort spent

Hours of work spent by each member of the team redacting the document:

- Matteo Franchi: 20 h 15 m
- Luca Guida: 40 h
- Alessandro Lavelli: 31 h
- *Additional teamwork*: 6 h