Power EnJoy

Design Document

Version 1.2.0

Redaelli Marco 877622 Zanoli Francesco 877471

27/02/2017

Contents

Figure Contents

Chapter 1

Introduction

This document contains the complete design description of *PowerEnJoy*. This includes the architectural features of the system down through details of what operations each code module will perform and the database layout. It also shows how the use cases detailed in the RASD will be implemented in the system using this design. The primary audiences of this document are the software developers but the level of the description is high enough for all the stakeholders to capture the information they need in order to decide whether the system meets their requirements or in order to begin the development work.

1.1 Scope

PowerEnJoy is a car sharing service developed to fit all the reality, as small or large city. The main goals of the system are:

- Guarantee the access of driver to the service
- Guarantee a fair management of reservation and use of electric car

The system architecture will be a four-tier architecture: client, server application, car system and server database. It will be created by using the MVC architectural pattern. The system will be divided into components with respect to the principles leading to good design:

- Each individual component will be smaller in order to be easier to understand
- Coupling will be reduce where possible
- Reusability and flexibility will be increase in order to make easier future implementation

The system will have efficient algorithm in order to increase its performance; in the document will be given special attention to the geolocalization algorithm.

1.2 Definitions, acronyms and abbreviations

In the document are often used some technical terms whose definitions are here reported:

- **Layer:** A software level in a software system.
- **Tier:** An hardware level in a software system.
- **Relational Database:** A digital database whose organisation is based on the relational model of data, as proposed by E.F. Codd in 1970.
- **Cocoa MVC:** A strict application of MVC principles.
- See the correspondent section in the **RASD** for more definitions.

For sake of brevity, some acronyms and abbreviations are used:

- **DD:** Design Document.
- **GPS:** Global Positioning System.
- **GUI:** Graphic User Interface.
- **API:** Application Programming Interface.
- **MVC:** Model View Controller.
- **ETA:** Estimated Time of Arrival.
- See the correspondent section in the **RASD** for more acronyms and abbreviations.

1.3 References

- Software Engineering 2 Project AA 2016/2017: Assignments AA 2016-2017
- ***PowerEnJoy* RASD v1.0:** Requirements Analysis and Specification Document for *PowerEnJoy*
- **IEEE Std 1016-2009:** IEEE Standard for Information Technology - Systems Design - Software Design Descriptions
- **ISO/IEC/IEEE 42010:** International Standard for Systems and software engineering - Architecture description

1.4 Document Structure

This document is essentially divided in seven main sections:

- **Introduction:** it gives a description of the document and some information about the system design and architecture.
- **Architectural Design:** This is the core of the document. It gives general information about the architectural design. It also describes how the system will be divided into components and how the components communicate. It also has a description of the design pattern and architectural styles that will be used.
- **Algorithm Design:** it gives a description of the main algorithm that will be implemented. Without using a specific language it describe the different steps the algorithm will do.
- **User Interface Design:** it gives a description of the user interfaces of the system and a detailed list of mockup
- **Requirements Traceability:** this section documents the life of a requirement and provides bi-directional traceability between various associated requirements.
- **Appendix:** it provides informations that are not considered part of the actual DD. It includes: software and tools used, project group organisation.

Chapter 2

Architectural Design

2.1 Overview

This chapter describes the software system, the relationships between software components and the interaction between the actors and the system. It also describes resources, that are all the elements used by the external component to design such a services. To do so this section is divided in three different parties:

- **Component View:** Describes the division of the system in layers and how the work flow is organized through them. It contains the UML component diagram and fast definition of all the parties
- **Deployment View:** Explains how the division of the system in Tiers is done.
- **Runtime View:** Describes with some UML Sequence diagram how each software component interacts with each other.

2.2 High level components and their interaction

This section contains all the architectural software specification. In particular it explains all the different parts displayed in the UML Component View (2.1) and link them with their dependencies. Before starting analyse all the component of the structure is important to know that the most part of the communications are made with a Client-Server style and only some are Point-to-Point, in plus the system is composed by different tiers.

- **Client:** This is the mobile application used by the users and the Computer board integrated in the car, as we consider it as another instrument for the user to communicate with the System. All the communications done by the client to the server are using the pattern Client-Server

- **Application Server:** This is the core of the system.It is responsible to manage all the user's interaction with the car and the customer service. It is composed by different parties as explain bellow
- **Database Server:** This is the memory of the system, all the interactions with this tier are made by the application server with a Point-to-Point communication
- **External Services:** To simplify the complexity of the system without deleting some functionality external services are connected with our Application Server

In particular the application server is composed by different components that are explained in this section.

2.3 Tiers

2.3.1 PowerEnJoy

Type System

Node PowerEnJoy

Description This is the core of the system and contains all the functionality provided to the user, in plus it provides API for the maintenance and the interoperability with other system and it is the main entry for every request coming from the mobile application

Dependencies Google APIs, PayPal API, Maintenance API, PowerEnJoy Database

Operations

- Account API (Sign-up, Sign-in, Login, Personal Information editing, Personal Information handling)
- Reservation API (Reservation Management, Reservation Time)
- Assistance API (Incident management from the system, Error and problem from the system)
- Rent API (Start a rent, End a rent, Bill management)
- Query Problem API (Request and Information to the Maintenance service)

2.3.2 PowerEnJoy Database

Type Database

Node PowerEnJoy Database

Description This is the part of the system devoted to store the data

Operations

- Database API
- Data storare management
- Data presentation
- Security management
- Multiuser access control
- Backup and recovery management and Data integrity management
- Data transaction management

2.3.3 Mobile App

Type Mobile App

Node Client

Description This is the interface that allow the users to interact with the system, rent car, look for near cars

Dependencies Account API,Reservation API,Rent API, Assistence API

Resources Supported devices (see RASD for further informations)

Operations

- Sign-in, Sign up, login, personal informations editing
- Reserve a car, Rent a car, Looking for a car
- Notification from service
- Assistance request

2.3.4 Car System

Type Car System

Node Client

Description This is the board computer in the car that allow the system to retriever all the needed data. This computer is also able to open and close the car and to do a fast check of the car condition in order to directly send a maintenance request to the system. Its main feature is to detect the car position in order to let the system localise the car. The communication between this computer and the system is done by an internet connection of 10 MB for month because it does not require an intense communication.

Operations

- Passenger counting.
- Open and close the car.
- Check the car condition and send notification to the central system
- Relieve the position of the car.
- Relieve the state of the battery even if it is plugged-in or not.

2.4 System Part

2.4.1 Ride Manager

Type Subsystem of PowerEnJoy

Node PowerEnJoy Server

Description This component provides all the data to calculate the total bill and all the information about the rent as the path followed, the time spent on the car and the number of passenger

Dependencies Account Manager,Data Manager, Car Manager API

Operations

- Rent API
- Reservation and Rent
- Data of the rent to calculate the bill
- Assistance request

2.4.2 Account Manager

Type Subsystem of PowerEnJoy

Node PowerEnJoy Server

Description This component manages users connected to the service. It manages sign-up, sign-in and login functionalities, profile editing. All data are stored on the database through Data API.

Dependencies Data Manager

Operations

- Account API

- Sign-up, Sign-in
- Login
- Personal Information editing

2.4.3 Car Manager

Type Subsystem of PowerEnJoy

Node PowerEnJoy Server

Description This component manages all the information and the interaction with the cars, and communicate with the other services letting they storing all the information they need. Especially with the Ride Manager who is in charge of obtaining all the informations about the number of passenger, the state of the battery and the position of the car.

Dependencies Map Service

Operations

- Reservation API
- Data of the car
- Status of the car

2.4.4 Bill Manager

Type Subsystem of PowerEnJoy

Node PowerEnJoy Server

Description It's in charge of calculate and ask for the rent payment, asking to the Ride Manager it can have all the information it needs to calculate the total cost and pretending to the external system the amount requested

Dependencies PayPal API, Ride Manager.

Operations

- Calculate total cost, apply discount and overtaxes
- Request the amount of the rent to the external payment method

2.4.5 Zone Manager

Type Subsystem of PowerEnJoy

Node PowerEnJoy Server

Description This part of the system is dedicated to manage all the parking/safe area in the system. It is in charge of counting the disposable place for each area based on the rent ended and the rent in course.

Dependencies Database API

Operations

- Retrieve the information about the parking and safe area
- Calculate the free area where an user can end the rent
- Communicate with the Ride Manager to avoid end of rent that are not correct

2.4.6 Problem Manager

Type Subsystem of PowerEnJoy

Node PowerEnJoy Server

Description This is the part of the system dedicated to the customer service, able to request assistance for some accident and to request maintenance on a particular car.

Dependencies Query Problems API

Operations

- Assistance API
- Ask for maintenance on a car
- Helping Customer with problems of the system

2.4.7 Data Manager

Type Subsystem of PowerEnJoy

Node PowerEnJoy Server

Description This component provides access to all of the data contained in the database. It provides various functions that allow entry, storage and retrieval of large quantities of information and provides ways to manage how that information is organized.

Dependencies PowerEnJoy Database

Operations

- Data API
- Data access
- Data presentation
- Data organisation

2.5 Component View

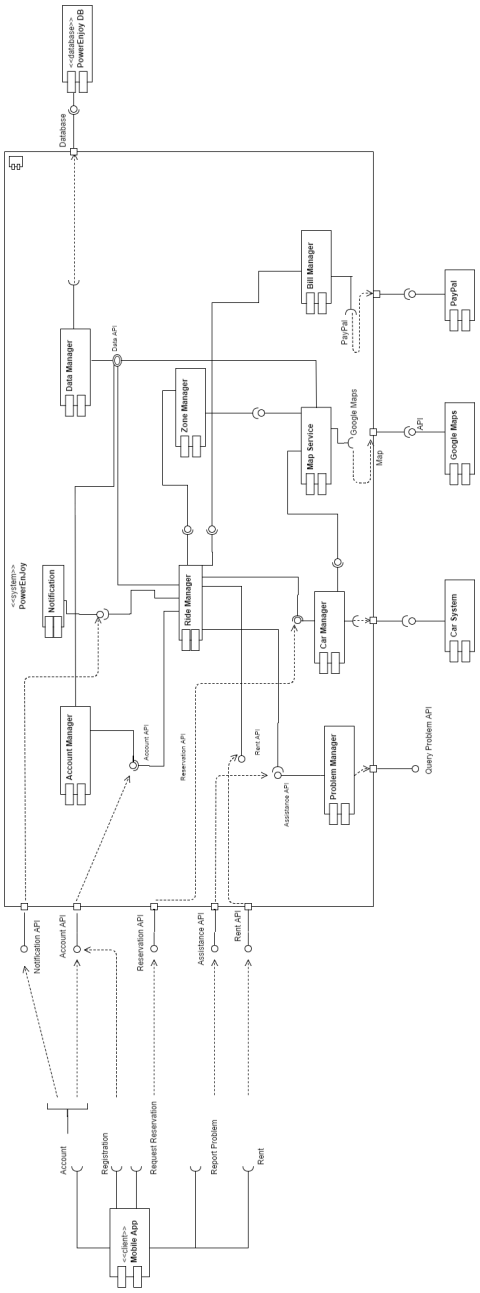

Figure 2.1: Component diagram of the entire system

2.6 Deployment view

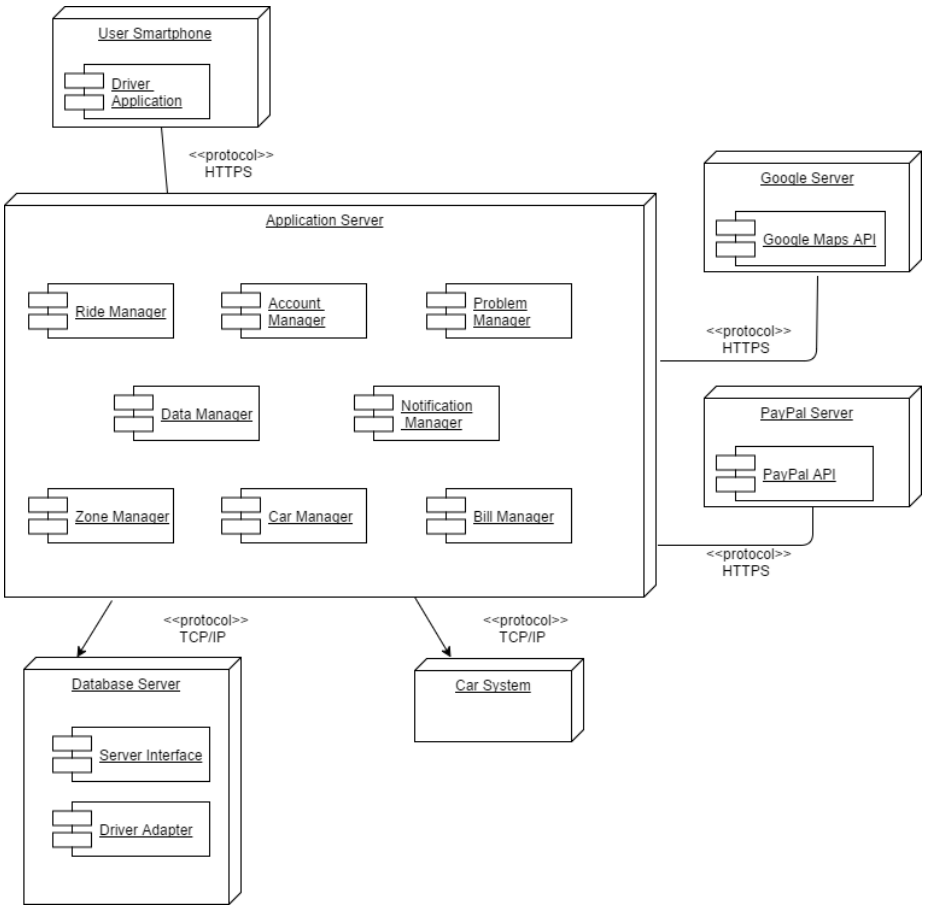

Figure 2.2: deployment diagram for PowerEnJoy

2.7 Runtime view

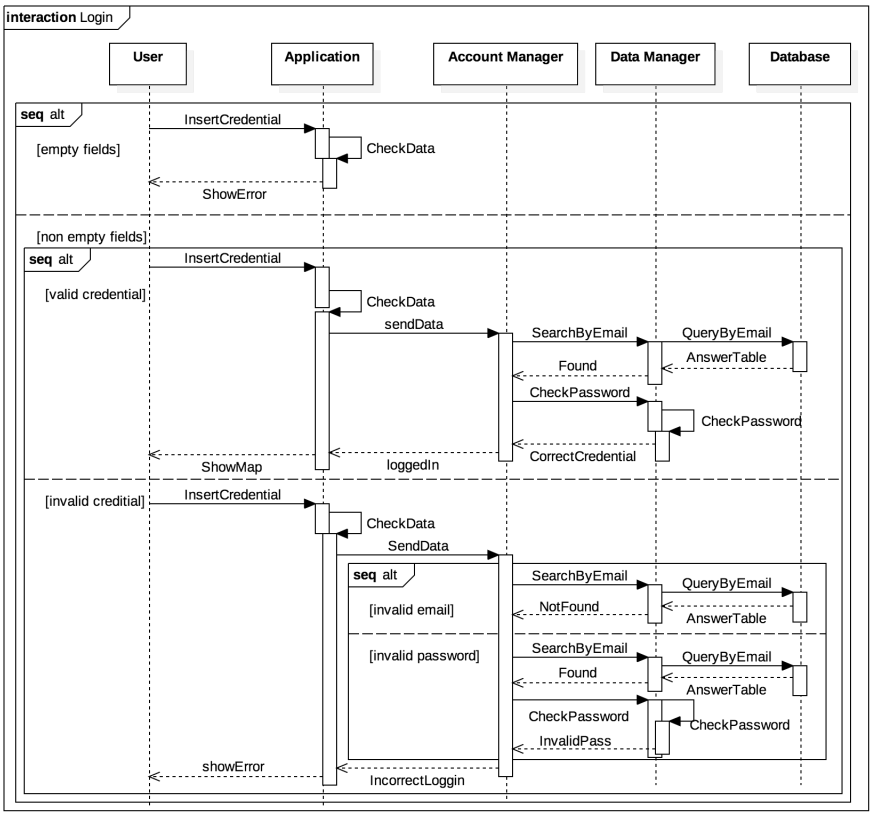

Figure 2.3: Login Sequence

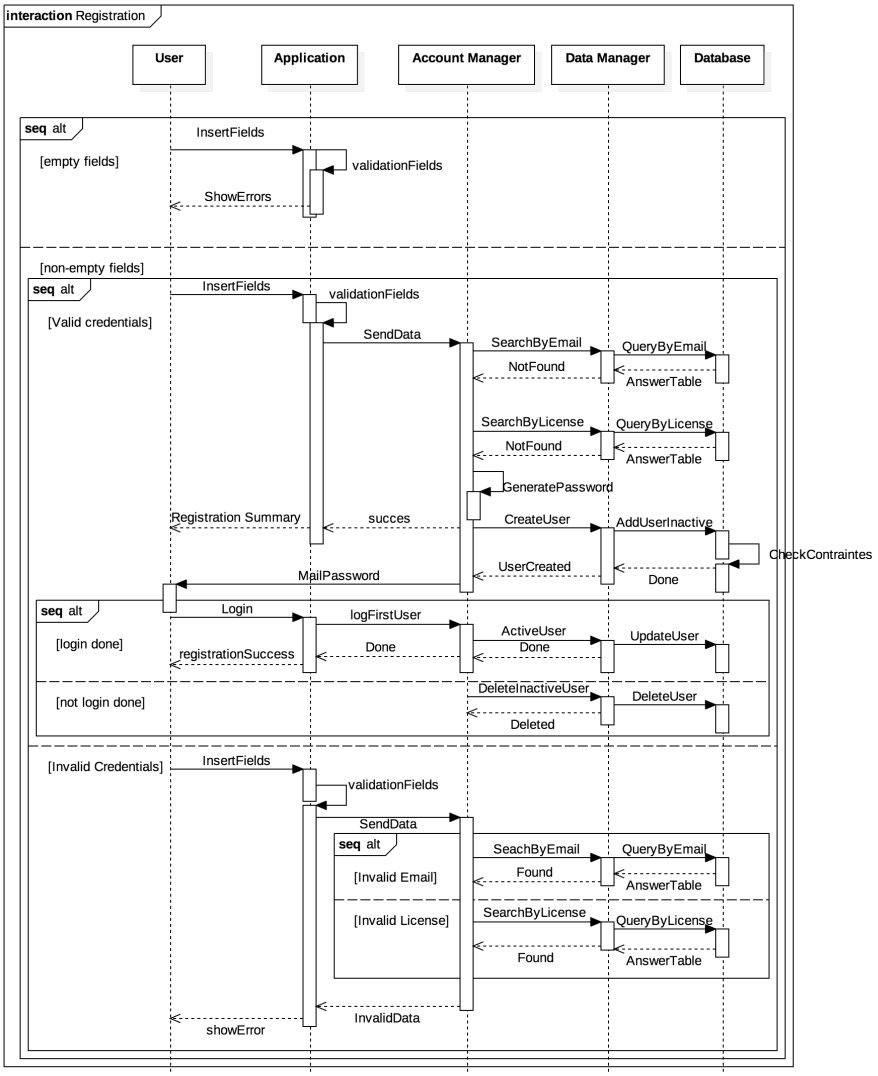

Figure 2.4: Registration Sequence

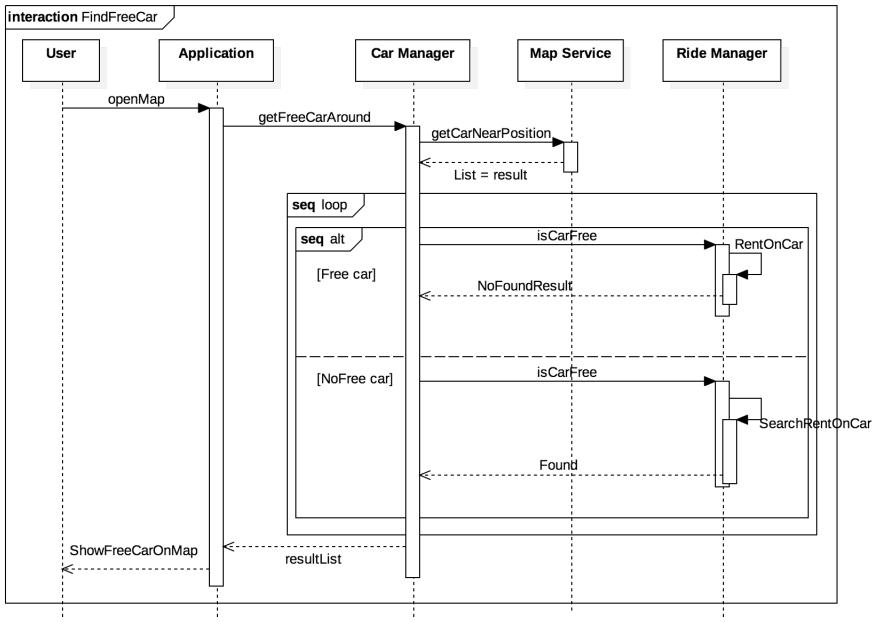

Figure 2.5: Free Car sequence

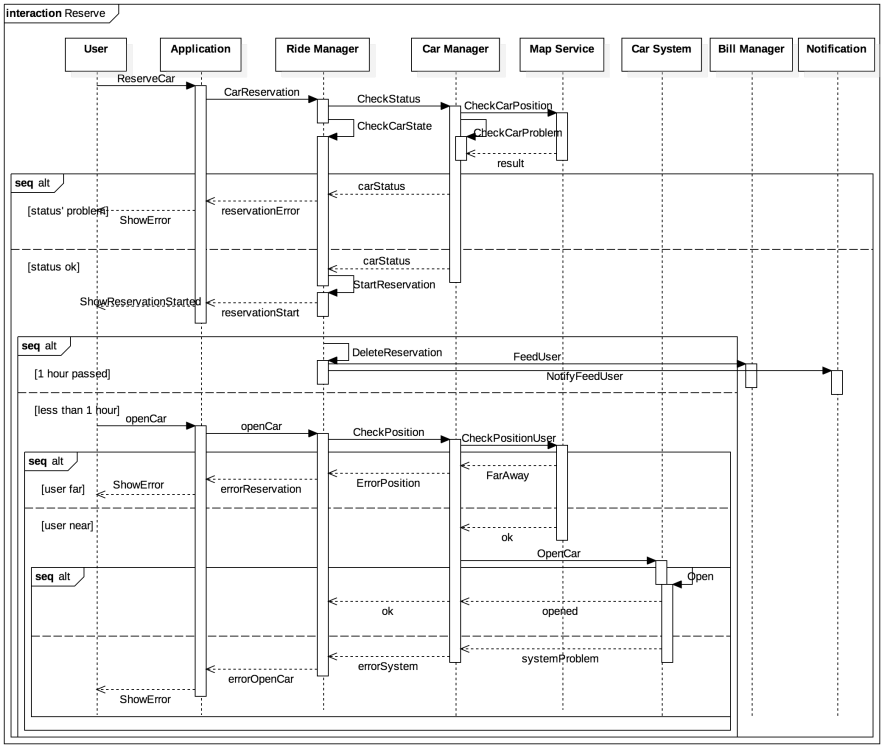

Figure 2.6: Reservation Sequence

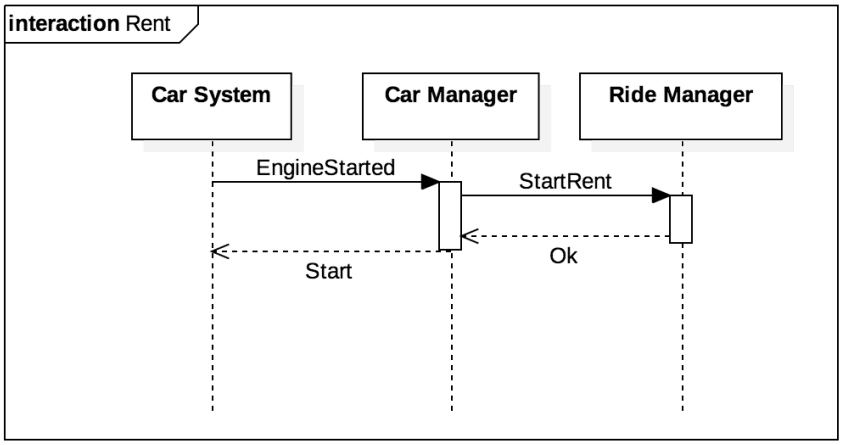

Figure 2.7: Rent Sequence

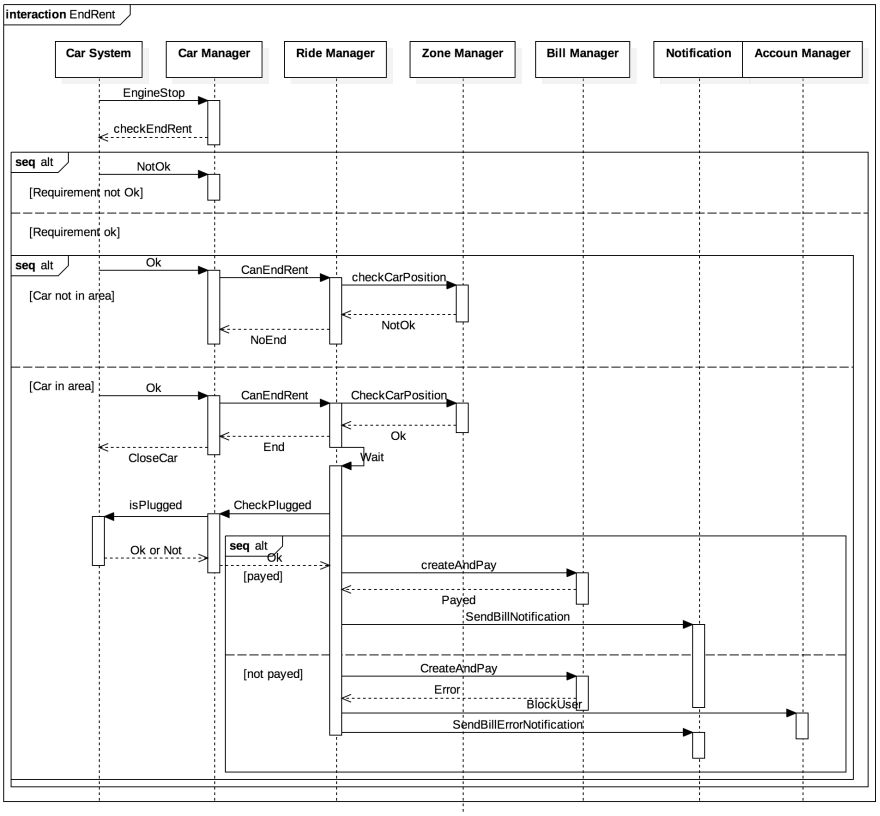

Figure 2.8: End rent sequence

2.8 Component interfaces

This section describes all interfaces between components, their interaction and their input/output parameters. For further, detailed explanations about their functioning, dependencies, resources, operations and parameters read the RASD release.

2.8.1 Application↔Client Interface

Components:

- PowerEnJoyServer (Application)
- Mobile App (Client)

Communication system: JPA, JDBC APIs;

Protocols: standard HTTPS protocol;

2.8.2 Application↔Database Interface

Components:

- Data Manager (PowerEnJoyServer) (Application)
- PowerEnjoy DB (Database)

Communication system: JPA, JDBC APIs;

Protocols: standard TCP/IP protocols;

2.8.3 Map services

Operation	**Involved Users**	**Input/Output Parameters** [type]
(General)	*(All)*	token [/] errors [/]
Zones Management	Drivers	location [Position] available car [Car]
Google Maps APIs	Drivers	address [string] location car [Position]

2.8.4 Account manager

Operation	**Involved Users**	**Input/Output Parameters** [type]
(General)	*(All)*	token [/] errors [/]
Registration	Visitors	email [string] password [string] license ID [string] .. .
Login	Visitors	email [string] password [string]
Email Confirmation	Driver	password [string]
Profile Editing	Drivers	new email [string] ...
Profile Deleting	Drivers	token [/] password [string]

2.8.5 Notification

Operation	**Involved Users**	**Input/Output Parameters** [type]
(General)	*(All)*	token [/] errors [/]
Car Reservation Notification	Drivers	car location [Position] ETA [interval] car state [boolean] ...
Bill Ride Notification	Drivers	time of rent [Time] total amount [float] discount [float] ...

2.8.6 Ride manager

Operation	**Involved Users**	**Input/Output Parameters** [type]
(General)	*(All)*	token [/] errors [/]
Ride Management	Drivers	user ID [string] start location [Position] start time [Time] end time [Time] end location [Position] num passengers [int] battery state [float] bill [float] ...
Reservation Update	Drivers	request status [enum] new request status [enum]
Request Status Update	Drivers	ride status [enum] new ride status [enum]

2.8.7 Zone manager

Operation	**Involved Users**	**Input/Output Parameters** [type]
(General)	*(All)*	token [/] errors [/]
Zone Management	Drivers	car location [Position] discount [float] car state [boolean] ...
Zone Update	Drivers	status [enum] new zone [Zone]

2.8.8 Bill manager

Operation	**Involved Users**	**Input/Output Parameters** [type]
(General)	*(All)*	token [/] errors [/]
Bill Management	Drivers	car location [Position] battery state [float] num passengers [int] time charged [float] ...
PayPal APIs	Drivers	email [string] total amount [float]

2.8.9 Car manager

Operation	**Involved Users**	**Input/Output Parameters** [type]
(General)	*(All)*	token [/] errors [/]
Car Management	Drivers	car location [Position] battery state [float] num passengers [int] time charged [float] ...
Car Update	Drivers	new car status [enum] new car location [Position]

2.8.10 Application↔Problem Manager Interface

Components:

- Problem Manager APIs
- Mobile App (Client)

Communication system: JAX-RS API (RESTful interface)

Protocols: standard HTTPS protocol;

2.8.11 Data manager

See subsection 2.6.1, *"Application↔Database Interface"*.

2.9 Selected architectural styles and patterns

Various architectural and logical choices have been justified as following:

- A **4-tier architecture** has been used: client, server application, car system and server database.
  This is due to the fact that we're developing an overall light application that doesn't need a lot of computing power, especially on the client side. Therefore, it is possible to structure the system in a logically simple and easily understandable way, without having to lose much in terms of optimization. Besides, this allows to obtain a good compromise between thin client and database tiers, and a clear correspondence between tiers and system' component.
- For similar reasons, a **SOA (Service Oriented Architecture)** has been chosen for the communication of the application server with the front ends. This improves flexibility, through modularity and a clear documentation, and simplicity, through an higher abstraction of the components.
- The entire system is designed within the principles of **Modular programming**, focusing on assigning each functionality to a different module. This greatly improves extensibility and flexibility, and allows for an easy implementation of the APIs.
- The **Client&Server** logic is the most common, simple way to manage the communication both between client and application server, and between application server and database.
- The **MVC (Model-View-Controller)** pattern, besides being a common choice in object-oriented languages, allows for a clear logical division of the various elements of the program.
- The **Adapter** pattern is largely used for portability and flexibility of the various modules.

2.10 Other Design Decision

The system uses **Google Maps** to perform all the operations related to maps, i.e. map and position visualization, geolocalization (either through GPS or user input) as well as distance, route and ETA calculations. This is an easy, fast to develop solution that relies on a worldwide, well-known and well-established software. Another third-part used by the system is the payment method furnished by **Paypal**, that allows the user and the system to manage a money transaction. This permit a fast development and a assurance about the correct end of a transaction as the third-part system is well-know and well-established as the first software.

Chapter 3

Algorithm Design

3.0.1 Bill' Algorithm

Below is represented the **Bill' algorithm**. Once the driver has stopped the car and exits the car, the system starts checking the state of sensors, the position of the car towards the position of the nearest safe area and last but not least the state of the battery (the driver has 5 minutes to eventually charge the battery and receive the discount).The events generated and their consequences are discussed in the following table.

[**Legend**]

- D: Driver;
- S: System;
- C: Car;
- B: Battery;
- LoP: List of passengers;
- SA: Safe area;

Event	***Consequences***
D exits C	$S.startChecking()$

Check the distance between the SA and the current position	**if** $sA.nearest() - D.currPos() \geq 3$ **then**
	$D.applyTax()$
	else
	if $i + k \leq maxval$ **then**
	$D.applyDiscount()$

Check the number of passengers	**if** $LoP.size() \geq 2$ **then** $D.applyDiscount()$

Check the battery state	**if** $B.getState() \leq 20$ **then**
	$D.applyTax()$
	if $B.getState() \geq 50$ **then**
	$D.applyDiscount()$

D ends the rent	$C.status \leftarrow Ready$
D has 5 minutes to charge the car and take a discount	$oldState \leftarrow B.getState()$ $wait(5)$ **if** $B.getState() \geq oldState$ **then** $D.applyDiscount()$

3.0.2 Reservation and Rent' Algorithm

Below is represented the **Reservation and Rent' algorithm**. The algorithm starts when a user clicks on the map provided by the system; immediately the controller of the system hides the selected car and starts the time of an hour (maximum amount of time that the user can wait before starting the rent). If the time exceeds the fixed constraint, the car returns available on the map, otherwise the status is "rented" (because it means that the user has pressed the start button and the ride can begin)
[Legend]

- U: User;
- R: Reservation;
- C: Car;
- B: Battery;
- S: Systen;

Event	***Consequences***
U selects car on the map	$C.status \leftarrow Reserved$ $S.hideCar()$

Check the reservation's time and compare it with the current time	**if** $S.getCurrTime() - R.time() \geq 1$ **then**
	$C.status \leftarrow Free$
	$U.applyTax()$
	$S.showCar()$
	else
	$C.status \leftarrow Rented$

Compare the positions	**if** $U.position - C.position \leq 1$ **then** $myApp.enableStartButton()$
User starts the engine	$C.startCharge()$

3.0.3 Geolocation' Algorithm

Below is represented the **Geolocation' algorithm**. Let's begin with the premise that we are imagine to build the algorithm with an object oriented Language and we're providing a pseudo-code. This algorithm checks if the given point is inside this Triangle. Infact thanks to a theorem about convex polygons, we can check if a point P is inside a given convex polygon (i.e. if the given vector associated to the point P is a convex combination of the polygon vertices). We can calculate if such coefficients exists solving a vector equation:

- P = dx * P1 + dy * P2 + dz * P3.
- P = dx*P1+ dy*P2+(1 - dx - dy)*P3.
- P - P3 = dx*(P1 - P3)+dy*(P2 - P3).

This equation can then be split into two scalar linear equations in the x and y components. The system is solved using Cramer's rule and then it is checked that alpha1 and alpha2 (and alpha3) found by solving the system satisfy the constraints.

[Legend]

- P: class Point;
- T: class Triangle;
- Z: class Zone;
- A: class Area;
- a,b,c,d,e,f: Double(or Float) values;

Event	***Consequences***
Declare variables that will allow to solve the linear equation system thanks to Cramer's method	$a \leftarrow p1.getX() - p3.getX()$ $b \leftarrow p2.getX() - p3.getX()$ $c \leftarrow p1.getY() - p3.getY()$ $d \leftarrow p2.getY() - p3.getY()$ $e \leftarrow p.getX() - p3.getX()$ $f \leftarrow p.getX() - p3.getY()$
Calculate the determinant to check the solution of the system	$tContains()*$ $d \leftarrow a * d - b * c$ **if** $d == 0$ **then** **return** $false$ $\mathrm{d}Px \leftarrow e * d - f * b$ $\mathrm{d}Py \leftarrow a * f - c * e$ $\mathrm{d}Z \leftarrow 1 - \mathrm{d}Px - \mathrm{d}Py$ $\mathrm{d}X \leftarrow \mathrm{d}Px/d$ $\mathrm{d}Y \leftarrow \mathrm{d}Py/d$
Check the results and draw conclusion	**if** $\mathrm{d}x \leq 0 \| \mathrm{d}y \leq 0 \| \mathrm{d}z \leq 0$ **then** **return** $false$ **return** $true$

Instance the Zone Class and set the Zone as a set of Triangles. Then check if a point (in our case our position) is contained in the triangle	$zContains()$
	for all Triangle t : triangles **do**
	  if t.tContains(p) **then**
	  return $true$
	  return $false$

Instance the Area Class and set the Area as a set of Zone. Then check if a point is contained in the zone. This allows us to determine in which zone is our point	$Point(latitude, longitude)$
	for all Zone z : zones **do**
	  if z.zContains(p) **then**
	  return z
	  return $null$

*the method tContains() should be defined in the Triangle Class as with for zContains() method that will be inside the Zone class

Chapter 4

User Interface Design

- Login Page

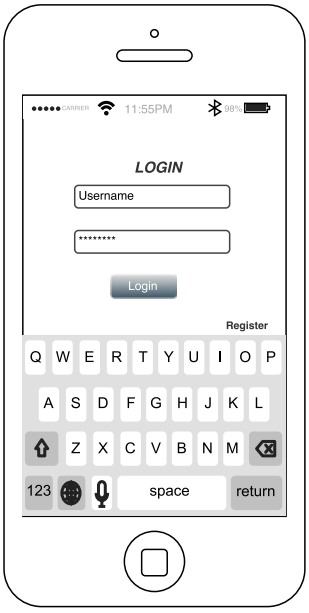

Figure 4.1: Login page

- Registration page

Figure 4.2: Registration page

- Registration page second step

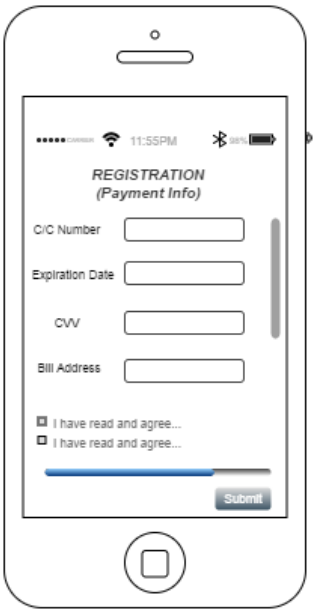

Figure 4.3: Registration page second part

- Home page or Map page

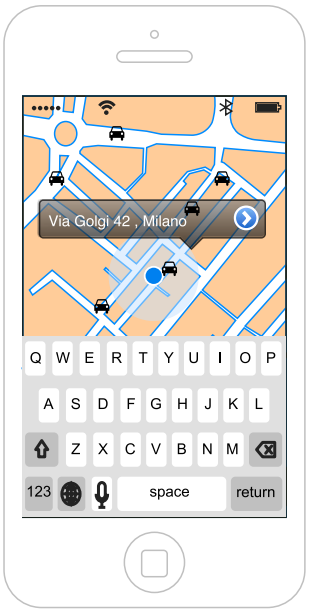

Figure 4.4: Home page or Map page

- Car information page

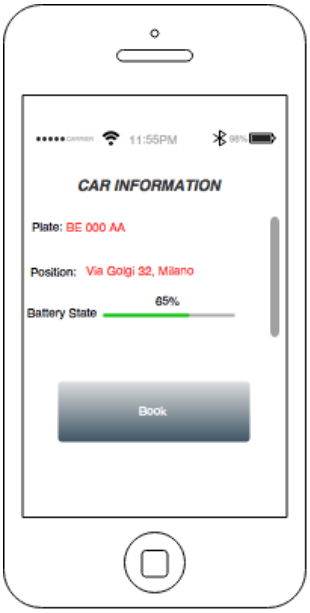

Figure 4.5: Car information page

- Car Reservation

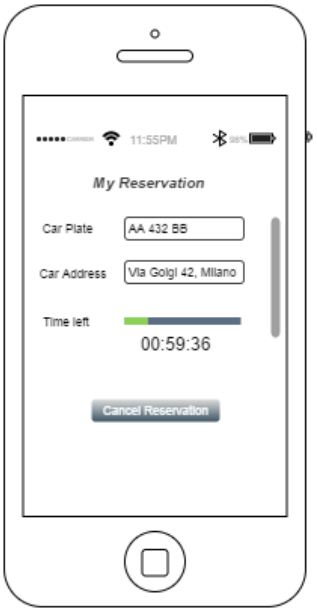

Figure 4.6: Car reservation page

- Edit Profile

Figure 4.7: Edit profile page

- End Rent

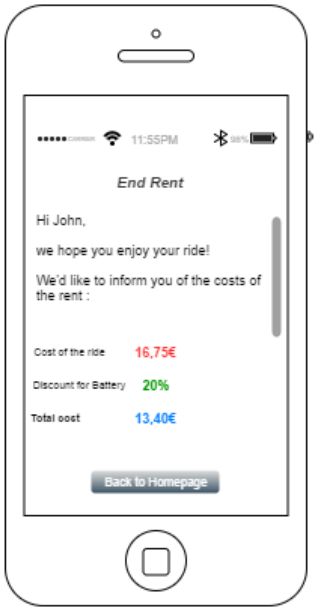

Figure 4.8: End rent page

Chapter 5

Requirements Traceability

5.1 Requirements Traceability

In this section are mapped all the functional requirements identified in the **RASD**, grouped by the **Use Case** they refer to.

5.1.1 Registration

Functional Requirements	***Design Elements***
FR1	**Account manager** does not consider unfulfilled registrations. No data is stored.
FR2	The **Account manager** consider a valid registered user (allowed to rent a car) after the first login, if the status of the user stays "inactive" for more than one day after the registration date the **Account manager** delete the data about this user.
FR3	**Account manager** does not consider unfulfilled registrations.
FR4	**Account manager** ensures the uniqueness of users.
FR5	Validation of all the form is done by the **Mobile Application** even the photos taken from the camera and not from the gallery

5.1.2 Login

Functional Requirements	***Design Elements***
FR6, FR7	**Account Manager** check the existence of the user's email and the correspondent password before allowing the user to log-into the system
FR8	Blank fields let the **Client Application** to ignore the login.
FR9	The **Client Application** provide this functionality and the **Account Manager** takes care of the procedure to follow to allow the user have another name
FR10,FR11,FR12	The **Account Manager** take only exiting user on the database and handle all the procedure to restore a password, from the generation of the new one until the sent of the mail with the new access code.
FR13	The **Account Manager** is allowed to generate only one password a day for one user.
FR14	The **Client Application** remeber email and password after a login until the user do a logout. It's also provide the email and password every time the system requires its.

5.1.3 Reserve a Car

Functional Requirements	***Design Elements***
FR15	The **Car Manager** change the state of the car according with the **Ride Manager** when it relieves that the car is started, it keep always communicate with the **Ride Manager** in order to know if there are some security problems. Only if the car is reserved it can be started
FR16	The **Car Manager** according with the **Ride Manager** set the status of the car. The **Ride Manager** shows to the **Mobile Application** only the car that are in free state.
FR17	The **Ride Manager** can set up the status of the car passing through the **Car Manager** when a reservation expires
FR18	The car position is managed from the **Car System** and sent to the **Car Manager** in order to log the car movement.
FR19	The **Account Manager** has the position of the user and the **Car Manager** has the position on the car and can open it communicating with the **Car System**. The **Ride Manager** handling the two link with this components will open the car only if all the requirement (as the same position and the previously reservation) are correctly soddisfied.
FR20,FR21	The reservation is handled by the **Ride Manager**

5.1.4 End a Rent

Functional Requirements	***Design Elements***
FR22,FR23, FR24,FR25	The **Ride Manager** communicating with the **Car Manager** is always aware about the conditions of the rent, thanks to that the **Bill Manager** can figure out the total bill and apply discount or overtaxes
FR26	Once the car is closed the **Car Manager** keep communicate with the car system in order to analyse the car condition, if after five minute the **Car Manager** has not communicate with the **Ride Manager** it send a payment request thought the **Bill Manager**.
FR27,FR28	The **Car Manager** through the **Car System** check the number of passenger when the car stop, if there is none on it, the **Car Manager** ask to the **Ride Manager** if the rent can end and the car can be closed. If the position of the car is concordant with a parking/safe area then the **Ride Manager** send the acknowledge to the **Car Manager** once the car is closed the rent is ended.
FR29,FR30	The **Bill Manager** verify to apply only one discount for rent or an overtaxes.
FR31	The communication between the **Car Manager** and the **Ride Manager** and the communication between the **Car Manager** and the **Car System** takes at most two minutes, it means that the user has two minute to get off from the car.

5.1.5 Report Problem

Functional Requirements	***Design Elements***
FR32	The **Problem Manager** offers some API of Vocal Recognition and Call Center to the **Mobile Application** in order to have a customer service always open.
FR33	The **Mobile Application** let the user access to his/her information.
FR34,FR35	The **Account Manager** can provide and modify information about any user.
FR36,FR37	The **Account Manager** can modify a user only if the new information respect the Requirement contained in the RASD.
FR38	**Account manager** does not consider unfulfilled registrations. No data is stored.
FR39	**Account manager** can delete an user from the database only if there is a double confirmation from the application in order to avoid errors.

Appendix A

Appendix

A.1 Tools

- **TeXstudio:** LaTeX editor used to write the document.
- **StarUML:** Used to build UML Component Diagram and Deployment View.
- **draw.io WebSite:** Used to design the UML Sequence Diagram and the Mockup (wep portal and smartphone app).

A.2 Hours of work

In the following are listed the hours of work that each member of the group did:

1. Marco Redaelli: 41 *hours*
2. Francesco Zanoli: 41 *hours*

A.3 Version History

In the following are listed the differences between versions:

1. **11/12/2016:** First version
2. **18/01/2017:** Corrected client definition, edit minor error
3. **27/02/2017:** Corrected deployment view, and missing words