PowerEnJoy

## Design Document

Cattaneo Davide El Hariry Matteo Frontino Francesco

Contents

1. ***Introduction***
   1. Purpose
   2. Scope
   3. Definitions, acronyms and abbreviations
   4. Reference documents
2. ***Architectural design***
   1. Overview
   2. High level components and interactions
   3. Component view
   4. Deploying view
   5. Runtime view
   6. Component interfaces
   7. Architectural styles and design patterns
      1. Overall architecture
      2. Protocols
      3. Design patterns
   8. Other design choices
3. ***Algorithm design***
   1. Algoritmo1
   2. Algoritmo2
   3. Algoritmo3
4. ***User interface design***
   1. Mockups
   2. UX diagrams
5. ***Requirements traceability***
6. ***Revision***
   1. Software and tools used
   2. Team work
7. ***Introduction***
   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. ***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. ***Definitions, acronyms and abbreviations***

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

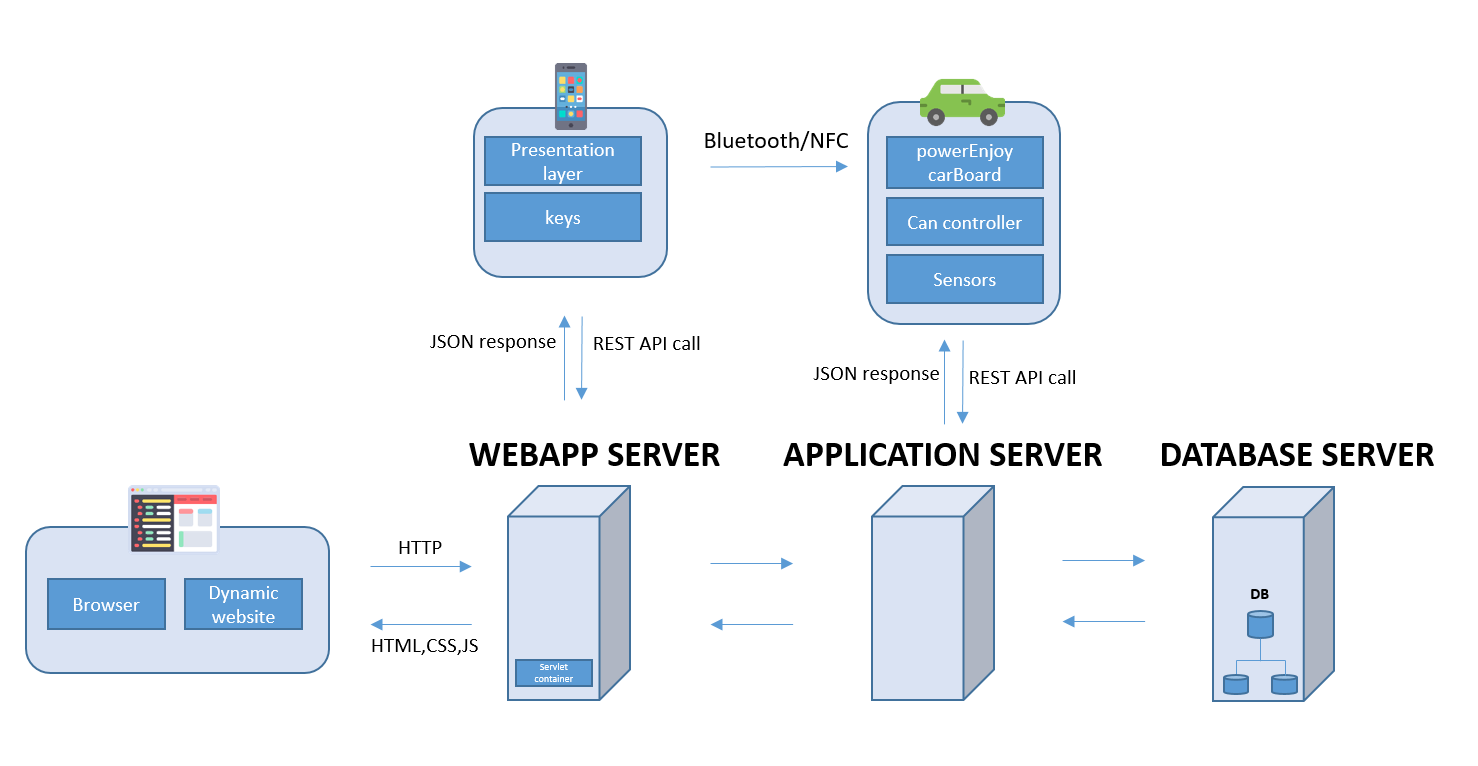
* **User:** by user is meant a person already registered in the system, so that has a profile, uses the features provided by the system and performs actions accordingly. (S)He can use all the functionalities described below (see Functional Requirements).
* **Guest:** a guest is a person that probably 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.
  1. ***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

1. ***Architectural design***
   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.

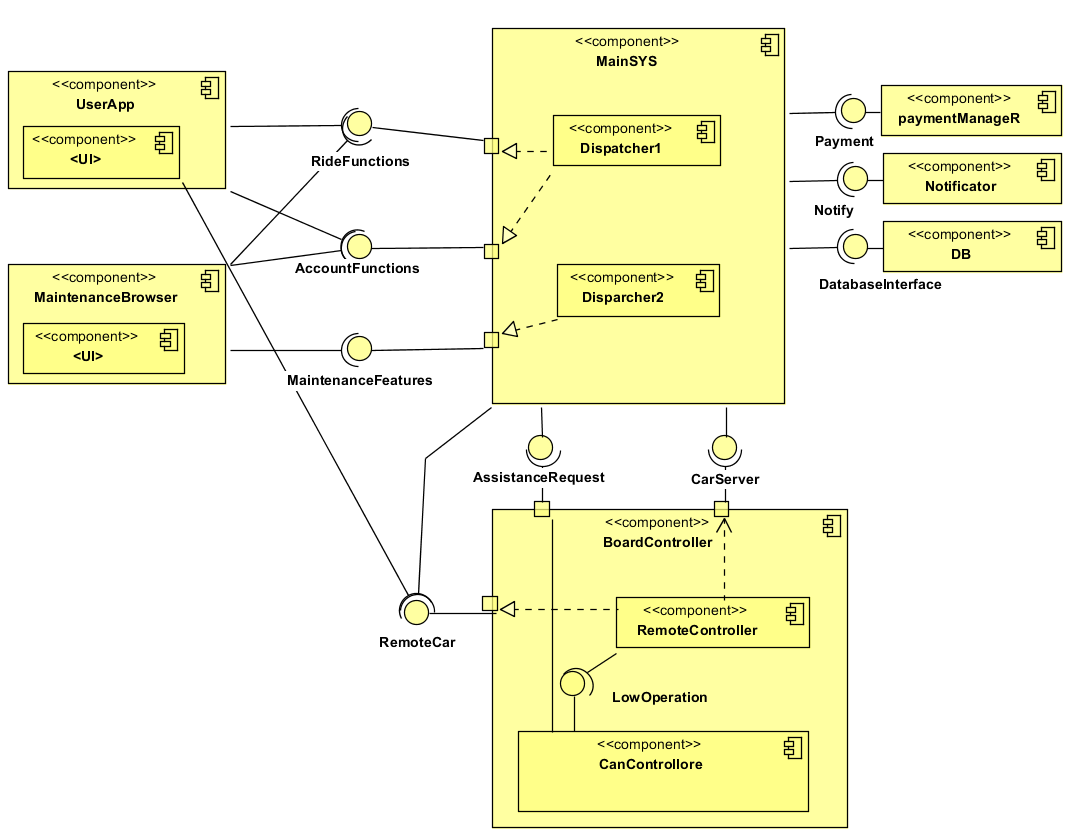
Administrators can manage and edit all data stored into the database, assign tasks to operators and manage car's remote functionality handling emergencies.

In order to provide interoperability between the main elements of PowerEnJoy system are (the Central System, the Board Controller and the users’ smartphones) [Internet](https://en.wikipedia.org/wiki/Internet) Representational state transfer (REST) [web services](https://en.wikipedia.org/wiki/Web_service) are used. REST-compliant web services allow requesting systems to access and manipulate textual representations of [web resources](https://en.wikipedia.org/wiki/Web_resource) using a uniform and predefined set of [stateless](https://en.wikipedia.org/wiki/Stateless_protocol) 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.

1. ***High level components and interactions***
   1. ***Component view***

III. Overview of PowerEnJoy Platform

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 platform to be designed provides a hardware/software interface, the Board Controller(BC), which allows the system to interact with all the vehicles in a uniform way. It relies on mobile devices (è un dispositivo che utilizza android, scelta attuata in ottica di una migliore scrittura e gestione del codice vlablsdcf ) 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.

**2. 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.)
* interagisce con l’utente attraverso un display touchscreen: le prinicipali funzionalità sono illustrate nella sezione relative I mockup;
* interact with the Central System to communicate the information on its status and the status of the vehicle, receive commands (eg. ADD TO RASD 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 Controllervehicles to manage the entire fleet, 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.

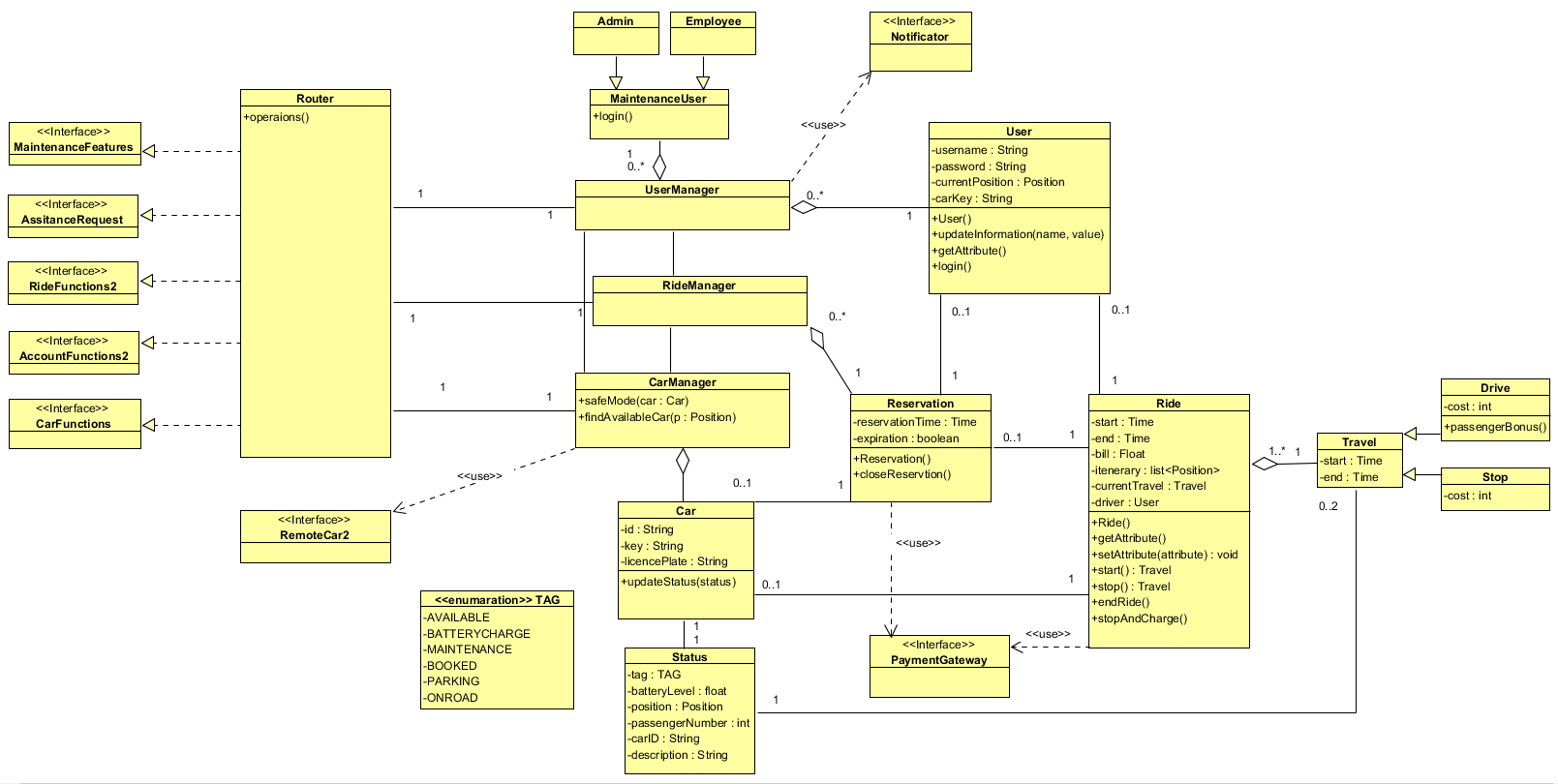
**Interaction between Board Controller and User’s Smartphone**

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.

RASD: mettere che l’nfc e il blutrhturhtu garantiscono che l’utente non posssa sbloccare l’auto a distanza.

**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.

* 1. ***Class***

The system composes of:

1. Interfaces:

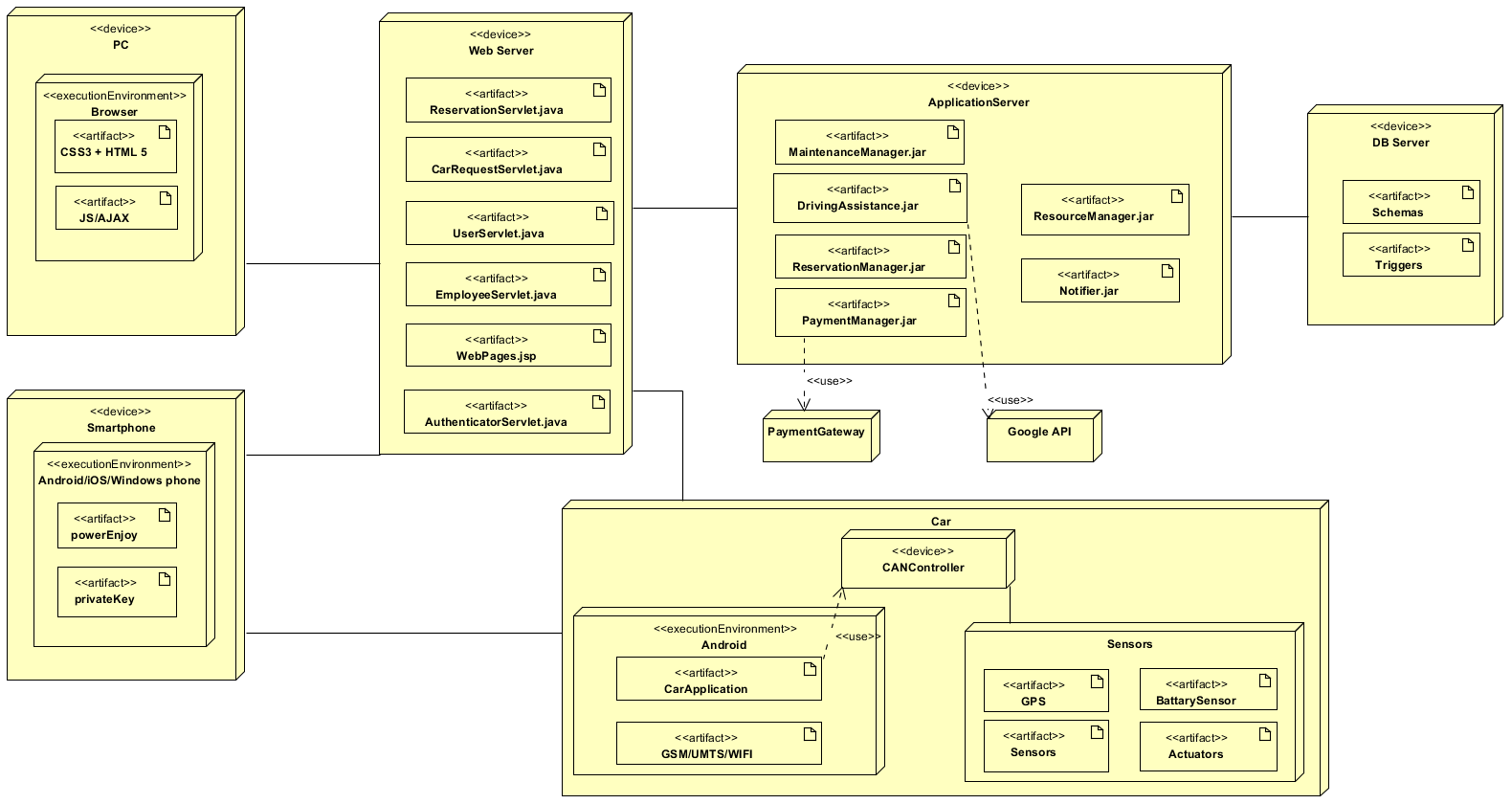
* maintenanceFeatures
* assistanceRequest (NOME FIXXARE)
* rideFunctions
* accountFunctions
* carStatus: contain the set of functionalities that the system provide to users.

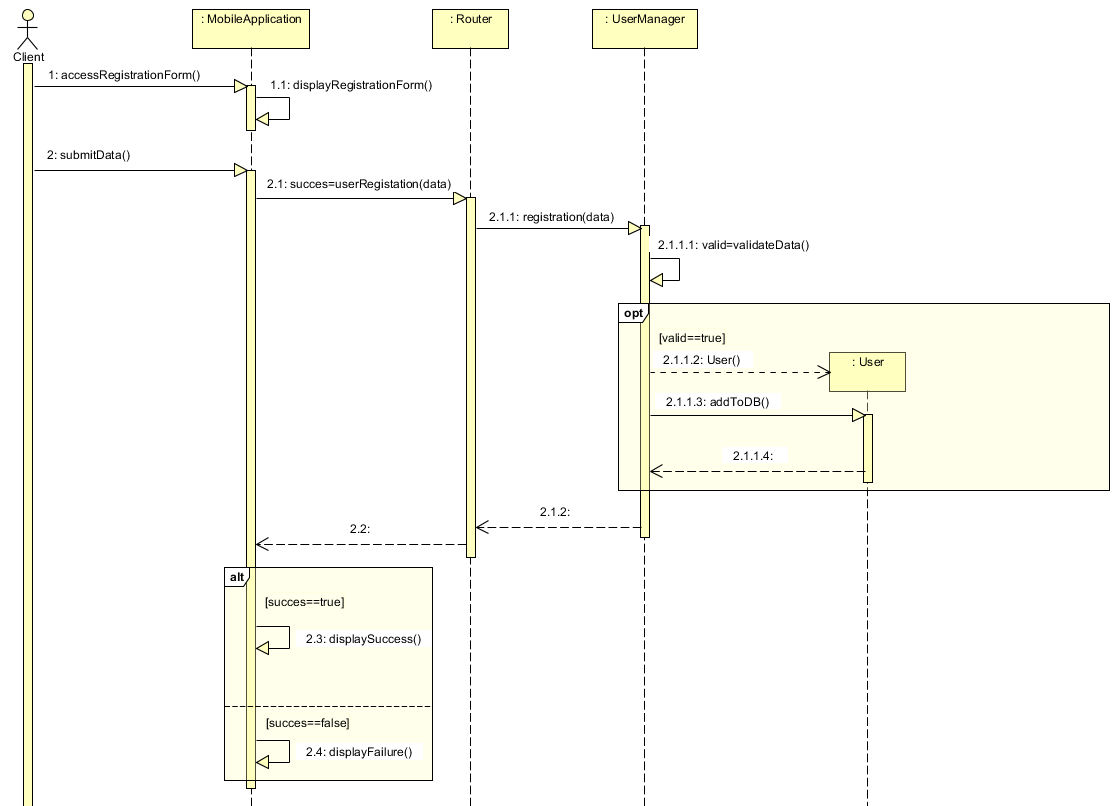
1. main elements (shown on the right):

* 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 the expiration of those not unlocked within an hour from reservation time. It is also able to manage ride functions, payments and travel management.
* CarManager handles all the cars, it updates their status and interacts with remote car functionalities.

Router will be programmed to guarantee security aspects: All the functions are activated 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.

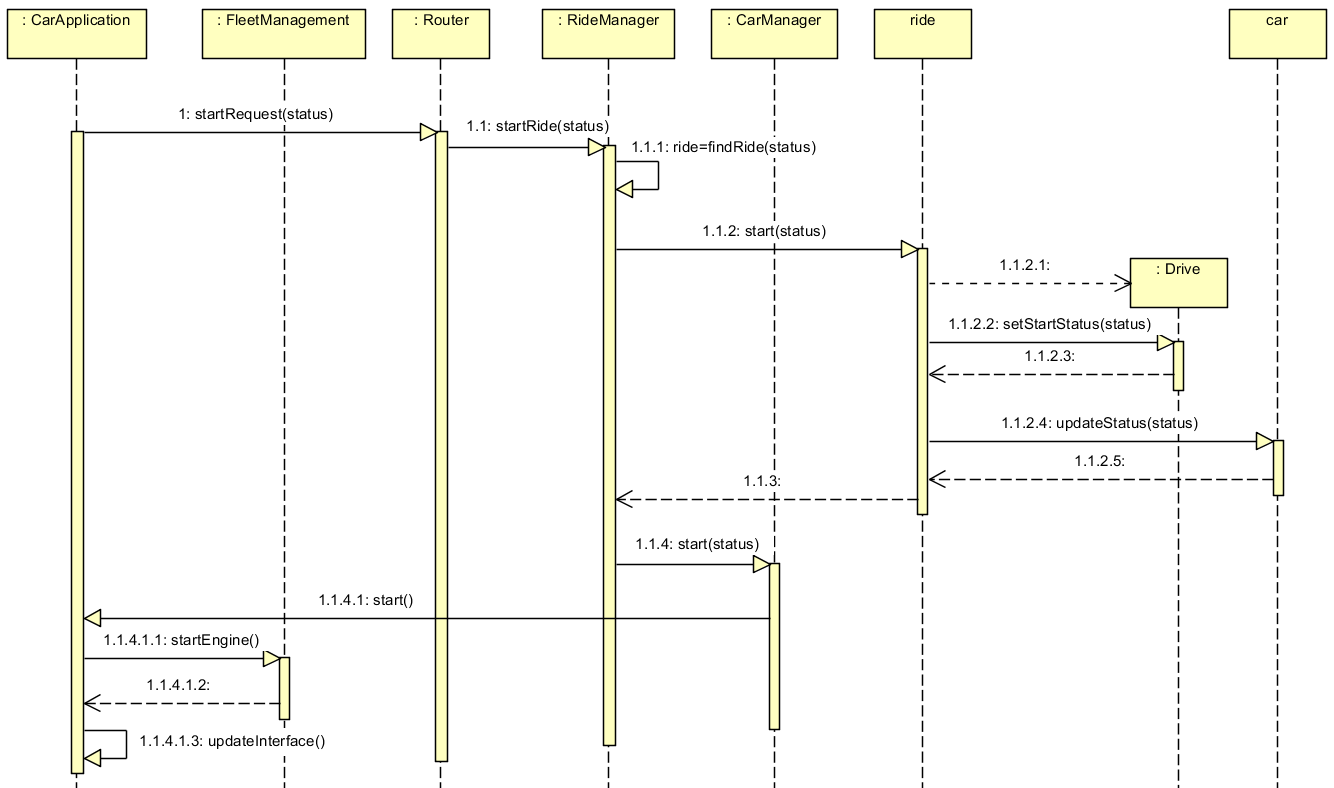
* 1. ***Deploying view***

******

* 1. ***Runtime view***
     1. ***SD: Registration***

this sequence 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 for the updating of the database. the last operation performed by the mobile application is the update of the user interface with the final outcome.

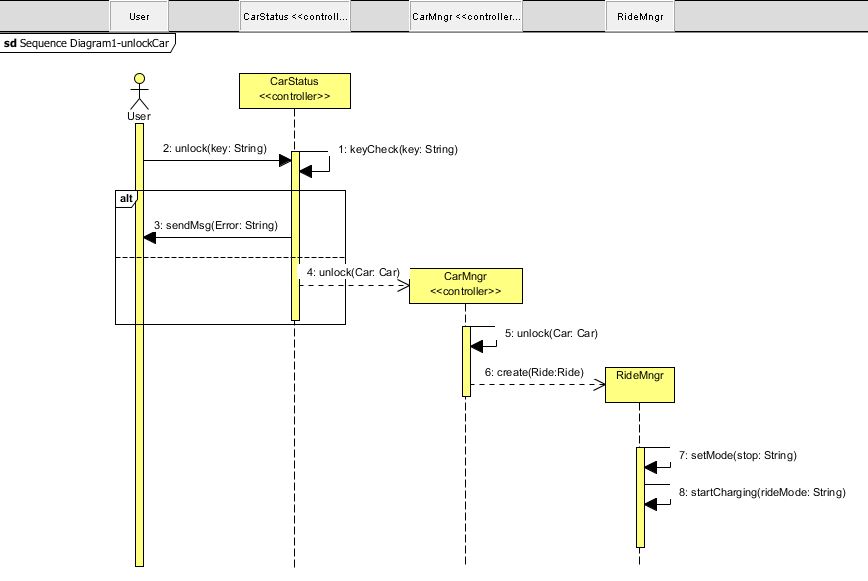
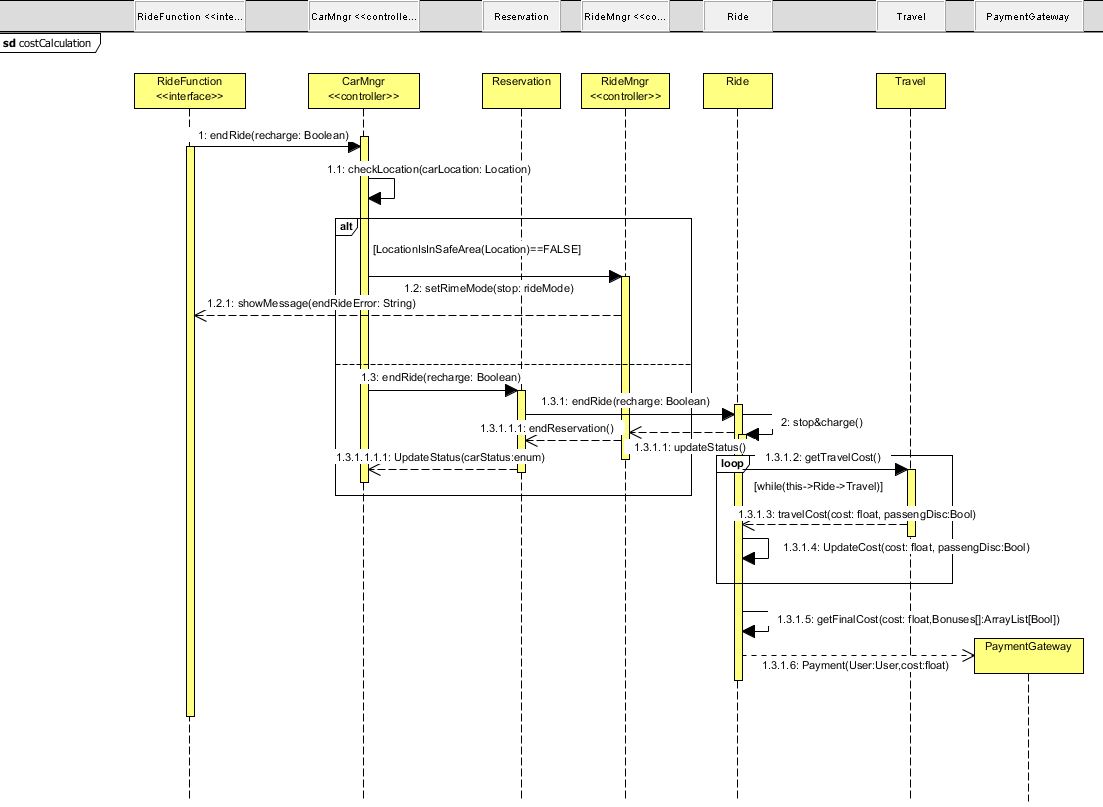
All the others functionalities enclosed in the AccountFuntions interface had the same structural behaviour of this sequence.

* + 1. ***SD:StartDriveTravel***

All the interactions between the car and the system are similar to 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: the RemoteCar interface provided by the server installed inside each car and the CarStatus interface provided instead 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.

* + 1. ***SD:UnlockCar***
    2. ***SD:Payment***
  1. ***Component interfaces***

The Component Interface, as an architecture-level artifact, is here used to illustrate the technical software architecture as an interconnected model.

Key to this model are the elements hereby described.

* 1. ***Architectural styles and design patterns***
     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 )

Using a REST web service, requests made to a resource's [URI](https://en.wikipedia.org/wiki/URI) will elicit a response that may be in [XML](https://en.wikipedia.org/wiki/XML), [HTML](https://en.wikipedia.org/wiki/HTML), [JSON](https://en.wikipedia.org/wiki/JSON) and other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide [hypertext](https://en.wikipedia.org/wiki/Hypertext) links to other related resources or collections of resources. Using [HTTP](https://en.wikipedia.org/wiki/HTTP), as is most common, the kind of operations available include those predefined by the HTTP verbs GET, POST, PUT, DELETE and so on. By making use of a stateless protocol and standard operations REST systems aim for fast performance, reliability, and the ability to grow, by using reused components that can be managed and updated without affecting the system as a whole, even while it is running.

“

The Central System of our platform acts both as a web front-end for users and as back-end that offers API (Application Programming Interface) used by cars’ Board Controllers to interact with the Central System itself. The connection medium used is a mobile data channel to allow board controllers and the main system to communicate in order to manage the fleet of vehicles. The same channel is used, at regular intervals, to transfer vehicle information such as diagnostic information, usage statistics, and information about the trip (for example the current position detected by GPS, the battery charge status etc). User interacts with the system via the smartphone App, which communicates with the Central System via a WiFi or mobile data connection in order to reserve the vehicles and to recover the software key that is required to unlock a vehicle booked. The software key is exchanged between the client App and Board Controller via a Bluetooth channel; it is used to open / close doors.

***“***

* + 1. ***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
  + 1. ***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.
  1. ***Other design choices***

The storage of data is handled by a 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.

1. ***Algorithm design***
   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;

}

}

* 1. ***Fair distribution***

*Distributor.java*

**import** java.util.ArrayList;

**public** **class** Distributor{

/\*

\* COME FUNZIONA LA COSA?

\* 1) L'ALGORITMO TIENE SEMPRE AGGIORNATO IL VALORE DELLA DISTANZA MEDIA (AVG) FRA TUTTI I VEICOLI

\* 2) PER OGNI MACCHINA SOGGETTA AD UNO SPOSTAMENTO ATTO A GARANTIRE UNA DISTRIBUZIONE EQUA VIENE IDENTIFICATA LA MACCHINA A DISTANZA MINORE

\* 3) VIENE IDENTIFICATO COME PARCHEGGIO "GIUSTO" QUELLO A DISTANZA (AVG) DALLA MACCHINA PIU' VICINA. LA DIREZIONE E' CASUALE.

\*

\* L'OBIETTIVO E' QUELLO DI TENERE LE MACCHINE LONTANE DAI GRANDI AGGLOMERATI DI ALTRE AUTO SENZA CHE QUESTE SIANO RELEGATE IN POSIZIONI ISOLATE.

\*/

**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);

}

// calcola la media delle distanze tra le macchine

**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;

}

// restituisce l'auto più vicina alla macchina 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;

}

// genera un nuovo valore della x per la macchina

**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;

}

// genera un nuovo valore della y per la macchina

**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;

}

// calcola la distanza euclidea fra due veicoli

**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;

}

// data una macchina c, le trova una collocazione che garantisca una distribuzione equa

**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;

// confini della città intesi in un range (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);

}

// muove la macchina casualmente entro i confini della città

**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;

}

// muove la macchina in una posizione specifica

**public** **void** move(**int** x, **int** y){

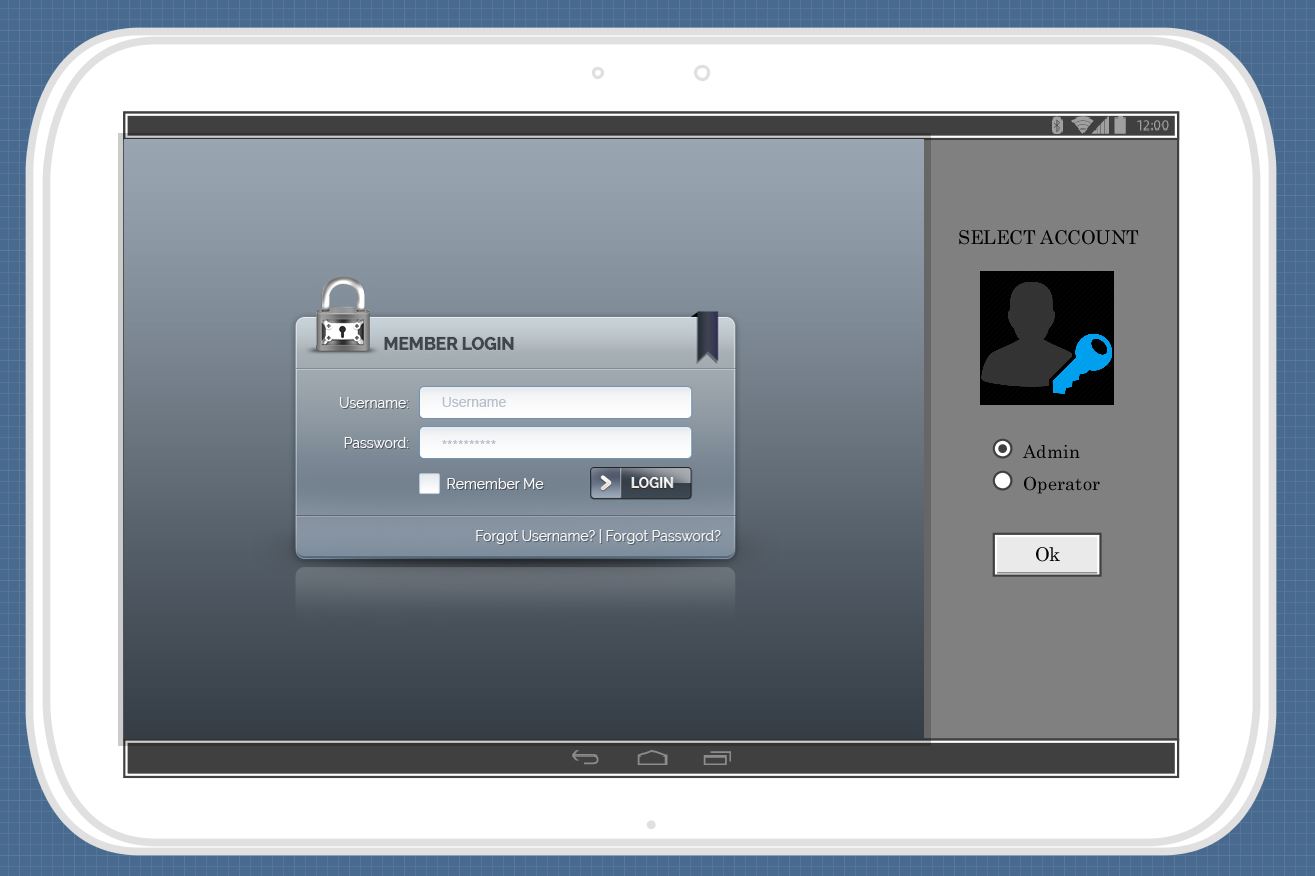
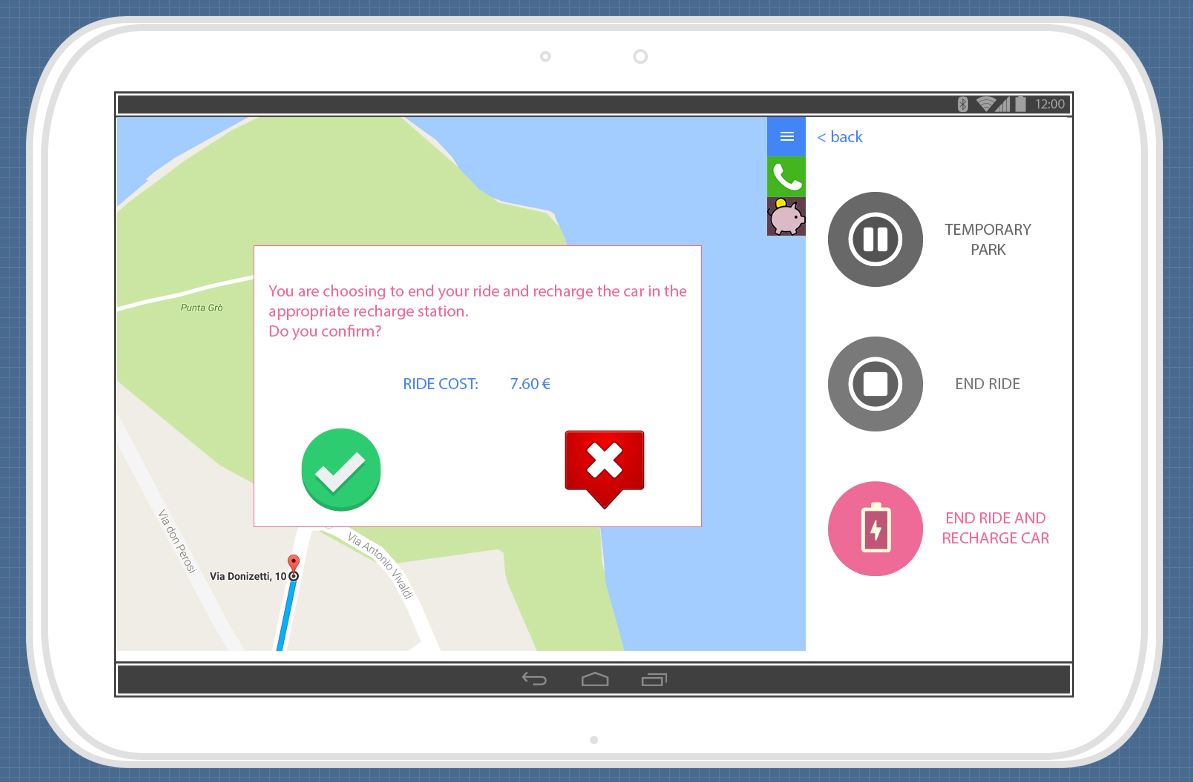
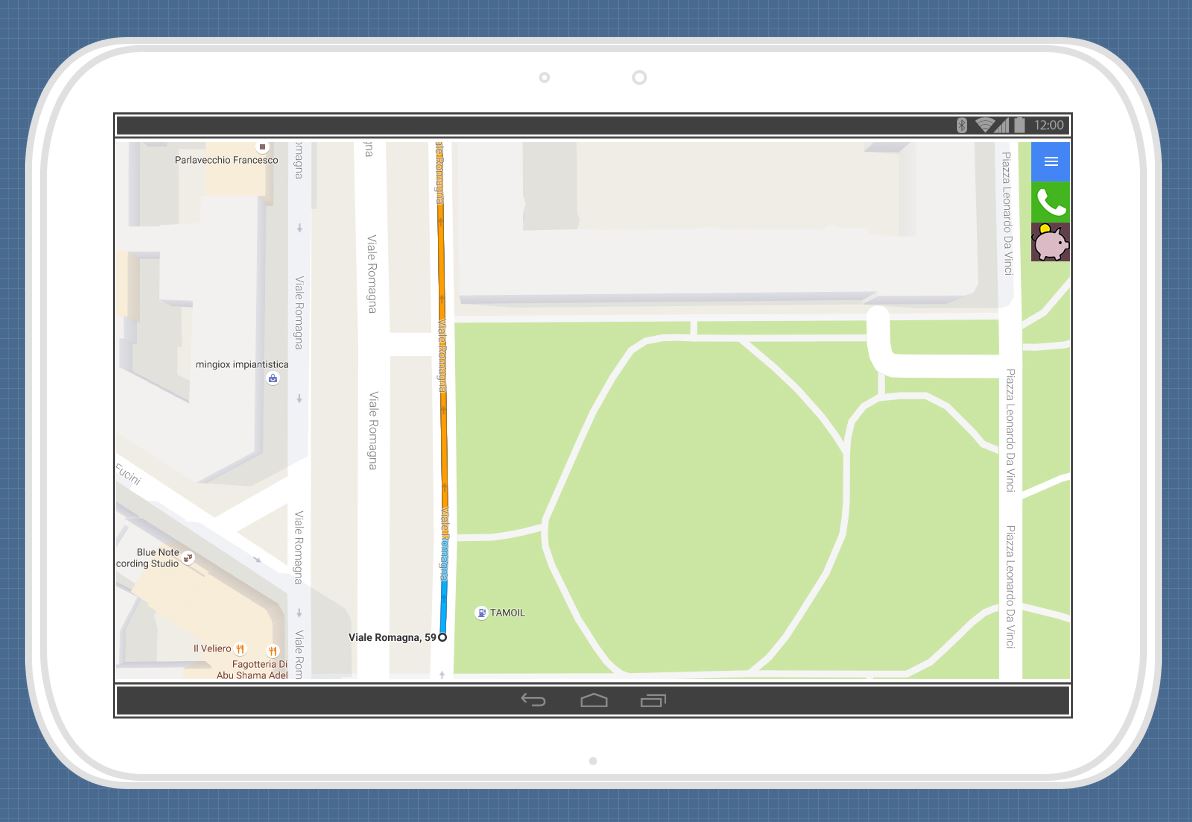
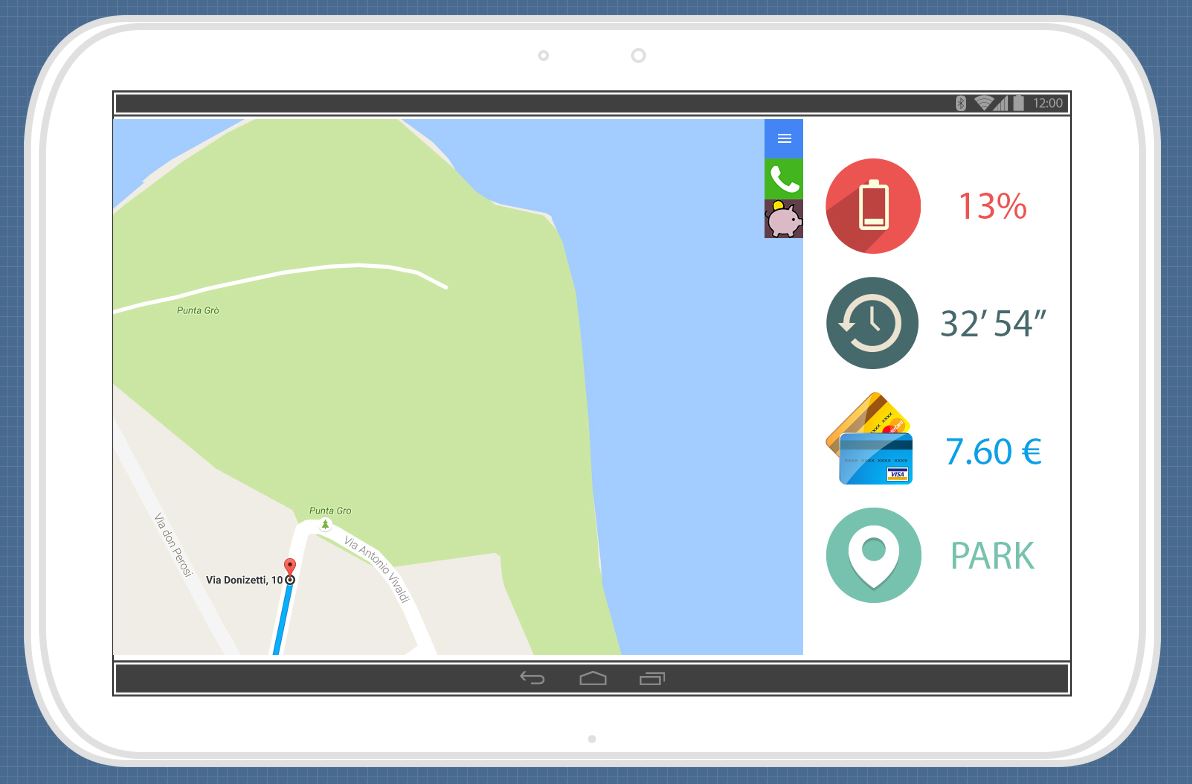
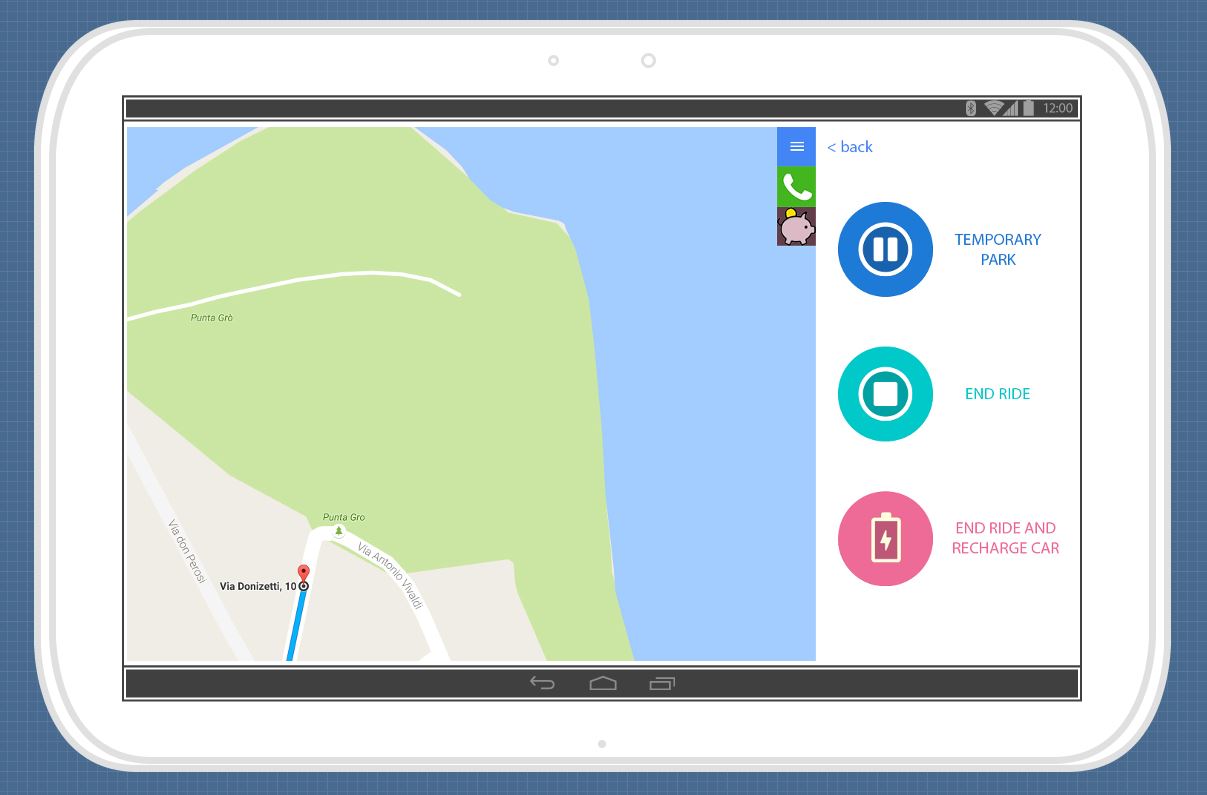
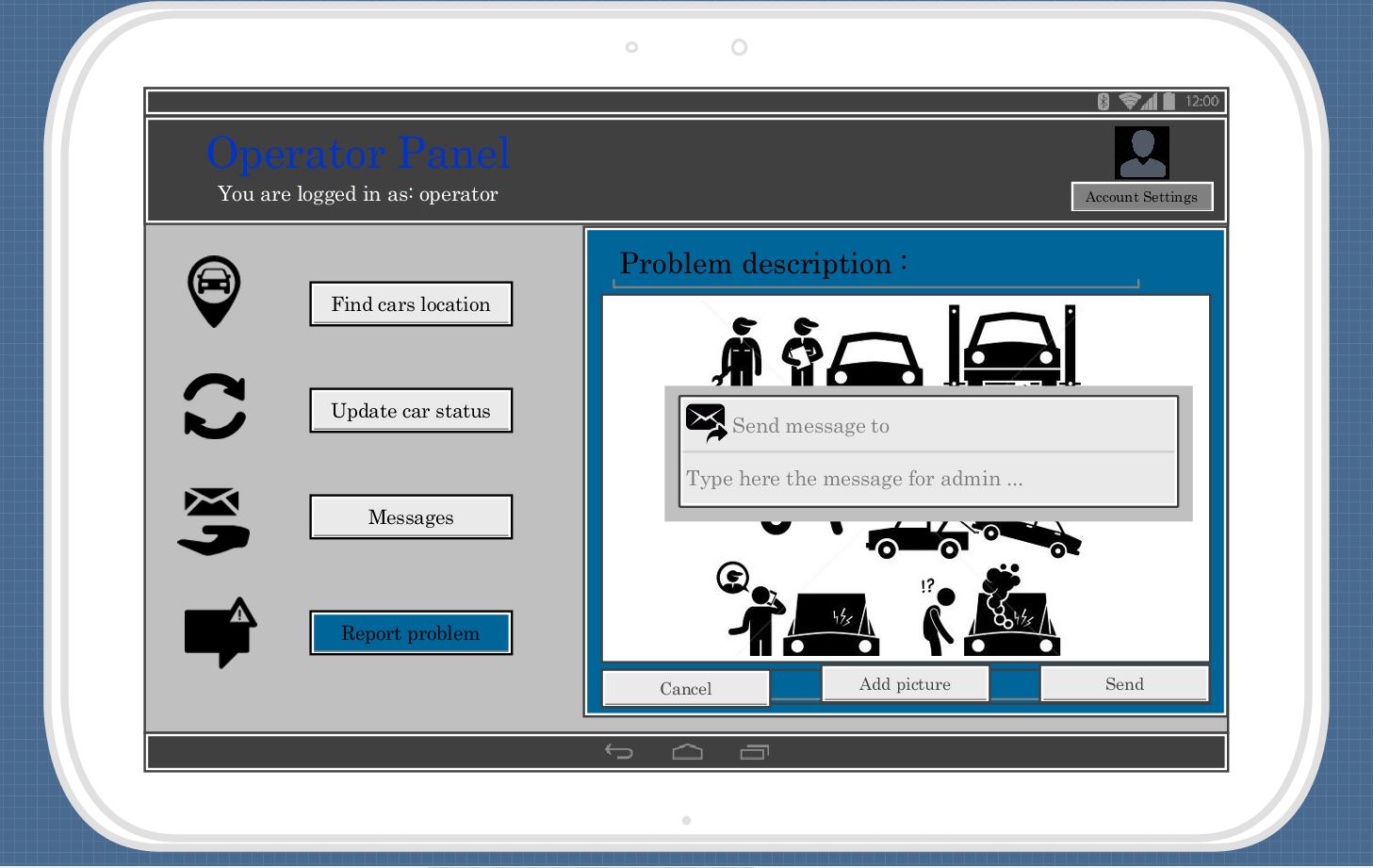
**this**.x = x;

**this**.y = y;

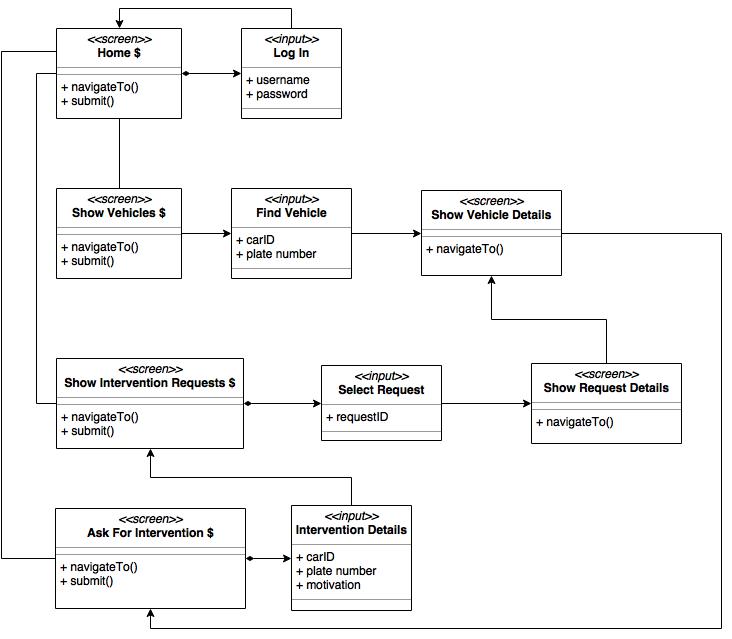
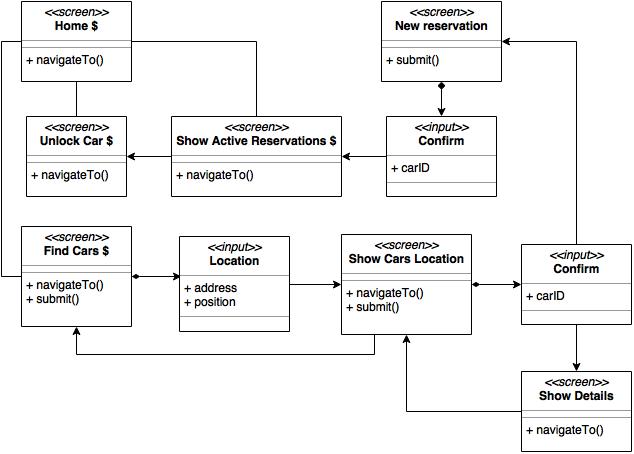
}

}

1. ***User interface design***
   1. ***Mockups***

******

* 1. ***UX diagrams***



1. ***Requirements traceability***

aaa

1. ***Revision***
   * 1. ***Software and tools used***

The following software have been used:

* Microsoft Word (document writing)
  + 1. ***Team work***

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

|  |  |
| --- | --- |
| *Member* | *Hours of work* |
| Cattaneo Davide |  |
| El Hariry Matter |  |
| Frontino Francesco |  |