POLITECNICO
MILANO 1863

PROJECT for SOFTWARE ENGINEERING 2

Design Document v2

Politecnico di Milano
A.A. 2016-2017
Prof.ssa Elisabetta Di Nitto

Students:
Diego Gaboardi
Giorgio Giardini
Riccardo Giol

INDEX

1 INTRODUCTION

1.1 PURPOSE

The main purpose of this document is to give a functional description of PowerEnjoy system, specifying better than in RASD our architectural choices and explaining to developers all technical details, unifying them in a global and stable vision of the project.

In particular, this document aims to identify:

- The high level architecture of the system
- The design patterns and main algorithms
- The components and their interfaces
- The runtime behaviour of the system

1.2 SCOPE

As just mentioned in RASD document, the project is about PowerEnJoy, a system of car-sharing that employs only electric cars, operating on the territory of Milan. It provides users to reserve and use shared cars of the system in a specific area, paying in function of driving time.

PowerEnjoy application needs two different types of people:

- Clients: daily users of the service;
- Assistants: employees of the company, with the task of adjusting and redistributing cars around the city.

Both of them must have an account and must be successfully login in order to take advantage of system functionalities provided by our mobile application.

Furthermore, clients, once they are using the car, can interface to the system also by car monitor, for example seeing real-time the amount of money they have to pay.

To be able to use the service a client has to associate his driver's licence and a valid credit card to its profile instead, for assistant, is sufficient to be correctly hired by the society.

After making the reservation, the client is able to open the car scanning a QR code. There are 5 minutes for the driver to get himself comfortable in the car after which the payment will start. At the end of the travel the driver has to leave the car in a safe area, and the payment will be automatically charged to his credit card.

Main purpose is to make an efficient system with an easy to use interface, in order to assure a full use of PowerEnjoy resources and an absolute respect of the environment. For this reason, the application wants to reward virtuous users with some special discounts: for example, if they transport other people or leave the car in a recharging

area, they manifest a particular attention to environment issues and so they will have a discount on last ride.

1.3 DEFINITIONS, ACRONYMS, ABBREVIATIONS

- **RASD**: requirements analysis and specifications document;
- **DD**: design document;
- **API**: application programming interface, it is the way in which our application communicates with other systems;
- **MVC**: model view controller;
- **REST**: REpresantional State Transfer;
- **UX**: user experience design;
- **BCE**: business controller entity;
- **RDBMS**: relational database management system;
- **MySQL**: my structured query language, is the rdbms used by our application for data administration;
- **PHP**: personal home page. Is the server-side scripting language used by our application;
- **JSON**: JavaScript Object Notation;
- **GPS**: global positioning system;
- **QR** Code: quick response code;
- **GUI**: graphical user interface.

1.4 REFERENCE DOCUMENTS

- RASD (RASD-Gaboardi_Giardini_Giol.pdf)
- Specification document : Assignments AA 2016-2017.pdf
- Example:
    - Structure of the design document.pdf
    - Sample Design Deliverable Discussed on Nov. 2-1.pdf (myTaxiService)

1.5 DOCUMENT STRUCTURE

- **Introduction**: in this section we explained the reasons of the Design Documents introducing the parts covered in the following chapters. In particular, we underlined the differences with the RASD document already released.

- **Architecture Design**: in this section we spoke about the following arguments:
  - Overview: in this part we explained how our application is divided in tiers.
  - High level components: in this part we introduced the main components of our application explaining how they exchange information.
  - Component view: we divided our application in several components explaining their functions.
  - Deployment View: we listed the components that have to be deployed necessarily to run our application.
  - Runtime view: we showed through several sequence diagrams how components interact to fulfil the most important tasks.
  - Component interface: we showed the interface implemented by components.
  - Selected architectural styles and patterns: we listed the protocols and design patterns used.

- **Algorithms Design**: in this chapter we explained the most critical parts of our application through pseudocode (similar to C and Java).

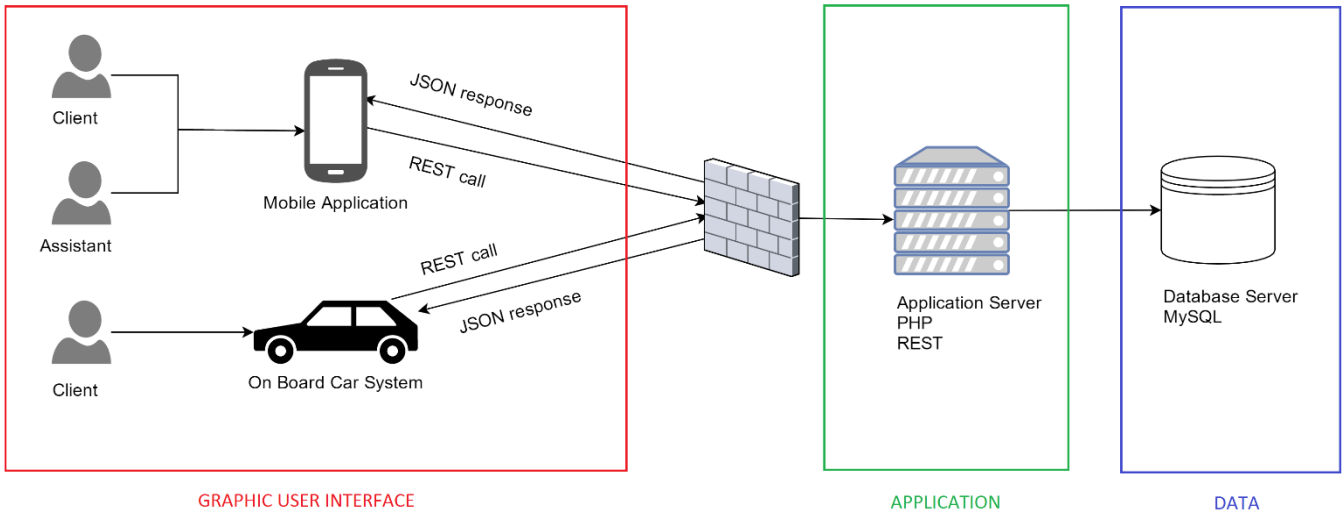
- **User Interface Design**: in this chapter we focused on user experience through UX and BCE diagrams.

- **Requirements Traceability**: in this chapter we explained how the components defined in this document are able to fulfil the requirements introduced in RASD document.

2 ARCHITECTURAL DESIGN

2.1 OVERVIEW

The system of PowerEnJoy is divided in three tiers.



On the client and assistant sides there is an app with a static GUI structure that charges data dynamically from the application server using the framework Apache-Cordova. So a request sent by the app arrives to the application server where it is elaborated. The data requested are so asked to the database server. Thus the answer obtained is sent to the app in the correct format.

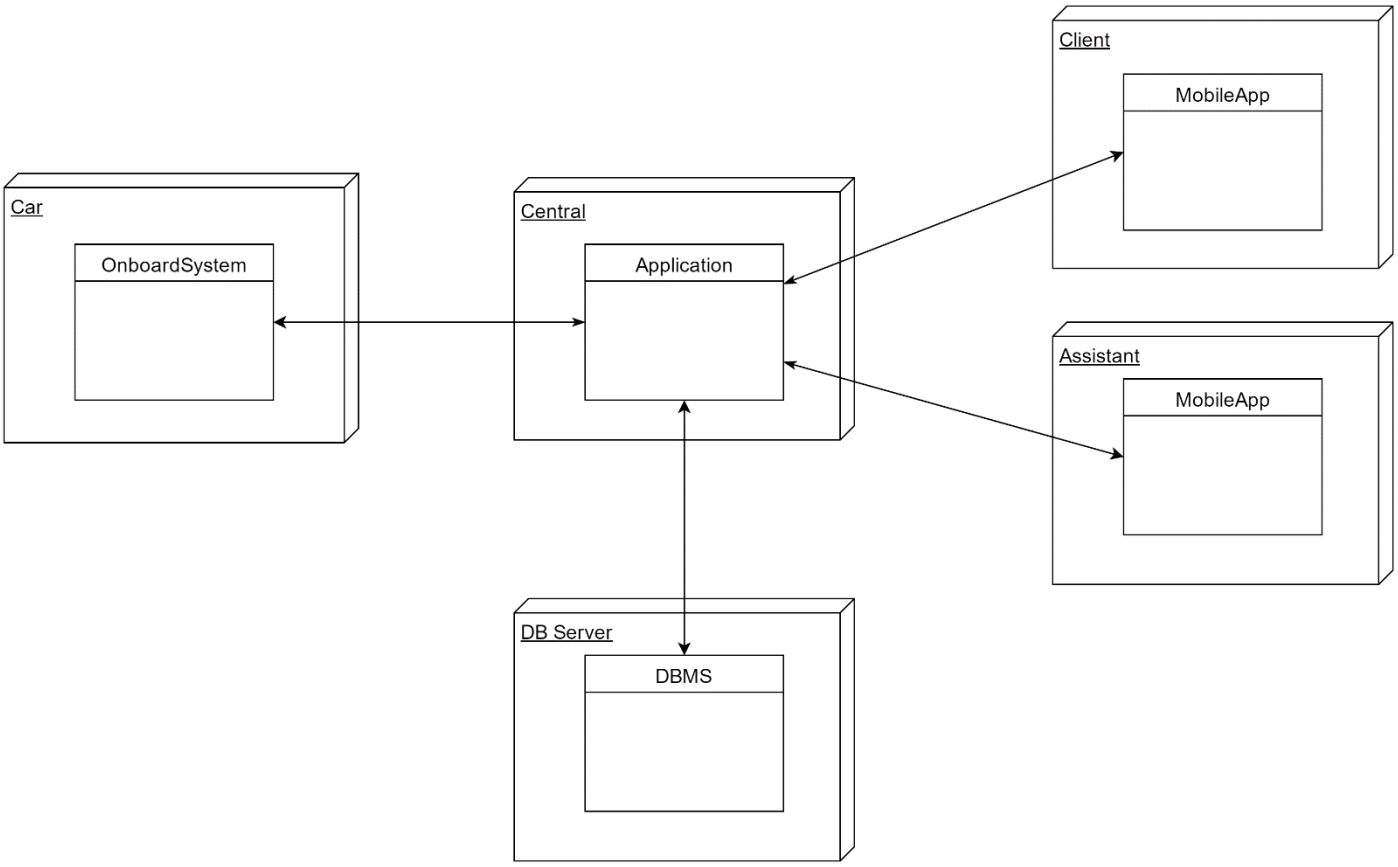
The three tiers are independent, they have only to communicate through the correct interface. The GUI tier and the Application tier communicate through internet, so the application server is protected by a firewall. The Application Server and the Database server are connected in a LAN, so there is no need of a firewall between them.

2.2 HIGH LEVEL COMPONENTS AND THEIR INTERACTION

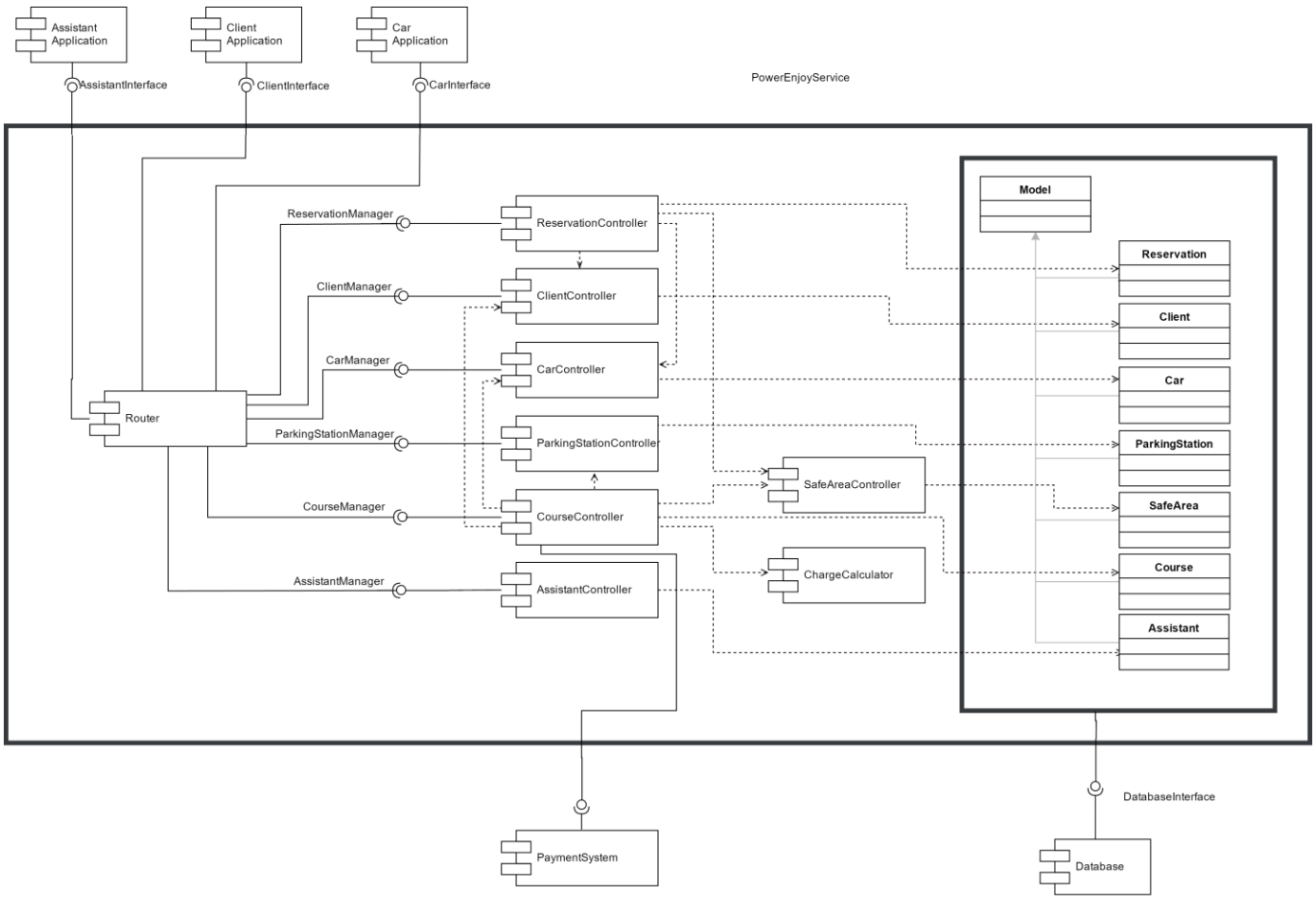
The high level components of our system are composed by a central component that manages all requests from the client and assistant components, and answers them using the data from the DB Server. All the information and responses from the central component are visualized by the client through the Mobile App. In some case the central sends a notification to the client.

The client and assistant login requests are made in asynchronous mode at the central component, that performs a request in synchronous mode at the DB Server. The information request is made in asynchronous way at the central, and gave to the client the data for starting a reservation. This kind of client request is made in asynchronous way, and after has been performed, the central component sends back the response, and activates the procedure. The central component sets correctly all the states of clients and cars through a synchronous instruction on the DB Server. The following instructions from client to central and *viceversa* are all made in asynchronous mode.

Another component is the car. The central has always to keep the positions of all the cars. The car position requests and responses are made in synchronous mode. The car position request-response when it is leaved for a break or for stop is made in asynchronous mode.



2.3 COMPONENT VIEW



2.3.1 GLOSSARY

- **AssistantController**: it is the component which manages assistant registration and login. It stores all useful information in the database.

- **ClientController**: it handles client registration, login and profile.

- **ChargeCalculator**: calculates the amount of money that has to be paid in function of driving time and special bonuses.

- **CourseController**: manages courses changing properly client and car state, controlling saving option course, locking the car when nobody is in and making payments effective.

- **SafeAreaController**: manages the set of safe areas.

- **ReservationController**: it is the component that creates a new reservation and sets the selected client and car in reserving/reserved state.
- **ParkingStationController**: manages the set of parking stations.

- **CarController**: manages the electric cars of the system updating properly their state and checking their current position.

- **ClientApplication**: it is the client's device in which there is our application.

- **AssistantApplication**: it is the assistant's device in which there is our application.

- **CarApplication**: it is the application situated in the electric cars.

- **Database**: it is the database of our system in which there are information about clients, assistants, cars, courses, reservations, safe areas and parking stations.

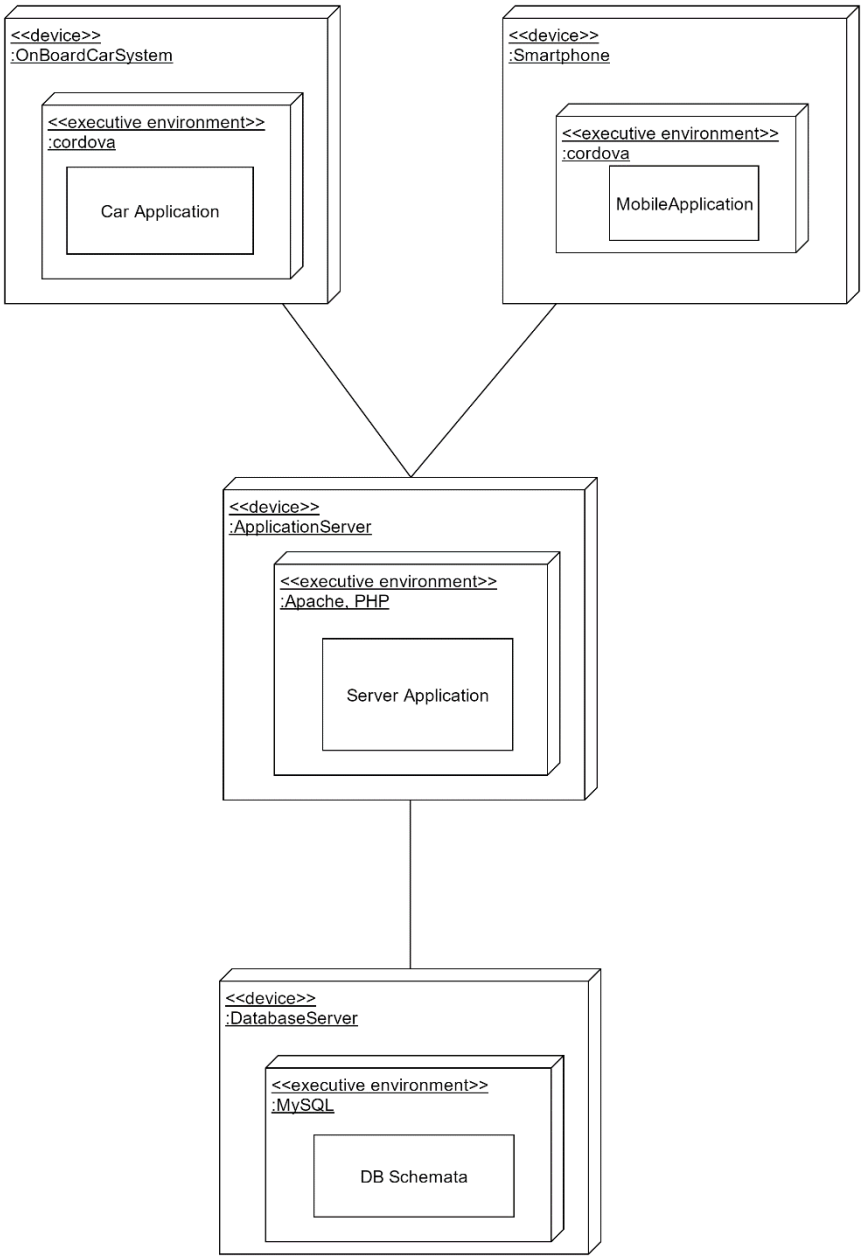
2.3.2 EXPLANATION

The component view shows a high-level representation of our system in which we can see all structural relationships between the components.

In particular, we can see how clients, assistants and car applications can require all functionalities through a specific interface that sends requests to the system which are immediately distributed to the appropriate controller by the router.

All controllers try to satisfy their tasks possibly communicating with other components or with the database which is situated outside the system and store all useful information for the correct working of the system.

2.4 DEPLOYING VIEW



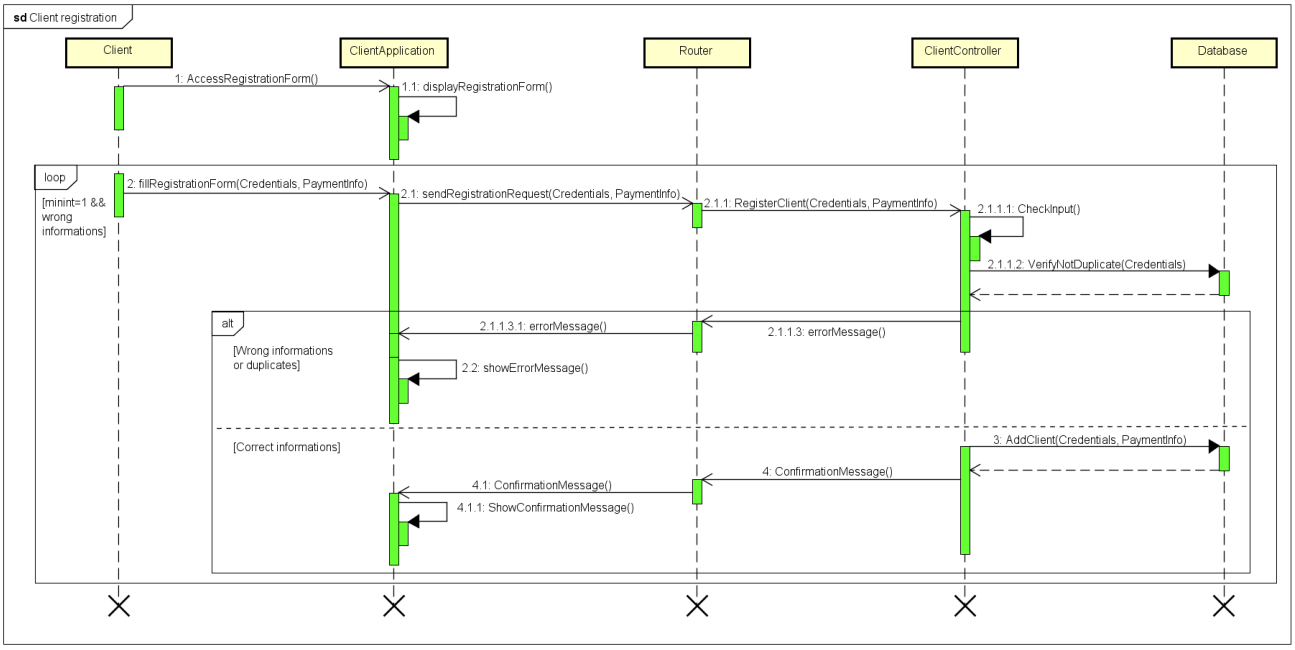
We have decided to use the framework Apache Cordova to handle our web application. Cordova allows us to easily create a multiplatform mobile app, indeed it supports Android, iOS and Windows Phone. With this framework we obtain a hybrid app that can manage some phone features through the specific API of the JS, and interchange data with the Apache server using REST requests, made in POST mode. The server responds with message in JSON format. The Business Logic is in PHP on the Apache Server. PHP provides specific APIs to interact with Apache and it is well integrated with MySQL database, always with specific APIs.

2.5 RUNTIME VIEW

In this chapter we are going to show how the most important tasks are carried out by the system components. For this reason, instead of analysing in an excessive detailed way all the steps of these operations, we concentrated on the participants of the tasks.
We considered the following situations:

- A client wants to register;
- A client wants to modify his profile;
- A client wants to enable saving option;
- A client uses the car;
- A client makes a reservation;
- An assistant changes a car state.

2.5.1 A CLIENT WANTS TO REGISTER

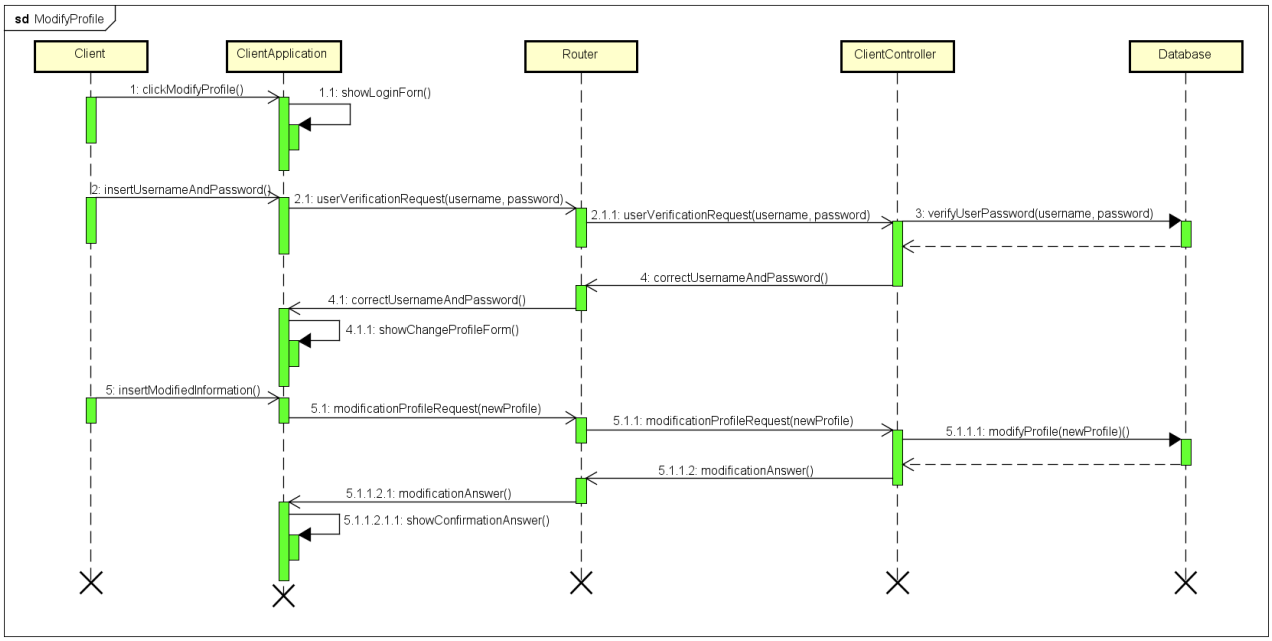


If a client wants to register, he has to go to the registration page with the mobile application and fill the correspondent form with all information needed: credentials and payment information.

Then the application sends to the system the client input in order to verify its correctness. So the request arrives to the router which transfers it to the ClientController that finally checks the validity of the user input.

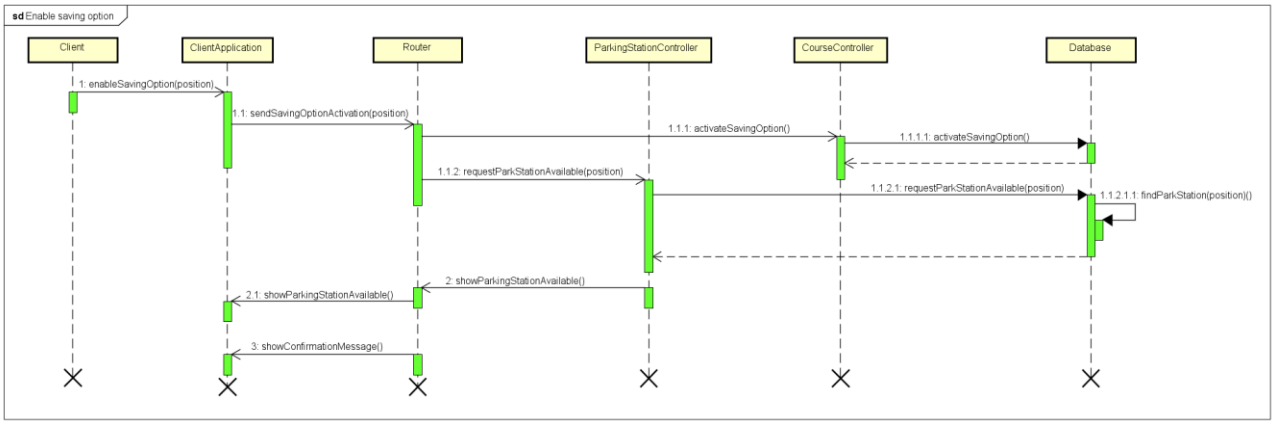
If they are correct, a new client is added to the database and the client receives a confirmation message. If data are erroneous or missing or the client is already registered, he receives an error message and his registration failed.

2.5.2 A CLIENT WANTS TO MODIFY HIS PROFILE



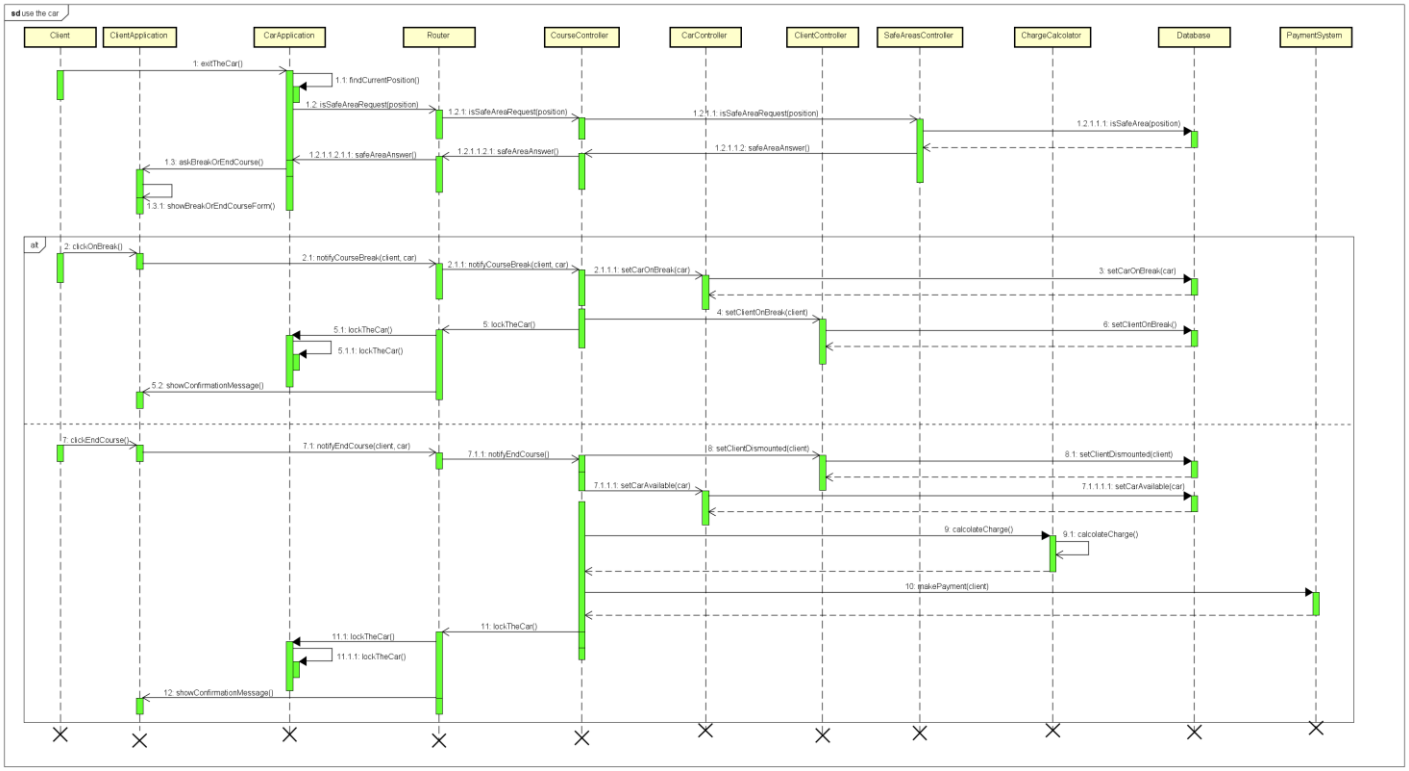
If a client has to modify his profile, he has to enter his username and password in the proper form through the mobile application. Then the ClientController, invoked by the router, verifies that the user input is correct and in that case it will ask the client to insert the modified personal information.
So the ClientController will change the client information in the database and a confirmation message is shown to the client.

2.5.3 A CLIENT WANTS TO ENABLE SAVING OPTION



The client in order to enable the saving option has to click on the relative checkbox sending a request to the System with the final destination. Then the Router will ask the CourseController to update the database while the ParkingStationCountroller will find the parking stations near the current position of the client.
So the ClientApplication will receive a map with all the requested parking stations and will show them on its monitor.

2.5.4 A CLIENT USES THE CAR

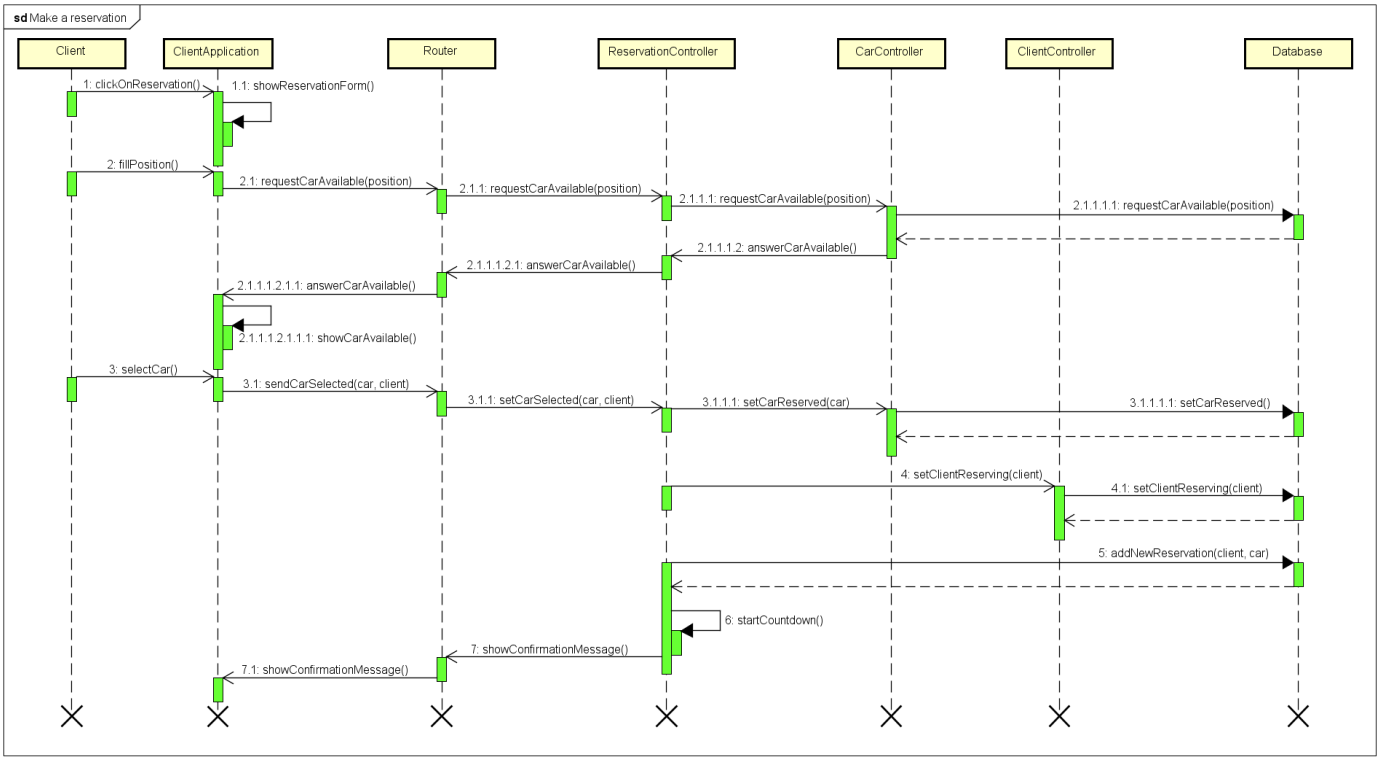


When a client finishes driving and all passengers leave the car, the CarApplication understands that nobody is on the car through the weight sensors and asks the system if its current position is a safe area. The system checks it querying the database and then notifies the ClientApplication which asks the client if he want to end his travel or to leave the car on break.

In both cases the system is notified and the CourseController (invoked by the Router) asks the CarController and the ClientController to update properly car and client states. At the same time the CarApplication is notified in order to lock the car. If the client decides to end his course the system through the ChargeCalcolator computes the amount of money that has to be paid and the payment transaction is made.

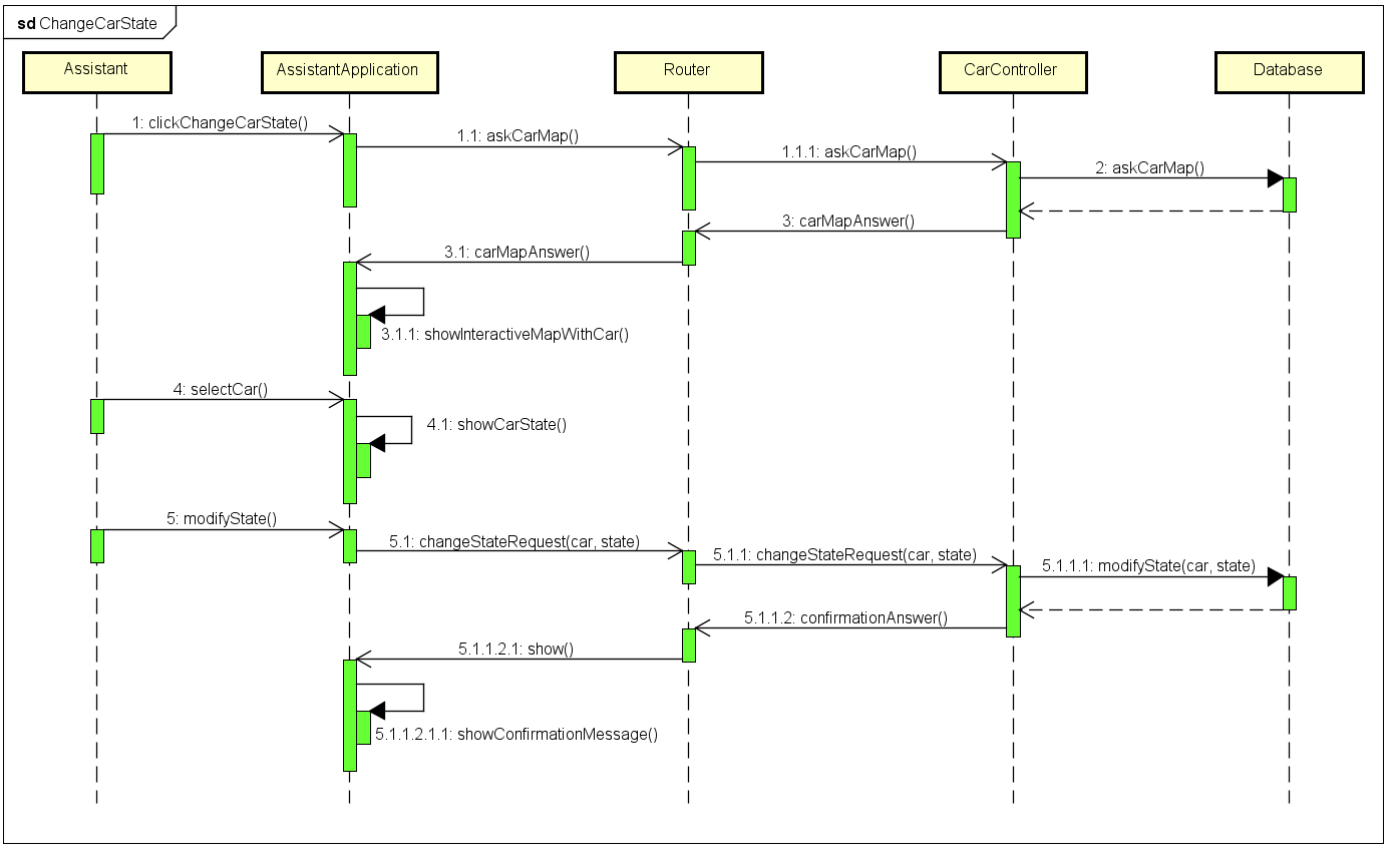
Finally, a confirmation message is shown on the ClientApplication.

2.5.5 A CLIENT WANTS TO MAKE A RESERVATON



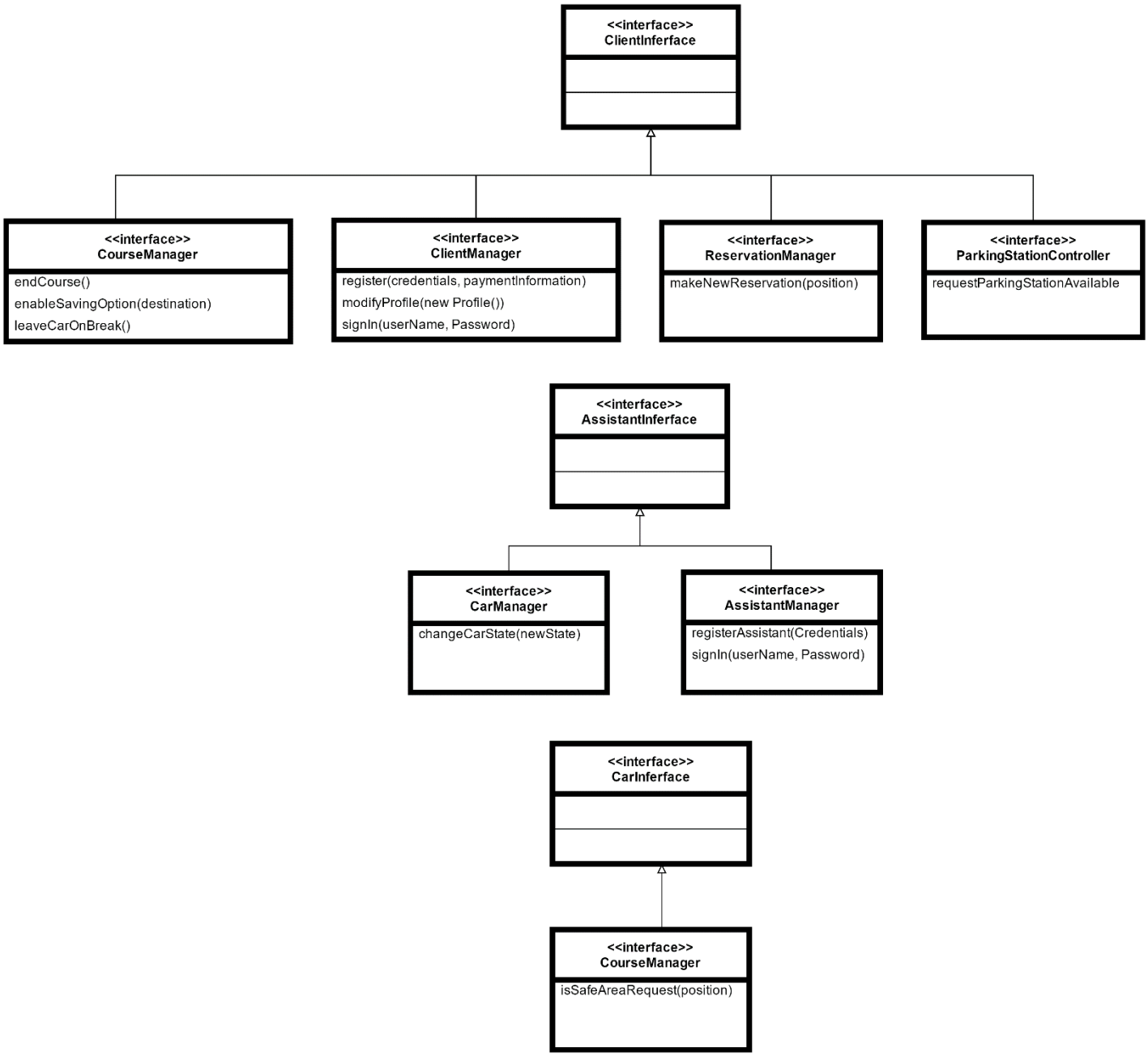
If a client wants to make a reservation, he has to insert the current position or a specific one through the mobile application.
Then the system is notified and the Router asks the Database (through the ReservationController and the CarController) to find all available cars near that position. So the ClientApplication receives all information necessary to show the user an interactive map in which he can select his car.
Then the system adds a new reservation in the database, updates car and client states and starts the reservation countdown.
Finally, the ClientApplication shows a confirmation message.

2.5.6 AN ASSISTANT CHANGES A CAR STATE



If an assistant want to modify a car state, he has to go in the proper part of his application. Then the system will be notified by the AssistantApplication and the Router will ask the CarController to give all necessary information to show an interactive map with all cars and their state.
Then the assistant will select a car and modify its current state. After modifying it, the Router will ask the CarController to change the car state also in the database.
Finally, the assistant will receive a confirmation message.

2.6 COMPONENT INTERFACES



2.7 SELECTED ARCHITECTURAL STYLES AND PATTERNS

2.7.1 OVERALL ARCHITECTURE

As seen before, our application is divided in 3 tiers:

1. The GUI tier: is a simple application that provides a graphic interface to the user, which can be classified as a thin client, because all data elaboration is managed by the second tier.
2. The Application tier: this level deals with business logic, handling the requests performed by the first tier and respond to them, often consulting the third tier.
3. The Data tier: manages all the persistent data of the system.

2.7.2 PROTOCOLS

In our system many components communicate through internet. Here there are some protocols used for manage correctly the communications.

The mobile App communicates through Cordova APIs. They use REST and JSON to formalize and interchange data with the Application Server.

The application server communicates with the data server through the protocol PDO, PHP Data Oriented, that permit it to handle the data like object with attributes.

The DBMS used is MySQL.

2.7.3 DESIGN PATTERNS

MVC

We make use of MVC architectural pattern in order to correctly separate application control (controller), data management and abstraction (model), and result visualisation (view). MVC considerably helps with the reuse of code and in the division of the work, making the whole application structure more flexible, readable and easily upgradable.

CLIENT-SERVER

In term of communication model we widely use the client-server pattern in order to correctly manage our distributed application, dividing the provider of resource (server) and resources requesters (client). All shared resource and application logic

are handled by server, while client are thin. This facilities data management and synchronization and allows also clients with a low-resources device to correctly run the application.

PROXY

With reference to the structure of the components, we make use of a design pattern of type proxy. In fact router component is an interface towards all other components. All requests are made at first to router, that is responsible for the invocation of the correct method in the specific component.

OBSERVER

The structure of the application is strongly event based so observer pattern is also widely used. This allows system to correctly response to the happening of events, like for example car parking, engine ignition, countdown expiration, etc.. promptly invoking the correct method.

FAÇADE

Another important pattern used to mask the complexity of some parts of application logic is the façade design pattern. It is used for example in the case of course controller: it acts as façade to all classes involved in the management of a single course, simplifying the access to all involved methods.

SINGLETON

Some of classes proposed in our UML model need to be instantiate one and only one time. For this particular type of classes is necessary to use singleton pattern, which guarantee the existence of a single instance in the model of involved classes. Class of this type are for example classes that manage the list of car, the list of clients or the list of safe areas of the city.

3 ALGORITHM DESIGN

This part is to give an idea to developers relating algorithms and techniques to use in the most critical sections. For this purpose it is used a mixture of C, Java and natural language.

Is taken for granted the use of an object oriented language.

All class named refers to our UML model in RASD document.

3.1 SAVING-OPTION

In order to ensure an equable distribution of cars in the city it is important to find the freest area, also considering the distance as the crow flies between the destination chosen by the client and the area itself. The proposed algorithm use a weighted average between these two factors.

```
#define DEST_WEIGHT = 1, AVAIL_WEIGHT = 1


function FindBestArea (Position destination, ParkStation [] parkStations) {

nearestAvailableParkStations (destination, parkStations); (3.2)

ParkStation currentBest = parkStations [0];

double bestValue = valueOf (destination, parkStations [0]);

for (ParkStation x : parkStations) { if valueOf (destination, x) < bestValue then
..update currentBest and bestValue }

}


function double valueOf (Position destination, ParkStation parkStation) {

return ( (parkStation.occupiedPercentage()*AVAIL_WEIGHT) + (distance(destination,
parkStation.position())*DEST_WEIGHT) ) / (AVAIL_WEIGHT+ DEST_WEIGHT);

}


function double distance (Position a, Position b) {

return sqrt((a.x+b.x)^2 + a.y+b.y)^2);

}
```

3.2 SELECTION OF NEAREST AVAILABLE PARKING AREAS TO A POSITION

This is to find the nearest available parking area to a position, restricting the global research on a specific circular area with a certain level of tolerance (radius). An algorithm of this type can be useful also to find available cars near a position or to verify if safe area constraints are respected when a client park his car

```
#define TOLL = 1

function nearestAvailableParkStations (Position destination, ParkStation [] parkStations) {

for (ParkStation x : parkStations) { if distance(destination, parkStation.position()) > TOLL or !x.available()than ..remove x from parkStations}

}
```

3.3 CALCULATING DISCOUNT

This algorithm is for calculating and applying special discounts and penalties. Note that application order does not influence final result.

```
function calculateDiscount (Course course) {

//Discount 1

If (course.numberOfPassengers >= 2 ) course.discount.add (new Discount(-0.1));

//Discount 2

If (course.car.battery > 50) course discount.add (new Discount(-0.2));

//Discount 3

If (course.car.recharging) course discount.add (new Discount(-0.3));

//Penalty 1

If (course.car.battery < 20 or distance(course.car.endPosition,
nearestParkStation(course.car.position).position) >30) course discount.add (new
Discount(0.3));

//Applying

for (Discount x : course.discount) {

course.cashAmount = course.cashAmount*(1+x);

} }
```

Number of passengers of the course are calculated making an average of the number of passengers noticed by car sensors over the whole duration of the course. During the course a function of this type is called about every 3 seconds.

```
function updatePassengersNumber (Course course, Car car) {

course.numberOfPassengers = (course.numberOfPassengers*NUMBER_OF_RELEVATION +
car.numberOfPassengers) / (NUMBER_OF_RELEVATION+1);

}
```

3.4 CONCURRENCY CONTROL

Locking of shared resources, like for example a car in the moment of reservation, is fundamental for this multiuser system in order to guarantee the consistency of the model. In the special case of cars the lock must be on the specific object and not on the whole car class. So is suggested to use a structure of this kind:

```
function Reservation reserveCar (Car car, Client client) {
if (..car is locked) than ..throw exception
else {
lock(car);
Reservation newRes = new reservation(car, client);
unlock(car);
return newRes;
} }
```

3.5 EVENT-DRIVEN ARCHITECTURE

This application is intrinsically event driven. So are needed handler on car object to react, for example, to the following events:

- Engine ignition
- Car parking
- Reservation countdown expiration

Happening of events is detected by sensors, that interface with system, or directly by software, like in the case of the countdown expiration.

```
object engineIgnitionHandler extend Listener {
Car car;
actionPerformed () {
..end reservation, create new course
} }

object carParkingHandler extend Listener {
Car car;
actionPerformed () {
if (car.isInSafeArea()) ..ask client if it's a break or a pause
} }

object reservationCountdownHandler extend Listener {
Car car;
actionPerformed () {
..end reservation, charge client
} }
```

4 USER INTERFACE DESIGN

4.1 MOCKUPS

The mockups of our client application and of the inboard car system are already present at the point 1.10 non-functional requirements of our RASD.

4.2 UX DIAGRAMS

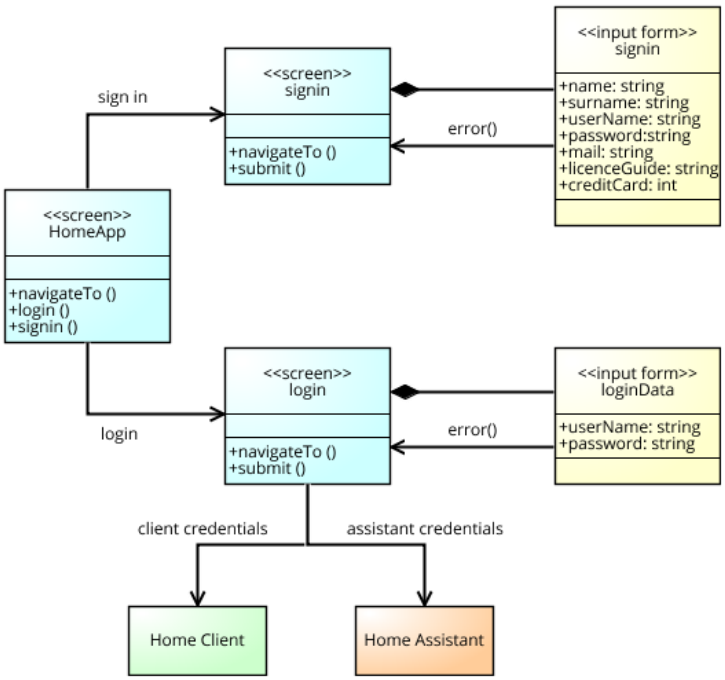
Here are presented the UX diagrams about:

- the login and sign in of the users;
- the navigation in the app for the client and for the assistant;
- the navigation on the system on board of the car.

With these diagrams we are meant to clarify the graphic interface of the mobile application. The UX diagram shows the different pages of the app and how to pass from one to another. Furthermore there are highlighted the various input forms, placed when it is requested an information from the user, like the QR scanning or the insertion of the credentials.

4.2.1 LOGIN AND SIGN IN UX

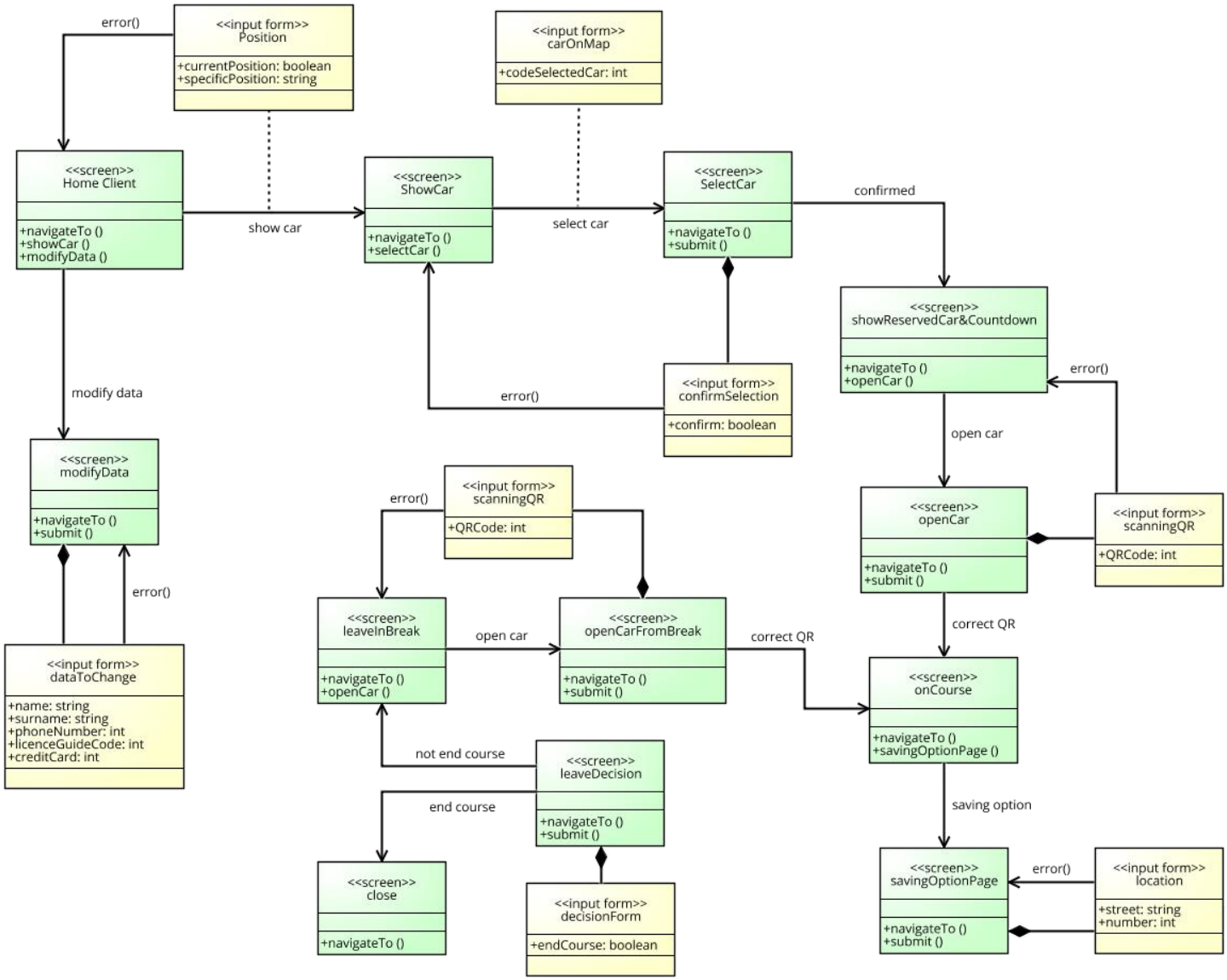
This is the UX for the login and sign in.



If the inputs are correct in the login page, the next screen will be the home client or the home assistant in function of the credentials inserted.

4.2.2 CLIENT MOBILE APP UX

This is the UX of the client mobile app.

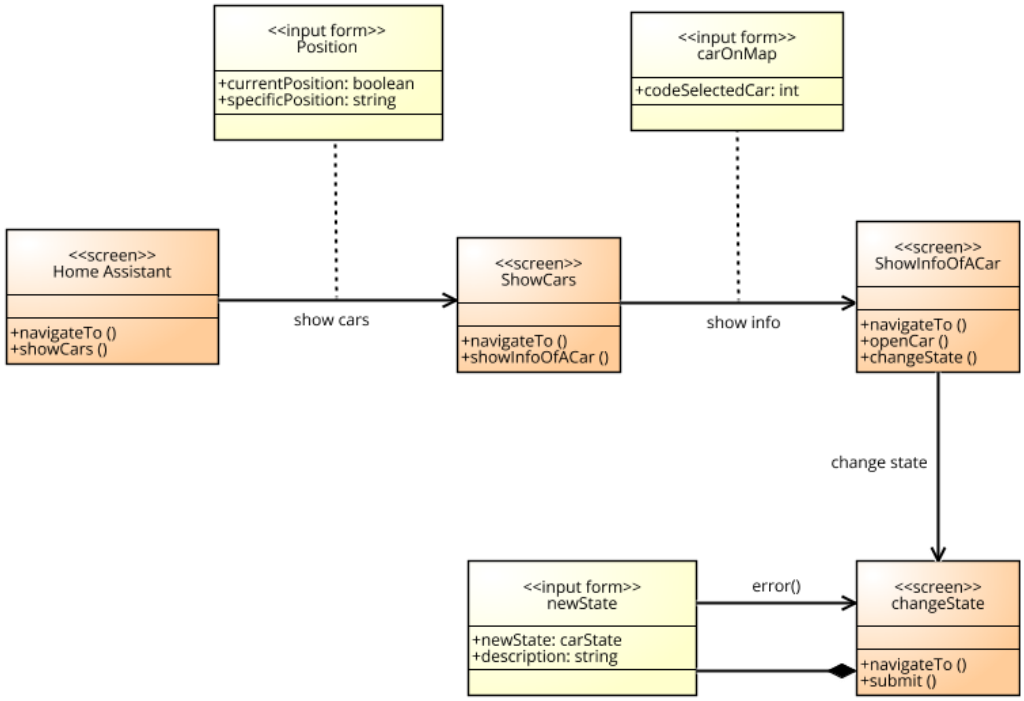


From the home the client can choose to reserve a car or modify his own data. Each step of the *iter* to reserve a car has an input to receive. When the car is opened, the client can search a recharge area near the destination inserted in order to take advantage of the Enable Option.

The leaveDecision screen is activated in a special condition: the car sensors have detected the doors closed and no passengers inside the vehicle. In the “close” screen there are shown the information about the travel, and how much it has cost.

4.2.3 ASSISTANT MOBILE APP UX

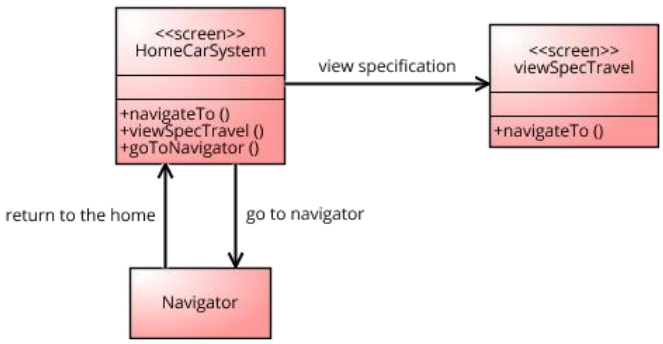
This is the UX for the assistant mobile app.



The assistant can see all the car and their state. Then he can open it with his *passe-partout*. After his intervention, he can change the car status.

4.2.4 CAR APPLICATION UX

This is the UX for the on board car system



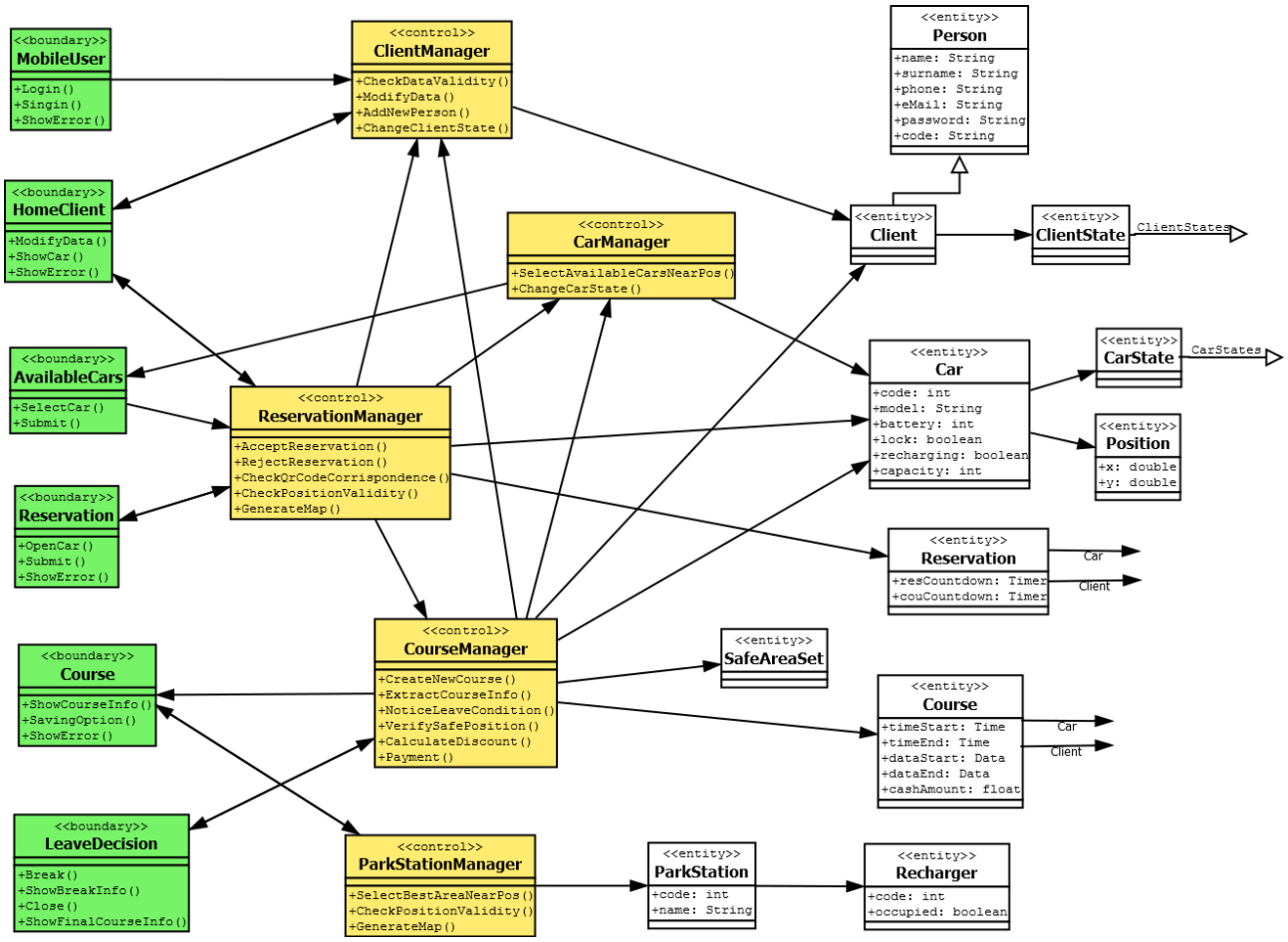
In the viewSpecTravel there are some data of the travel like the current amount of money to pay. From the home it is possible to reach the navigator, that is a system completely autonomous and independent but it is shown in the same display. From the navigator system it is possible to return to the home of the car system.

4.3 BCE

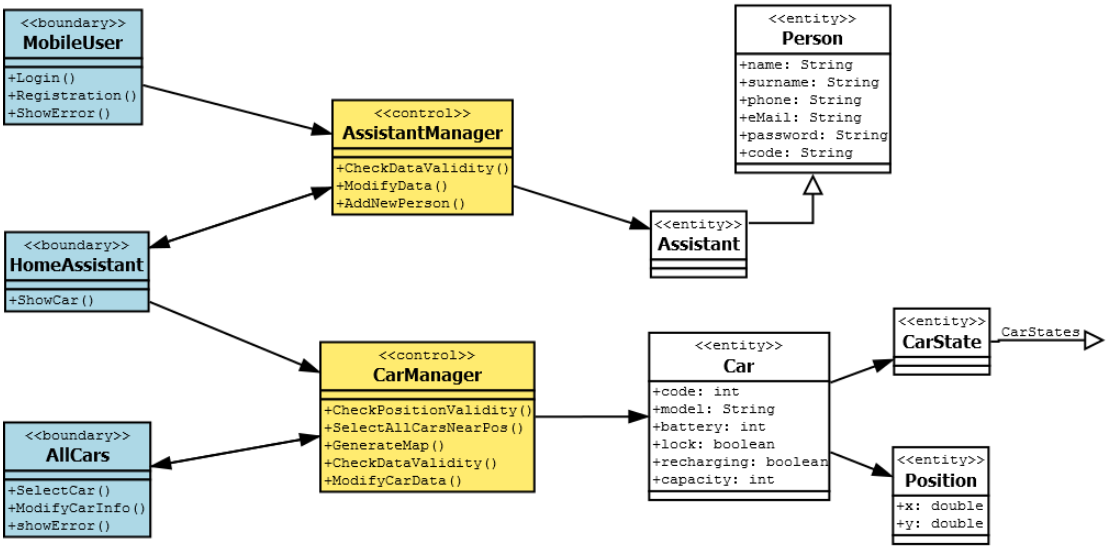
In order to show the management of user actions and its consequences in modifying model, here below are reported two BCE (business controller entity) diagrams, one for client and the other for assistant. Since our application follow particularly MVC pattern it can be very useful. In particular, it is inserted at the end of the document in order to summarize and unifying what has been modelled separately in previous diagrams.

In the entity part only relevant class are reported. For a complete version of our UML diagram please go to section 4.2 of RASD document.

4.3.1 CLIENT BCE



4.3.2 ASSISTANT BCE



5 REQUIREMENTS TRACEABILITY

In this document we specified the design of our project in order to satisfy all the requirements defined in the RASD. In this chapter for each goal we will list all the components necessary for the fulfilment of it.

- [G1] Clients are allowed to register to the system giving their credentials and payment information.
    - ClientController
    - ClientApplication

- [G2] Registered clients are able to see through an interactive map the positions of the available cars near a specific address (current position or inserted).
    - ReservationController
    - CarController
    - ClientApplication

- [G3] Registered clients can reserve a single electric car for at most an hour before picking it up.
    - ReservationController
    - CarController
    - ClientController
    - ClientApplication

- [G4] Clients that get the reservation countdown expired are punished with a fee of one euro.
    - ReservationController
    - CarController

- [G5] Clients can open the reserved car scanning the QR code.
    - CourseController
    - ClientController
    - CarController
    - ClientApplication

- [G6] Clients can monitor the amount of money to be payed, updated in real time on car display.
    - CourseController
    - ClientController
    - CarApplication

- [G7] Client can enable the saving option in order to ensure a uniform distribution of car in the city.
    - CourseController
    - ParkingStationController
    - ClientApplication

- [G8] Clients can leave the car locked in break continuing to pay, keeping it reserved.
    - CourseController
    - CarController
    - ClientController
    - ClientApplication

- [G9] Cars are blocked automatically when client ends his travel.
    - CourseController
    - CarController
    - ClientController
    - ClientApplication

- [G10] Client are charged proportionally to the driving time with some penalties or discounts.
    - CourseController
    - ChargeCalcolator

- [G11] Assistants can know the position and the state of all electric cars.
    - CarController
    - AssistantApplication

- [G12] Specific employee can register at the system with a special account, becoming Assistant.
    - AssistantController
    - AssistantApplication

- [G13] Assistant can change the state of a car.
    - CarController
    - AssistantApplication

- [G14] Registered clients can modify their account, such as their credit card.
    - ClientController
    - ClientApplication

6 EFFORT SPENT

The writing of this document took overall about 18 hours of head-work.

The amount of work has been divided equally between the 3 members of the group.

About 8 hours of 18 were spent working together in order to set up correctly and better coordinate each part of the work.

Furthermore, last 2 hours of head-work have been spent for reread, revise and assemble the whole document.

7 REFERENCES

USED TOOLS

The tools we used to create this document are:

- Microsoft Word 2016 : to write and assemble the document
- Dropbox e GitHub : to share work
- Signavio Academic and DIA: for all UML model
- Astah : for Sequence diagrams
- Draw.io : for Component diagrams

8 CHANGELOG

- Version 2
    - Fix in sequence diagram “a client uses the car”
    - Fix in component diagram (controllers don’t access to all the model but only to some parts of it)
    - Fix in the introduction of BCE diagram