Software Engineering 2: PowerEnJoy

Design Document (DD)

Version 1.0

Politecnico di Milano, A.A. 2016/2017

Agosti Isabella, 874835
Cattivelli Carolina, 879359

December 11, 2016

Contents

1 INTRODUCTION **4**
- 1.1 Purpose . . . 4
- 1.2 Scope . . . 4
- 1.3 Definitions, Acronyms, Abbreviations . . . 5
  - 1.3.1 Definitions . . . 5
  - 1.3.2 Acronyms and Abbreviations . . . 6
- 1.4 Reference Documents . . . 6
- 1.5 Document Structure . . . 7

2 ARCHITECTURAL DESIGN **8**
- 2.1 Overview . . . 8
- 2.2 Component view . . . 11
  - 2.2.1 Client component . . . 11
  - 2.2.2 Employee manager component . . . 12
  - 2.2.3 Ride manager component . . . 13
  - 2.2.4 User manager component . . . 14
  - 2.2.5 Registered user manager component . . . 15
  - 2.2.6 Database component . . . 16
- 2.3 Deployment view . . . 17
- 2.4 Runtime view . . . 18
  - 2.4.1 User registration . . . 18
  - 2.4.2 Registered user and employee login . . . 19
  - 2.4.3 Registered user views his/her profile . . . 20
  - 2.4.4 Registered user manages his/her personal information . 20
  - 2.4.5 Registered user makes a reservation . . . 21
  - 2.4.6 Registered user reports an issue . . . 22
  - 2.4.7 Registered user cancels his/her current reservation . . . 23
  - 2.4.8 Employee manages a car's information . . . 24
- 2.5 Component interfaces . . . 25
  - 2.5.1 Client interfaces . . . 25
  - 2.5.2 Employee manager interfaces . . . 25
  - 2.5.3 User manager interfaces . . . 25
  - 2.5.4 Registered user manager interfaces . . . 26
  - 2.5.5 Ride manager interfaces . . . 26
- 2.6 Selected architectural styles and patterns . . . 27
  - 2.6.1 Architectural styles . . . 27
  - 2.6.2 Design patterns . . . 28
- 2.7 Other design decisions . . . 28

3 ALGORITHM DESIGN **29**
3.1 Constants 29
3.2 User 30
3.3 Registered User 31
3.4 Employee 33
3.5 Car 34
3.6 Screen 37
3.7 Reservation 38
3.8 Address 39
3.9 Map 40
3.10 Safe Area 41

4 USER INTERFACE DESIGN **42**
4.1 Mockups 42
4.2 UX diagram 43
4.3 BCE diagram 44

5 REQUIREMENTS TRACEABILITY **45**

6 EFFORT SPENT **48**
6.1 Agosti Isabella 48
6.2 Cattivelli Carolina 48

7 REFERENCES **49**
7.1 Used tools 49

1 INTRODUCTION

1.1 Purpose

The purpose of this document is to illustrate in details the concepts expressed in the RASD document, clarifying the architectural choices that have been made and the functionalities that will be developed.

1.2 Scope

The system allows registered users to reserve a car via mobile or web application, by letting the system track their position or by inserting a specific address.

A user is considered registered only after filling out a form, receiving a password by the system and logging in for the first time.

The system's main purpose is to protect the environment. For this reason it is based on a car-sharing service using only electric cars.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Keyword	Definitions
User	A person that interacts with the PowerEnJoy mobile or web application to register to the system.
Registered user	A person who already registered to the system, that interacts with the PowerEnJoy mobile or web application in various ways.
Employee	A member of the PowerEnJoy staff.
Car-sharing service	Model of car rental where people rent cars for short periods of time, often by the hour.
Electric car	Automobile that is propelled by one or more electric motors, using electrical energy stored in rechargeable batteries.
Registration	The act or process of filling out an online form providing credentials and payment information.
Log-in	Process by which a user gains access to the system by identifying and authenticating himself/herself.
Reservation	Arrangement through which a registered user holds a car for his use at a later time.
Safe area	Area whose position is predefined by the management system. Safe areas are the only ones in which a user is allowed to park a car.
Special safe area	Special type of safe area where a car can be recharged.
Discount percentage	Discount applied on the users last ride only in certain circumstances.
Low battery	The cars battery level is considered low when less than 20%.

1.3.2 Acronyms and Abbreviations

Acronym/abbreviation	Definitions
DB	Database
UI	User Interface
GUI	Graphical User Interface
RASD	Requirements Analysis and Specification Document
DD	Design Document
UX	User eXperience
BCE	Boundary Control Entity
UML	Unified Modeling Language
AA	Anno Accademico (Academic Year)
SOA	Service Oriented Architecture
MVC	Model View Controller

1.4 Reference Documents

- Specifications document: Assignments AA 2016-2017.pdf
- Sample Design Deliverable Discussed on Nov. 2.pdf
- RASD.pdf

1.5 Document Structure

The document is organized as follows:

- **Introduction**, gives an overview of the design document's contents.
- **Architectural design**:
  - *Overview*, describes the high level components and their interaction.
  - *Component view*, gives a more detailed description of the application's components.
  - *Deployment view*, presents the components that must be deployed in order for the application to run correctly.
  - *Runtime view*, describes through sequence diagrams the way components interact to accomplish specific tasks related to our use cases.
  - *Component interfaces*, presents the interfaces of the application's components.
  - *Selected architectural styles and patterns*, presents the styles and patterns we used, why and how.
  - *Other design decisions*.
- **Algorithm design**, focuses on the definition of the most relevant algorithmic parts.
- **User interface design**, provides an overview on how the user interface(s) of our system will look like.
- **Requirements traceability**, explains how the requirements defined in the RASD map to the design elements defined in this document.
- **Effort spent**, includes information about the number of hours each group member has worked towards the fulfillment of this deadline.
- **References**, describes the tool used to redact this document and its components.

2 ARCHITECTURAL DESIGN

2.1 Overview

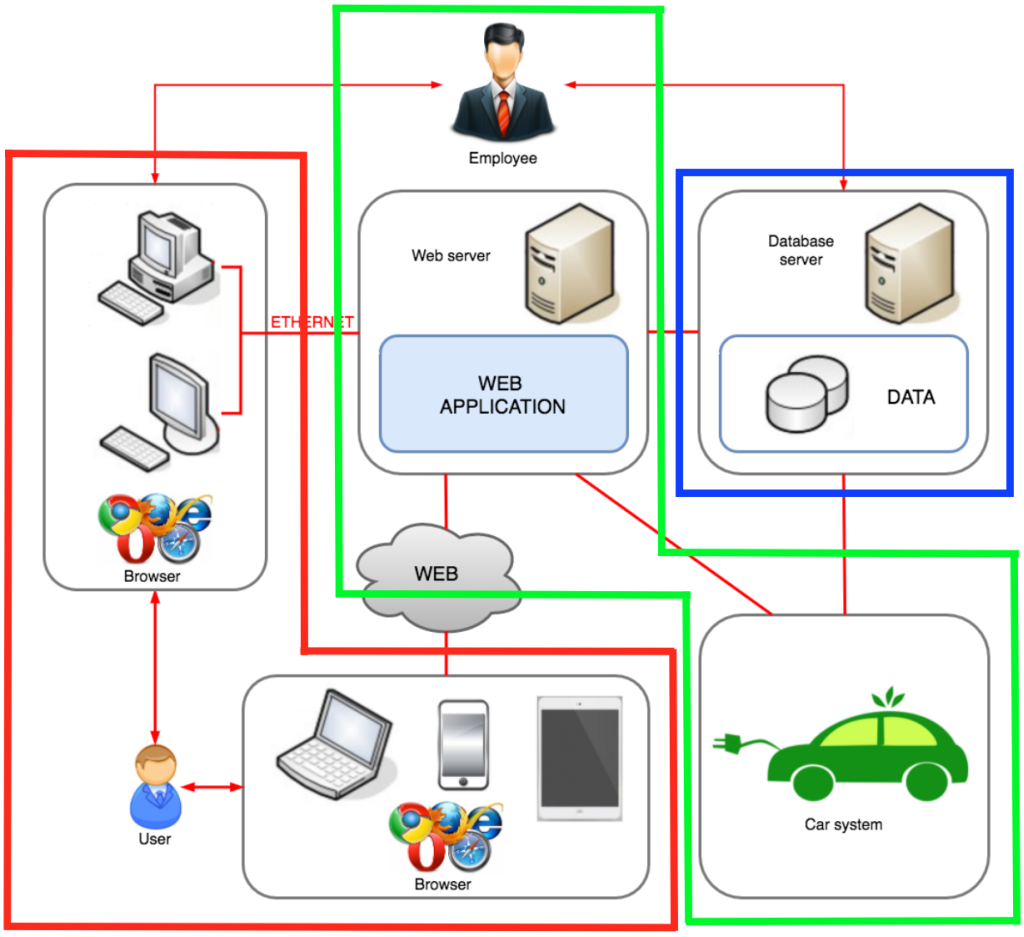


Figure 1: General architecture

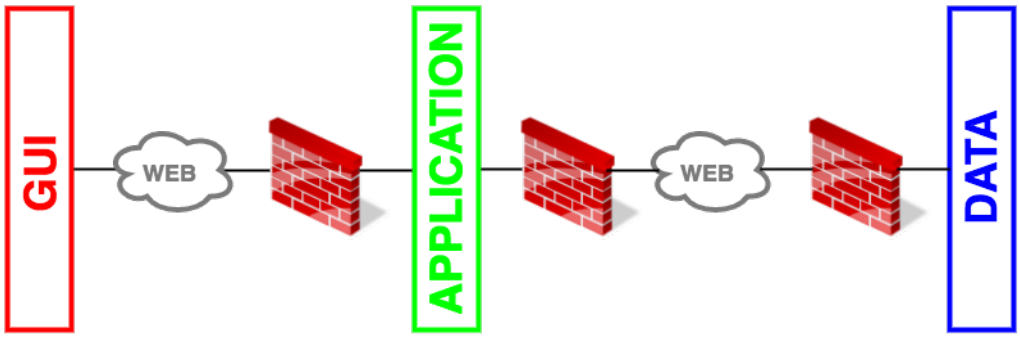


Figure 2: Three-tiered architecture

The images represent the main components of our system and show how each component interacts with the others.

The system is based on a three-tiered architecture:

- **Client tier**, which contains web pages (when the users interact with the system through the web application) or an application client (when the users interact with the system through the mobile application).
- **Server tier**, which is divided into two tiers:
  - Web tier, which is in charge of generating the web pages and sending them to the client.
  - Business tier, which contains the application's logic.
- **Database tier**, which contains the database server.

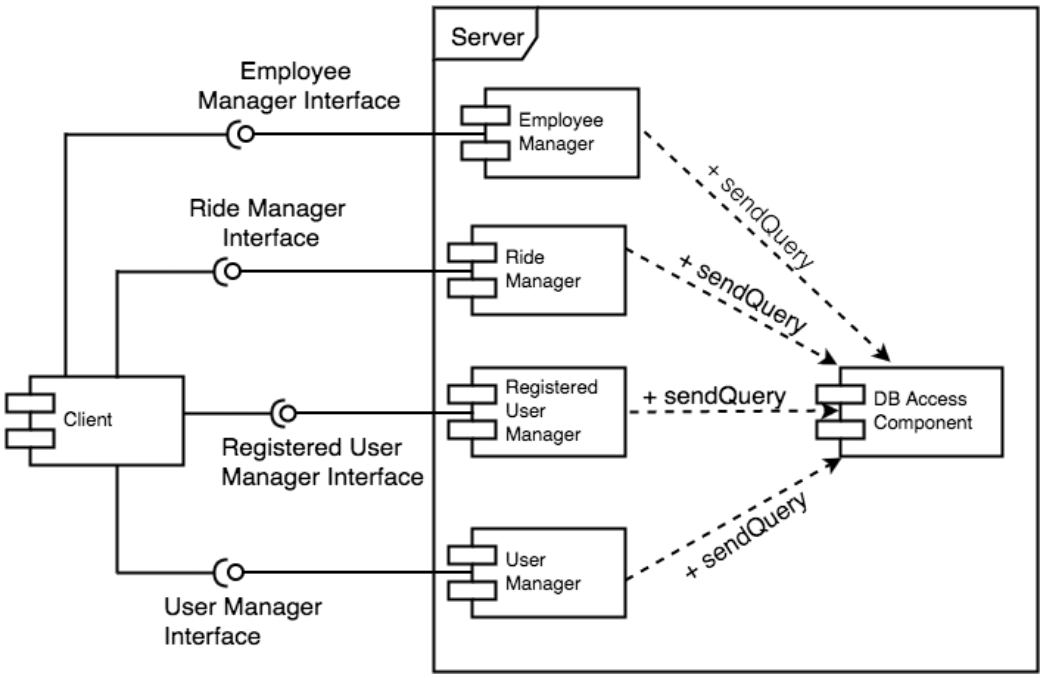


Figure 3: High-level architecture

The high-level architecture is composed of three different elements.

The client initiates the communication with the server by using the mobile application or the application's website. After making a request, the client has to wait for an answer from the server (synchronous communication). For example, once the client has filled out and submitted the registration form, he has to wait for the server to send him the password via email.

The server is composed of:

- An employee manager, that will manage the system's information either with a direct access to the database or through the GUI (in this case he will only be able to change the PowerEnJoy cars information).
- A ride manager, that will take care of the car management side of our application and will provide an interface to the client.
- A user manager, that will provide functions to manage users information and will provide an interface to the client.
- A registered user manager, that will provide functions to manage registered users information and will provide an interface to the client.

Finally the database, which contains the information about users, cars and reservations, will be accessible by the other three server components.

2.2 Component view

Here the components presented in the previous section are described in details.

2.2.1 Client component

The client has three subcomponents:

1. **Client manager**, which possesses all the functionalities related to the generic user, the registered user and the employee.
2. **User interface**, to communicate with the users (registered, not registered and employees).
3. **Connection manager**, to communicate with the server.

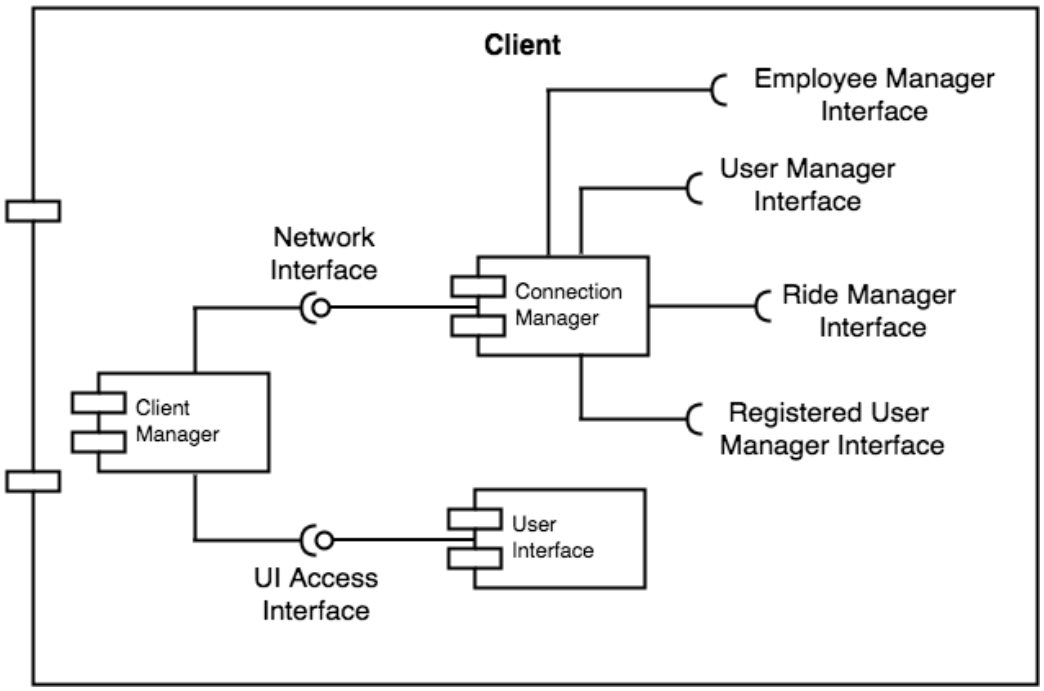


Figure 4: Client component

2.2.2 Employee manager component

The employee manager has four subcomponents:

1. **Connection manager**.
2. **Car manager**, that in this case allows the system to change the selected car's information.
3. **Account manager**, that provides the login functionalities.
4. **Database access component**, used to store persistent data.

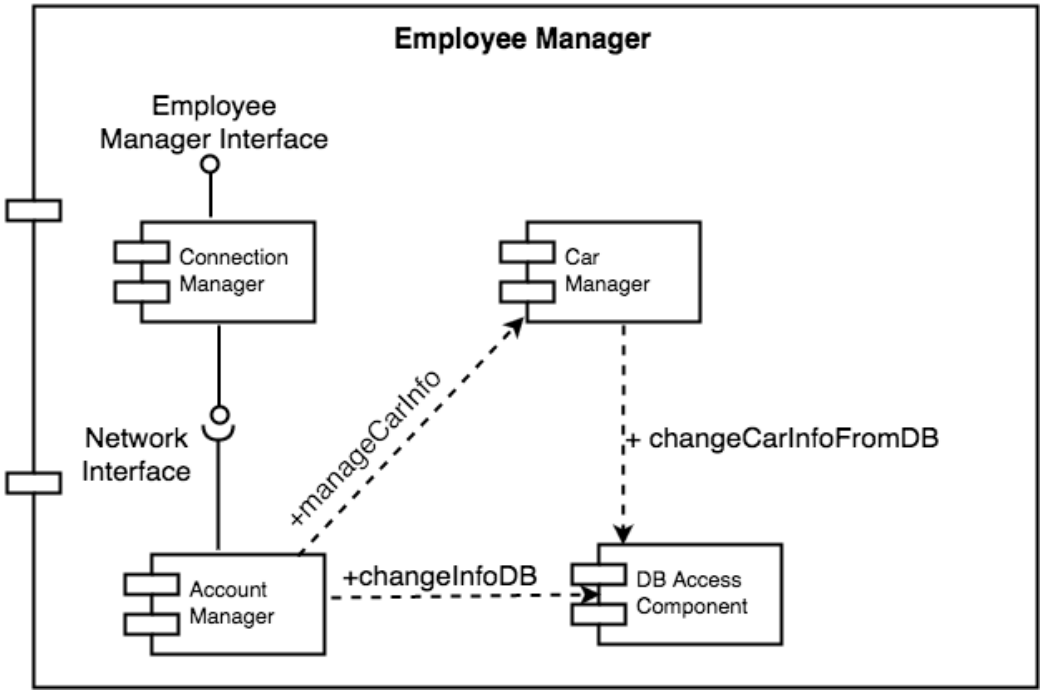


Figure 5: Employee manager component

2.2.3 Ride manager component

The ride manager has seven subcomponents:

1. **Connection manager**.
2. **Car manager**, that in this case allows the system to find all the available cars.
3. **Reservation manager**, that takes requests from the registered users.
4. **Reservation generator**, that creates the reservation after the registered user chooses the car.
5. **Database access component**.
6. **Notification manager**, that manages the notifications the server needs to send to the client.
7. **Email gateway**, manages the sending of emails.

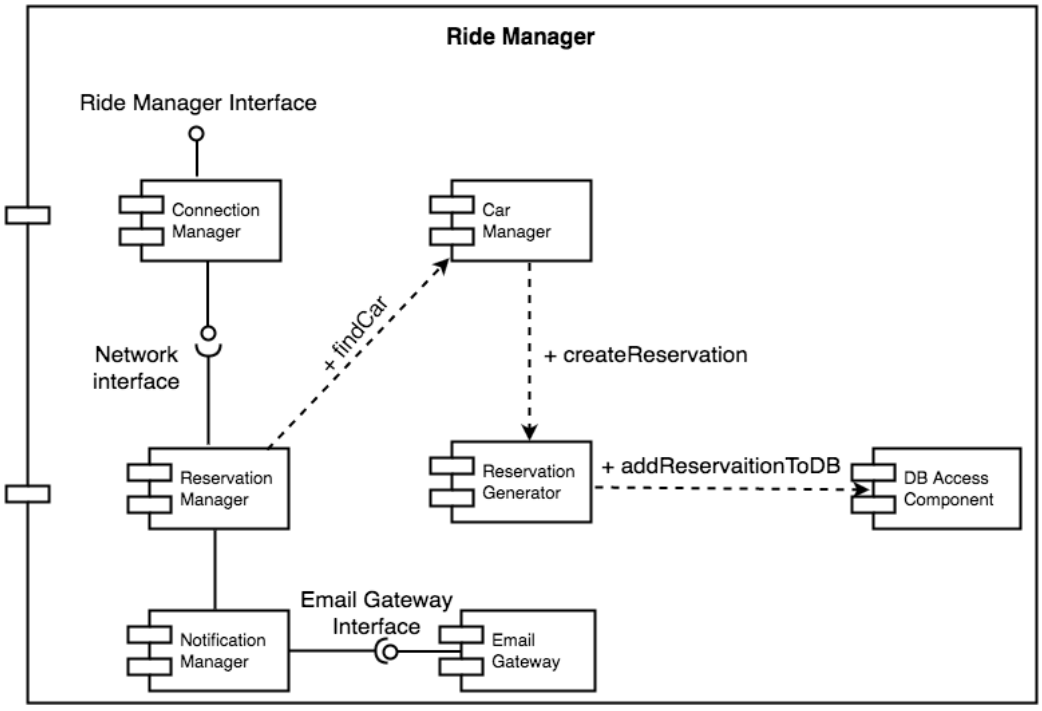


Figure 6: Ride manager component

2.2.4 User manager component

The user manager has five subcomponents:

1. **Connection manager**.
2. **Notification manager**.
3. **Email gateway**.
4. **Database access component**.
5. **Registration manager**, that adds the user's information to the database.

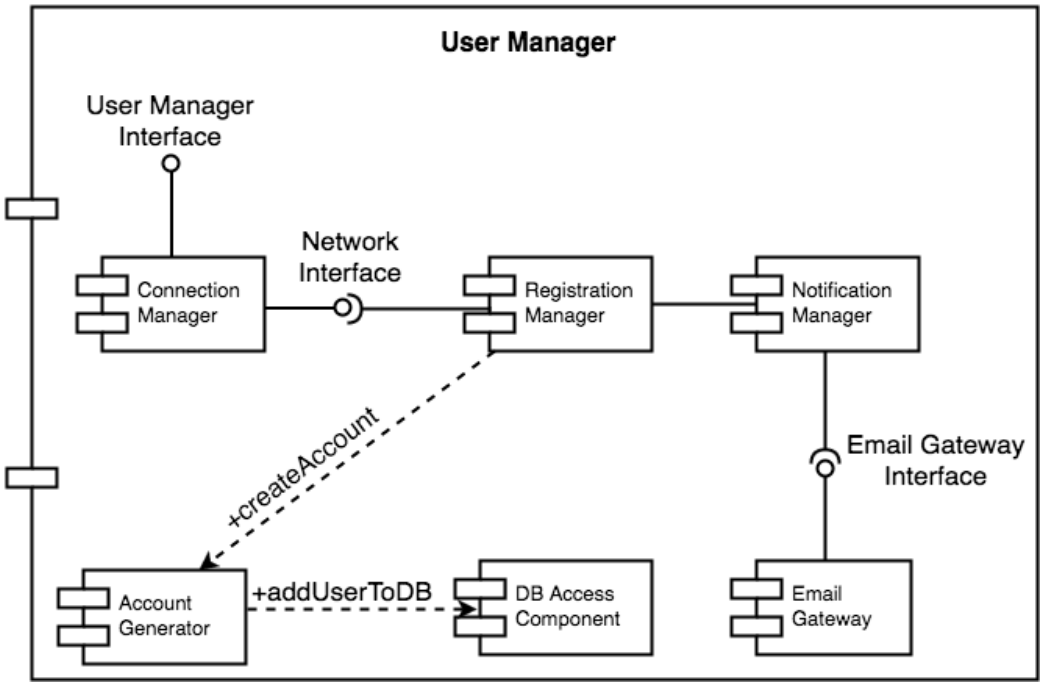


Figure 7: User manager component

2.2.5 Registered user manager component

The registered user manager has four subcomponents:

1. **Connection manager**.
2. **Reservation manager**.
3. **Database access component**.
4. **Account manager**.

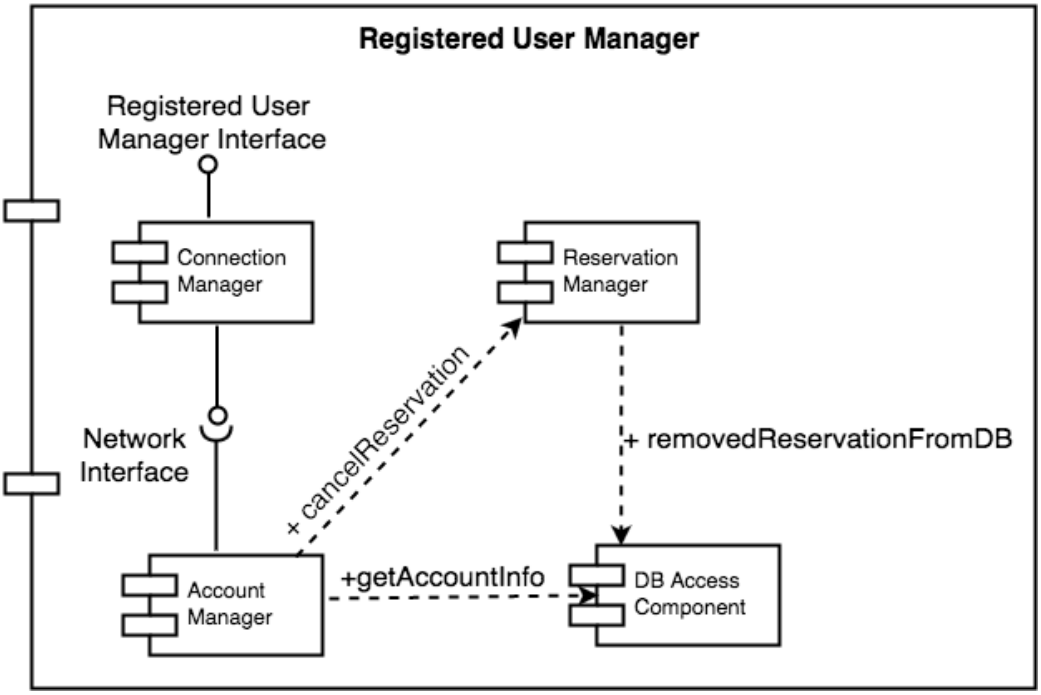
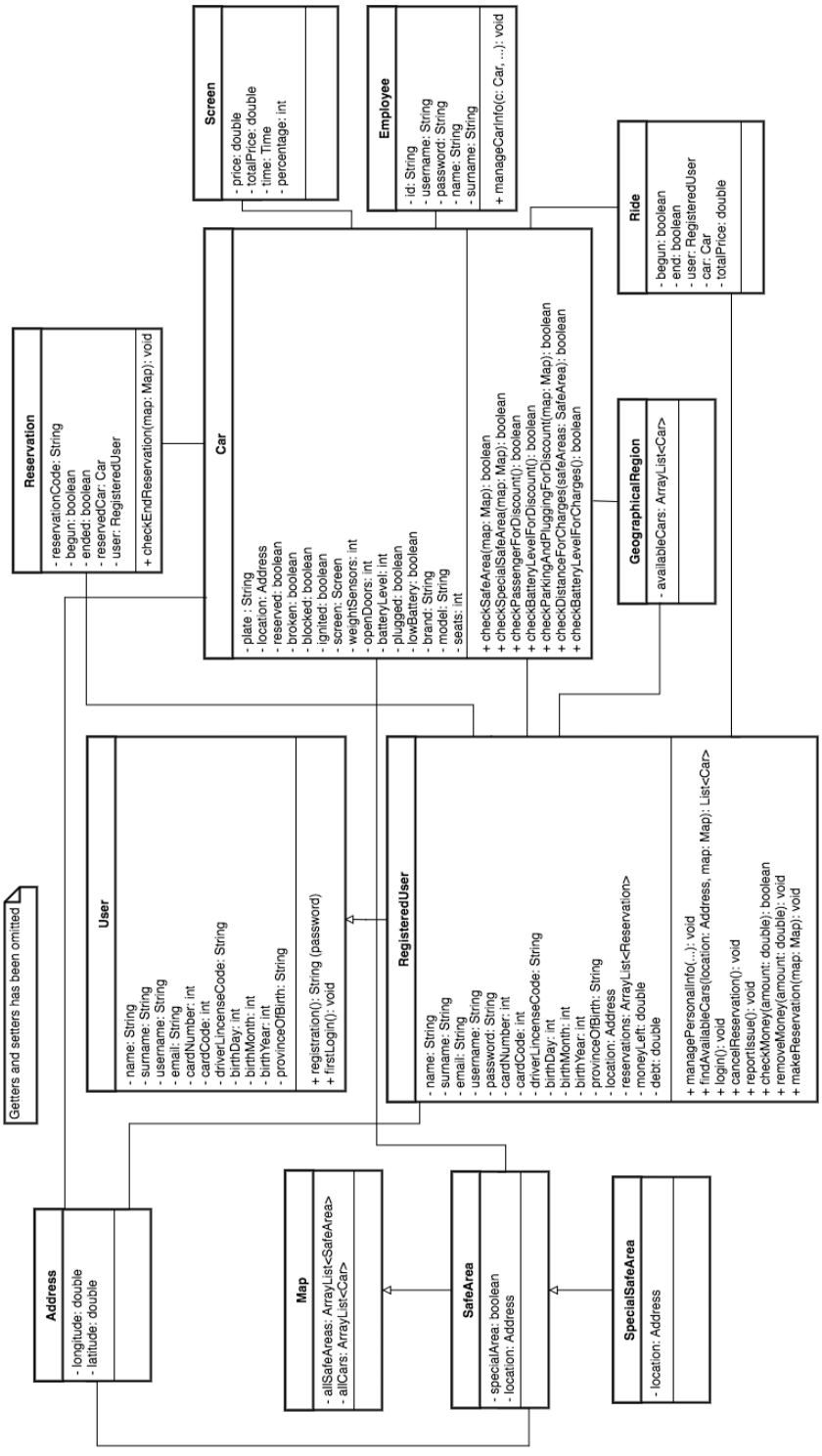


Figure 8: Registered user manager component

2.2.6 Database component



2.3 Deployment view

The following diagram shows how the whole system will be deployed.

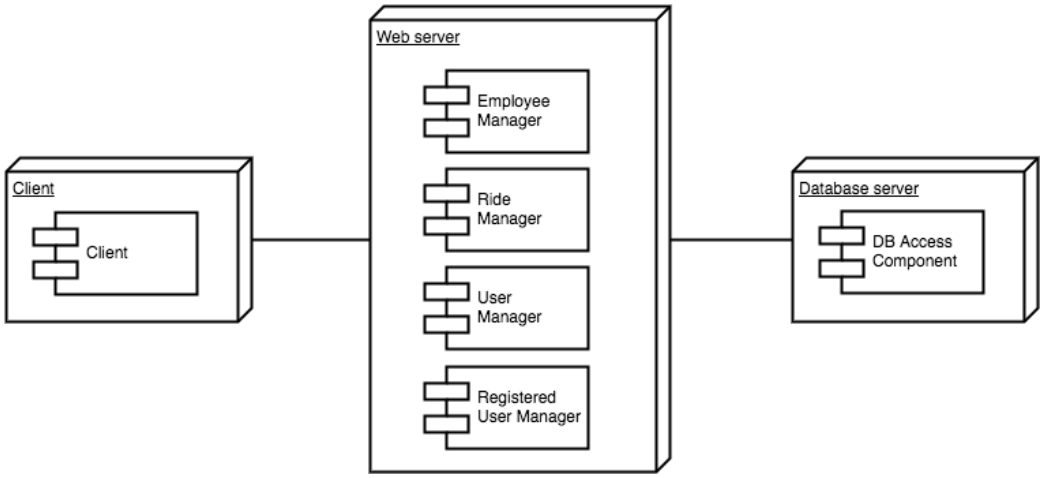


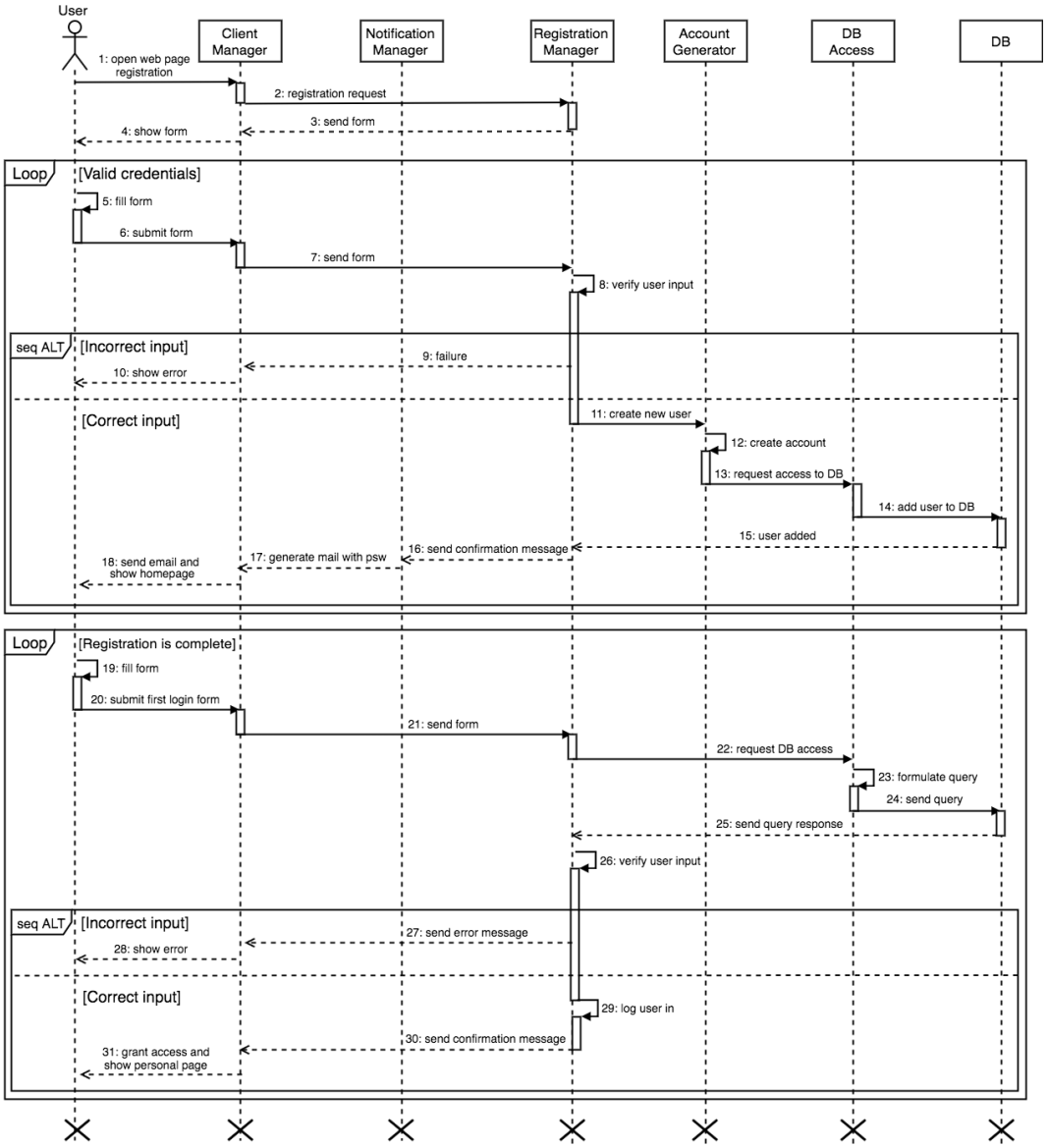
Figure 9: Deployment view

2.4 Runtime view

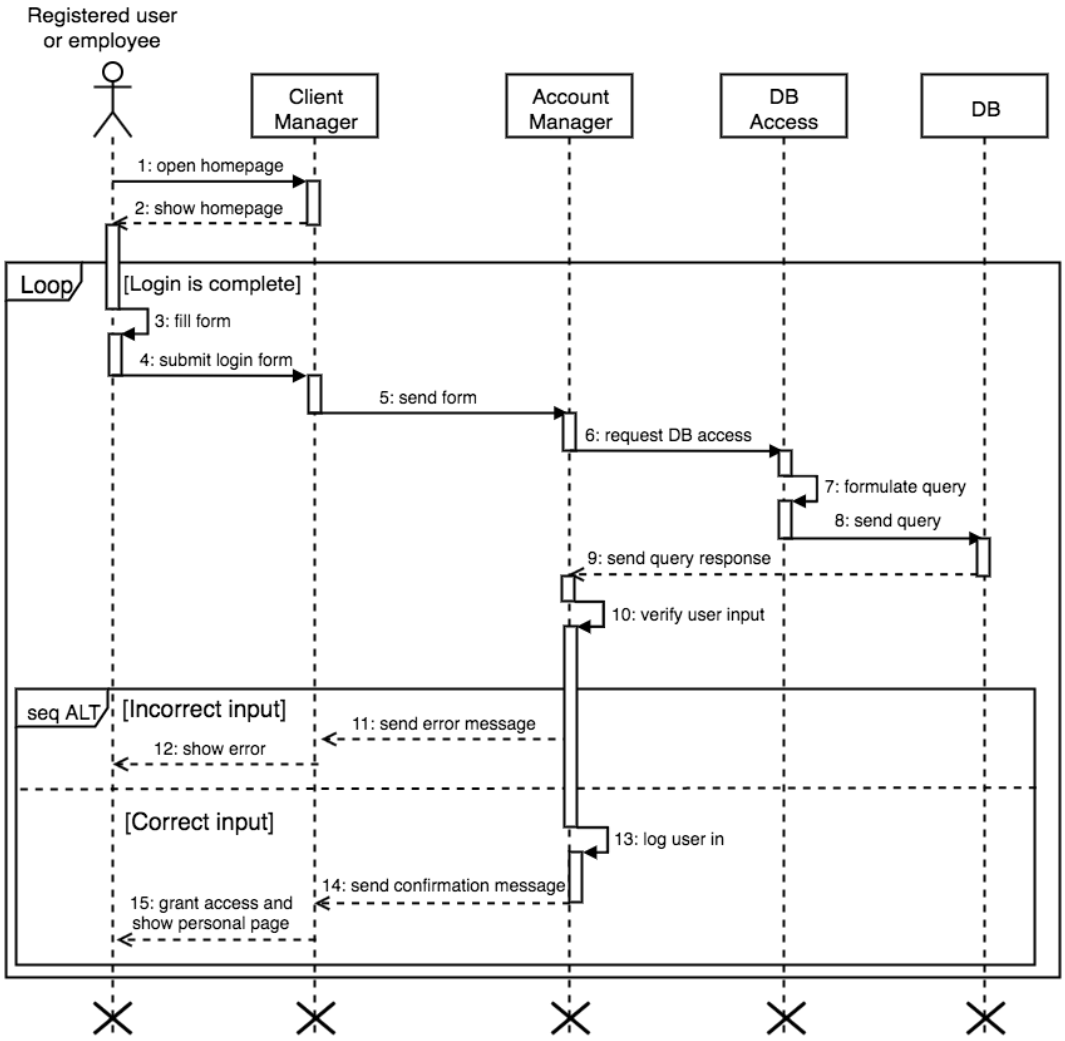
The following sequence diagrams describe the way components interact to accomplish specific tasks related to our use cases.

If the initial action of a diagram corresponds to an already existing sequence diagram, instead of drawing the whole diagram again we thought would be best to just write its name.

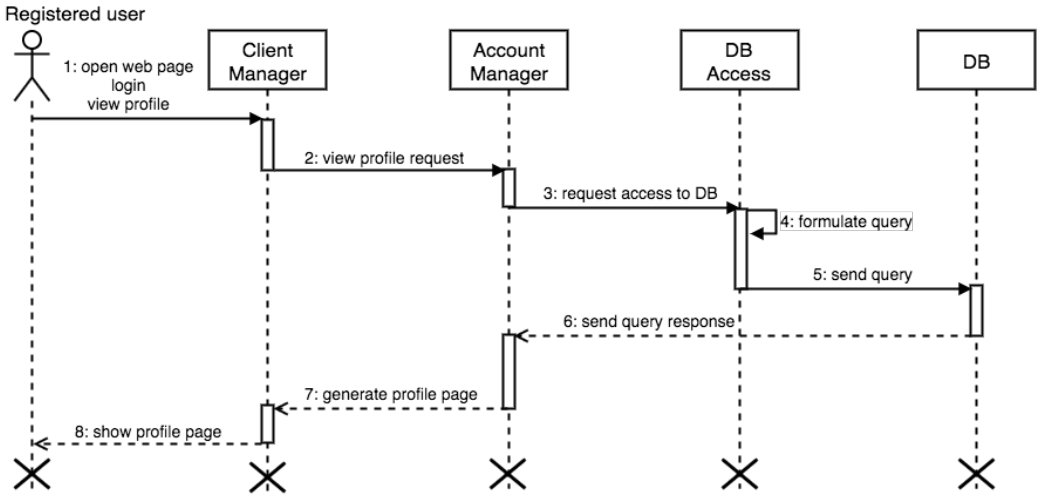
2.4.1 User registration



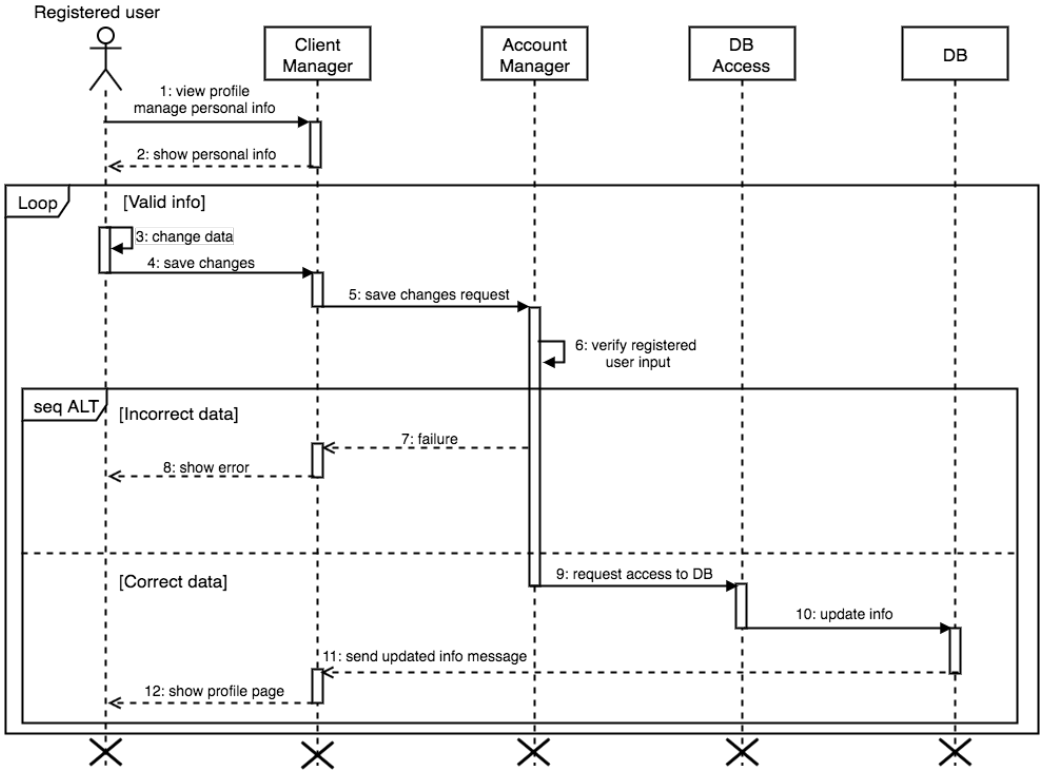
2.4.2 Registered user and employee login



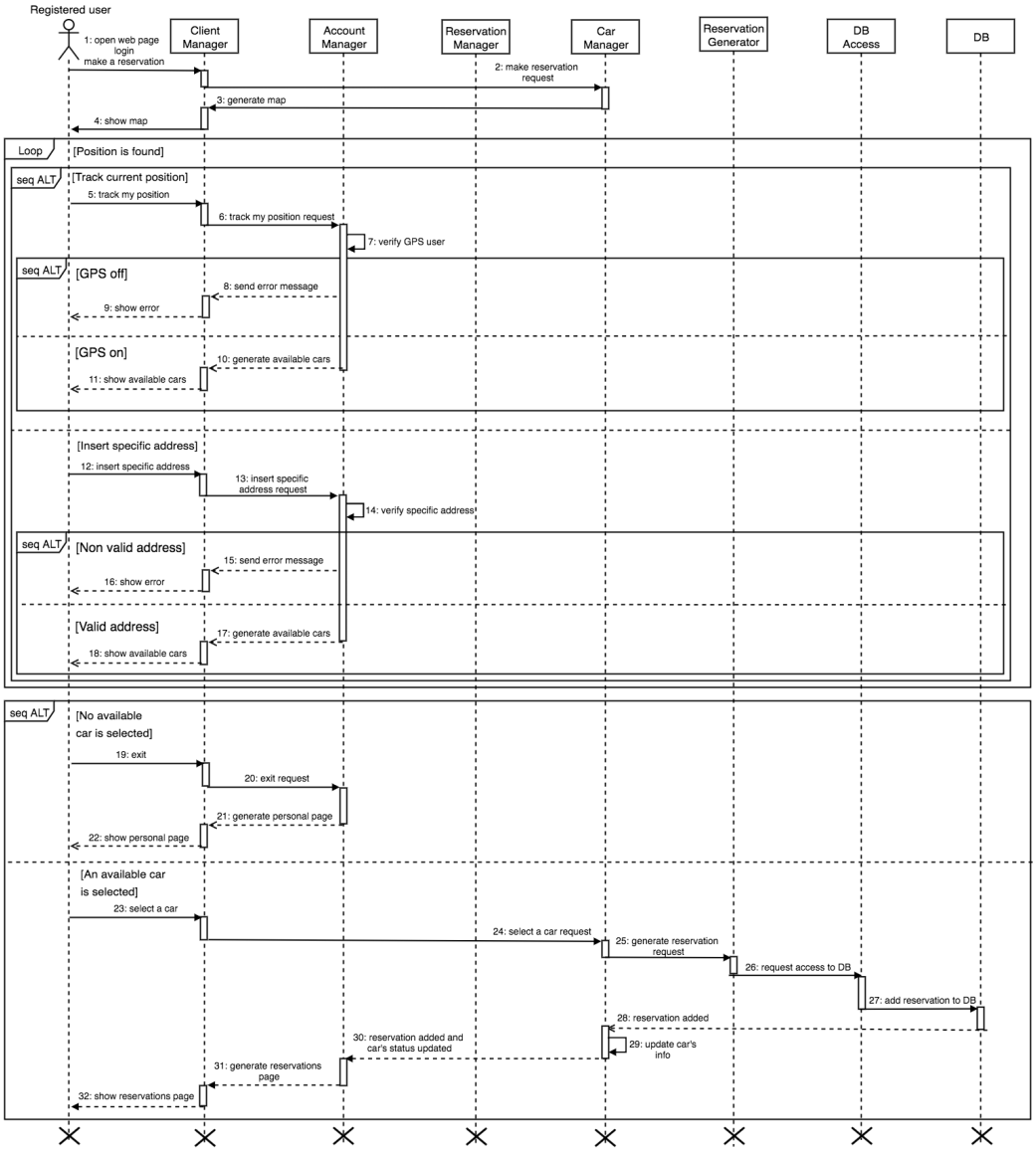
2.4.3 Registered user views his/her profile



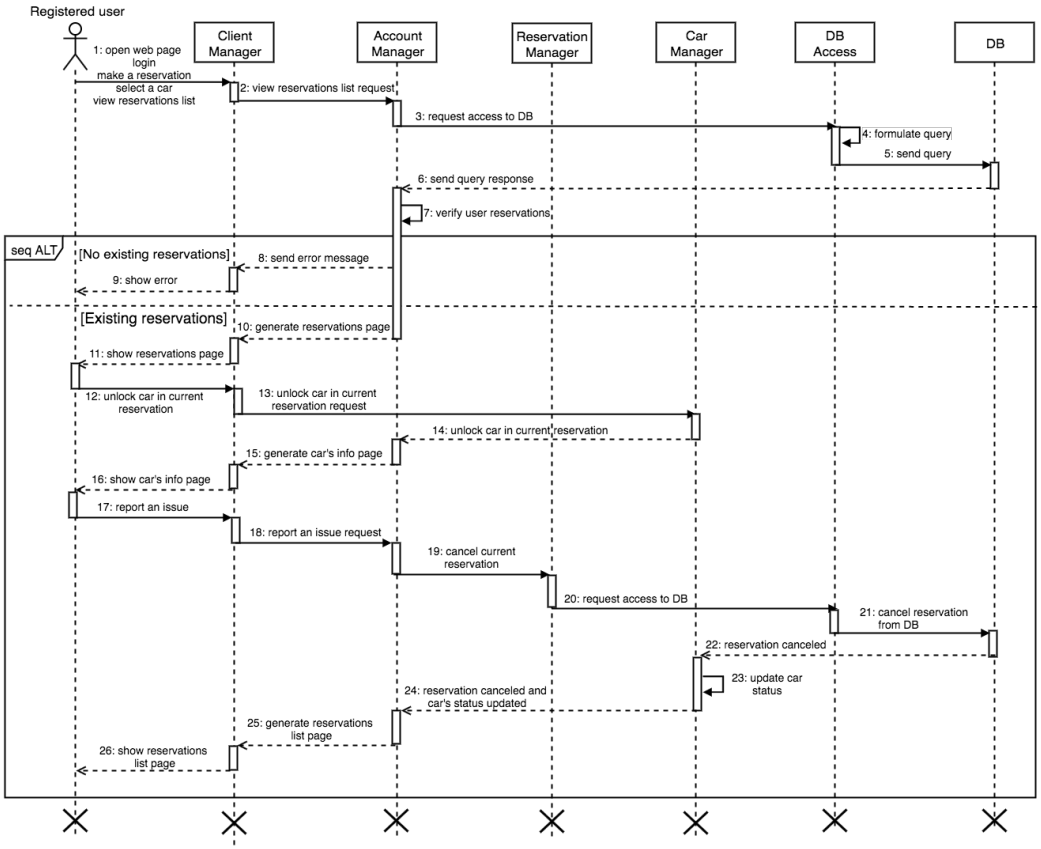
2.4.4 Registered user manages his/her personal information



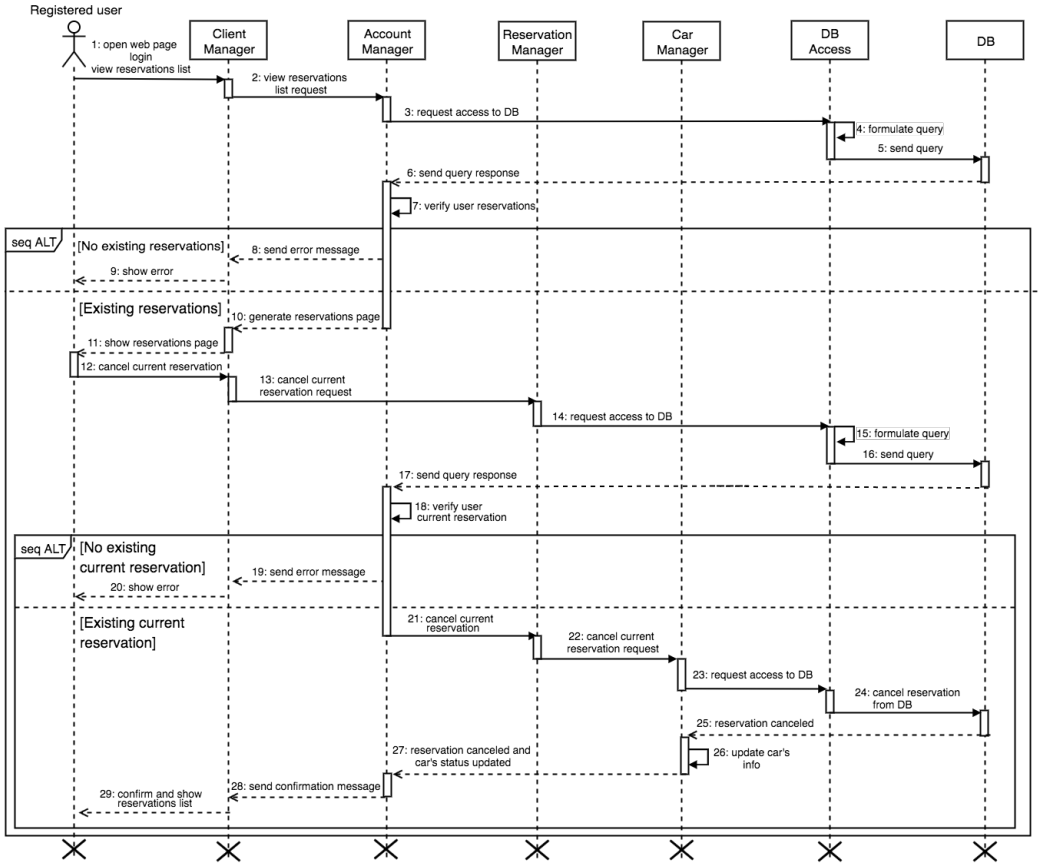
2.4.5 Registered user makes a reservation



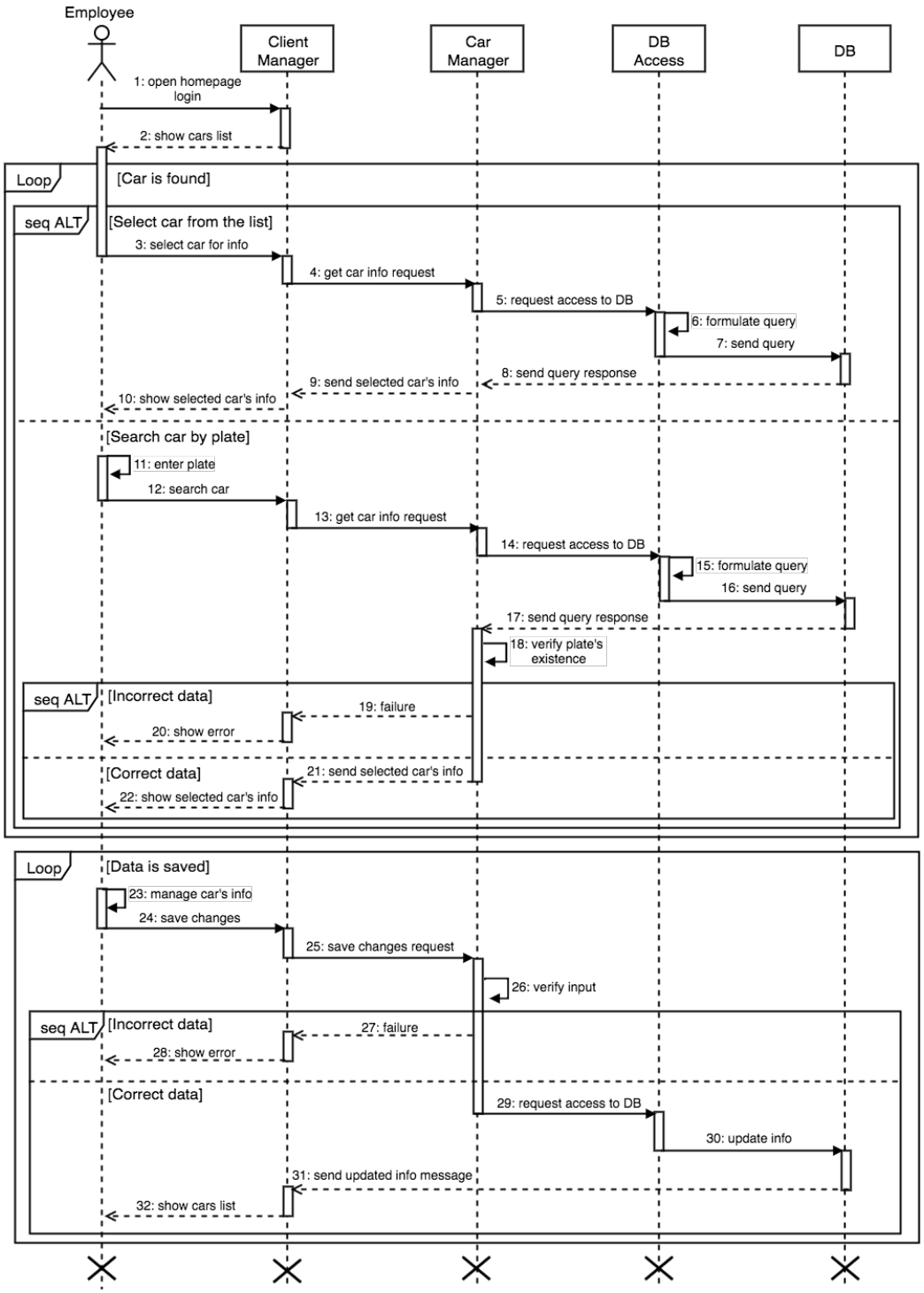
2.4.6 Registered user reports an issue



2.4.7 Registered user cancels his/her current reservation



2.4.8 Employee manages a car's information



2.5 Component interfaces

This section contains a more detailed description of all the application's interfaces.

2.5.1 Client interfaces

- **UIAccess interface**, which provides methods to get user inputs through the user interface.
- **Network interface**, which provides methods to manage the sending and receiving of information with users, server and database.

2.5.2 Employee manager interfaces

- **User manager interface**, which provides methods through which the client can send a registration request to the server.
- **Registered user manager interface**, which provides methods through which the client can send requests to the server (i.e. login, view profile).
- **Ride manager interface**, provides methods through which the client can send requests to the server (i.e. make a reservation, cancel reservation).
- **Network interface**, which provides methods to manage the sending and receiving of information with users, server and database.

2.5.3 User manager interfaces

- **Employee manager interface**, which provides methods through which the employee can send requests to the server (i.e. login, manage cars information).
- **Ride manager interface**, provides methods through which the client can send requests to the server (i.e. make a reservation, cancel reservation).
- **Registered user manager interface**, which provides methods through which the client can send requests to the server (i.e. login, view profile).
- **Network interface**, which provides methods to manage the sending and receiving of information with users, server and database.
- **Email gateway interface**, which provides methods to manage the sending and receiving of emails between users and server.

2.5.4 Registered user manager interfaces

- **Employee manager interface**, which provides methods through which the employee can send requests to the server (i.e. login, manage cars information).
- **User manager interface**, which provides methods through which the client can send a registration request to the server.
- **Ride manager interface**, provides methods through which the client can send requests to the server (i.e. make a reservation, cancel reservation).
- **Network interface**, which provides methods to manage the sending and receiving of information with users, server and database.

2.5.5 Ride manager interfaces

- **Employee manager interface**, which provides methods through which the employee can send requests to the server (i.e. login, manage cars information).
- **User manager interface**, which provides methods through which the client can send a registration request to the server.
- **Registered user manager interface**, which provides methods through which the client can send requests to the server (i.e. login, view profile).
- **Network interface**, which provides methods to manage the sending and receiving of information with users, server and database.
- **Email gateway interface**, which provides methods to manage the sending and receiving of emails between users and server.

2.6 Selected architectural styles and patterns

In this section the styles and patterns we used are described.

2.6.1 Architectural styles

- **Client-Server**
  We have decided to use the Client-Server model because:
  - Having centralized control helps to avoid redundancy and conflict issues. It also makes it easier to find files and manage them.
  - Since all data is stored on the server, it is easy to make a back-up. Also, if data is lost, it can be recovered easily and efficiently.
  - Scalability.
  - Accessibility: the server can be accessed remotely by various platforms in the network.
  - The server can play different roles for different clients.
- **Service-oriented architecture (SOA)**
  We have decided to use a service-oriented architecture for the communication between the application server and the client because:
  - Flexibility and scalability: possibility to implement the features of the architecture's components in whatever language and platform we choose.
  - Easier Testing and Debugging: having all the components isolated into various services makes it easier to test and debug all of them individually.
  - Reusability: since various components are built out separately, it becomes much easier to reuse them later.
- **Three-tiered architecture**
  We have decided to use a three-tiered architecture because:
  - It makes the logical separation between business layer, presentation layer and database layer.
  - As each tier is independent it is possible to enable parallel development of each tier by using different sets of developers.
  - Easy to maintain.

  - Since the application layer is a sort of intermediary between the database layer and the presentation layer, the client will not have a direct access to the database, so the database will be safer.
  - Easy to apply object oriented concepts.
  - Easy to update data provider queries.

2.6.2 Design patterns

- **MVC**: we decided to use the Model-View-Controller pattern in order to improve our application. Its main benefits are:
  - Makes it easier to reuse the code.
  - Useful for dividing the work into application control, data structure and data visualization.
  - Easy to maintain.
  - Makes it easier for other programmers to understand the code.
  - Flexible.

2.7 Other design decisions

Since we also need to integrate our mobile and web application with a map service, we decided to use Google Maps.

3 ALGORITHM DESIGN

This section focuses on the definition of the most relevant algorithmic parts of our application.

3.1 Constants

```
1  public class Constants {
2
3      public static int PASSWORDLENGTH = 12;
4      public static double DISTANCERADIUS = 0.001; //degrees
5      public static int MINWEIGHTSENSORS = 1;
6      public static int MAXWEIGHTSENSORS = 5;
7      public static int DISCOUNT1 = 10;
8      public static int DISCOUNT2 = 20;
9      public static int DISCOUNT3 = 20;
10     public static int OVERCHARGE = -30;
11     public static int MAXPERCENTAGE = 100;
12     public static int MAXSEATS = 5;
13     public static int MINSEATS = 4;
14     public static String MODEL = "Nuova Panda";
15     public static String BRAND = "Fiat";
16 }
```

3.2 User

```java
import java.util.Random;

public class User {
    private String name;
    private String surname;
    private String email;
    private String username;
    private int cardNumber;
    private int cardCode;
    private String driverLicenseCode;
    private int birthDay;
    private int birthMonth;
    private int birthYear;
    private String provinceOfBirth;

    /**
     * Creates a random password after the registration form is filled.
     * @return
     */
    public String registration() {
        // show registration form
        // fill registration form
        char[] chars = "abcdefghijklmnopqrstuvwxyz0123456789".toCharArray();
        StringBuilder sb = new StringBuilder();
        Random random = new Random();
        for(int i=0; i<Constants.PASSWORDLENGTH; i++) {
            char c = chars[random.nextInt(chars.length)];
            sb.append(c);
        }
        return sb.toString();
    }
    /**
     * A new registered user is created with the password return by registration().
     */
    public void firstLogin() {
        String psw = registration();
        new RegisteredUser(name, surname, email, username, cardNumber, cardCode,
                driverLicenseCode, birthDay, birthMonth, birthYear, provinceOfBirth, psw);
        // show personal page to ru
    }
}
```

3.3 Registered User

```java
import java.util.ArrayList;
import java.util.List;

public class RegisteredUser {
    private String name;
    private String surname;
    private String email;
    private String username;
    private int cardNumber;
    private int cardCode;
    private String driverLicenseCode;
    private int birthDay;
    private int birthMonth;
    private int birthYear;
    private String provinceOfBirth;
    private String password;
    private Address location;
    private List<Reservation> reservations; // first reservation is the current one
    private double moneyLeft;
    private double debt;

    public RegisteredUser(String name, String surname, String email, String username,
            int cardNumber, int cardCode, String driverLicenseCode, int birthDay,
            int birthMonth, int birthYear, String provinceOfBirth, String password) {

        this.name = name;
        this.surname = surname;
        this.email = email;
        this.username = username;
        this.cardNumber = cardNumber;
        this.cardCode = cardCode;
        this.driverLicenseCode = driverLicenseCode;
        this.birthDay = birthDay;
        this.birthMonth = birthMonth;
        this.birthYear = birthYear;
        this.provinceOfBirth = provinceOfBirth;
        this.password = password;
        location = new Address(9.1122, 45.2751);
        reservations = new ArrayList<Reservation>();
    }
```

```java
    /**
     * The registered user wants to change his password, card number and card code.
     * @param password
     * @param cardNumber
     * @param cardCode
     */
    public void managePersonalInfo(String password, int cardNumber, int cardCode) {
        this.password = password;
        this.cardNumber = cardNumber;
        this.cardCode = cardCode;
    }
```

```java
    /**
     * The registered user wants to find the location of all available cars within a
     * certain distance (Constants.DISTANCERADIUS) from an address (location).
     * The map contains all the available PowerEnJoy cars in Milan.
     * @param location
     * @param map
     * @return
     */
    public List<Car> findAvailableCars(Address location, Map map) {
        List<Car> availableCars = new ArrayList<Car>();
        for(int i=0; i<map.getCars().size(); i++){
            double x = map.getCars().get(i).getLocation().getLongitude();
            double y = map.getCars().get(i).getLocation().getLatitude();
            if((x-location.getLongitude())*(x-location.getLongitude()) +
                    (y-location.getLatitude())*(y-location.getLatitude())
                    <= Constants.DISTANCERADIUS*Constants.DISTANCERADIUS) {
                availableCars.add(map.getCars().get(i));
            }
        }
        return availableCars;
    }
```

```java
    /**
     * The registered user makes a reservation by selecting one of the available cars.
     * @param map
     */
    public void makeReservation(Map map) {
        // select make reservation from personal page
        // enter a valid address
        List<Car> availableCars = new ArrayList<Car>();
        availableCars = findAvailableCars(location, map);
        if(availableCars.size()!=0){
            // show map with available cars
            // choose a car among the available ones
            new Reservation(availableCars.get(0), this);
            // show reservations page
        }
        else {
            // show error message
        }
    }

    /**
     * The registered user cancels his current reservation, which is always the first
     * one in the reservations array.
     */
    public void cancelReservation() {
        // select reservations list from personal page
        // click on "Cancel reservation" button
        reservations.remove(0);
    }
```

```
    /**
     * The registered user reports an issue once the car is unlocked.
     * -> The reservation is canceled.
     */
    public void reportIssue() {
        // select reservations list from personal page
        // click on "Unlock car" button
        Car reservedCar = reservations.get(0).getCar();
        reservedCar.setBlocked(false); // unlockCar()
        // show car's info
        // click on "Report issue" button
        reservedCar.setBroken(true);
        reservedCar.setReserved(false);
        cancelReservation();
    }
    /**
     * Checks if the amount of money the registered user has on his/her card is enough to
     * pay for the ride.
     * @param amount
     * @return
     */
    public boolean checkMoney(double amount) {
        if(moneyLeft >= amount)
            return true;
        return false;
    }
    /**
     * Removes from the registered user's bank account the ride's price. If the amount of money
     * on his/her card is not enough to pay for the ride, the bank account is zeroed and a debt
     * is assigned to him/her. This debt will be paid once the card is recharged.
     * @param amount
     */
    public void removeMoney(double amount){
        if(checkMoney(amount))
            moneyLeft -= amount;
        else {
            debt = amount - moneyLeft;
            moneyLeft = 0;
        }
    }
}
```

3.4 Employee

```
public class Employee {

    private String id;
    private String name;
    private String surname;
    private String username;
    private String password;

    public void manageCarInfo(Car c, boolean broken, int seats) {
        c.setBroken(broken);
        c.setSeats(seats);
    }
}
```

3.5 Car

```java
import java.util.Random;

public class Car {

    private String plate;
    private Address location;
    private boolean reserved;
    private boolean broken;
    private boolean blocked;
    private boolean ignited;
    private boolean plugged;
    private boolean lowBattery;
    private Screen screen;
    private int weightSensors;
    private int openDoors;
    private int batteryLevel;
    private String brand;
    private String model;
    private int seats;

    // possible cars position
    private double[] longitudes = {9.1122 , 9.1123, 9.1124, 9.1125 , 9.1126};
    private double[] latitudes = {45.2751 , 45.275111, 45.275102, 45.275109 , 45.275119};
```

```java
    public Car() {

        reserved = false;
        broken = false;
        blocked = true;
        ignited = false;
        plugged = false;
        lowBattery = false;
        openDoors = 0;
        batteryLevel = 100;
        weightSensors = 0;
        screen = new Screen();
        brand = Constants.BRAND;
        model = Constants.MODEL;

        Random random = new Random();
        seats = random.nextInt(Constants.MAXSEATS - Constants.MINSEATS + 1) + Constants.MINSEATS;

        // create random plate
        char[] chars = "abcdefghijklmnopqrstuvwxyz0123456789".toCharArray();
        StringBuilder sb = new StringBuilder();
        for(int i=0; i<Constants.PASSWORDLENGTH; i++) {
            char c = chars[random.nextInt(chars.length)];
            sb.append(c);
        }
        plate = sb.toString();

        // random location
        int a = random.nextInt(5);
        int b = random.nextInt(5);
        location = new Address(longitudes[a], latitudes[b]);
    }

    public void setSeats(int seats) {
        this.seats = seats;
    }
```

```
60⊖    public void setBlocked(boolean blocked) {
61         this.blocked = blocked;
62     }
63
```

```
64⊖    public void setBroken(boolean broken) {
65         this.broken = broken;
66     }
67
68⊖    public void setReserved(boolean reserved) {
69         this.reserved = reserved;
70     }
71
72⊖    public Address getLocation() {
73         return location;
74     }
75
76⊖    public void setWeightSensors(int weightSensors) {
77         this.weightSensors = weightSensors;
78     }
79
80⊖    public int getWeightSensors(){
81         return weightSensors;
82     }
83
84⊖    public Screen getScreen() {
85         return screen;
86     }
87
88⊖    /**
89      * Verifies whether the car is in a safe area, by checking if its address
90      * corresponds to the address of a safe area in the map.
91      * @param map
92      * @return
93      */
94⊖    public boolean checkSafeArea(Map map) {
95         for(int i=0; i<map.getSafeAreas().size(); i++){
96             double x = map.getSafeAreas().get(i).getLocation().getLongitude();
97             double y = map.getSafeAreas().get(i).getLocation().getLatitude();
98             if(x == location.getLongitude() && y == location.getLatitude())
99                 return true;
100        }
101        return false;
102    }
```

```java
    /**
     * Checks if the number of passenger is enough to get the 10% discount.
     * @return
     */
    public boolean checkPassengersForDiscount() {
        if(weightSensors >= 3) // the system saves the number of active weight sensors during the ride
            return true;
        return false;
    }

    /**
     * Checks if the battery level is enough to get the 20% discount.
     * @return
     */
    public boolean checkBatteryLevelForDiscount() {
        if(batteryLevel >= 50)
            return true;
        return false;
    }

    /**
     * Verifies whether the car is in a special safe area, by checking if its address corresponds to
     * the address of a safe area in the map with the special attribute set to true.
     * @param map
     * @return
     */
    public boolean checkSpecialSafeArea(Map map) {
        for(int i=0; i<map.getSafeAreas().size(); i++){
            double x = map.getSafeAreas().get(i).getLocation().getLongitude();
            double y = map.getSafeAreas().get(i).getLocation().getLatitude();
            if(x == location.getLongitude() &&
                    y == location.getLatitude() && map.getSafeAreas().get(i).getSpecialArea())
                return true;
        }
        return false;
    }
```

```java
    /**
     * Checks if the car is parked in a special safe area and if it is plugged to get the 20%
     * discount.
     * @param map
     * @return
     */
    public boolean checkParkingAndPluggingForDiscount(Map map) {
        if(checkSpecialSafeArea(map) && plugged)
            return true;
        return false;
    }

    /**
     * Checks if the battery level is lower than 20% to get an overcharge (it should also check if
     * the car is left at more than 3KM from the nearest power grid station).
     * @return
     */
    public boolean checkBatteryLevelForCharges() {
        if(batteryLevel <= 20)
            return true;
        return false;
    }
}
```

3.6 Screen

```
1  public class Screen {
2
3      private double price;
4      private double totalPrice;
5      private int percentage;
6
7      public Screen() {
8          price = 0;
9          totalPrice = 0;
10         percentage = 0;
11     }
12
13     public void setPrice(double price) {
14         this.price = price;
15     }
16
17     public void setPercentage(int percentage) {
18         this.percentage += percentage;
19     }
20
21     public void setTotalPrice(double totalPrice) {
22         this.totalPrice = totalPrice;
23     }
24
25     public double getPrice() {
26         return price;
27     }
28
29     public double getTotalPrice() {
30         return totalPrice;
31     }
32
33     public int getPercentage() {
34         return percentage;
35     }
36 }
```

3.7 Reservation

```java
import java.util.Random;

public class Reservation {

    private Car reservedCar;
    private RegisteredUser user;
    private boolean ended;

    public Reservation(Car reservedCar, RegisteredUser user) {
        this.reservedCar = reservedCar;
        Random random = new Random();
        // random number of passengers
        int weightSensors = random.nextInt(Constants.MAXWEIGHTSENSORS - Constants.MINWEIGHTSENSORS) + 1;
        reservedCar.setWeightSensors(weightSensors);
        reservedCar.setReserved(true);
        this.user = user;
    }

    public Car getCar() {
        return reservedCar;
    }
```

```java
    /**
     * Every time a car's "ignited" attribute is set to false, this method is invoked.
     * It checks if the ride corresponds to one of the discount or overcharge cases and
     * then calculates the total prince and sets the screen price to it.
     * @param map
     */
    public void checkEndReservation(Map map){
        if(reservedCar.checkSafeArea(map)) {
            reservedCar.getScreen().setPrice(18.3);
            ended = true;
            reservedCar.setReserved(false);
            reservedCar.setBlocked(true);
            if(reservedCar.checkPassengersForDiscount()) {
                reservedCar.getScreen().setPercentage(Constants.DISCOUNT1);
            }
            if(reservedCar.checkBatteryLevelForDiscount()) {
                reservedCar.getScreen().setPercentage(Constants.DISCOUNT2);

            }
            if(reservedCar.checkParkingAndPluggingForDiscount(map)) {
                reservedCar.getScreen().setPercentage(Constants.DISCOUNT3);
            }
            if(reservedCar.checkBatteryLevelForCharges()) {
                reservedCar.getScreen().setPercentage(Constants.OVERCHARGE);
            }
            // show discount on screen
            double totalPrice = reservedCar.getScreen().getPrice() -
                    (reservedCar.getScreen().getPrice() *
                            reservedCar.getScreen().getPercentage()/Constants.MAXPERCENTAGE);
            reservedCar.getScreen().setTotalPrice(totalPrice);
        }
    }
}
```

3.8 Address

```
1  public class Address {
2
3      private double longitude;
4      private double latitude;
5
6      public Address(double longitude, double latitude) {
7          this.longitude = longitude;
8          this.latitude = latitude;
9      }
10
11     public double getLongitude(){
12         return longitude;
13     }
14
15     public double getLatitude(){
16         return latitude;
17     }
18 }
```

3.9 Map

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class Map {
5
6      private List<Car> allCars = new ArrayList<Car>();
7      private List<SafeArea> allSafeAreas = new ArrayList<SafeArea>();
8
9      public Map() {
10         // 5 PowerEnJoy cars
11         for(int i=0; i<5; i++) {
12             allCars.add(new Car());
13         }
14
15         for(int i=0; i<5; i++) {
16             allSafeAreas.add(new SafeArea());
17         }
18     }
19
20     public List<Car> getCars() {
21         return allCars;
22     }
23
24     public List<SafeArea> getSafeAreas() {
25         return allSafeAreas;
26     }
27 }
```

3.10 Safe Area

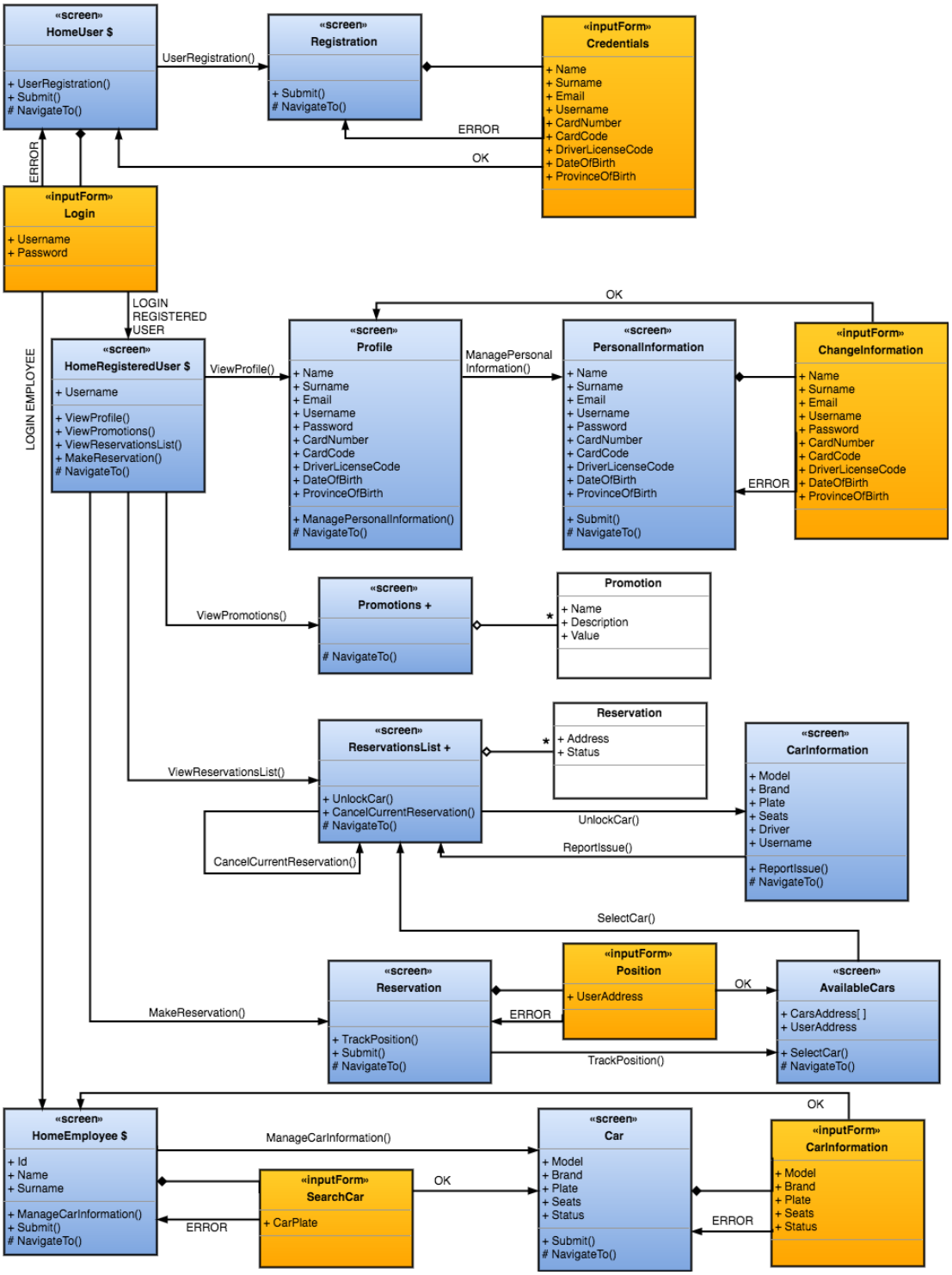
```
1  import java.util.Random;
2
3  public class SafeArea {
4
5      private boolean specialSafeArea;
6      private Address location;
7
8      private double[] longitudes = {9.1122 , 9.1123, 9.1124, 9.1125 , 9.1126};
9      private double[] latitudes = {45.2751 , 45.275111, 45.275102, 45.275109 , 45.275119};
10
11     public SafeArea() {
12         // random location
13         Random random = new Random();
14         int a = random.nextInt(5);
15         int b = random.nextInt(5);
16         location = new Address(longitudes[a], latitudes[b]);
17     }
18
19     public Address getLocation() {
20         return location;
21     }
22
23     public boolean getSpecialArea() {
24         return specialSafeArea;
25     }
26 }
```

4 USER INTERFACE DESIGN

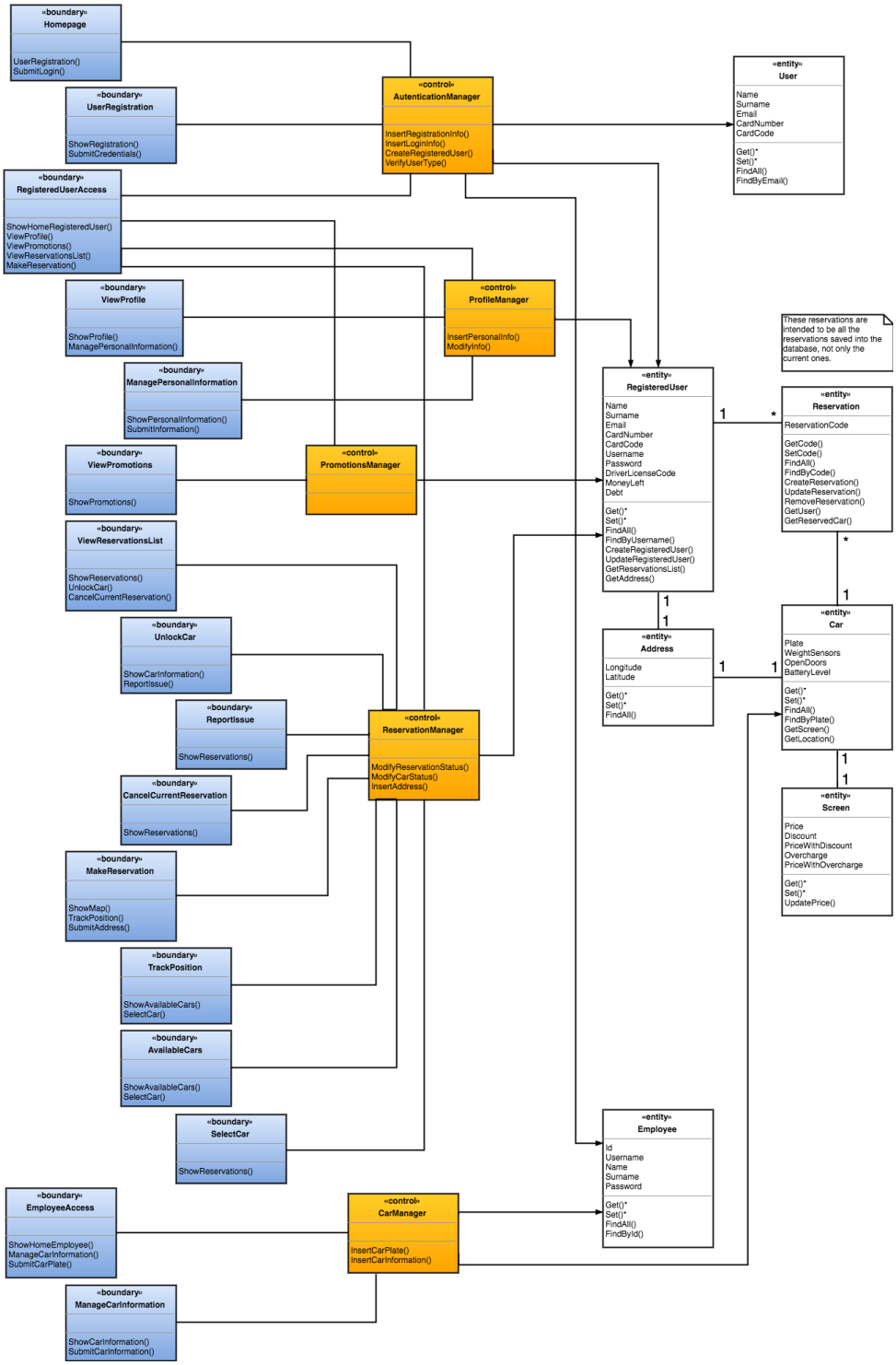
4.1 Mockups

We have already presented our mockups in sections 3.2.1, 3.2.2 and 3.2.3 of the RASD document.

4.2 UX diagram



4.3 BCE diagram



5 REQUIREMENTS TRACEABILITY

Here are listed, under each goal, all the design components that will assure its fulfillment.

- [G1] Allow users to register to the system by filling out a form.
  - Client manager.
  - Notification manager.
  - Registration manager.
  - Account generator.
  - DB access manager.
- [G2] Allow registered users to access the system by entering their username and password.
  - Client manager.
  - Account manager.
  - DB access manager.
- [G3] Allow registered users to manage their personal information.
  - Client manager.
  - Account manager.
  - DB access manager.
- [G4] Allow registered users to find the locations of all available cars within a certain distance from their current location or from a specified address.
  - Client manager.
  - Reservation manager.
  - Car manager.
  - DB access manager.
- [G5] Allow registered users to reserve a single car among the available ones in a certain geographical region for up to one hour before they pick it up.
  - Client manager.

  - Reservation manager.
  - Car manager.
  - Reservation generator.
  - DB access manager.

- [G6] Allow registered users to tell the system they are nearby when they reach a car they previously reserved.
  - Client manager.
  - Connection manager.
  - Car manager.
  - Account manager.
  - Reservation manager.
  - DB access manager.

- [G7] Allow registered users to see on a screen the amount of money they are being charged for while they are driving.
  - Client manager.
  - Car manager.
  - DB access manager.

- [G8] Allow registered users to see on a screen a map showing all the safe areas they can park in.
  - Client manager.
  - Car manager.
  - DB access manager.

- [G9] Allow registered users to see on a screen the discount percentage (if any) applied on their bill once the ride has ended.
  - Client manager.
  - Car manager.
  - DB access manager.

- [G10] Allow registered users to cancel a reservation paying a fee of 1 EUR.
  - Client manager.

  - Reservation manager.
  - Car manager.
  - Account manager.
  - DB access manager.

- [G11] Allow registered users to benefit from a discount percentage in certain cases.
  - Client manager.
  - Account manager.
  - Reservation manager.
  - Car manager.
  - DB access manager.

- [G12] Allow registered users to report an issue when they realize a car they reserved is somehow broken.
  - Client manager.
  - Reservation manager.
  - Account manager.
  - Car manager.
  - DB access manager.

- [G13] Allow employees to interact with the system and manage cars information.
  - Client manager.
  - Account manager.
  - Car manager.
  - DB access manager.

6 EFFORT SPENT

This section includes information about the number of hours each group member has worked towards the fulfillment of this deadline.

Since we decided to work together every day, the worked hours are going to be the same for each group member. We think this is the best way to achieve good results.

6.1 Agosti Isabella

- 26/11/2016: 8h
- 27/11/2016: 3h
- 02/12/2016: 6h
- 03/12/2016: 3h
- 05/12/2016: 3h
- 06/12/2016: 4h
- 08/12/2016: 4h
- 09/12/2016: 8h

6.2 Cattivelli Carolina

- 26/11/2016: 8h
- 27/11/2016: 3h
- 02/12/2016: 6h
- 03/12/2016: 3h
- 05/12/2016: 3h
- 06/12/2016: 4h
- 08/12/2016: 4h
- 09/12/2016: 8h

7 REFERENCES

7.1 Used tools

The tools we used to create this document are:

- **draw.io**
  To create the UX and BCE diagrams.
  https://www.draw.io
- **GitHub** and **GitHub Desktop**
  To collaborate with the team and keep track of the changes in the document.
  https://github.com and https://desktop.github.com
- **TeXstudio**
  LaTeX editor we used to write this document.
  http://www.texstudio.org
- **BasicTeX**
  Distribution of the LaTeX system.
  http://www.tug.org/mactex/morepackages.html
- **Eclipse IDE for Java developers**
  To design the algorithmic parts.
  https://eclipse.org