Power EnJoy

Design **D**ocument

Software Engineering 2

A.A. 2016/2017

Version 1.2

Emanuele Ghelfi (mat. 875550)
Emiliano Gagliardi (mat. 878539)

January 6, 2017

Contents

1 Introduction

1.1 Purpose

This software design document describes the architectures and the system design of Power EnJoy. It's mainly intended for developers but it has a hierarchical structure. It completes the RASD and defines the component that led to the satisfaction of the goals previous defined. It starts from an high level description of the architecture and then it goes into detail.
This document has to identify:

- Architecture of the system
- Interactions between components
- Main algorithms of the system

The main purpose is to gain a general understanding of how and why the system is decomposed and how individual components work together.

1.2 Scope

PowerEnJoy is a software that manages a car sharing service for electric cars. The aim of this software is to make the reservation of cars simple and quick. So the system should provide users with real time information about availability of cars, their status and their positions.After the reservation, users can directly get their car in pre-defined parking areas. The service will be accessible only to registered users, giving some personal information and data needed to the payment. The price of the ride is computed with a fixed amount of money per minute, displayed by the car, and finally charged. To avoid useless reservation, that is to say that a user doesn't pick up the car, the reservation expires after a fixed time, the car returns available, and the user is charged with a fee. Cars must be locked in the safe areas, and only users that have made the reservation can unlock them. The software has to provide also management functionality for administrators and operators in order to ensure a simple managing of the system.

1.3 Definitions, Acronyms, Abbreviation

- RASD: Requirements and Specifications Document
- DD: Design Document (this document)
- JSON: JavaScript Object Notation
- REST: Representational State Transfer
- JDBC: Java Database Connectivity
- API: Application Programming Interface
- JEE: Java Enterprise Edition

1.4 Reference Documents

- RASD released before this document.
- Assignments AA 2016-2017.
- DD from previous years.

1.5 Document Structure

Introduction: this is a general overview of the document.
The *Purpose* part describe the audience and the main goals of this document.
The *Scope* part has to provide a description and scope of the software and explain the goals, objectives and benefits of the project.
Reference Documents are previous documents of this project and documents used as examples and reference.
Architectural Design: this section explains the relationship between the modules to achieve the complete functionality of the system (requirements defined in the RASD).
It contains an high level overview of how responsibilities of the system were partitioned and then assigned to subsystem (components).
In this part of the document are identified each high level subsystem and the roles or responsibility assigned to it in order to achieve a more detailed comprehension of the software to be. It's also described how these components collaborate with each other in order to achieve desired functionality. There is a focus on the interface provided by individual components in *Component Interfaces* section.
Deployment View gives a description of how the software to be it's intended to be deployed.
Runtime View gives a description of the interaction between components in the most important use case of the system.
In the section of *Selected Architectural Styles and Patterns* are described which styles and patterns have been followed in the realization of the system. There is a focus on the rationale of these decisions.
Algorithm Design: this section explains the most important algorithms of the software to be. Pseudo–code has been used in order to avoid unnecessary implementation details.
User Interface Design: this section refers to the same section in RASD and provides some extensions.
Requirements Traceability: this section describe how requirements defined in RASD have been mapped to system components defined in section 2.

2 Architectural design

2.1 Overview

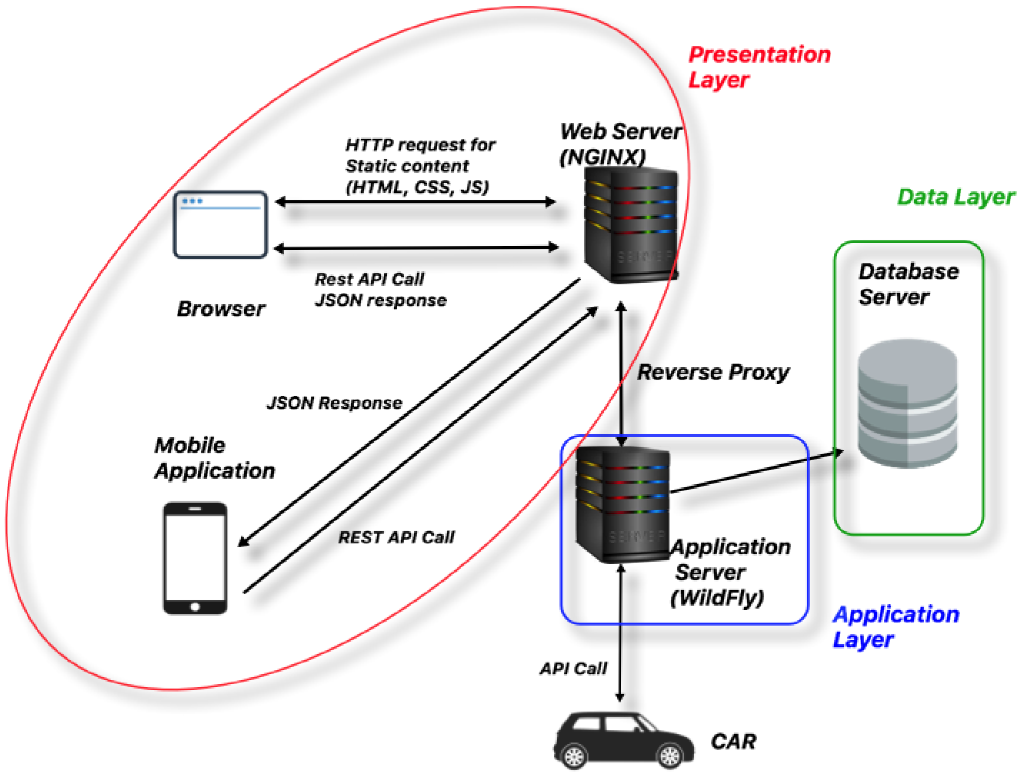

Figure 1: High level architecture

In this section is presented an high level architecture. It shows how the system is structured and the main interaction between subsystems.
There is a WebServer with Nginx technology that serves static content (like .html, .js, .css) to browser. When WebServer receives certain API calls it acts as reverse proxy. Proxying is used to distribute load among several servers. When web server proxies a request, it sends the request to the Application Server, fetches the response and sends back to the client.
Application server contains the business logic of the software to be and interacts with the Database Server and with the CarOS.
Mobile Application only makes API calls to WebServer and receives back the JSON response of Application Server.
This project starts as a monolithic application because of simplicity of development, but with particular attention to the modularization. So, in case will be the need of scaling, the refactoring to micro-services won't be difficult.

2.2 High level components and their interaction

In this section the high level component are presented and it's described how they interact with each other. The client component is made of the Web Browser and the Mobile Application. Both communicates with the server through its interface.
The application server communicates with DBMS through DBMS API and with CarOS through its API.
In this way the central system always knows the status (position, battery, etc.) of all cars and also cars can initiate a communication with server when they need to communicate important event. This is done with observer pattern as specified later in this document.
The communication between client and server can be synchronous or asynchronous depending on the kind of interaction. The server can communicate asynchronously with client with notification or messages (email).This is the class diagram of the data core (likely during the coding phase a more detailed data structure will come up). There are two important considerations to do about this model: the first one is that the two relations user-reservation and user-ride will guarantee the reconstruction of users "story" in the use of the Power Enjoy service, and the second one related to a bit more technical solution about car system. In the car class there are some information that in general shouldn't be stored, like battery life and position, because they change rapidly. The explanation of this decision is in the component diagram description, in particular in the explanation of how the car proxy component works.

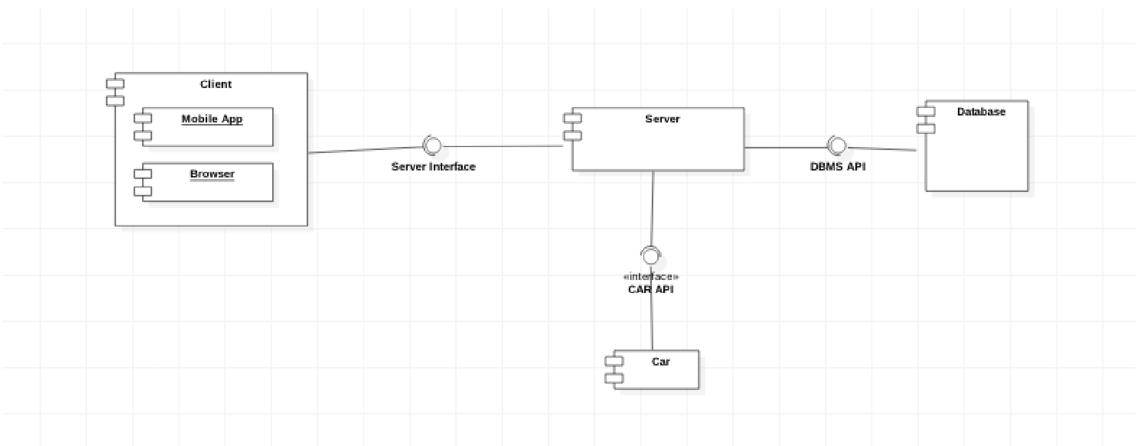

Figure 2: High level component diagram

2.3 Data model

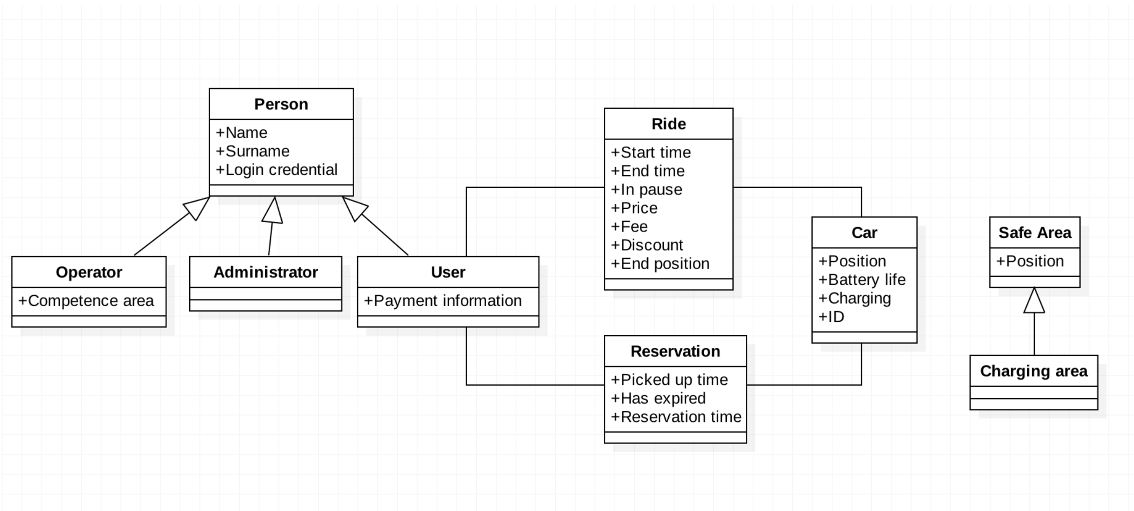

Figure 3: Data class diagram

This is the core of the data stored in the db. It is not complete, it only contains the more important information and it is possible that during the coding phase more structures will come up. There are two important considerations to do: the first one, is that the two relations User-Ride and User-Reservation lead to the construction of the whole story of a user, and the second one, more technical, is related to the Car class. This class contains some information that in general aren't stored in a database, because they change rapidly, like Position and battery life. This decision is motivated and explained in the description of the car proxy component, in the following section.

2.4 Component view

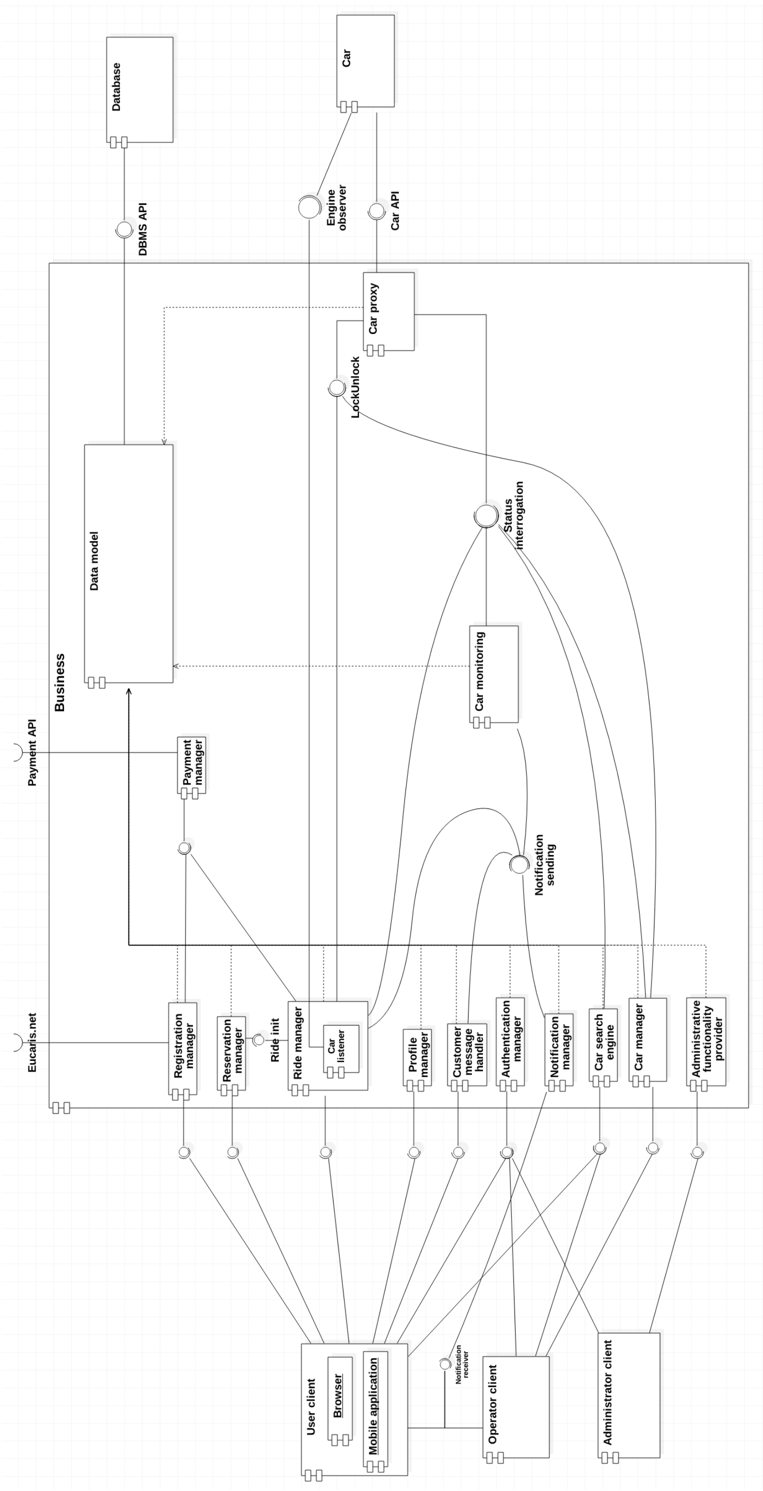

Figure 4: More detailed component diagram

2.4.1 Car proxy component

Car proxy is the abstraction of cars in the server. It must be invoked by other internal subsystems that need information about cars physical status (real time informations), like position, battery life etc. This component absolutely doesn't care about information related to reservations or rides.
The main purpose of this component is to interact with cars using their API. Its implementations should also guarantee that the system working on cars doesn't have to support high amount of parallel request. In order to do that, car proxy will use the database, storing a physical characteristic related to the timestamp of the API call that has provided that information. When a request arrives, car proxy decides which data should be provided: the one got through the API call to the car, or the one stored in the database. If the stored data are sufficiently recent then they can be provided, otherwise it is needed to make the request.
As it can be seen in the diagram, this component has a considerable fan-in, it could be necessary in future to make it scalable.

2.4.2 Car monitoring component

This component is substantially a daemon. It periodically asks to the car proxy component the information that operators need to do car maintenance. If a damage has been detected, the notification component should be called, in order to communicate it to an operator.

2.4.3 Ride manager

Ride manager is the component that takes care abut the data in the server corresponding to rides in the real world. Furthermore, when a ride is set to pause, it uses the car proxy component to lock or unlock the car.

2.4.4 Reservation manager

This component manages users reservation request, setting the status of a car from available to reserved and vice versa. It should also take care about reservation expiration, resetting the car state from reserved to available, and taxing the user through the component payment manager.

2.4.5 Registration manager

This component accepts request from guest of joining the Power Enjoy service. It checks if the guest has a valid drive license and adds it to the users. It also verifies the validity of the payment informations received by the guest using the services provided by the payment manager component.

2.4.6 Profile manager

This component reply to users that want to see their past utilization of the power enjoy services.

2.4.7 Customer messages handler

This component receives messages about users regarding malfunctions of cars, and use the notification manager to notify the operator that should take care about that.

2.4.8 Authentication manager

The authentication manager the component that manages the login of user, operator, and administrator, and administrates session.

2.4.9 Car search engine

This component interrogates the car proxy component and get cars position, performing the two type of car research.

2.4.10 Car manager

The car manager component can be used by operators to change car status in the server. For instance, if an operator is going to work on a car, it should use this component to switch the car status to "under maintenance", and then to unlock it.

2.4.11 Administrator functionality provider

This component provides the functionalities accessible only by the administrators that are listed in the RASD document.

2.4.12 Payment manager

This component uses external API of the accepted payment service. It should be used not only for carry out the payments, but also to verify during the registration of a user that the provided payment informations are correct.

2.4.13 Notification manager

Notification manager is used to send notifications to users (when their reserved car changes status), and to operator (when the system detects that an intervention on a car is needed, or a user make a communication).

2.4.14 Car listener

This is a subcomponent of the ride component, that is instantiated for a ride when the car is successfully unlocked. After the unlock of a car, probably the engine will be turned on. This component fulfill the problem of getting the engine ignition time, in order to calculate the correct ride cost. The runtime flow is explained with a sequence diagram in the section Runtime view.

2.5 Deployment view

The deployment diagram in Figure 5 shows the hardware of the system and the software that it's installed on it. The Mobile application is installed on Mobile Phone of the user while the browser runs on PC of the user/operator/admin. The Application Server and the Web Server are on different Nodes and on different environments. In this way they are totally decoupled and it's ensured the scalability of the system. The DBMS runs on a different node in order not to overload a node. If the need of scalability becomes more important for the system there could be more application server on different nodes with a load balancer before them. This is compatible with a cloud approach. In Figure 5 the web server serves the static content such as html, javascript and css. On the web server it's installed the reverse proxy module in order to act as a reverse proxy with respect to the Application Server. In this figure it's clear that the dbms is accessible only from the application server and the application server it's called only by the web server. The business logic runs on the Application server on J2EE container like WildFly(the new name of jBoss). WildFly provides full access to the functionality that Java Enterprise Edition provides including EJB and many other technologies. The web server could be Nginx because it uses an asynchronous event-driven architecture to handle a massive amount of connections. This architecture makes handling high and fluctuating loads much more predictable in terms of RAM usage, CPU usage, and latency.

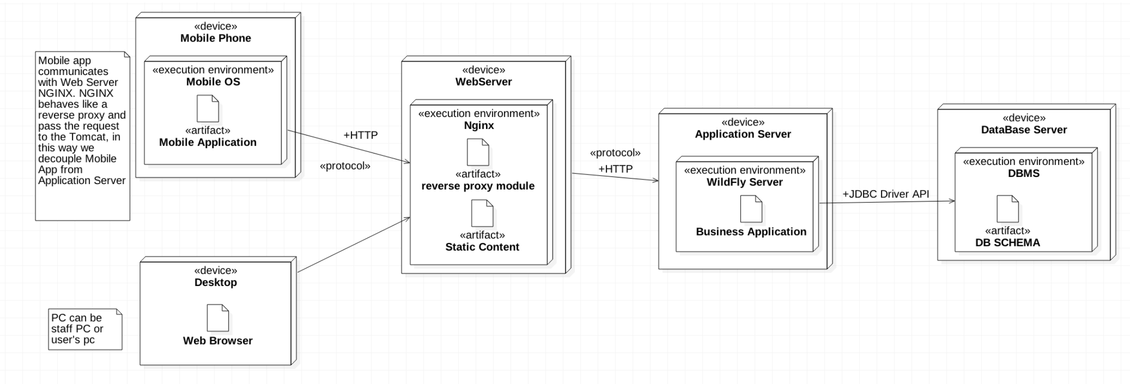

Figure 5: Deployment view

2.6 Runtime view

2.6.1 Registration

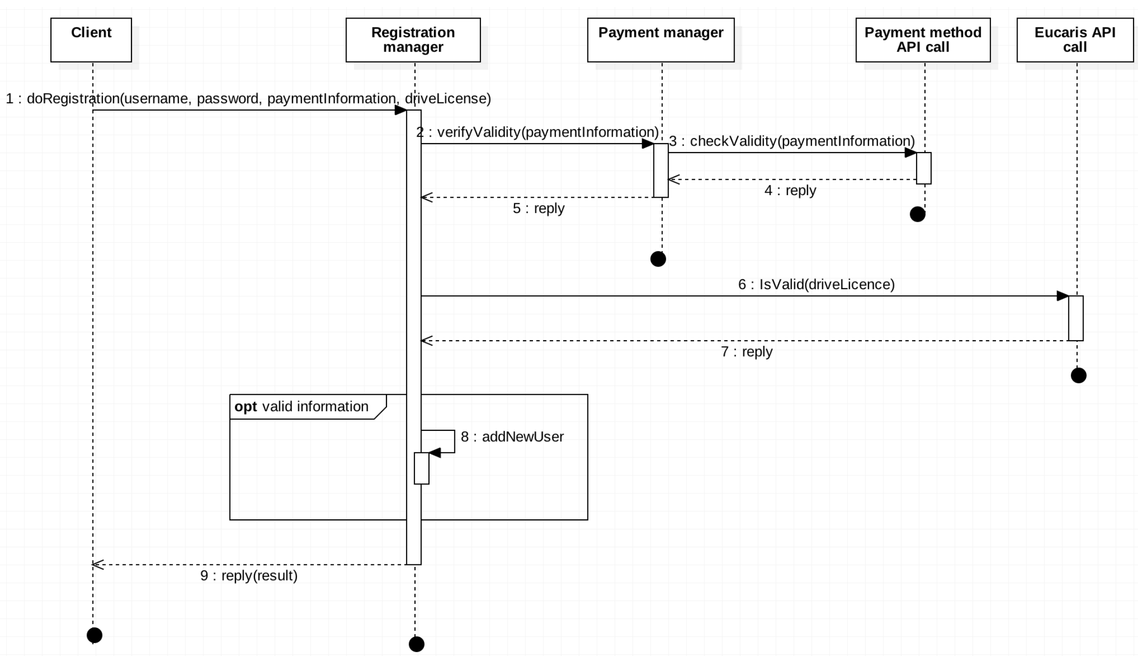

Figure 6: Registration runtime view

The client application request is sent to the registration manager component, that has to verify that the payment information and the driver license are valid. To do this, as mentioned in the product perspective section of the RASD document, external services are used. Then, if the information are valid, a new user is created and inserted in the database, and the result of the operation is sent to the user.

2.6.2 Car search

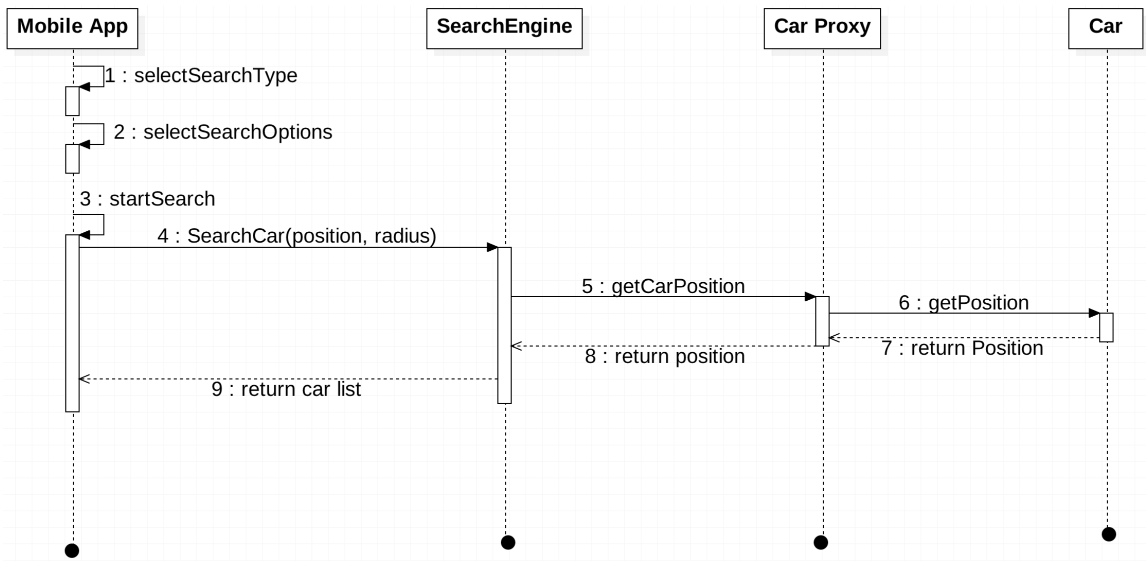

Figure 7: Car search runtime view

The user through the application insert the information needed in order to do the research, then the information are sent to the search engine that performs the research using the algorithm described in the algorithm section of this document. The position of the cars are obtained interrogating the car proxy component.

2.6.3 Reservation

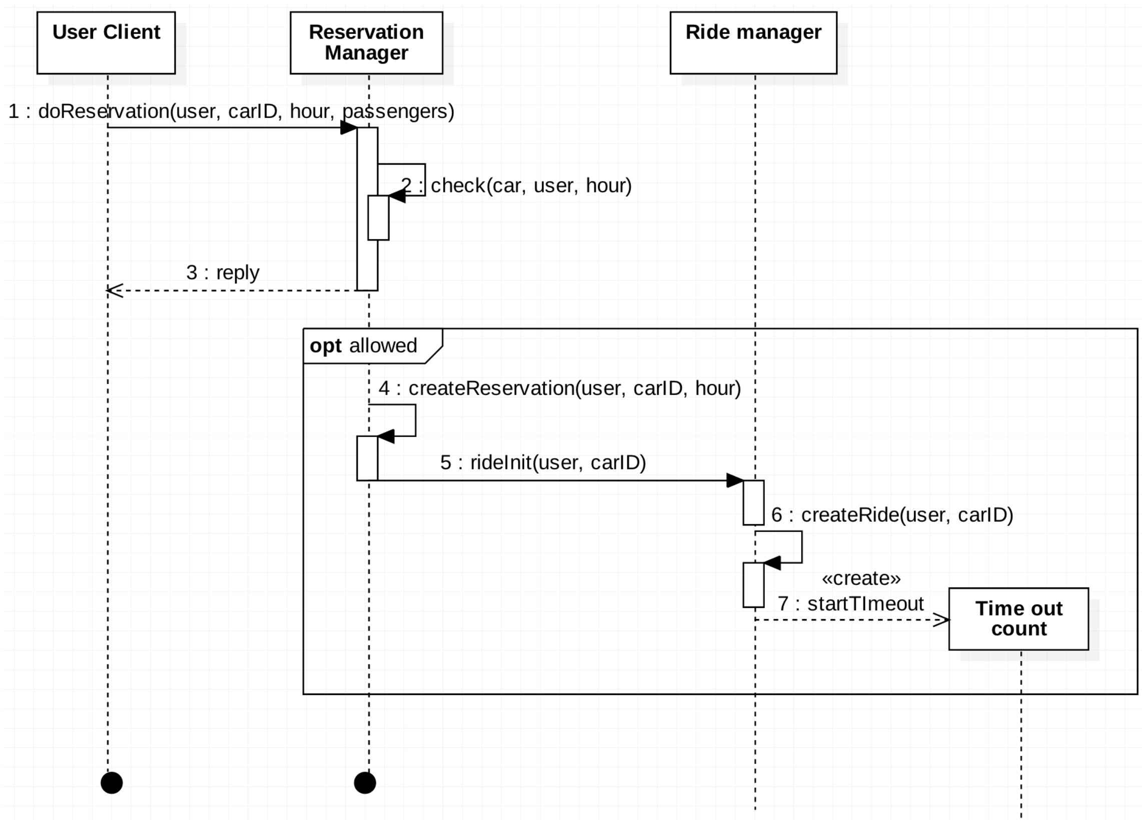

Figure 8: Car reservation runtime view

When the request is received the reservation manager check if it the user can reserve the car identified by carID, and sends back the outcome. Then if the reservation is allowed, the reservation manager delegates to the ride manager the managing of the car picking up.

2.6.4 Unlock car

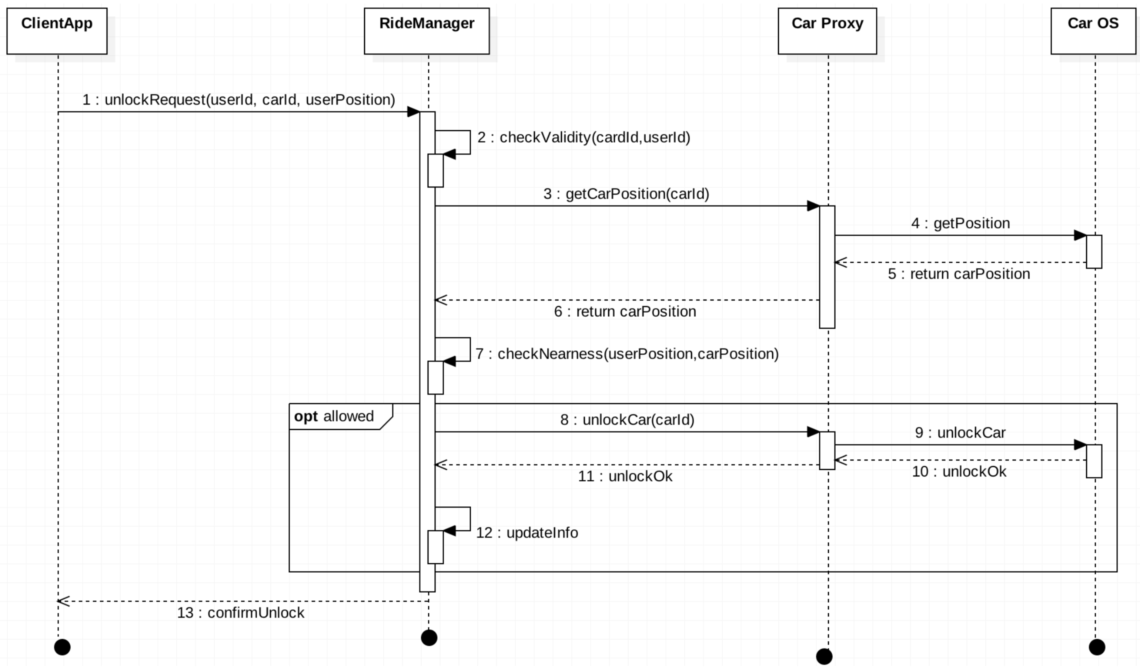

Figure 9: Unlock car runtime view

When a client makes the unlock request, the system verifies that the request owner is allowed to unlock the specified car. This operation is carried out checking two things: that the user has actually reserved the car, and that it is sufficiently close to that car. Then through the car proxy the car is unlocked, and the outcome is sent to the user.

2.6.5 End ride

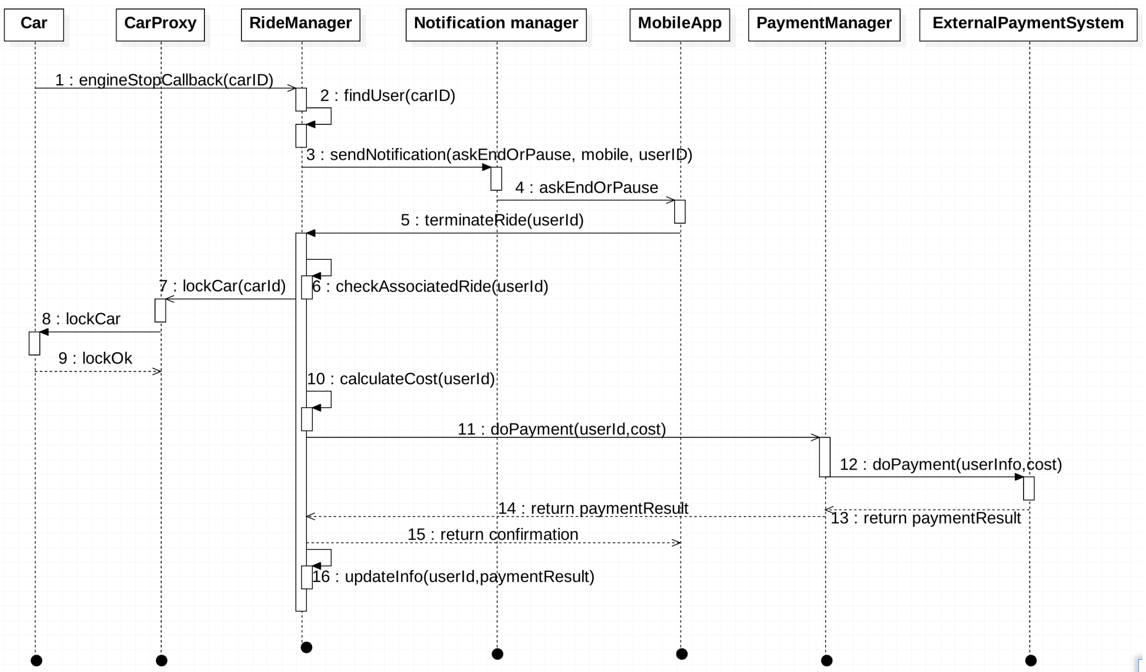

Figure 10: End ride runtime view

The stopEngineCallback remote method is called by the car system on the subcomponent of the ride manager (car listener) when the stop of the engine is detected. The call contains as parameter the cadID, so the ride manager component can find in the persistent data the user that is riding the car. In this way the system can ask to a user that has stop the engine if he/she wants to end the ride, or put it in pause (it is necessary to manage the situations in which there is no reply). Then if the ride is finished, the payment can be carried out through the payment method API call.

2.6.6 Car monitoring

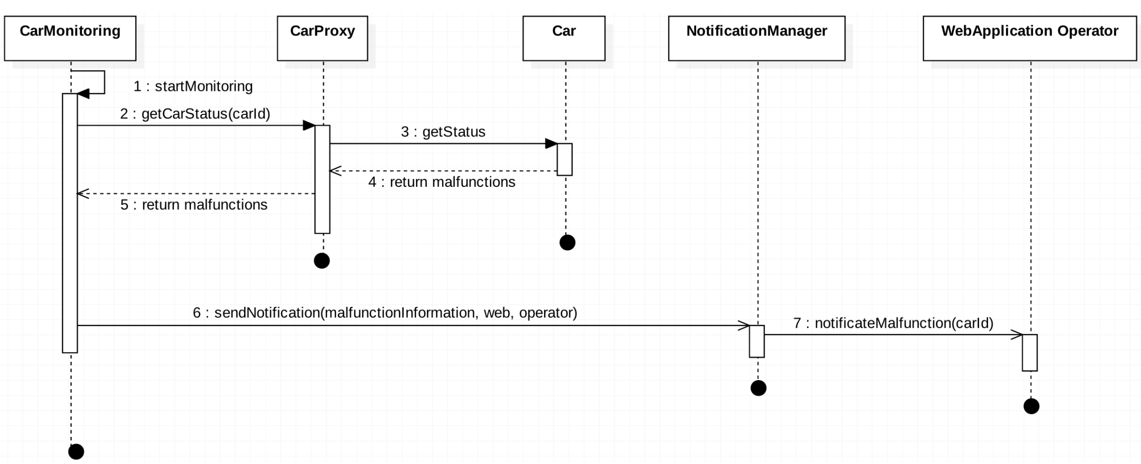

Figure 11: Car monitoring runtime view

Here is explained how the system try to keep cars in a good state. The car monitoring component makes periodic requests to the cars system, asking if the system needs some kind of operators intervention. If for instance a malfunction is detected, or the battery level is too low, the central system can notify the operator that is working in that area.

2.6.7 Communication of a malfunction

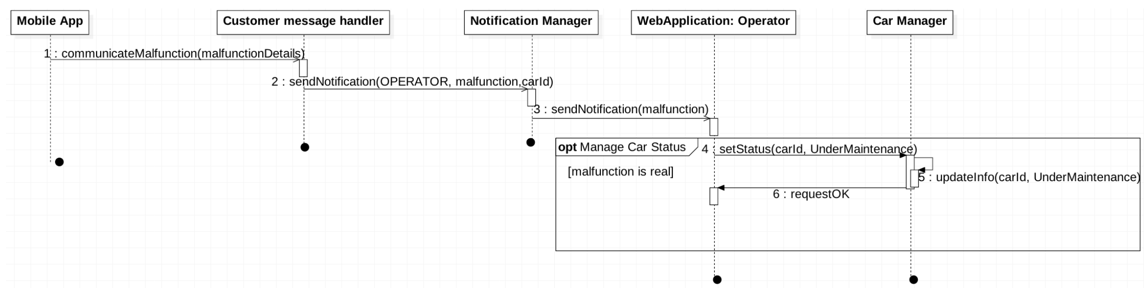

Figure 12: Communication of a malfunction runtime view

In this diagram is explained how a user can communicate to the system eventual cars problem. Furthermore, when an operator decides to work on a car, it can set its status to “under maintenance”, so no user can reserve it.

2.6.8 Car proxy

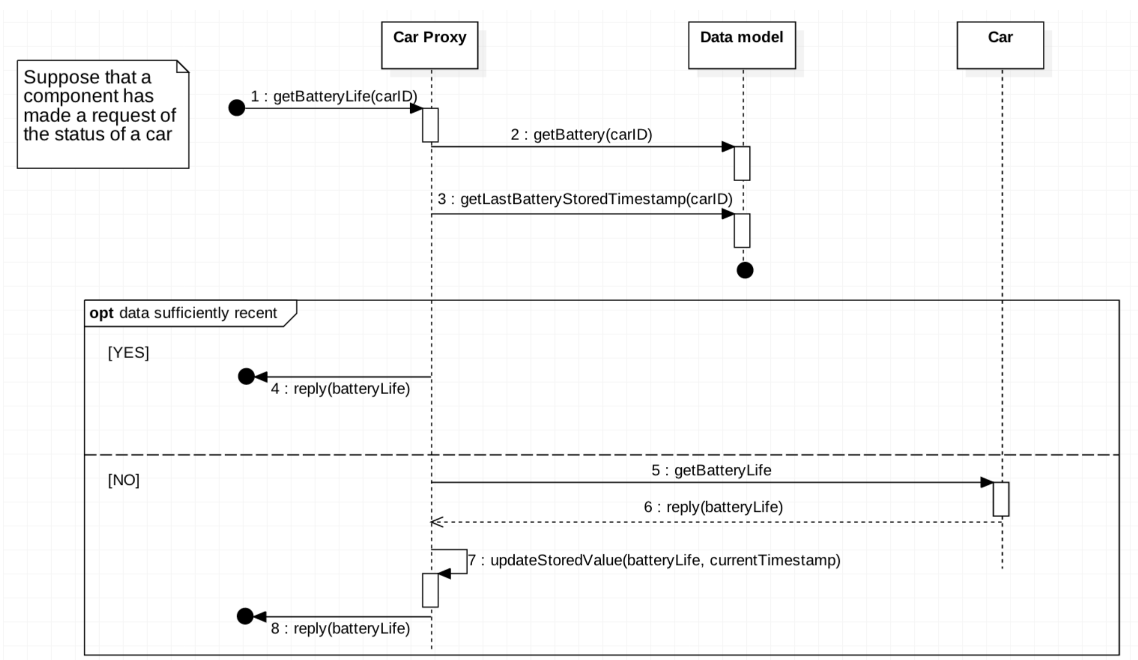

Figure 13: Example of car proxy runtime view

In this diagram is shown an example of the car proxy functioning. Attention was paid in particular on how it checks, when a request to a car should be done, if the stored value is sufficiently recent or not. In this way it can decides if it is necessary to make an actual remote request, or if the cached value can be used.

2.7 Component Interfaces

Components provide some interfaces through which it's possible to other components to use the services provided.
Some interfaces are protected and are accessible only to specific components.
Here is the list of components and their interfaces.

Registration Manager:

- doRegistration(username, password, paymentInformation, driveLicense): creates a new user if possible.

Reservation Manager:

- doReservation(userId, reservationInfo): creates a new reservation and starts the timer.

Ride Manager:

- unlockRequest(userId, carId, userPosition): calls unlock interface provided by Car Proxy
- lockRequest(userId, carId): the same as before.
- pauseRide(ride): called when the user wants to pause a ride.
- terminateRide(ride): called when the user wants to terminate a ride.
- rideInit(ride): called by Reservation Manager when the user creates the ride.

Profile Manager:

- getProfileInfo(userId): returns the profile information associated to userId.

Customer Message Handler:

- sendCommunication(communication): called when the user wants to send a communication regarding malfunctions or damages to the system.

Authentication Manager

- doLogin(username, password): manages login and sessions.

Notification Manager

- sendNotification(notificationInfo, notificationType, receiver): sends a notification of type notificationType with the content notificationInfo.

Car Search Engine

- searchCars(position, radius): searches for cars near the position.

Car Manager

- setCarStatus(carId, status): set the status of car
- getCarStatus(carId): get the status of the car (i.e. available, busy)
- unlockCar(carId)
- lockCar(userId)

Administrative Functionality Provider

- addSafeArea(position, safeAreaType): adds a safe area. SafeAreaType stands for simple safe area or charging area.
- deleteSafeArea(safeAreaId)
- addOperator(username, password, competenceArea)
- deleteOperator(username, password)
- addAdmin(username, password)
- deleteAdmin(username, password)

Car Monitoring: it doesn't provide any interface to other components.

Car Proxy:

- lockCar(carId)
- unlockCar(carId)
- getCarStatus(carId)

Payment Manager:

- doPayment(paymentInformation, price): it calls the external payment API in order to complete the payment.
- verifyValidity(paymentInformation): it calls the external payment API to check if the information are valid.

Notification Receiver: this interface is provided by both Web Application and Mobile application in order to receive notifications.

Car Listener:

- startEngineCallback(carId): called by the carOS when the engine starts. It's a callback method in order to be informed by important event of the car.
- stopEngineCallback(carId): the same as above.

2.8 Selected architectural styles and patterns

2.8.1 Tiers

The system will be divided into 5 tiers:

1. Database Server
2. Application Server
3. Web Server
4. Mobile Application
5. Car

Note that here the car is considered as a tier but actually is a blackbox capable of receiving request from our system and to call some interfaces provided by components of the system.
This division is in order not to overload any machine. Application Server and Web Server are deployed on different machines because if, due to great load, there will be the need of another Application Server will be easier to instantiate another machine.

2.8.2 Layers

As said before the system will be divided into 3 Layers:

- Presentation Layer: it's distributed on the mobile application or on the Web Server.
- Business Layer: it's on the Application Server.
- Data Layer: it's on the Database Server.

It is important to divide software in layers for decoupling and for dividing responsibilities. In this way different developers can focus on different tasks abstracting from other layers. Diving in layers is also important for mental clearness because in the case of a problem it should be clear where the problem is located.

2.8.3 Protocols

In this section is described how different tier communicate with each other and how they exchange data.

JDBC

Used by the Application Server to communicate with the database server. JDBC API is the standard for database-independent connectivity between the Java programming language and a wide range of databases SQL databases and other tabular data sources. The JDBC API provides a call-level API for SQL-based database access.

REST API

Used by both Mobile Application and Web Application to access the services provided by Application Server component. Rest stands for Representation State Transfer. It relies on a stateless, client-server, cacheable communication protocol and the HTTP protocol is used. The Application use HTTP request to post data, read data and delete data. Rest approach permits the fact that the server and the client can interact in complex ways without the client knowing anything beforehand about the server and the resources it hosts.

2.8.4 Design Patterns

Client-Server

Client server is the base of the architecture. The central system is the server and provides services to both web application and mobile application. The central server communicates with car that acts as a server with respect to central system.

Monolithic

The software to be follows the Monolithic pattern. Particular attention is given to decoupling and modularization. In this way the refactoring to micro-services will be easy if there will be the need of scalability. With the attention to decoupling a cloud approach it's possible and it's possible to deploy different part of the system on different machines.
This pattern has been decided because of its benefits:

- Simple to Develop: the goal of current development tools and IDEs is to support the development of monolithic applications.
- Simple to Deploy: you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime.

- Simple to Scale: you can scale the application by running multiple copies of the application behind a load balancer.

However once application becomes large this approach has a number of drawback so there's the need to decoupling components as much as possible in order to make possible the "migration" to micro-services. Most of well known internet service have initially started with a monolithic architecture and then migrated to a micro-services architecture.

Proxy

Proxy Pattern is used to communicate with the car. Car Proxy is the abstraction of the car in the system. When a component needs to communicate with the car, it calls the method provided by car proxy. Then car proxy decides whether to call car API or to get information from the DB.
Proxy The intent of the proxy is:

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

Observer

Observer Pattern is used to get callback from car's important event like engine start or engine stop. CarObserver is instantiated when the user unlocks the car and the car is informed that it needs to call the observer when an important event takes place.

3 Algorithm Design

In this section are described the most important algorithm of the software to be.
Algorithms are divided by component in order to disambiguate functions with the same name but executed by different component.
Algorithms are written in a sort of pseudocode to maintain an high level of abstraction and to make them more clear and readable.

3.1 Search Engine Algorithms

Search Cars

```
// function of search engine
// optionSearch is the option related to the search
// userPosition in the current position of user
Function searchCar(OptionSearch optionSearch, Position userPosition){
    // list of cars to return
    List toReturnCars;
    // get max distance of car to user from optionSearch
    var maxDistance = optionSearch.distance;
    // get all cars from database
    List cars = DB.getCars();
    For Car car in cars {
        var carPosition = CarProxy.getCarPosition(car.id);
        var carToUser = calculateDistance(carPosition,userPosition);
        // if distance is lower than selected distance, add car to
            toReturnCars
        if(carToUser<maxDistance)
            toReturnCars.add(car);
        }

    return toReturnCars;
}
```

3.2 Ride Manager Algorithms

Calculate Cost

```
// function of ride manager
Function calculateCost(Ride ride){
    // calculate cost due to time
    var timeCost = ride.duration*Constants.costPerMinute;
    // get max discount from ride object
    var discount = ride.maxDiscount;
    // get fee related to ride
    var fee = ride.fee;
    // apply discount and fee to ride cost
    var realCost = applyDiscountAndFee (timeCost,discount,fee);
    // pass all to payment manager
    PaymentManager.doPayment(realCost,ride.user);
}
```

Unlock Car

```
// function of ride manager
Function unlockCar(User user, String carID, Position userPosition){

    // get car position from carProxy
    var carPosition = CarProxy.getCarPosition(carID);

    If car is not reserved by user
        return Error

    If carPosition and userPosition Are near
        //create a new ride
        ride = new Ride(carID,user);
        ride.setState(CREATED);
        // ask car proxy to unlock car
        CarProxy.unlockCar(carID);
        // register an observer to the car
        CarProxy.registerObserver(carID,new CarObserver());

    Else return Error
}
```

3.3 Car Observer Algorithms

Engine Start Callback

```
// callback of carobserver called by car when engine starts
Function engineStartCallback(String carID){

    ride = get ride related to car;

    //set started state to ride
    ride.setState(STARTED);
}
```

Engine Stop Callback

```
//callback of carObserver called by car when engine stops
Function engineStopCallback(String carID){
    same as engineStart but set state STOPPED
}
```

3.4 Reservation Manager Algorithms

Do Reservation

```
//function of reservation manager
Function doReservation(User user, String carID, Hour hour, int
    passengers){
    // if user has already some reservation can't reserve a car
    If user.currentReservation>0
        return Error

    // if car is reserved or is busy return error
    If car.isReserved or car.isBusy
        return Error
```

```

    // if user is suspended, it can't do a reservation
    If user.isSuspended
        return Error

    //  users must be able to reserve a car for up to one hour before
       they pick it up.
    If Time.currentHour+1>hour
        return Error

    user.reservedCar = car;
    create new Reservation(car,user,hour,passengers);
    update DB information;
}
```

3.5 Car Proxy

Get Car Status

```
// function of car proxy called by other components
Function getCarStatus(String carID){

    // get the time of last update from the db
    lastStatusUpdate = get last update of car from DB;

    // if the last update is sufficiently recenent return that status
    // Cache function
    If lastStatusUpdate is recent
        return status from DB;

    // if the last update is not recent the status could have changed
       so request status to the car
    If lastStatusUpdate is not recent
        get ip of the car;
        create new status request to car;
        wait car response;
        status = car response;
        return status;
}
```

Unlock Car

```
// function of car proxy called by other components when they need to
   unlock a car
Function unlockCar (String carId){
    get ip of the car;
    create a new unlock request to car;
    wait car response;
    return response;
}
```

4 User Interface Design

Mockups have already been done in RASD document in the section regarding the Non-Functional requirements.

4.1 User Experience Diagrams

In this section the navigation flow of the user in the application is explained.

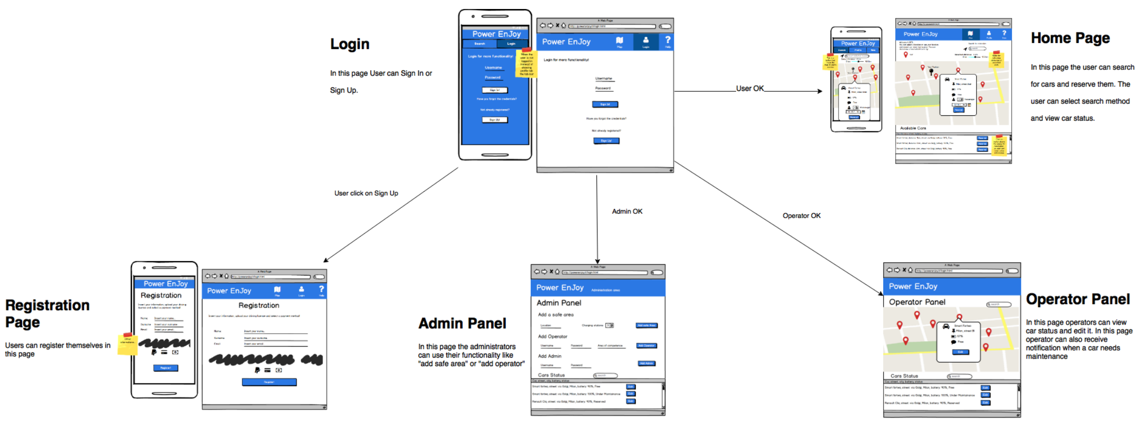

Figure 14: Login Diagram

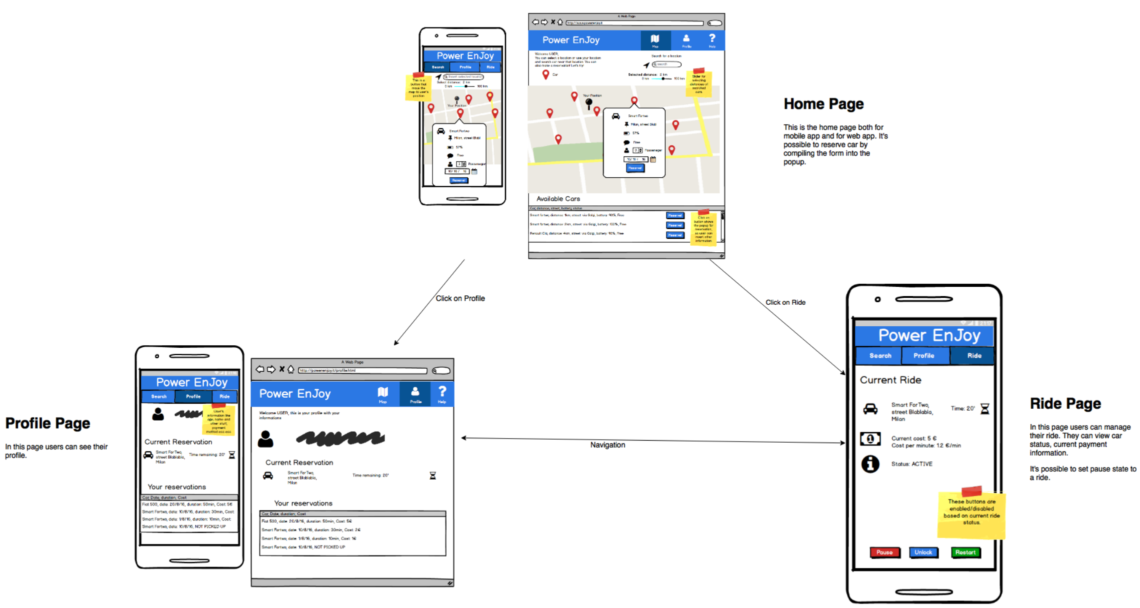

Figure 15: Navigation Flow

figure 14 and figure 15 show the high level navigation flow of the user in the application.

The next figures show the mapping between screens, functionality and related components. In this way the functions provided by each component are clear and understandable. It's useful also for developers in order to understand dependencies between functions and components.

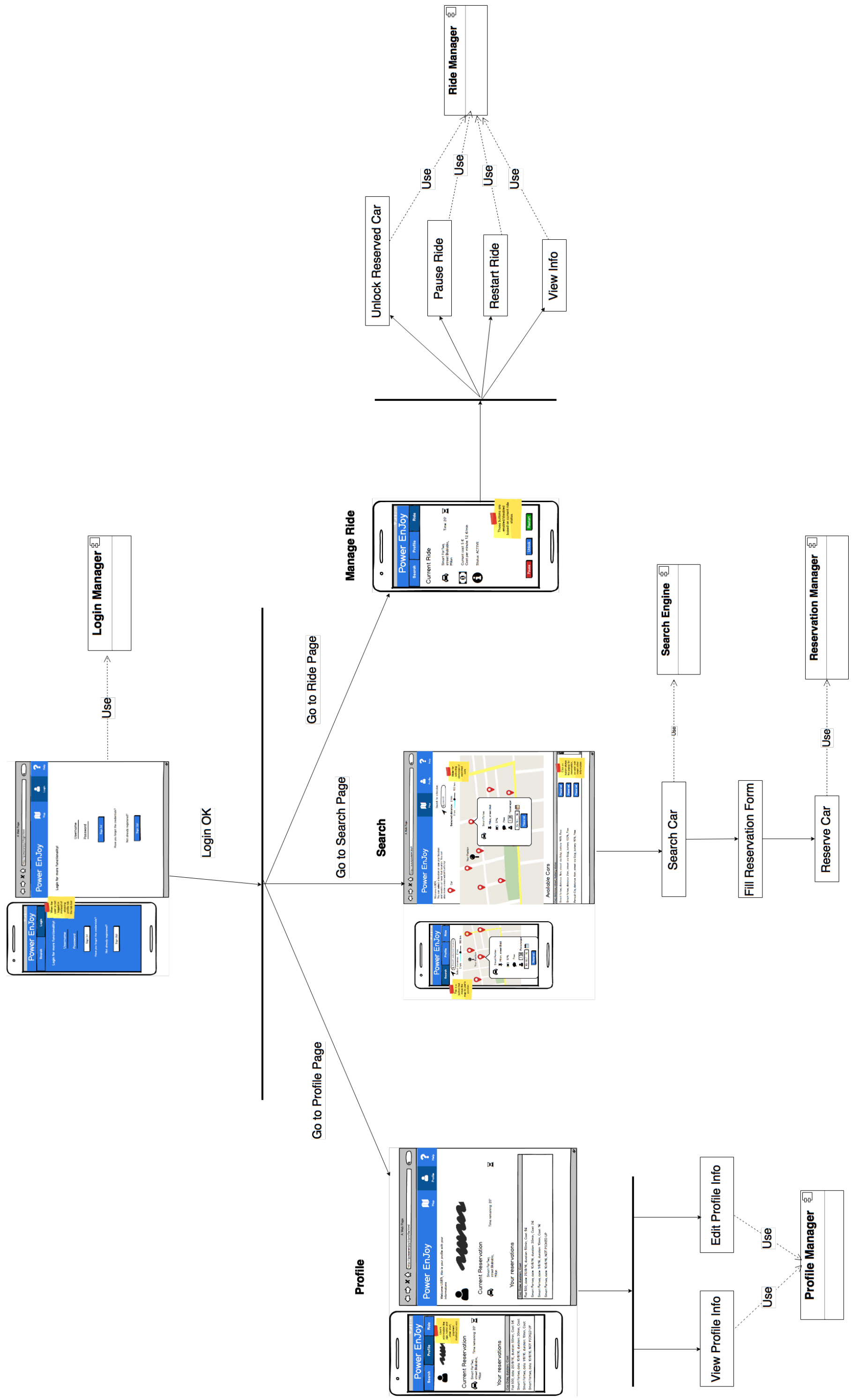

Figure 16: Navigation Flow Detailed

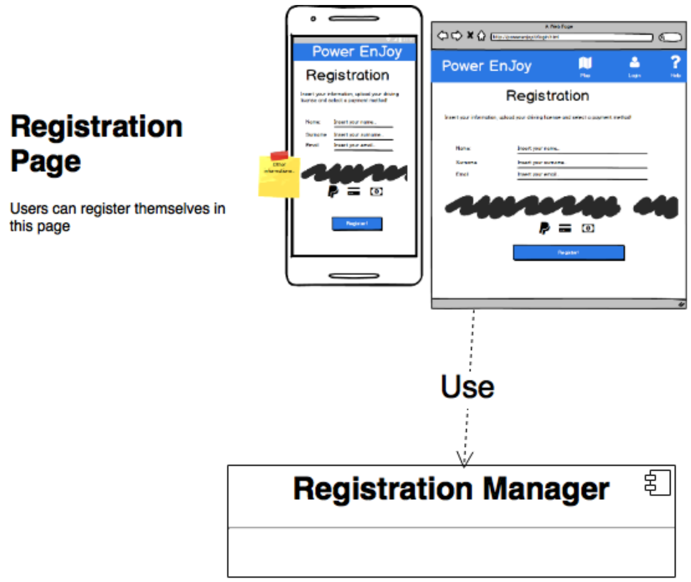

Figure 17: Registration Screen

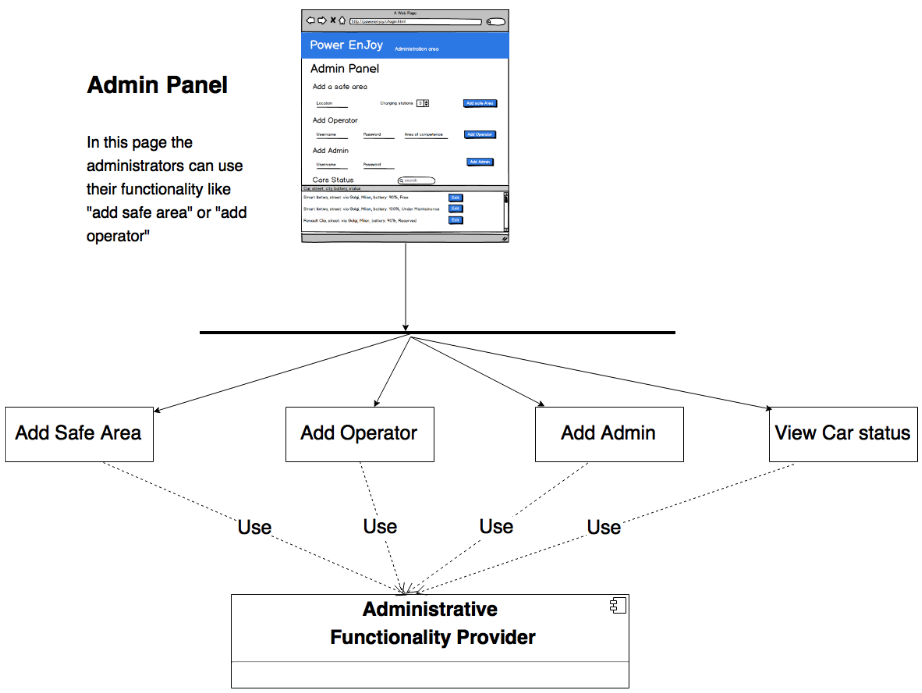

Figure 18: Administrator Page

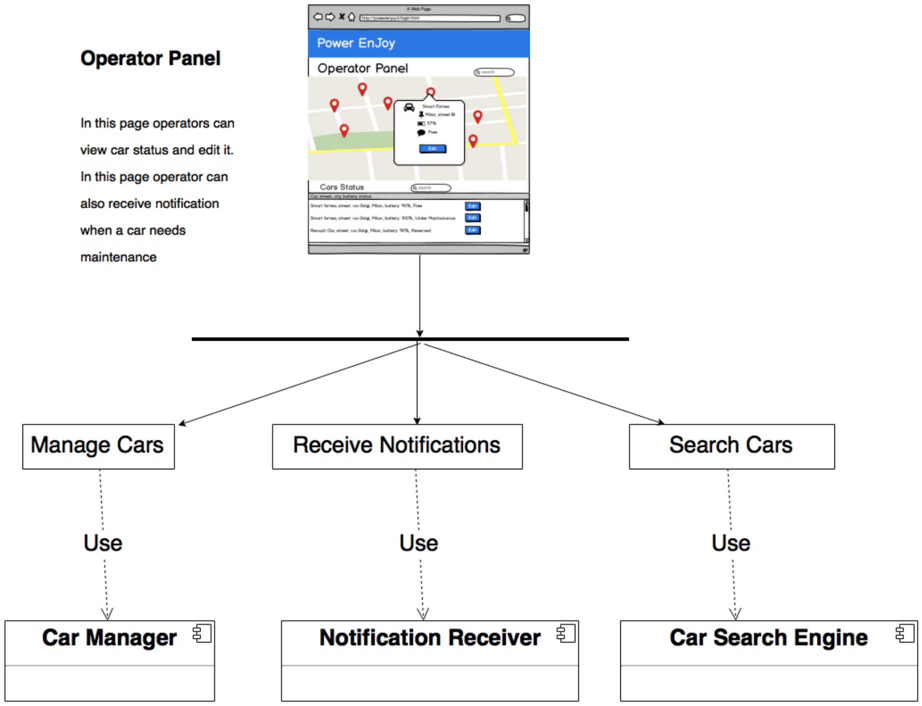

Figure 19: Operator Page

Figures 16,17,18,19, are not standard UX Diagrams because there isn't only a focus on the navigation flow of the user in the application. The focus is also on functionality and components. This approach has been used because the navigation flow of the user can change in the future but the mapping between functionality and components needs to be clear at this level of analysis.

Figure 16 is the navigation flow of the user. Here there is the profile page, the search page and the ride page.

In the profile page the user can see his/her information and edit them using the Profile Manager Component.

In the search page the user can search cars and reserve them by filling a form. Search is performed by the Search Car Engine.

In the ride page the user can manage its ride by pausing or restarting it. In this page the user can also unlock the reserved car. These functionality regarding the ride are performed by the Ride Manager.

Figure 17 is the registration screen that uses the Registration Manager component.

Figure 18 is the Administrator Page. In this panel administrators can use their functionalities. The Administrator Functionality Provider is the component responsible for them.

Figure 19 is the Operator Page. Operators can manage cars by the Car Manager Component. Operators can also receive notifications through the Notification Receiver Component.

5 Requirements Traceability

In this section is described how requirements defined in the RASD are mapped to components described in this section. The set of components have to fulfill all the requirements defined in the RASD.
It is understood that Mobile Application and CarOS are used in interactions with users and cars so here aren't listed in order to avoid useless repetition. Also the Persistency Manager is used in order to keep coherency with data.
Here there is a list of all goals of out system (component here are associated to goals but indirectly to all requirements that ensure the fulfillment of the goal).

- [**G1.1**]Only user should be able to reserve a car.
  - **Authentication Manager**: manages user logins and sessions.
  - **Reservation Manager**: manages reservation and allows only registered user to make a reservation.
  - **Registration Manager**: allows the registration of guests and manages all information related to registration.
- [**G1.2**]User should be able to unlock reserved car when they are close to it.
  - **Ride Manager**: allows user to unlock a car.
  - **Reservation Manager**: allows user to reserve a car.
  - **Car Proxy**: allows communication with car.
- [**G1.3**]User should be aware of how much they are going to pay during the ride.
  - **Ride Manager**: communicates to the car the current ride cost.
  - **Car Proxy**: same as before.
- [**G1.4**]If reserved car pass under maintenance, the user that made the reservation must be notified.
  - **Car Monitoring**: monitors periodically the status of the cars and calls notification manager.
  - **Notification Manager**: notifies user and operators.
  - **Car Proxy**
- [**G1.5**]User should be able to set pause status during a ride.
  - **Ride Manager**: manages all stuff related to the ride.
- [**G1.6**]User should be able to restart a paused ride.
  - **Ride Manager**
- [**G2**]The reservation of two (or more) cars at a time must be forbidden.
  - **Reservation Manager**
- [**G3**]Users and Guests must be able to search cars.
  - **Car Search Engine**: searches car with respect to user's preferences.
  - **Car Proxy**
- [**G4**]Induce users to keep a virtuous behavior.

  - **Ride Manager**: manages the calculation of discounts and fees related to a ride.
  - **Car Proxy**
  - **Payment Manager**: manages the payments.
  - **Reservation Manager**: if a reservation expires it manages the fee.

- [**G5**]Users have to pay an amount of money based on the ride's duration.
  - **Ride Manager**
  - **Payment Manager**

- [**G6**]Guarantee a ready maintenance of cars.
  - **Car Monitoring**
  - **Notification Manager**:notifies operators when a car needs maintenance.
  - **Car Manager**: allows operators to know car status and car positions. Allows also operators to set car as available after maintenance.
  - **Customer Message Handler**: receives messages from users about malfunctions and notifies operators through notification manager.
  - **Car Proxy**

- [**G7**]Admins must be able to manage the system.
  - **Administrative Functionality Provider**: allows administrators to manage safe areas and operators

6 Appendix

6.1 Hours of Work

Emanuele Ghelfi:

- 24/11/16: 4 h, Overview of the problem
- 29/11/16 5 h, Focus on architecture
- 30/11/16: 5 h, Diagrams
- 1/12/16: 1 h, introduction
- 2/12/16: 3 h, Architecture Overview, diagrams, algorithms, data model
- 3/12/16: 3 h, Comments on architecture, requirements traceability
- 6/12/16: 4 h, Components and interfaces
- 7/12/16: 3 h, Algorithms and refactoring

Total hours: 28 h

Emiliano Gagliardi:

- 24/11/16: 4 h, Overview of the problem
- 29/11/16 5 h, Focus on architecture
- 30/11/16: 5 h, Diagrams
- 1/12/16: 1 h, introduction
- 2/12/16: 3 h, Architecture Overview, diagrams, algorithms, data model
- 3/12/16: 3 h, Comments on architecture, requirements traceability
- 6/12/16: 4 h, Components and interfaces
- 7/12/16: 3 h, Algorithms and refactoring

Total hours: 28 h

6.2 Used Tools

The tools used to create this RASD document are:

- Github: for version control.
- Lyx: to redact and organize this document.
- StarUML: to create UML diagrams (component diagram, data model diagram, deployment diagram, sequence diagram, statechart diagram, use case diagrams).
- draw.io : to create some diagrams.

6.3 Changelog

- 1.2 removed startRide in RideManager component interface

References

[1] https://www.nginx.com

[2] http://wildfly.org

[3] http://microservices.io/patterns/microservices.html

[4] http://www.agilemodeling.com/artifacts/deploymentDiagram.html

[5] http://www.uml-diagrams.org/deployment-diagrams.html

[6] http://www.uml-diagrams.org/deployment-diagrams-overview.html

[7] http://www.uml-diagrams.org/component-diagrams.html

[8] http://www.agilemodeling.com/artifacts/componentDiagram.htm

[9] http://www.ibm.com/developerworks/rational/library/dec04/bell/

[10] http://microservices.io/patterns/monolithic.html

[11] https://sourcemaking.com/design_patterns/proxy

[12] www.stackoverflow.com