PowerEnJoy

Politecnico di Milano

Industrial and Information Engineering Computer Science and Engineering

Cattaneo Davide

El Hariry Matteo

Frontino Francesco

Design Document

Contents

1 Introduction

1.1 Purpose

Purpose of this document is to define and design an architecture on which to rely when developing the software platform.

Addressed to developers, this document contains all the major guidelines to follow while developing the PowerEnJoy system. Both the application and server side architecture are treated in this document.

1.2 Scope

This project aims at designing an electric-car sharing software system.

Car Sharing is a very cost-effective and useful service for anyone who needs a car occasionally. It allows people to use and pay for the car according to their personal use, without the hassle and costs of owning their own vehicle (parking, purchase costs, maintenance, insurance etc.).

The system we will develop is meant for cities which are provided with an efficient amount of parking lots and a wide distribution of electric car-charging platforms throughout the urban areas.

The application must allow the users which are registered to perform several easy and effective operations. Once logged in, the user can find available cars around him/her or in specified locations of the city, and chose the one to reserve.

Afterwards the user, who needs to reach the car before a given time slot expiration, will be able, by unlocking the car using the app, to easily enter the vehicle and drive to his/her destination.

1.3 Definitions, acronyms and abbreviations

Here is a brief description of the most important actors and words used in our system:

- **User:** by user is meant a person already registered in the system, so that has a profile, uses the features provided by the system and performs actions accordingly. (S)He can use all the functionalities described below (see Functional Requirements).
- **Guest:** a guest is a person that probably for the first time accesses the system or that hasn't already signed up. Guest has less power in the system; his/her actions are limited to access an introduction view and register to the service.
- **System:** is the application core. The software system which will perform all the operations and monitor interactions and be a medium between users and cars.
- **Reservation:** the allocation of a car to a user, which starts when the booking request arrives and ends either when the expiration time ends or when the car is unlocked. In this last case it triggers the start of the first travel so it initiates a ride.
- **Car:** the vehicle used by the users, which contains different sensors and an embedded computer. It has seat sensors to detect passengers, sensor to know battery level and charging actions. The computer, of course, has as main functionality to provide navigation facilities through a GPS system and to send all the relevant data to the main system server.
- **Ride:** conceptually is the use of the car, and it can be identified by the time duration of the user's journey, from unlocking the vehicle until the final parking (having user selecting "end ride" or "end ride & charge" on the car screen) with the car locked.
- **Travel:** is considered as the ride segment and is identified by a change of the status of the car. More travels can be part of a single ride.
- **Operator:** is a flexible actor in our system. He's part of a set of people operating under the administrator directions. Their normal tasks are to bring to charging stations cars left with less than 15% battery level, interact with users which call for help during a ride, intervene when necessary (e.g. a wheel brakes during a ride).

Their exceptional task can be the case in which they have to go and get back cars taken by the police or cars involved in incidents etc.

- **Administrator:** the administrator of the system is the person allowed to manage eventual unexpected cases (like incidents and damaging situations). He is the person notified every time a problem occurs, and once analyzed the situation (s)he'll decide how to handle it (call for support, send operators, call the police etc.).
- **Safe Area:** is a part of a set of areas considered safe for parking cars after a ride is over. Temporary stops can be everywhere, but long term parks can only occur in safe areas. They must be very spread and every neighborhood should have at least one.
- **Normal rate:** the charging rate applied when the car engine is ON.
- **Halt rate:** the charging rate applied when the engine is OFF and either the user is inside or (s)he has parked the car in temporary stop mode.
- **Board Controller:** BC is the car system, which includes all the hardware and software components interacting with the vehicle itself, with the users' smartphone and the Central System. The Main components part of the BC are the CAN controller, the Android System and the car display. All of them are interconnected in order to guarantee an efficient flow of information from and to sensors and with the outside environment (users and central system).
- **CAN:** A Controller Area Network (CAN bus) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other in applications without a host computer. It is a message-based protocol, designed originally for multiplex electrical wiring within automobiles, but is also used in many other contexts.

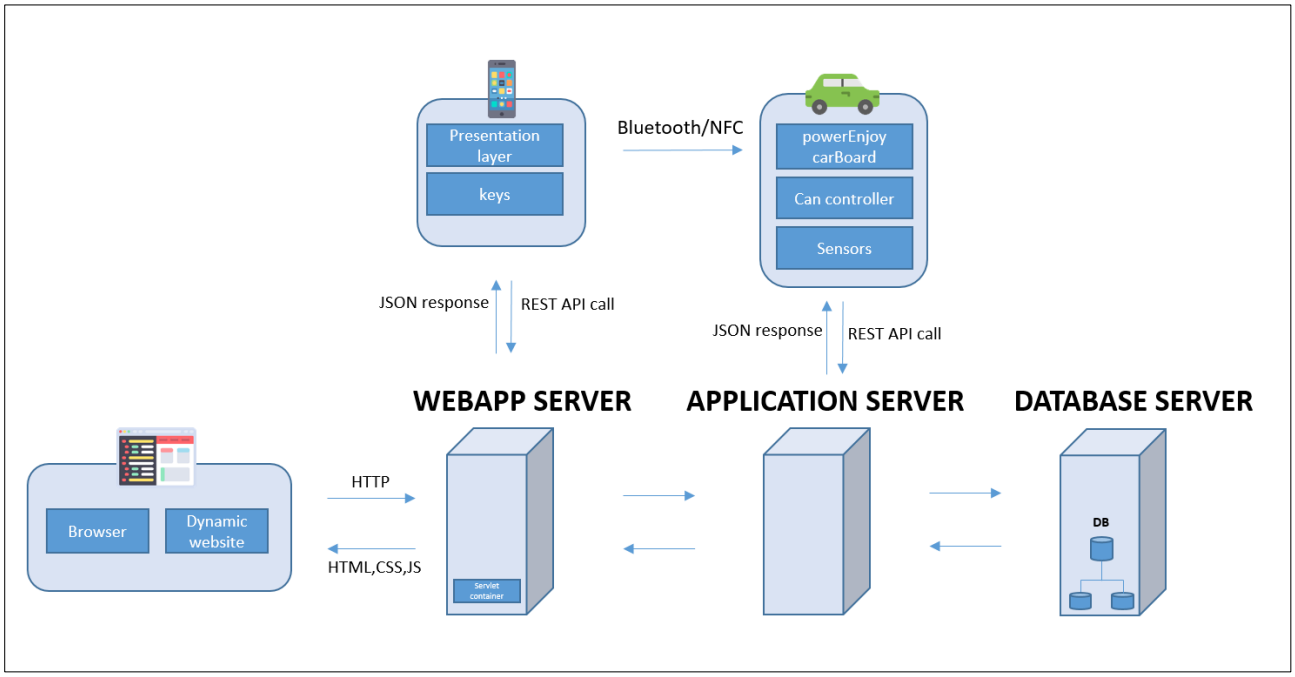
1.4 Reference documents

Specification Document:
* Assignments 1 and 2 (RASD and DD).pdf
* Design Part I.pdf and Design Part 2.pdf from lecture slides

Examples documents:
* Sample Design Deliverable Discussed on Nov. 2

2 Architectural design

2.1 Overview

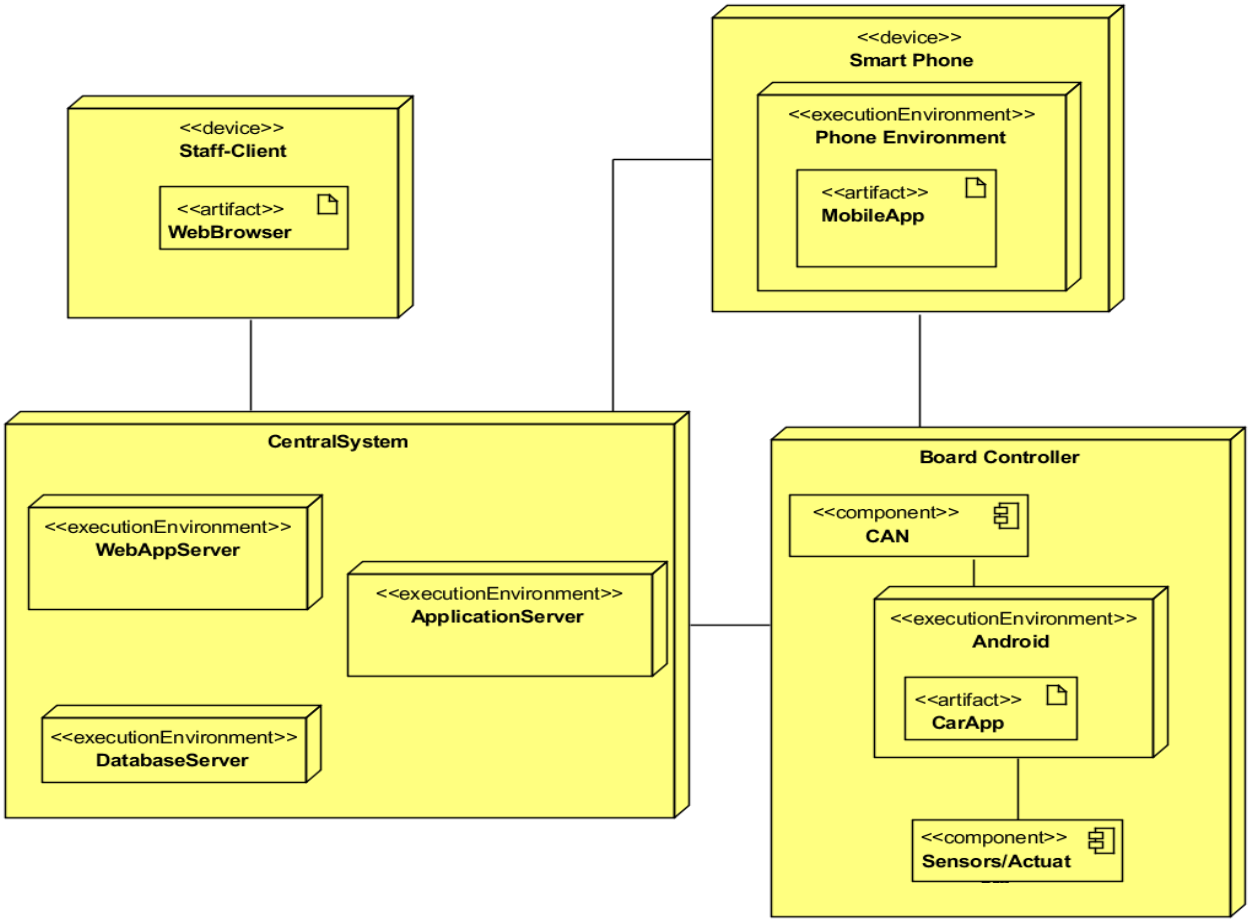


In PowerEnjoy platform, users interact with the system via the client App installed on their smartphone. The App communicates with the Application Server on the main system via a 3g or WiFi channel in order to book vehicles and retrieve the key software that is required to unlock a vehicle booked. The software key is exchanged between the client App and the car system via Bluetooth channel; It is used to open/close doors, and to enable/disable the vehicles.
We have decided to implement also a web application for the system. PowerEnJoy's website (provided by the WebApp Server) will be accessible either as a set of Java Servlet Pages to unregistered users who are seeking information about the platform, or as a private portal for system administrators and operators, who will be able to access their personal account.
Administrators can manage and edit all data stored into the database, assign tasks to operators and manage car's remote functionality handling emergencies.
In order to provide interoperability between the main elements of PowerEnJoy system are (the Central System, the Board Controller and the users' smartphones) Internet Representational state transfer (REST) web services are used. REST-compliant web

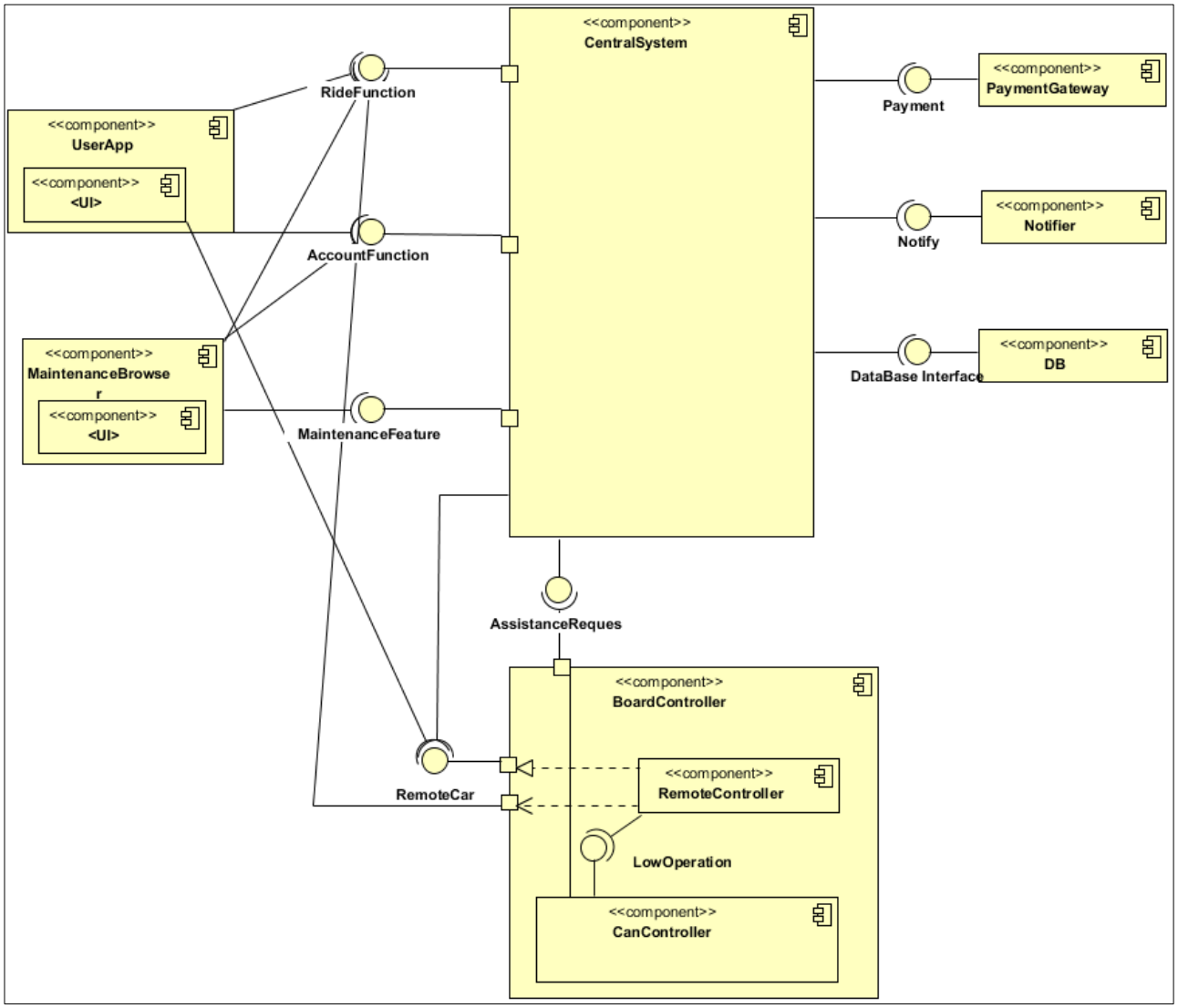
services allow requesting systems to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations.
A network of distributed database is chosen to guarantee reliability.

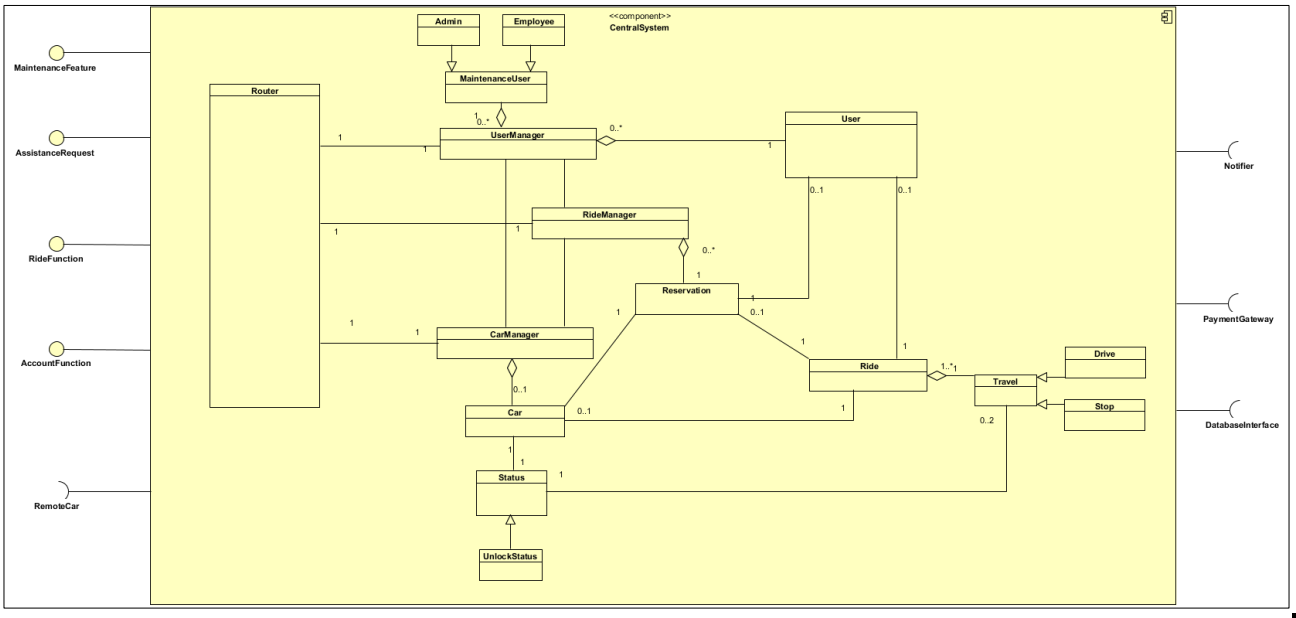
The main protocol used by our system is TCP/IP.
Some security aspects are ensured by means of the use of firewall positioned between all the servers.

2.2 High level components and interactions



2.3 Component view





The main elements of PowerEnJoy system are: the Central System, which coordinates the system, the Board Controllers, which constitute the interface between the vehicles and the rest of the system, and the users' smartphones, on which the client app is installed.
The Central System (CS) coordinates the activities of the whole system and offers services such as user and vehicle registration, vehicle reservation/ acquisition/release/monitoring, and so on.
The Board Controller (BC), which allows the system to interact with all the vehicles in a uniform way. It relies on Android platforms to let users access and interact with the system—take possession/release a reserved vehicle, open/close its doors. It uses standard protocols to allow staff members to access information and operations (e.g., the reservation of vehicles) on the coordination center.
The mechanisms designed allow application to access the data present on the vehicle—car location, number of passengers onboard, state of charge of the battery, etc.—to perform each step part of the user's ride always having a complete view over the important process's aspects.
The client app, used my users to control their accounts and start and manage car reservations and ride sessions.
The dynamic web site, from which the staff members (administrators and operators) can access their account and use the provided functionalities to perform their work.

System and Board Controller

BC is an electronic box which is interposed between the vehicle and the Central System. Its role is to allow each vehicle to interface with the system. The Board Controller is able to capture all the internal signals to the vehicle so as to make the vehicle itself independent of the surrounding system. In particular, the BC is able to:

- interact with the vehicle on which it is installed to collect information (state of battery charge, presence of persons on the seats, etc.) and send commands (doors open / close etc.)
- it interacts with the user through a touch-screen display. (The main functionalities provided to users are shown in Mock-up session).
- interact with the Central System to communicate the information on its status and the status of the vehicle, receive commands (the vehicle locking in case of malfunctions, the closure of a vehicle remained open at the end of reservation, etc.) and receive updates (device software updates).

Interaction between Board Controller and Central System

The Central System must be able to communicate with Board Controller, retrieve information from the vehicle (car status). Each BC must be designed so as to establish a communication channel with the Central System in order to transmit the necessary information. To do so, the BC has to integrate:

- a module for mobile data connections to ensure GPRS / EDGE / HSPA / 3G / 4G connections and transmit / receive data to / from Central System; a WiFi module to allow connection to any access point in the area.

Interaction between Board Controller and User's Smartphone

A PowerEnJoy customer, after making a reservation, interacts with the private vehicle through his/her smartphone, using the application installed on the mobile device. The user must be able to activate the reservation (through the apposite button) and, once entitled to drive, to take advantage of the additional services offered by PowerEnJoy. To successfully complete these tasks a direct connection can be established for communication between the user's mobile device and the BC. The communication channel chosen to allow direct exchange of information between smartphone and BC is the Bluetooth channel. You can use as an alternative channel, the NFC (Near Field Communication), if the user's mobile device is the latest generation and integrates this technology.

Interaction between Board Controller and vehicle

The BC has to be designed for connecting car electronic components to vehicle control units to acquire the information and, where possible, to send commands to the vehicle (how to open / close the doors or acquire info from sensors / send commands to actuators / allow parking). To retrieve the information listed above, the BC should include: a CAN controller, to manage the connection with vehicles equipped with CAN-bus; digital and analog inputs (and outputs), to retrieve data (send commands) to vehicles without CAN-bus; a GPS receiver, to recover the position of the vehicle.

Interaction between User Application and Central System

The smartphone App, provided agreement by the user during the installation, is able to access phone hardware components interfaces such as Bluetooth, eventual NFC and GPS.

User App developed to be used in the three native OSs (Android, iOS, Windows Phone) as thin Client. The main task for the app is to generate interfaces related to user and coordinates the various events that occurs during each interaction with the CS and BC.

Requests from users are performed through RESTful API web service. Requests made to a resource's URI will elicit a response that may be in HTML, JSON.

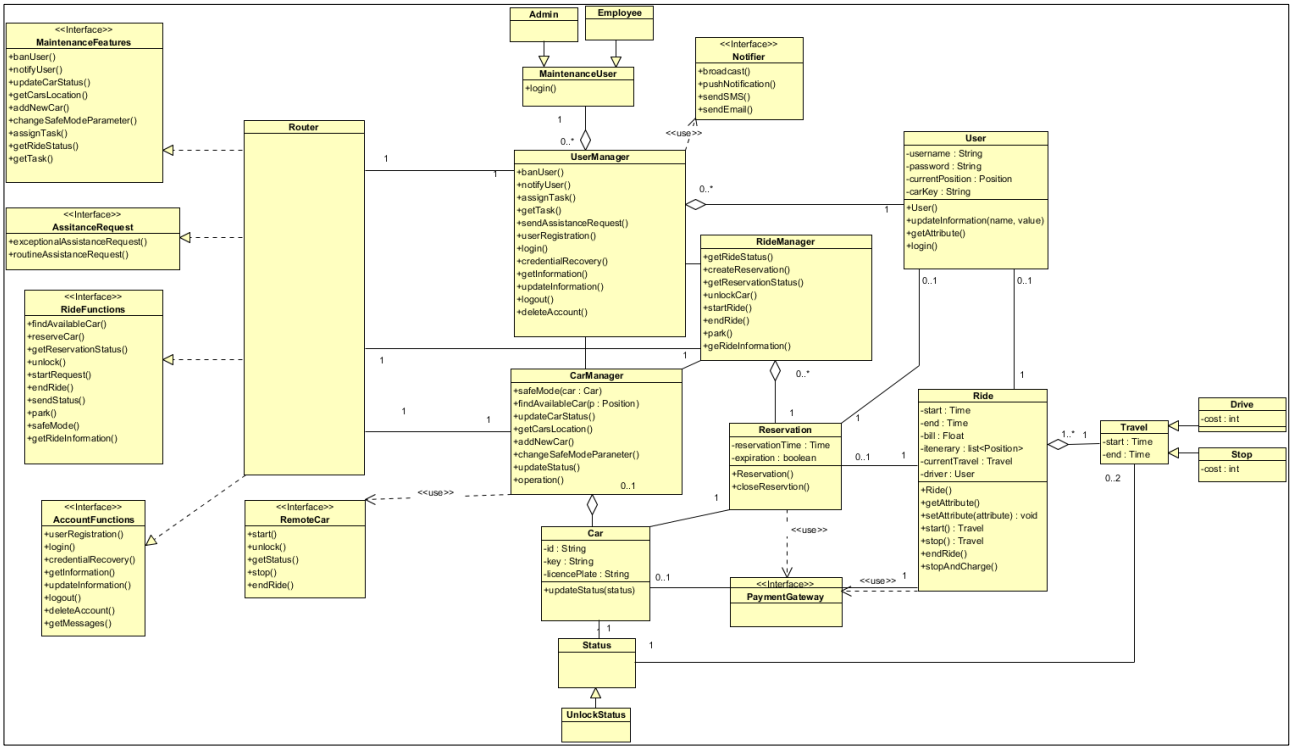
Interaction between Web Application and Central System

The web Application is used by the PowerEnJoy staff (administrators and operators), who, through a browser connection, can get inside the Central System functionalities. It provides two UI interfaces: a desktop and a mobile CSS. The Client Browser uses asynchronous calls through the AJAX protocol enhancing processes performance.

User Experience is improved thanks to a tailored CSS interfaces and a local function method developed in JS.

Requests are handled by servlets, able to generate dynamic content in order to allow a process reuse (limiting redundancy in the system calls), in the Web Server.

2.4 Class diagram



The system composes of:

Interfaces:

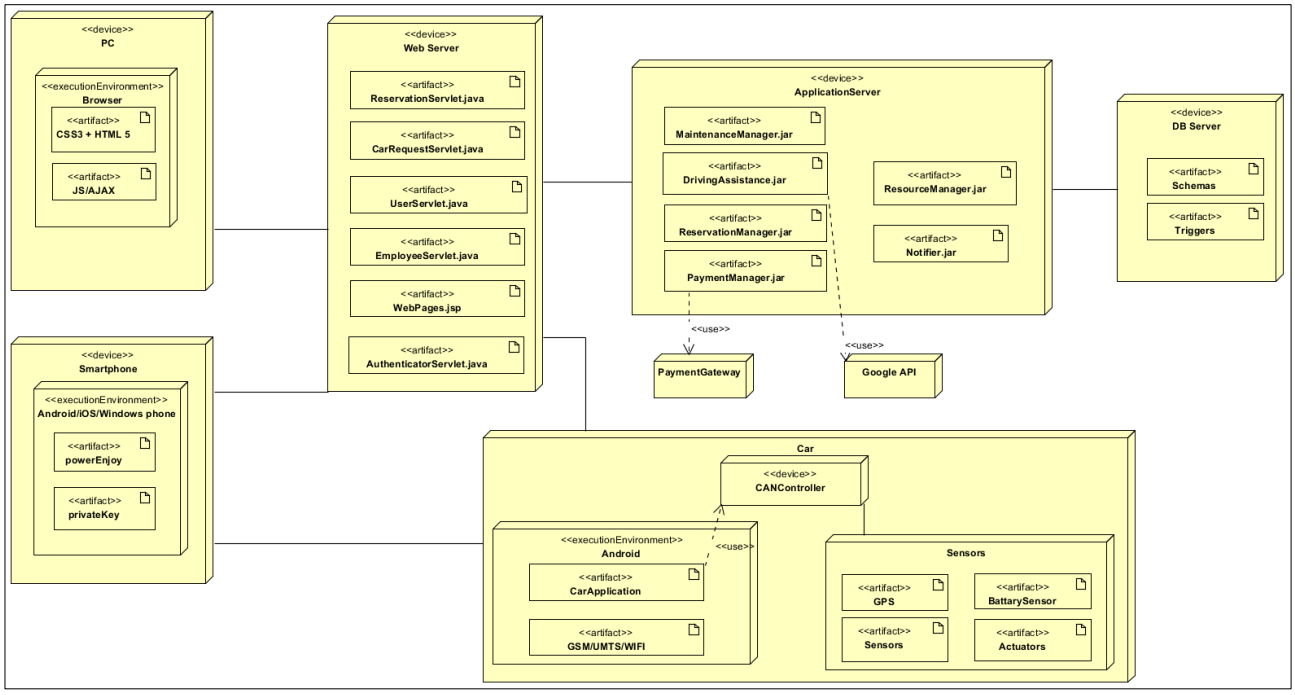
- maintenanceFeatures
- assistanceRequest
- rideFunctions
- accountFunctions
- carStatus

main elements (shown on the right):

- The router class works as a coordinator for the entire system, starting from a methods called through an interface, it subdivide the whole problem in elementary jobs that will be assigned to the various managers.
- UserManager has the task of managing all aspects regarding users and operators such as the assignment of the keys used for the unlocking procedure or the check of the user's sessions, the authentication and registration process, the update of the personal information and the various queries on the database.
    - RideManager handles all the reservations currently active. It also monitors t he expiration of those not unlocked within an hour from reservation time. It i s also able to manage ride functions, payments and travel management.
    - CarManager handles all the cars, it updates their status and interacts with r emote car functionalities.

Router will be programmed to guarantee security aspects: All the functions are activat ed only if they are authenticated by the user manager. Scalability and performance are guarantee with a parallel execution of the various class managers: each manager can be duplicate when requests increase.

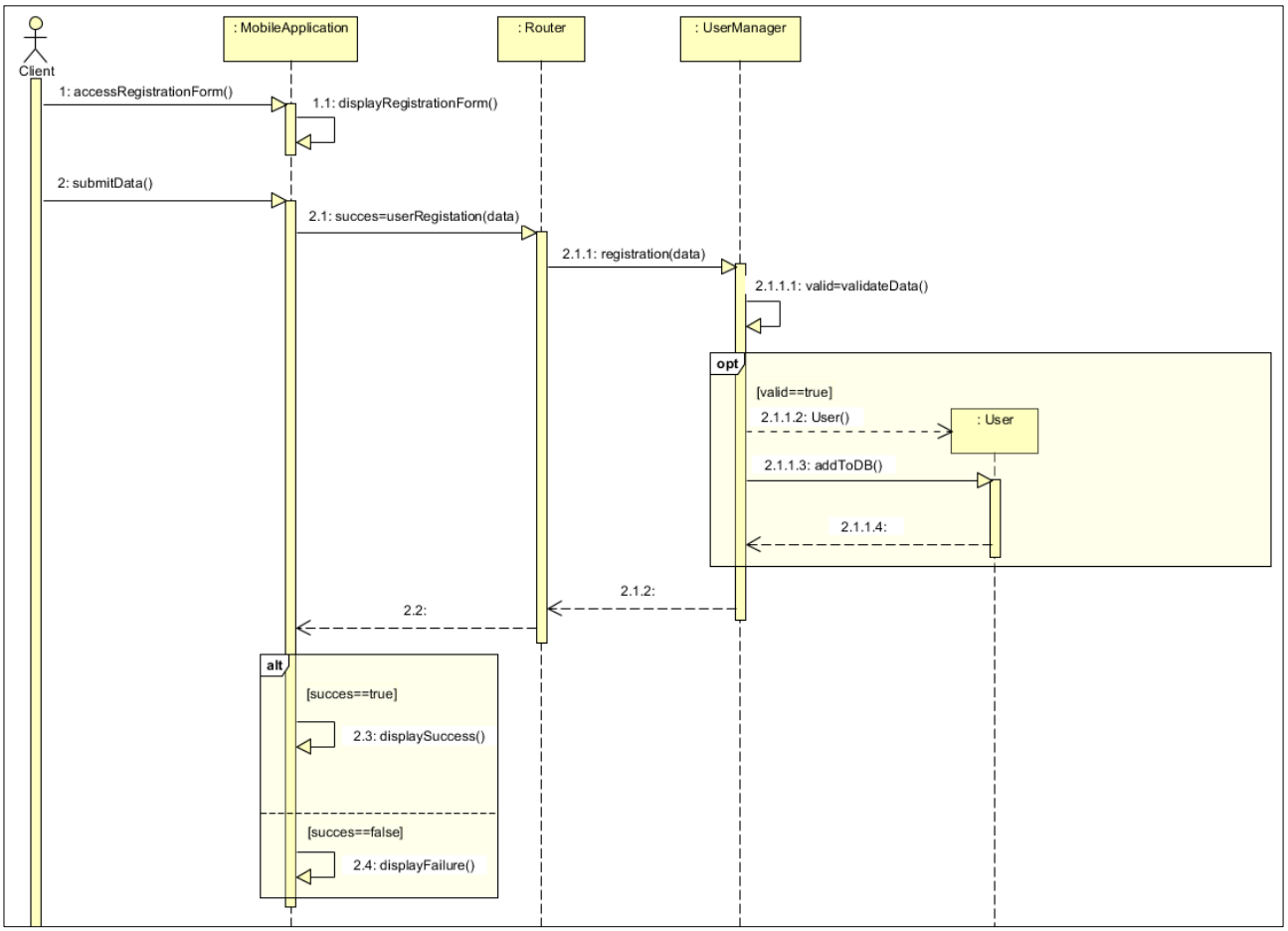
2.5 Deploying view


<<device>>
PC
<<executionEnvironment>>
Browser
<<artifact>>
CSS3 + HTML 5
<<artifact>>
JS/AJAX
<<device>>
Smartphone
<<executionEnvironment>>
Android/iOS/Windows phone
<<artifact>>
powerEnjoy
<<artifact>>
privateKey
<<device>>
Web Server
<<artifact>>
ReservationServlet.java
<<artifact>>
CarRequestServlet.java
<<artifact>>
UserServlet.java
<<artifact>>
EmployeeServlet.java
<<artifact>>
WebPages.jsp
<<artifact>>
AuthenticatorServlet.java
<<device>>
ApplicationServer
<<artifact>>
MaintenanceManager.jar
<<artifact>>
DrivingAssistance.jar
<<artifact>>
ReservationManager.jar
<<artifact>>
PaymentManager.jar
<<artifact>>
ResourceManager.jar
<<artifact>>
Notifier.jar
<<use>>
<<use>>
PaymentGateway
Google API
<<device>>
DB Server
<<artifact>>
Schemas
<<artifact>>
Triggers
Car
<<device>>
CANController
<<executionEnvironment>>
Android
<<use>>
<<artifact>>
CarApplication
<<artifact>>
GSM/UMTS/WIFI
Sensors
<<artifact>>
GPS
<<artifact>>
BattarySensor
<<artifact>>
Sensors
<<artifact>>
Actuators

2.6 Runtime view

Here are reported some of the most significant sequence diagrams aiming at giving fur ther details about how the system is conceived.

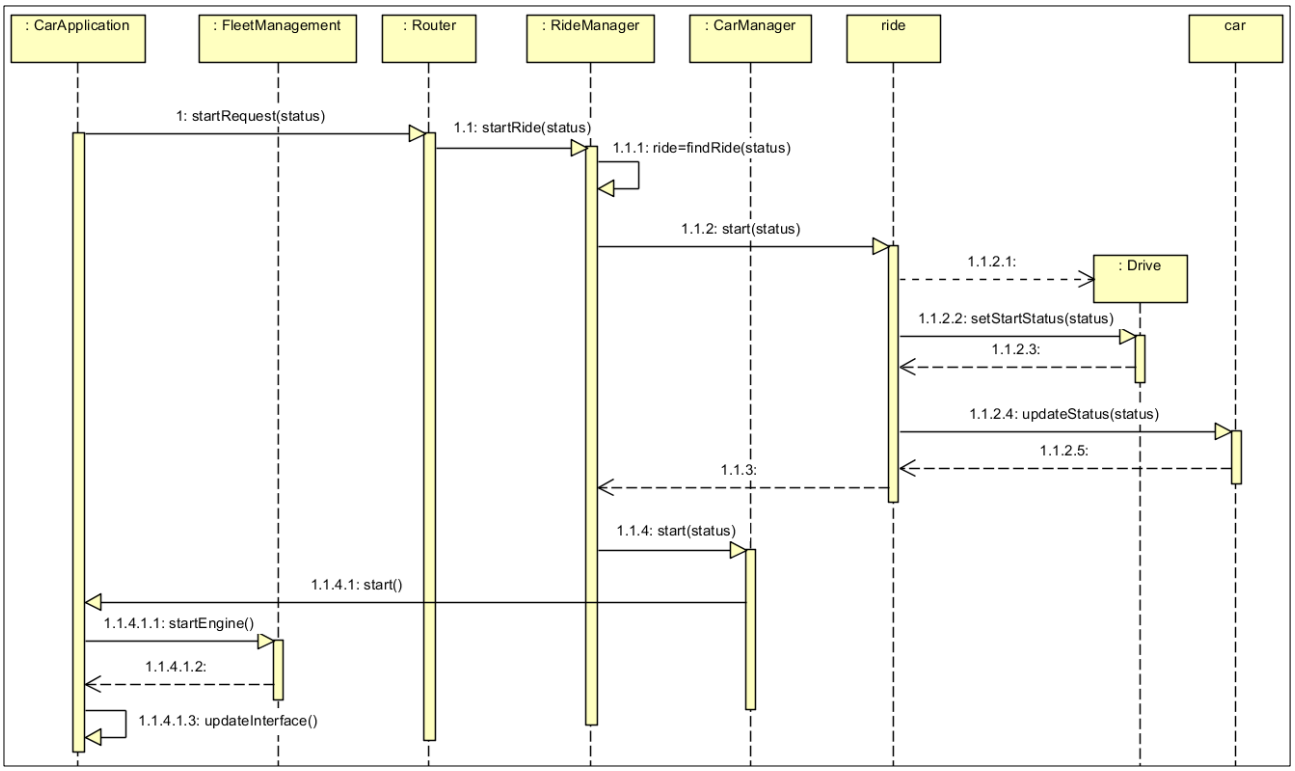
2.6.1 Registration



This sequence diagram explains how the registration process is performed by the system and how the requests are routed to the managers. UserManager has the task to validate input data, create new users and call methods to update the database. The last operation performed by the mobile application is the update of the user interface with the final outcome.

All the others functionalities enclosed in the AccountFuntions interface have the same structural behavior of this sequence.

2.6.2 Start drive travel

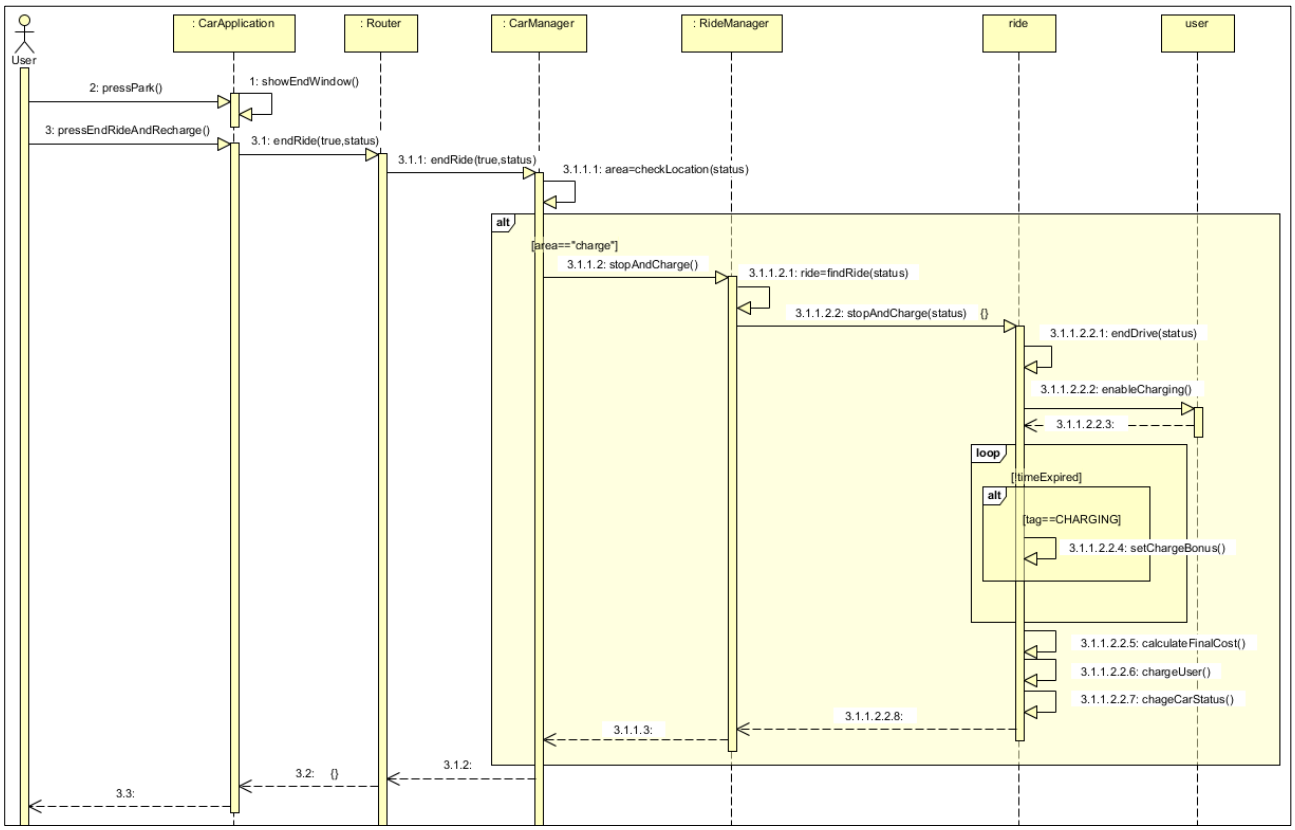


Interactions between the car and the system are represented by this sequence which explains how a new ride is created starting from a "startEngine" event.

The communication between the system and the car is allowed through two interfaces: "RemoteCar" interface provided by the server installed inside each car and the "CarStatus" interface provided by the main system.

RideManager is able to find a ride starting from a carStatus because has all the references to the active reservations. starting from that Ride previously found the Ride object creates a new Travel Ride type, updates carStatus and finally communicates with a remote car activating the remote function startEngine.

2.6.3 End and charge



The user ready to end his car use can choose "end ride and recharge" function. By doing so, the car sends to the server its status along with the ride termination request.

Car's position is verified with the method "checkLocation" called by the RideManager and the procedure can continue only if the value of the variable area is "charge".

The first phase of this procedure is accomplished by the ride object: the ride is stopped calling the endDrive method and the user is enabled to use the charging station.

From the architectural point of view, the entire mechanism works in this way:

- The system updates the user account from the database changing the value "charge" from false to true. A trigger is delegated for the restore of this variable after a time slot.

- The Power Enjoy application queries the system to download locally the key used then for the activation of the power station through NFC/Bluetooth technology.

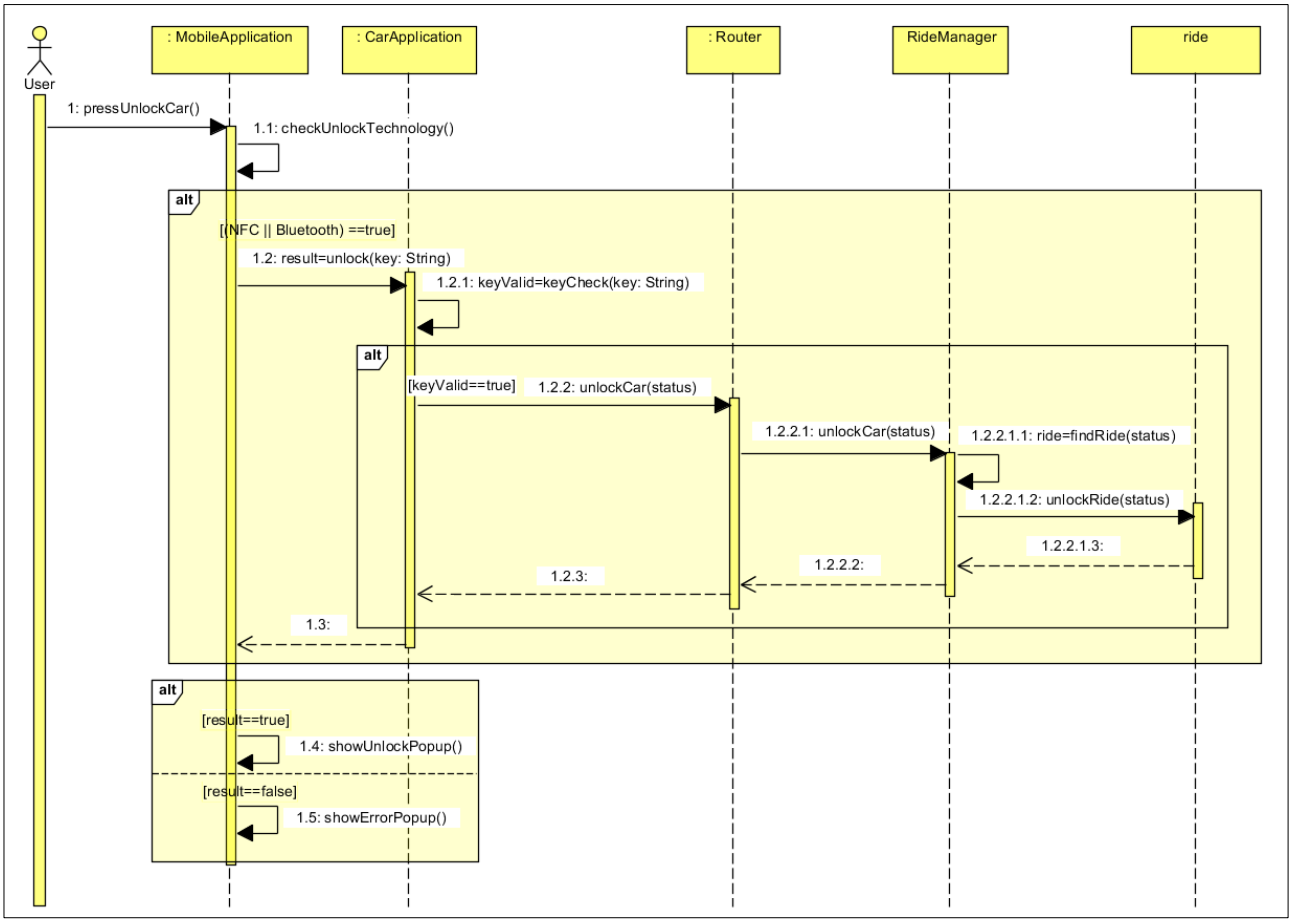
During second phase, system is kept in listening for an event:

-System detect a car status with parameter set on "charging" → the bonus is assigned

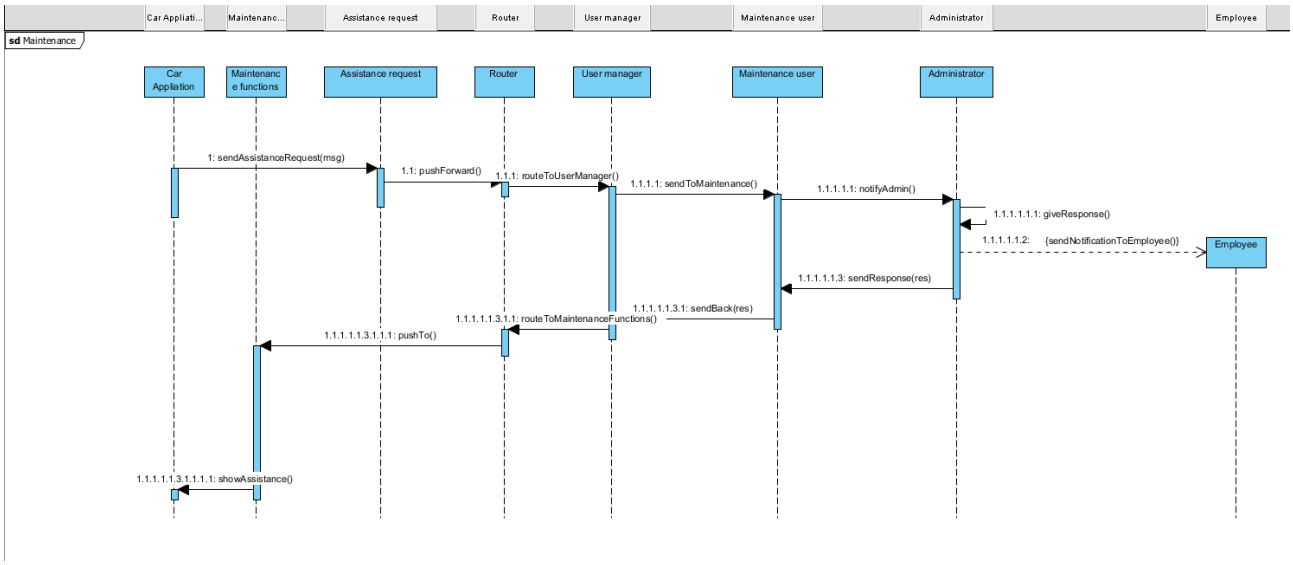
-time expired event →nothing happens

Finally the ride costs with the related bonuses are calculated and the user is charged.

2.6.4 *Unlock car*



2.6.5 *Maintenance*



2.7 Architectural styles and design patterns

Here are reported details regarding the overall architecture, the design patters which are meant to be used and the protocols on which the system application Is thought to rely.

2.7.1 Overall architecture

The platform to be designed will be divided into 3 tiers:

1. Databases (HDDMS: Homogeneous Distributed Databases Management System)
2. Application Logic (BLL: Business Logic Layer)
3. Thin Client (interface to BLL)

2.7.2 Protocols

The system makes use, for the exchange of data, of the following protocol:

- HTTP – TCP/IP
- Bluetooth 4.2/ 5
- NFC
- GPRS / EDGE / HSPA / 3G / 4G
- CAN – Controller Area Network
- REST – Representational State Transfer

2.7.3 Design patterns

The system makes use of the following design patterns:

- MVC: Model-View-Controller pattern is use in the development of the client app.
- Client-Server: is used for to manage the interaction between the clients and the main application system.
- The observer pattern: used as part of the notification component
- The mobile push pattern, used to send content to devices automatically, eliminating the dependency on web app servers for pulling content updates.
- Factory method, methods to deal with the (problem of creating objects without having to specify the exact class of the object) “Ride” class.
- Singleton pattern: used for the “Router” element.

2.8 Other design choices

The storage of data in PowerEnJoy platform is handled by a HDDMS (Homogeneous Distributed Databases Management System) which has identical software and hardware running all databases instances, and appears through a single interface as if it were a single database.

The homogeneous system is much easier to design and manage. The following conditions must be satisfied for homogeneous database:

Choosing this kind of DB system a condition must hold:

- Operating system, data structures, and database application used at each location must be same or compatible.

3 Algorithm design

3.1 Costs calculation

Ride.java

```
import java.util.ArrayList;

public class Ride {

	private ArrayList<TravelDrive> travelDrives;
	private ArrayList<TravelStop> travelStops;
	private boolean finalCharge;
	private final float CHARGE_BONUS = (float) 0.2;

	public Ride(){
		this.travelDrives = new ArrayList<>();
		this.travelStops = new ArrayList<>();
		this.finalCharge = false;
	}

	public float calculateTotalCosts(){
		float cost = 0;

		for(int i=0; i<travelDrives.size(); i++)
			cost+=travelDrives.get(i).calculateCosts();

		for(int i=0; i<travelStops.size(); i++)
			cost+=travelStops.get(i).calculateCosts();

		if(this.finalCharge)
			cost-=cost*CHARGE_BONUS;

		return cost;
	}

	public void putCarInCharge(){
		this.finalCharge = true;
	}
}
```

TravelDrive.java

```
import java.util.Calendar;

public class TravelDrive {
	private int passengersNumber;
	private Calendar startTime;
	private Calendar endTime;
	private final float PASSENGERS_BONUS = (float) 0.1;
	private final float COST_PER_MINUTE = (float) 0.3;


	public TravelDrive(int passengersNumber){
		this.startTime.getTime();
		this.passengersNumber = passengersNumber;
	}

	public void endTravel(){
		this.endTime.getTime();
	}

	public boolean deservePassengersBonus() {
		return this.passengersNumber > 3;
	}

	public float calculateCosts(){

		float minutes;
		float cost;

		minutes = (endTime.getTimeInMillis() * 1000 / 60 -
startTime.getTimeInMillis() * 1000 / 60);
		cost = minutes * COST_PER_MINUTE;

		if(this.deservePassengersBonus())
			cost-= cost * PASSENGERS_BONUS;

		return cost;
	}
}
```

TravelStop.java

```
import java.util.Calendar;

public class TravelStop {
	private Calendar startTime;
	private Calendar endTime;
	private final float COST_PER_MINUTE = (float) 0.1;

	public TravelStop(){
		this.startTime.getTime();
		this.inCharge = false;
	}
```

```
	public void endTravel(){
		this.endTime.getTime();
	}


	public float calculateCosts(){

		float minutes;
		float cost;

		minutes = (endTime.getTimeInMillis() * 1000 / 60 -
startTime.getTimeInMillis() * 1000 / 60);
		cost = minutes * COST_PER_MINUTE;;

		return cost;
	}
}
```

3.2 Fair distribution

Distributor.java

```
import java.util.ArrayList;

public class Distributor{

/*
* HOW IT WORKS
* 1) THE ALGORITHM KEEPS ALWAYS UPDATED THE AVERAGE DISTANCE (AVG) BETWEEN ALL VEHICLES
* 2) FOR EACH CAR THAT NEEDS TO BE MOVED TO GUARANTEE A FAIR DISTRIBUTION, THE NEAREST
CAR IS FOUND
* 3) THE RIGHT PARKING POSITION IN THE ONE WITHIN A RAGE EQUAL TO (AVG) FROM THE NEARES
VEHICLE.
*
* THE OBJECTIVE IS TO PROVIDE A FORCED SPARSE DISTRIBUTION AND ALSO KEEP CARS AT A
CERTAIN DISTANCE ONE FROM THE OTHERS.
*/

	private ArrayList<Car> cars;
	private int averageDistance;

	public Distributor(){
		this.cars = new ArrayList<>();
		this.averageDistance = 0;
	}

	public void addCar(Car car){
		this.cars.add(car);
	}

	public Car getCar(int idx){
```

```
        return this.cars.get(idx);
}

// average of the distance between all cars
private void calculateAverageDistance(){

        int tot = 0;
        int dist;

        for(int i=0; i<cars.size()-1; i++){
                for(int j=i+1; j<cars.size(); j++){
                        dist = distanceBetweenCars(cars.get(i), cars.get(j));
                        tot += dist;
                }
        }

        tot /= cars.size();
        this.averageDistance = tot;
}


// returns the nearest car to c
private Car findNearestCar(Car c){
        int minDist = 0;
        Car found = null;

        for(Car car : cars){
                if(minDist == 0){
                        minDist = distanceBetweenCars(c, car);
                        found = car;
                }
                else
                        if(distanceBetweenCars(c, car) < minDist && !car.equals(c)){
                                minDist = distanceBetweenCars(c, car);
                                found = car;
                        }
        }

        return found;
}

// generates a new x for the car
private int getNewX(Car c){

        int valX;
        int X_UPPERBOUND = 800;

        do{
                valX = c.getX();
                int signX = (int) (Math.random()*10%2);
                int incX = (int) (Math.random()*1000%20);

                if(signX == 0)
                        valX -= incX;
                else
                        valX += incX;
        }while(valX > X_UPPERBOUND || valX < 0);

        return valX;
}
```

```
	// generates a new y for the car
	private int getNewY(Car c){
		int valY;
		int Y_UPPERBOUND = 600;


		do{
			valY = c.getY();
			int signY = (int) (Math.random()*10%2);
			int incY = (int) (Math.random()*1000%20);

			if(signY == 0)
				valY -= incY;
			else
				valY += incY;
		}while(valY > Y_UPPERBOUND || valY < 0);

		return valY;

	}

	// euclidean distance between cars
	private int distanceBetweenCars(Car c1, Car c2){
		float distance;

		distance = (float) Math.sqrt(Math.pow(c2.getX()-c1.getX(),2) +
Math.pow(c2.getY()-c1.getY(),2));

		return (int)distance;
	}

	// given a car c it finds a position guaranteeing a fir distribution
	public void findNewPosition(Car c){

		Car nearestCar = null;

		calculateAverageDistance();
		nearestCar = findNearestCar(c);
		c.move(getNewX(nearestCar), getNewY(nearestCar));
	}
}
```

Car.java

```
public class Car {
	private int x;
	private int y;

	// city bounds (0, k)
	private final int X_UPPERBOUND = 800;
	private final int Y_UPPERBOUND = 600;

	public Car(){
		this.x = (int) (Math.random()*1000 % X_UPPERBOUND);
		this.y = (int) (Math.random()*1000 % Y_UPPERBOUND);
	}
```

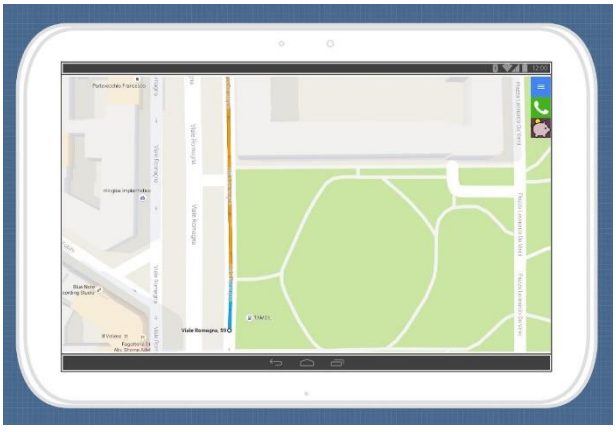
```
	// randomly moves the car within city limits
	public void move(){

		int valX;
		int valY;

		do{
			valX = this.x;
			int signX = (int) (Math.random()*10%2);
			int incX = (int) (Math.random()*1000%120);

			if(signX == 0)
				valX -= incX;
			else
				valX += incX;
		}while(valX > X_UPPERBOUND || valX < 0);

		do{
			valY = this.y;
			int signY = (int) (Math.random()*10%2);
			int incY = (int) (Math.random()*1000%120);

			if(signY == 0)
				valY -= incY;
			else
				valY += incY;
		}while(valY > Y_UPPERBOUND || valY < 0);


		this.x = valX;
		this.y = valY;
	}

	public int getX(){
		return this.x;
	}

	public int getY(){
		return this.y;
	}


	public void move(int x, int y){
		this.x = x;
		this.y = y;
	}
}
```

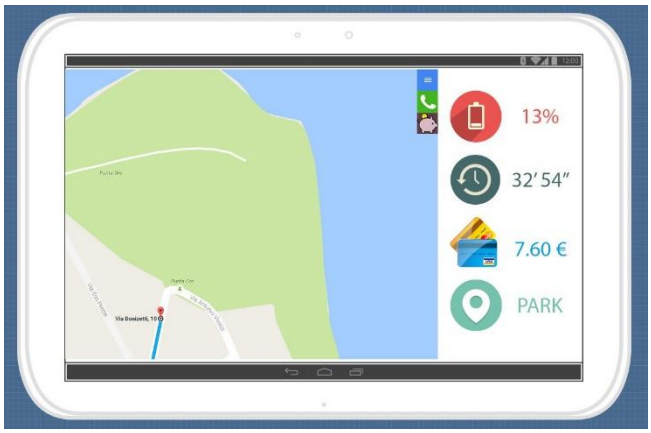
4 User interface design

4.1 Mockups

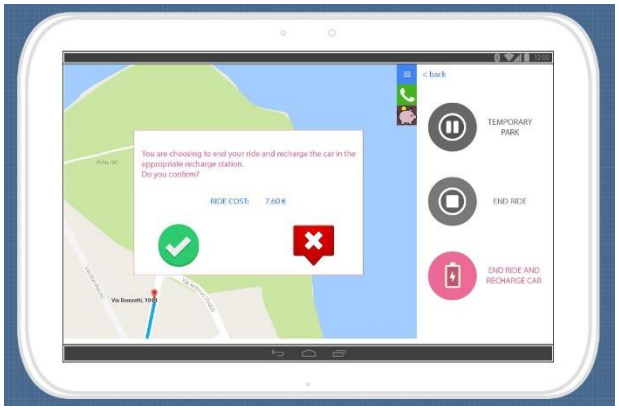
Administrator login and maintenance

Navigation and info





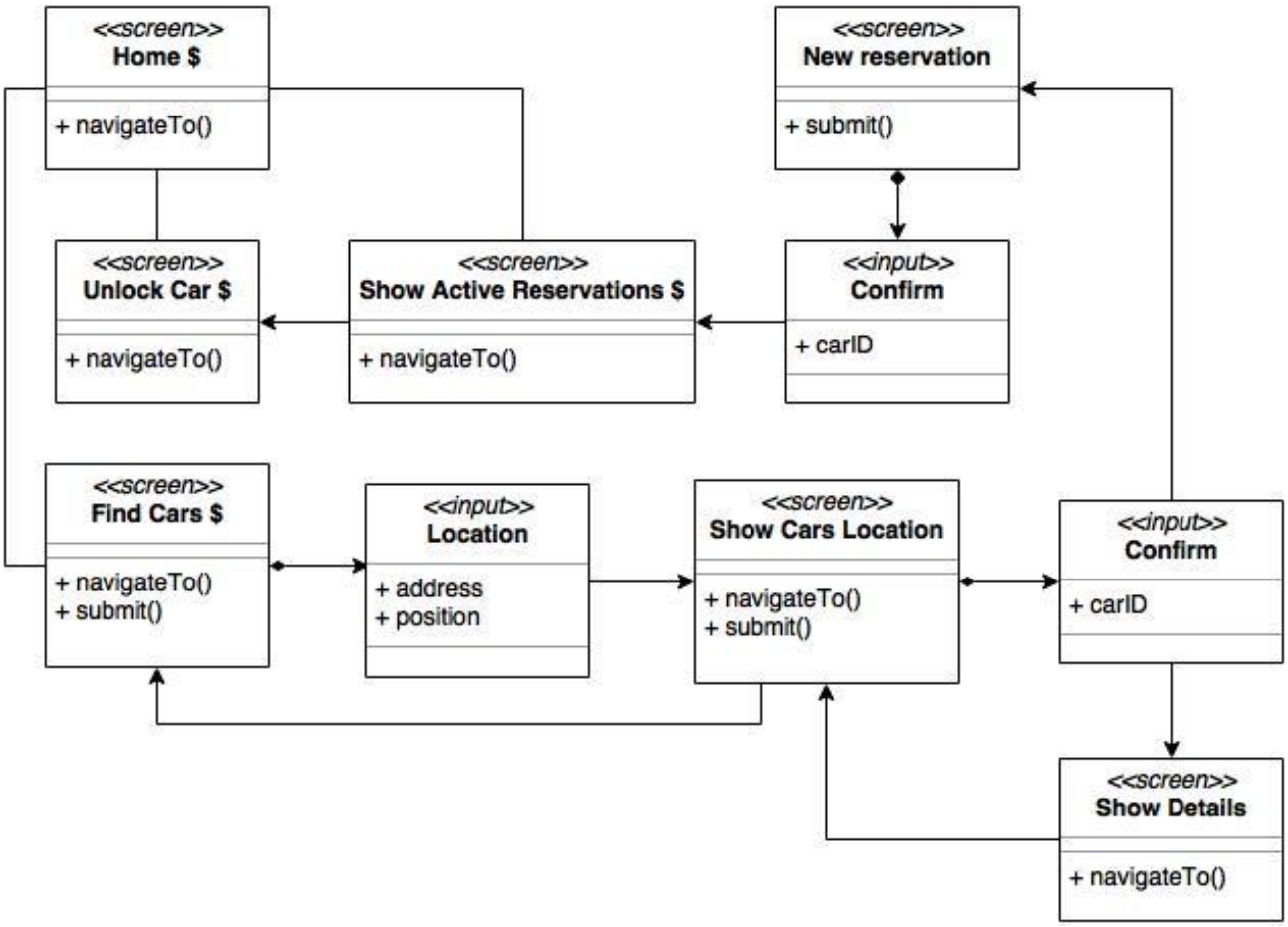
Parking options and selection



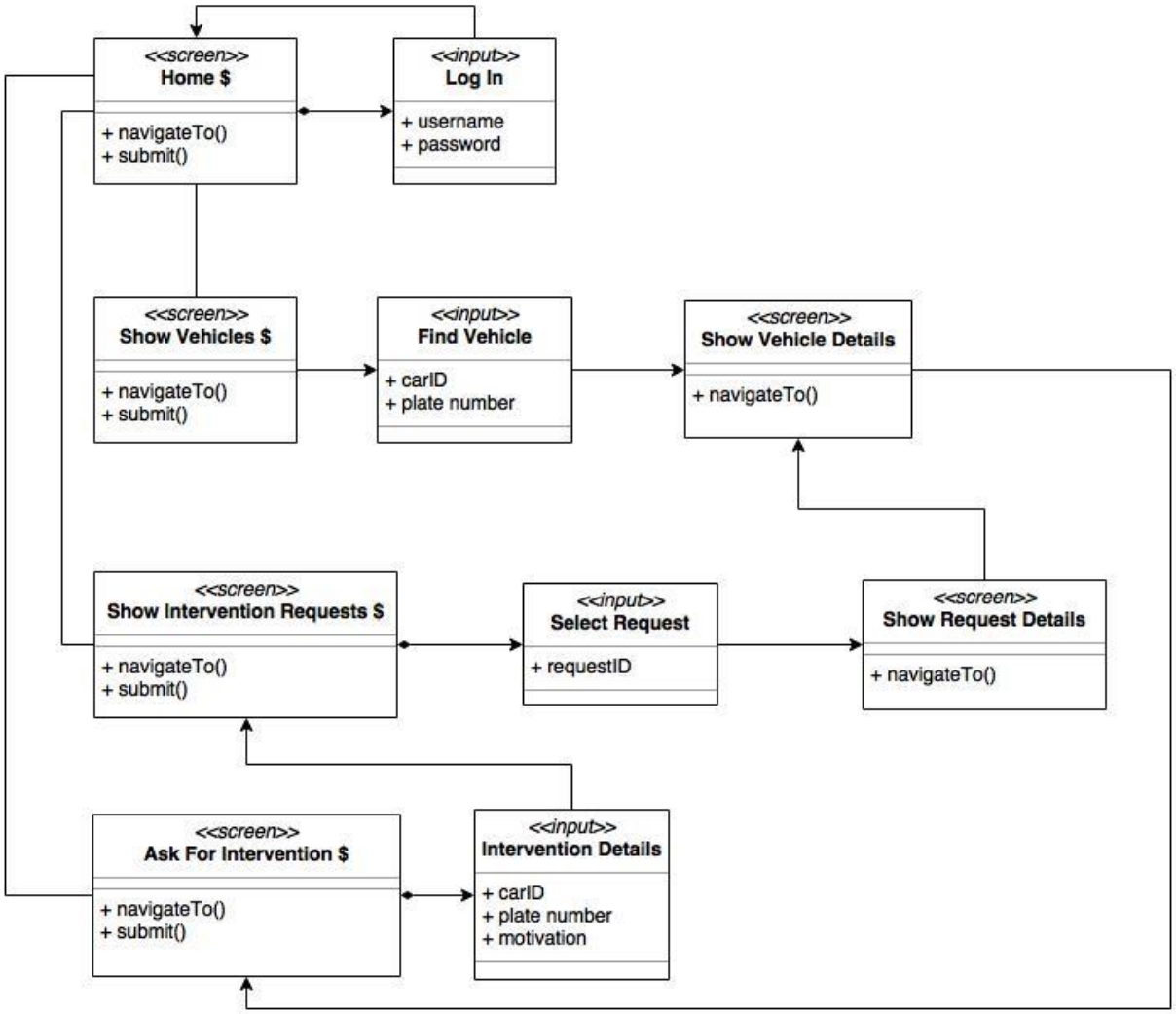
Evaluation

4.2 UX diagrams

The UX diagram reported below represents how the interaction process among screens present in the mobile app has been thought to be.



This second UX diagram represents how the user experience, with regards to the administration process do through the web page offered to the administrator, has been conceived.



5 Requirements traceability

The components used in goal fulfillment are listed below:

- [G1] Users must be able to register and access to the system.
    - User manager
    - Router
    - Account functions
- [G2] Registered users must be able to find the locations of available cars within a certain distance.
    - Car manager
    - Car
    - Status
- [G3] Users must be able to reserve a single car with a one-hour time limit.
    - Car manager
    - Car
    - Status
    - Reservation
- [G4] If a car is not picked-up within one hour from the reservation, the system tags the car as available again, and the reservation expires; user will be charged with a fee of 1 €.
    - Ride manager
    - Reservation
    - Payment gateway

•[G5] A user that has reserved successfully a car must be able to unlock it using the app.

- Car manager
  - Remote car
  - Router
- Car functions

•[G5+] Once a user is inside the vehicle (s)he must be able to indicate eventual damages in the car.

- Ride manager
- Router
- Ride functions

•[G6] As soon as the engine ignites, the system starts charging the user for a given amount of money per minute; the user is notified of the current charges through a screen on the car.

- Ride manager
- Router
- Ride functions
- Car functions

•[G7] The system stops charging the user as soon as the car is parked in a safe area and the user exits the car; at this point, the system locks the car automatically.

- Ride manager
- Router
- Ride functions
- Car functions

•[G7+] The cars parked not in safe area must keep charging with the halt rate up to a given limit of time Max-Time-Stop.

- Ride manager
- Router
- Ride functions

•[G8] The set of safe areas for parking cars is pre-defined by the management system.

- Ride manager

•[G8+] The set of safe areas must always be displayed on the car display's map during the rides.

- Ride manager
- Router
- Ride functions
- Car functions

•[G9] system must know parking location, the battery level, status (in charge or not) and if there were two passengers onboard every time a ride is over in order to calculate the right discount.

- Car manager
- Remote car
- Car
- Status

•[G10] The system must charge the proper cost for every user at the end of the ride, so that eventual discounts or fee are included.

- Ride manager
- Reservation
- Ride
  - Travel
- Payment gateway

•[G11] Mechanical problems that occur during a ride session and accidents must be solved by the maintenance operators, whom can be contacted by users through the car display.

- Assistance request
- Router
- User manager
- Maintenance user
- Administrator
- Maintenance user
- User manager
  - Notificator
  - (Employee)
- Router
- Maintenance functions

6 Revision

6.1.1 Software and tools used

The following software have been used:

- Microsoft Word (document writing)
- Visual Paradigm Trial (diagrams)
- Draw.io (UX diagrams)
- Photoshop Trial (mockups)
- Eclipse IDE (developing algorithms)

6.1.2 Team work

Here is reported a compact table showing how the work was brought on by all the members of the group.

Document work finished on the 9th December 2016.

effort spent by each group member - assignment 2

francesco				matteo				davide			
ora inizio	ora fine	dettaglio lavoro	totale ore	ora inizio	ora fine	dettaglio lavoro	totale ore	ora inizio	ora fine	dettaglio lavoro	totale ore
2/12 14.00	2/12 16.00	general understanding of DD	2.00.00	04/12 13.00	04/12 16.00	reading DD descriptions	3.00.00	03/12 15.00	05/12 18.30	REST API information collection	3.30.00
3/12 14.00	3/12 18.00	study phase	4.00.00	05/12 17.00	05/12 21.00	High level Component & Deployment Diagram	4.00.00	04/12 17.00	05/12 19.00	Mockup design	2.00.00
3/12 15.00	3/12 18.00	main overview	3.00.00	06/12 11.00	06/12 13.30	Component Diagram & descriptions	2.30.00	05/12 17.00	05/12 18.30	High level architecture and interaction between components	1.30.00
5/12 17.00	5/12 21.00	High level Component & Deployment Diagram	4.00.00	06/12 15.00	06/12 21.00	descriptions & diagrams understanding	6.00.00	06/12 11.00	05/12 12.00	Algorithms writing	1.00.00
06/12 15.00	06/12 21.00	component diagram & descriptions	6.00.00	07/12 10.30	07/12/2016 14.00	document writing & ...	3.30.00	06/12/2016 18.00.00	06/12 21.30	Algorithms writing	3.30.00
07/12 14.00	07/12 19.00	class diagram	5.00.00	07/12 16.30	07/12 19.00	mockups & class	2.30.00	07/12/2016 16.00.00	07/12 18.00	UX diagrams & fixes	2.00.00
08/12 11.00	08/12 15.30	Sequence diagram + descriptions	4.30.00	07/12 22.30	08/12 2.30	class description + sequence diagrams	4.00.00	08/12/2016 13.00.00	07/12 17.30	Document writing, corrections and refactoring	4.30.00
8/12 17.30	8/12 22.00	sequence diagram + description + document revision	4.30.00	08/12 15.00	08/12 20.00	sequence + document formatting	5.00.00	09/12 19.00	09/12 21.30	Done layout,maintenance sequence, code comments, traceability and indexing	2.30.00

Update:

DD V1 : class diagram fixed