Design Document

December 11, 2016

POLITECNICO
DI MILANO

Version 1.0

- Davide Alessandro Cottica (mat. 806868)
- Stefano Antonino Badalucco (mat. 809322)

Contents

1 Introduction

1.1 Purpose

The purpose of this design document is to provide a high-level description of the PowerEnojoy system, providing insight into the structure and the design of each component.
This document is adressed to the developers and aims to identify:

- The high-level architecture
- The design patterns
- The main components and their interfaces provided one for another
- The Runtime behavior

1.2 Scope

The project PowerEnjoy will provide a service based on mobile application and web application, and the only target of this service is the clients.
All the clients of the service must be registered into the system.
The system allows the client to reserve a car, not already reserved, among the ones available. The client could make a reservation via the mobile application or web application selecting the car from a map that shows all the car of the company.
After the client makes a reservation, he has a limited time to reach the selected car. If the reservation expires a penalty is subtracted from the user bill.
When the client turns on the engine of the car, the ride starts. At the end of the ride the system subtract the cost of the ride from the user's bill.
This system could be very useful for all the people that would like to reach places not covered by public transportation. This service is attainable everyday and at any time of the day. This project has also as purpose the decrease of the pollution in the city using electric car instead petrol-based car.
The system will track in each moment car's position with a GPS system located inside the car.
So the main purpose of the system is offering a car rent service simple to use that takes care about the environment.

1.3 Definition, acronyms, abbreviations

- RASD: Requirements analysis and specifications documents
- DD: design document
- UX: user experience design
- UX: user experience design
- API: application programming interface; it is a common way to communicate with another system
- MVC: model view control
- ServiceNow: offers everything-as-a-service cloud computing that offers everything-as-a-service cloud computing
- SOA: Service-oriented Architecture
- MEP: message exchange pattern
- SOAP: simple Object Access Protocol
- XML: Extensible Markup Language
- Cyphertext: encode information that is unreadable without the proper cipher to decrypt it

1.4 Reference documents

- RASD produced before
- Paper by Jackson and Zave on the word and the machine.pdf
- Paper on the green move project.pdf
- Second paper on the green move project.pdf

1.5 Documents structure

- Introduction: this section provides general information about the DD document and the system to be developed
- Architecture design: this section shows the main components of the system giving a global view of the application and how that interacts with the system
- Algorithms Design: this section is composed by some algorithms that describes the most critical parts of the system functionalities, independently from their concrete implementation
- User Interface Design: this section describes how the user interface will look like and behave showing mockups and UX and BCE diagrams
- Requirements Traceability: this section explains how the requirements in the RASD document are satisfied by the design choices of the DD

2 Architectural design

2.1 Overview

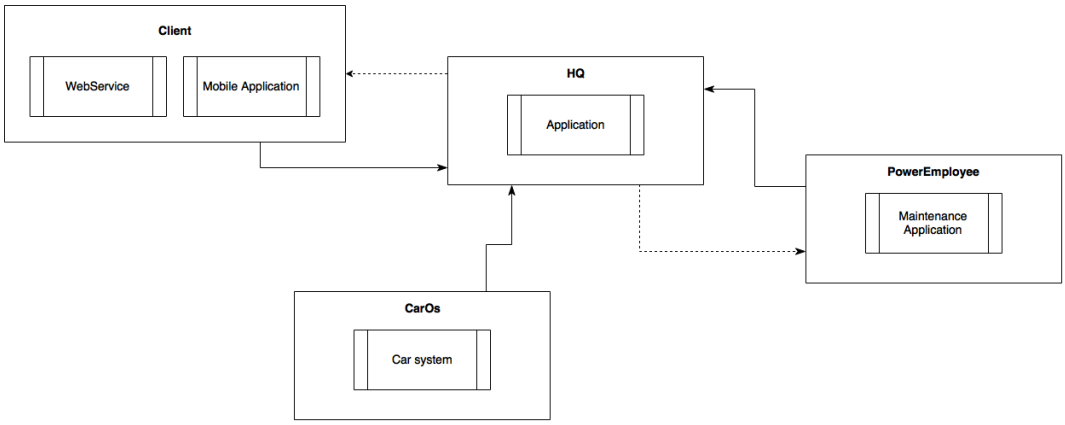
PowerEnjoy will be based on a tier-3 architecture as explained in the RASD documentation.

A system built in this way can be moved to cloud based platforms. As an advantage we haven't anymore the duty to maintain servers or expand the existing server farm.
One option we took into consideration is deploy the system in the ServiceNow cloud platform, in order to easily handle complaints and maintenance works logically subdividing these in two categories:

- Incidents, opened by both employees and clients (e.g. accidents and reports)
- Problems, opened by employees (e.g. technical issues related to incidents)

System relies on the SingleSignOn (SSO) authentication method that grants users migrating from different platforms in the application to access all the other parts of the application without the need to login other times.

2.2 High level components and their interaction



We have identified four components acting in the system:

1. **HQ**
   Incidents, opened by both employees and clients (e.g. accidents and reports)

2. **Client**
   It is the main actor of the system.
   The client communicate with our platform via mobile application or web service in a synchronous way when:

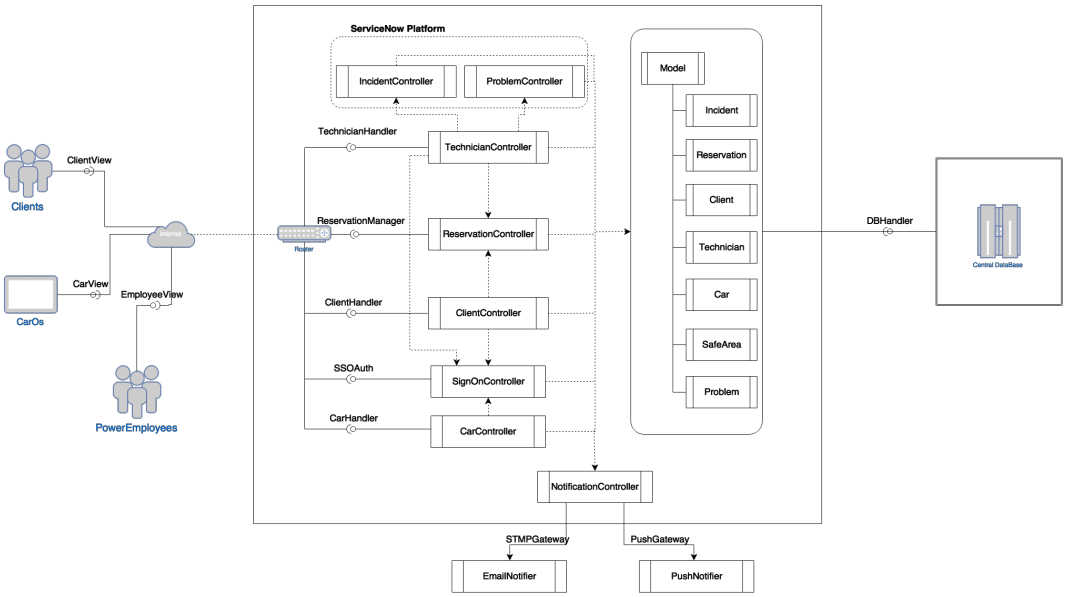
   - A reservation has been confirmed to the system and the client must wait the acknowledgment of the request
   - A complaint or an issue has been submitted to the system and the client must wait for the acknowledgment of it
   - The client queries our DB for any kind of information

   The client receives asynchronous notifications from the HQ (e.g. a reminder that user's reservation is going to expire in 15 minutes).

3. **CarOs**
   In each car will be installed a part of the system able to communicate with the HQ, mostly in an asynchronous way; for example a user pressing on the 'Temporary Stop' in the car's computer display sends a single message to HQ that do not need any response.
   Instead, if a client wants to inform our employees that an accident has occurred, that could be done as well from car's computer display, but in this case a synchronous message is sent to HQ; the acknowledgment response is sent to the user whose active reservation is linked to the car.

4. **PowerEmployee**
   He/she is an employee (technicians, mechanics and software maintainers) communicating with HQ mostly in a synchronous way, as they take charge of problems and incidents and must wait for system acknowledgment.

2.3 Component View



IncidentController: manage incidents.

ProblemController: manage problems.

TechnicianController: manage technicians.

ReservationController: manage reservations.

ClientController: manage clients.

SignOnController: manage the SingleSignOn Authentication method.

CarController: manage cars.

STMPGateway: communication channel able to send and receive STMP messages.

PushGateway: communication channel able to send push notifications to app.

Model: logic organization of data in the database.

CentralDatabase: physic place where data is stored.

Clients: main target of our system.

PowerEmployees: technicians able to resolve incidents and problems.

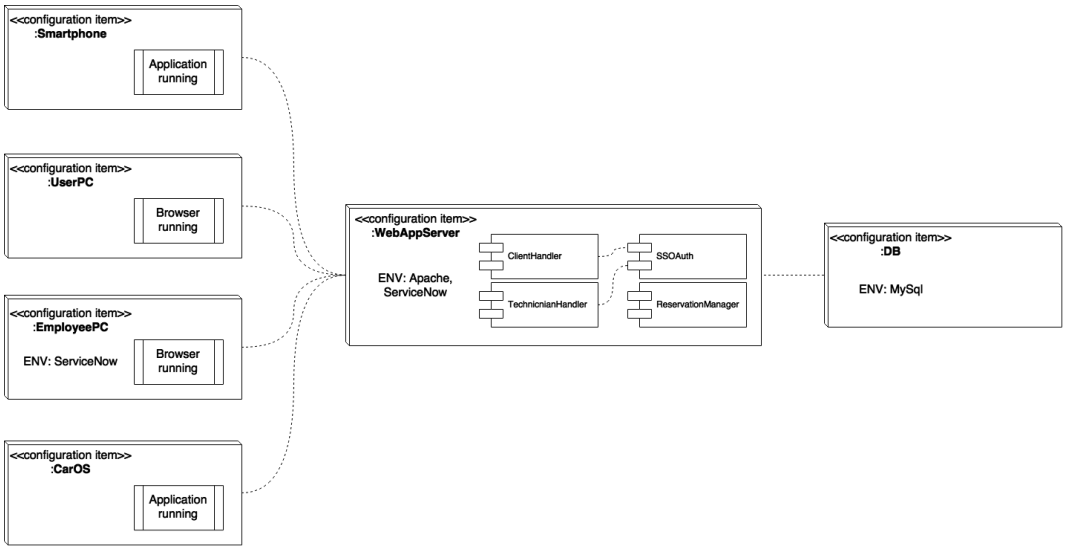
CarOS: part of a car able to communicate with the system.

Router: device able to route information outside enterprise intranet.

ServiceNowPlatform:cloud service platform.

As already explained before, a system relying on a SSO authentication method grants users to access different parts of the system without logging-in other times to all the parts requiring additional authentication.
This way, the router first route the message to the SignOnController and save a session for the selected user/car/technician for further uses in the system. The PushGateway sends notifications to mobile applications, while with the SMTPGateway emails are sent to both user and technicians email accounts.

2.4 Deploying View



2.5 Runtime View

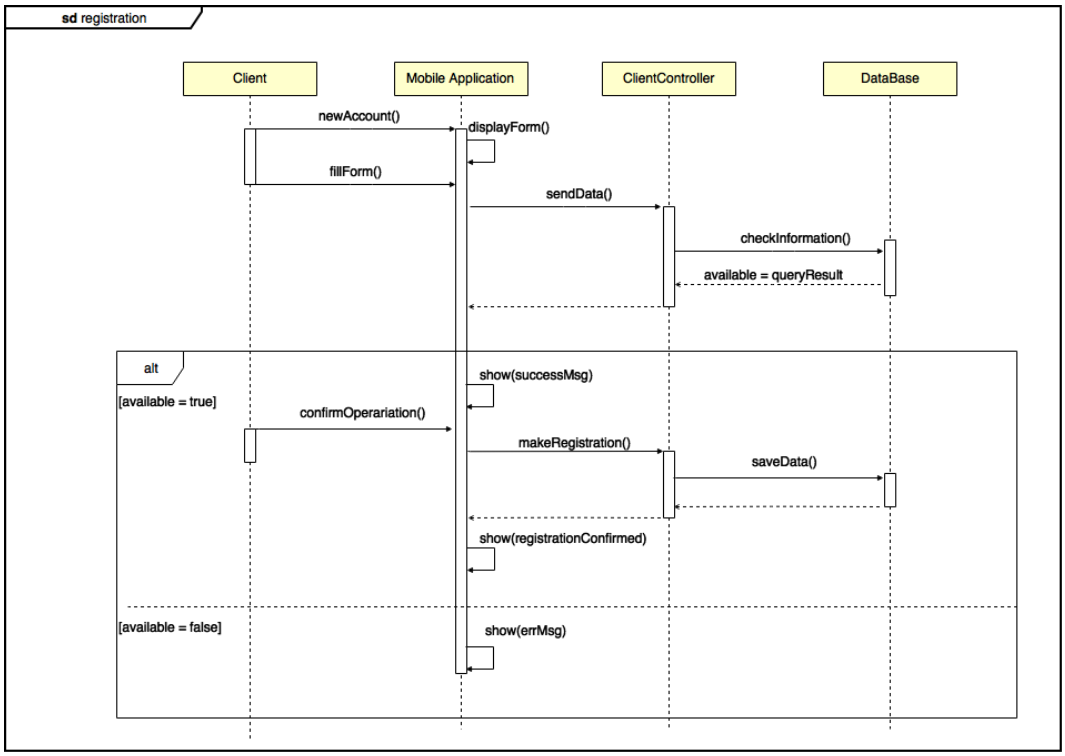


Figure 1: Sequence diagram of a client registration.

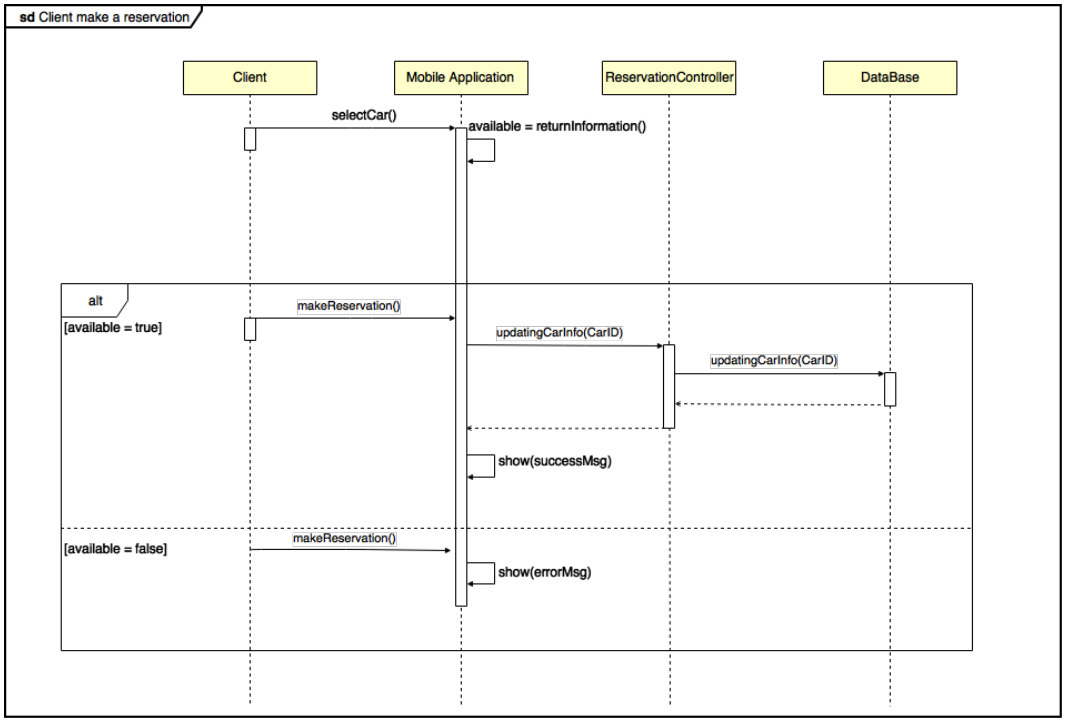


Figure 2: Sequence diagram of a client that makes a reservation.

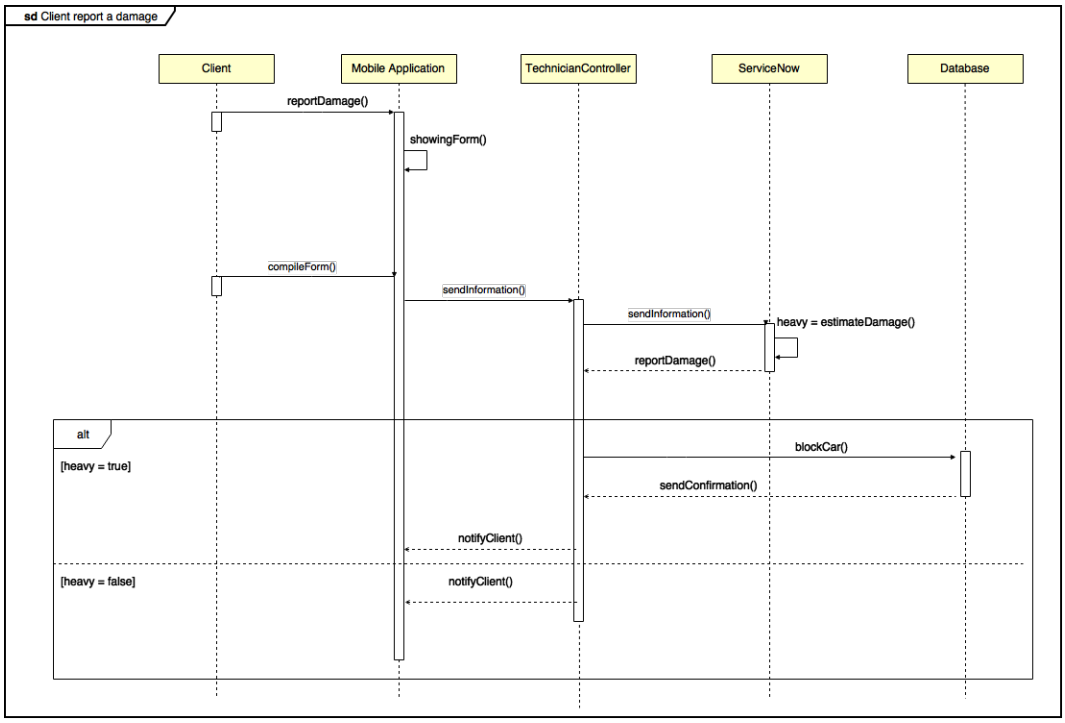


Figure 3: Sequence diagram of a client that reports a damage.

2.6 Selected Architectural Styles and Patterns

We decided to use a **Service-oriented Architecture** (SOA), architectural approach that aims at developing modular applications consisting of independent services, which fulfill a specific task and communicate with each other in concordance.
The main advantages of this architectural approach are:

- Service Reusability: the services of the system can be reused in multiple applications independent of their interaction with other service
- Easy Maintainability: each service is an independent entity, thus it can be easily updated or maintained
- Reliability: services are easier to test and debug as compared to massive pieces of code
- Improved Scalability and Availability: multiple instances of a single service can run on different servers at the same time to increase scalability and availability of the service
- Improved Software Quality: services can be reused so there is no scope for redundant functionality. This helps could reduce errors to inconsistent data

The application is also strongly based on a **Client-Server** communication model. The server component provides a function or service to one or many client, which initiate requests for such services. Clients and servers exchange messages in a **request-response messaging pattern**: The client sends a request, and the server returns a response. The main reasons that led us to this architectural choice are:

- Centralization: there is a centralized control
- Security: servers have better control access and resources to ensure that only authorized clients can access or manipulate data and server-updates are administered effectively
- Easy maintenance: since client/server architecture is a distributed model representing dispersed responsibilities among independent computers integrated across a network, it's an advantage in terms of maintenance. It's easy to replace, repair, upgrade and relocate a server while clients remain unaffected. This unawareness of change is called as encapsulation

MVC has been widely used in our application. Thus we decided to divide our application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. In this way due to the separation of the model from

the view, the user interface can display multiple views of the same data at the same time.
Another benefit of MVC pattern is that user interfaces tend to change more frequently than business rules because the model does not depend on the views.
For the communication channel we decided to use the message exchange pattern

(MEP) **SOAP** (Simple Object Access Protocol) that allows processes running on disparate operating system to communicate using Extensible Markup Language (XML). Advantages of SOAP are:

- Language neutrality: SOAP can be developed using any language
- Interoperability and Platform Independence: SOAP can be implemented in any language and can be executed in any platform
- Simplicity: SOAP messages are very simple XML format
- Scalability: SOAP uses HTTP protocol for transport due to which it becomes scalable

2.7 Other design decision

The system uses an external service, Google Maps API, to offload all the geolocalization and map visualization processes. The reason of this choice are:

- Google maps could be used on the server side and on the client side (map visualization)
- Google maps API let to create custom map for any location, adding locations marks, places of interest
- Google maps is well know from the most of the people so the uses feel comfortable using it

We decided to transform data stored in database into cyphertex. The used method is the Symmetric database encryption that involves a private key being applied to data that is stored and called from database. This private key alters the data in a way that causes it to be unreadable without first being decrypted. The main benefits of this choice are:

- Complete data protection
- Secure transmission: encrypting data provides security benefits during transmission
- Data Integrity: if encrypted data is used, the recipient will definitely notice that it has been tampered with if a spiteful person tries to manipulate specific data

3 Algorithm Design

In this section we want to show a preview how we want to implement the system of this application. interaction between them.

```
public class ReservationController {


    public void startRide(Car car, Ride ride, Client client){
        ride.setStartTime(new Date());
        ride.setCar(car);
        ride.setClient(client);
    }
    public double endRide( Ride ride){
        ride.setEndTime(new Date());
        double cost;
        cost = (ride.getStartTime() - ride.getEndTime())* ride.getMinuteCost();
        Car car = ride.getCar();
        float latitude = car.getLatitude();
        float longitude = car.getLongitude();

        Position position = new Position(latitude, longitude);
        double discount = 0;
        double penality = 0;


        if(position.isInSafeArea()){
            discount += cost * (0.3);
        }
        if(car.energyLevel() > 50){
            discount += cost * (0.2);
        }
        if(car.hasMorePassenger()){
            discount += cost * (0.1);
        }
        if(car.isDistant() || (car.energyLevel() < 20)){
            penality = cost * (0.3);
        }
        cost = cost - discount + penality;

        return cost;
    }
}
```

4 User interfaces design

4.1 Mockups

Mockups are already present on the RASD document.

4.2 UX diagrams

The UX diagram shows the different user's main actions performed and the interaction between them.

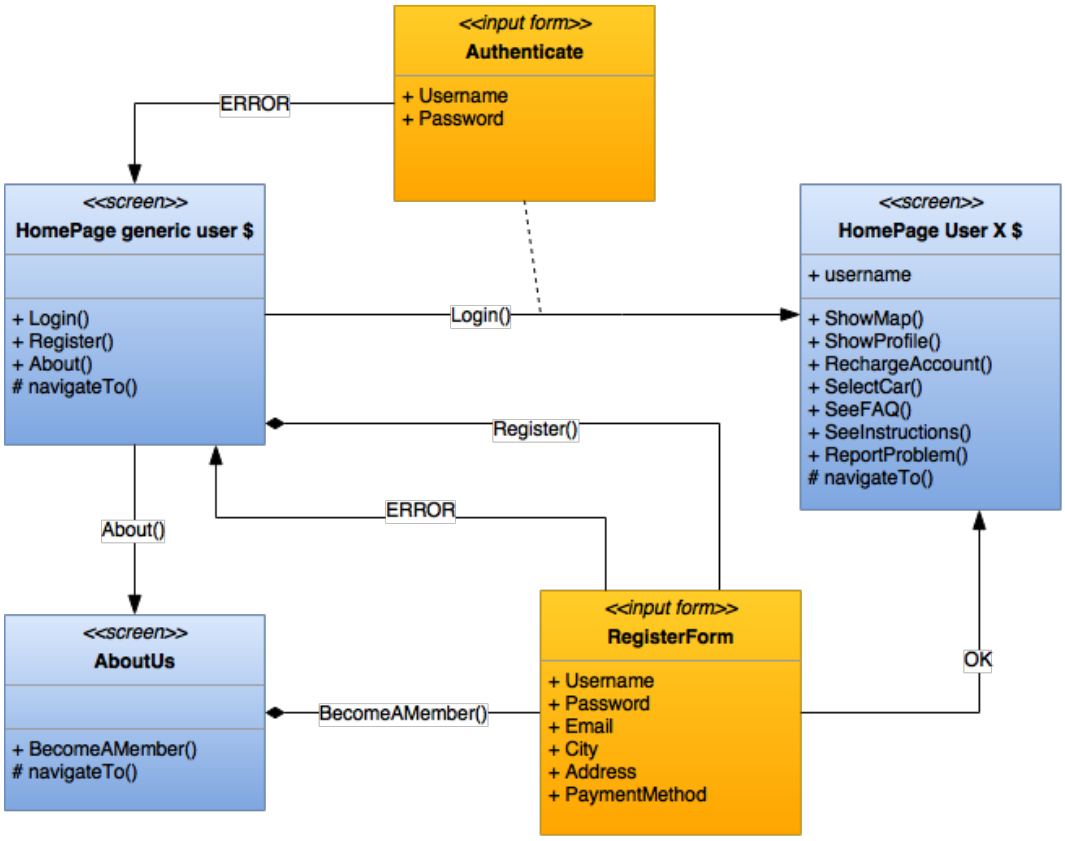


Figure 4: UX user registration.

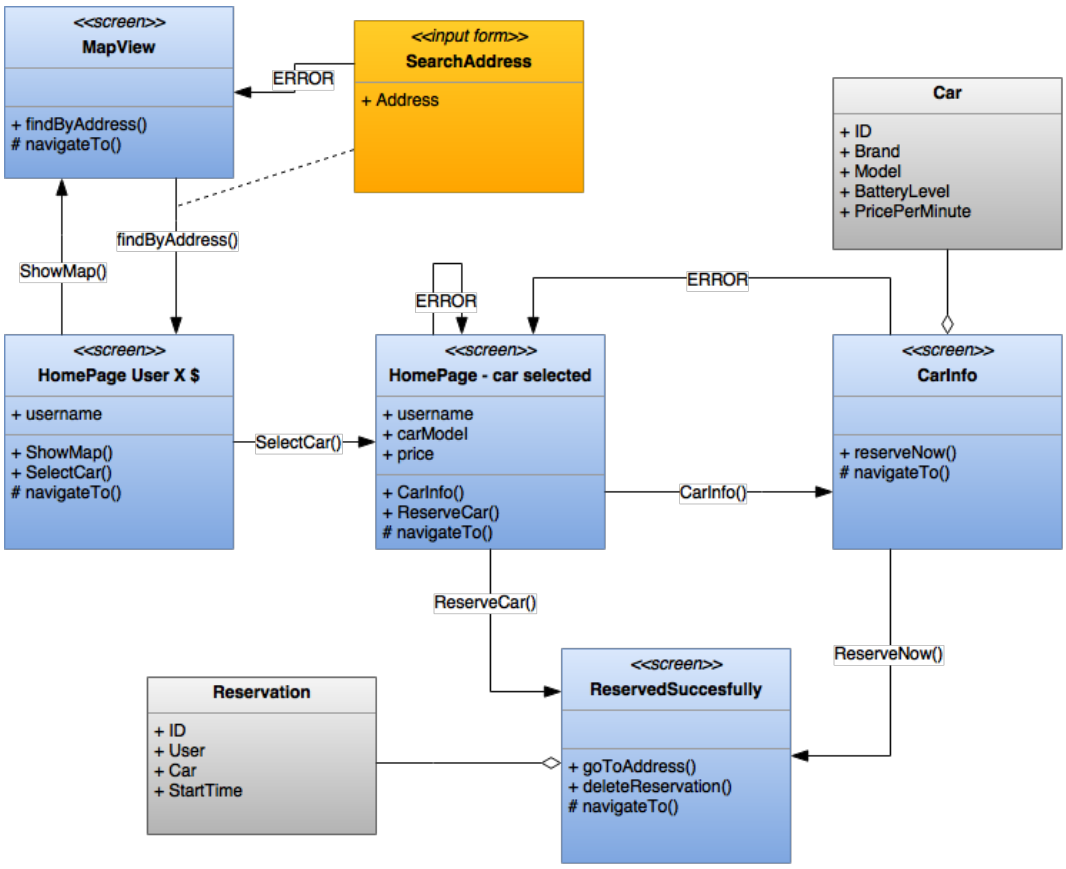


Figure 5: UX user reservation.

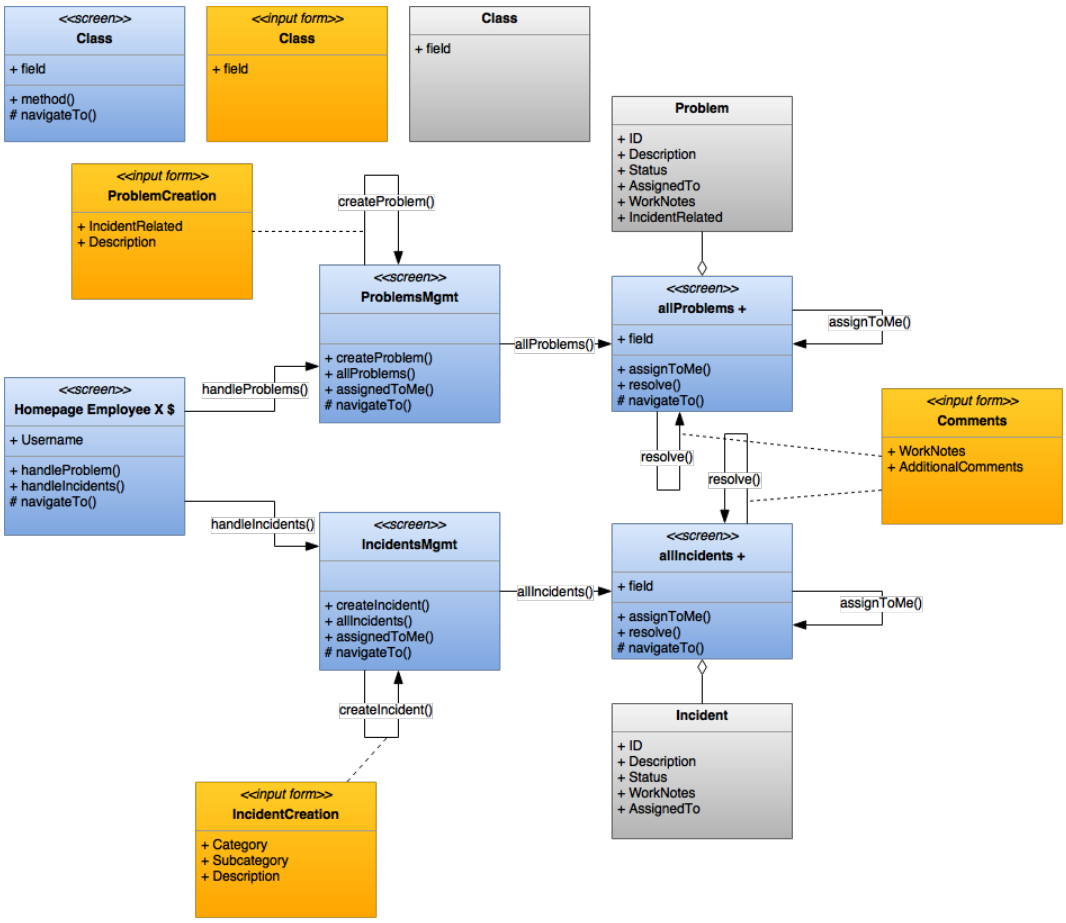


Figure 6: UX technicians.

4.3 BCE diagrams

BCE diagrams show how each user action is managed internally.

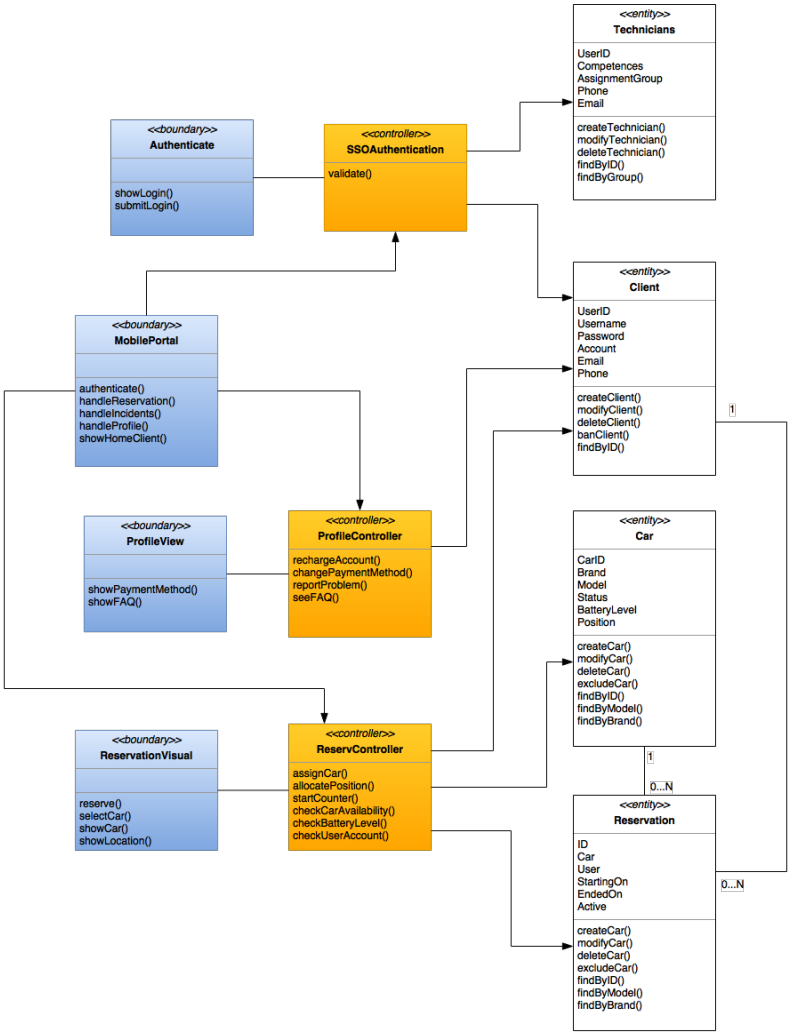


Figure 7: BCE diagram.

5 Requirements traceability

Here we see how our components relate themselves to the system in order to accomplish the goals stated in the RASD document:

- [G1] Users will be able to create a profile in the system.
  - ClientController
- [G2] Users will be able to log into the system with the credentials provided.
  - SignOnController
  - ClientController
- [G3] Users can reserve a car from the website or the mobile application.
  - ReservationController
  - ClientController
- [G4] System will provide an updated map showing all the available cars in the area.
  - ReservationController
  - CarController
- [G5] System could show user the available cars from a given address.
  - ReservationController
  - CarController
- [G6] System will show user all the safe area and the power grid stations.
  - ReservationController
- [G7] When the user with the specific reservation is near the car, system will provide him a way to unlock this car.
  - NotificationController
  - CarController
  - UserController
- [G8] User will be able to monitor the amount of money he is going to spend by a screen located in the car.
  - UserController
- [G9] System will detect if there is another passenger by weight sensor located under the seats.
  - NotificationController

- CarController

- [G10] System will be able to find cars tracking GPS signal every time.
  - CarController
- [G11] System will be able to accept or refuse a reservation by looking at car reservation status.
  - ReservationController
  - CarController
- [G12] Cars will be provided with GPS navigators.
  - CarController

6 References

The tools we used to create this document are:

- Draw.io: for diagrams
- sharelatex.com: to create the Latex version of this document
- Google Drive: to share the file
- Github: for version controller and distributed work
- Eclipse: to create the code for the Algorithm design

7 Hours of work

The amount of working hours:

- Davide Cottica: 19 hours
- Stefano Badalucco: 19 hours