Politecnico di Milano

Software Engineering 2

DESIGN DOCUMENT

PowerEnjoy

Authors:

Simone Bruzzechesse

Luca Franceschetti

Gian Giacomo Gatti

INDEX

1 INTRODUCTION

1.1 PURPOSE

The purpose of this document is to give a better explanation of the PowerEnjoy system with respect to the RASD document. The aim of this document is to define aspects of the software that are useful for the programmers to develop the system according to the specifications that are proposed in this document. The document purpose is to identify:

- High level infrastructure of the system
- The main components of the system and the interfaces between them
- The main interfaces of the system with existing components or 3rd part software
- The runtime behaviour of the system

1.2 SCOPE

The aim of PowerEnjoy software is to provide a system for a car-sharing service that exclusively employs electric cars, so it should provide the functionality normally provided by car-sharing services. The users must be able to register to the system by providing their credentials and payment information, then they receive back a password that can be used to access the system. This procedure can be done both by the mobile application or web application.

Registered users must be able to find the locations of available cars within a certain distance from their current location or from a specified address. The system provides also the possibility to reserve a single car, but with some constraint: for example, if a car is not picked up, the user must pay a fee. On the other hands, if a user reaches a reserved car, he must be able to tell the system he’s nearby his reserved car, so the car will be unlocked and the user can enter and start his rent.

Car-sharing system initiates the charging of money as soon as the engine ignites, and the system starts charging the user for a given amount of money per minute. Indeed, the user is notified of the current charges through a screen on the car. The system stops charging the user as soon as the car is parked in a safe area and the user exits the car.

The set of safe areas where is possible to park cars is predefined by the management system, so we can contact a database to catch some information about the current position of the car, and then the system can decide if it is parked in a safe area.

Although, the system must be able to define how user behaves with respect to the service, applying some discounts (or chargings) in consequence of certain action.

1.3 Definitions, Acronyms, Abbreviation

Here there are some definitions and acronyms that we will use later on this document:

- **RASD**: requirements analysis and specifications document
- **DD**: design document
- **SMS**: short message service; it is a notification sent to a mobile phone, an SMS gateway is needed to use it.
- **ChargingStationAreas**: some areas around a charging station, in our case the areas have a radius of 3Km that is the bound for our system over which the system applies a charge to the user.
- **SafeAreas**: areas where is possible to park the car. They are defined and stored in database.
- **API**: application programming interface; it is a common way to communicate with another system.
- **MVC**: model view controller
- **URL**: uniform resource locator
- **Path**: it’s a structure containing at least 2 positions
- **REST**: Representational State Transfer
- **RESTful**: REST with no session
- **UX**: user experience design

1.4 Reference Documents

- Our RASD document
- Specification Document: Assignments AA 2016-2017.pdf
- Structure of the design document.pdf
- Sample Design Deliverable Discussed on Nov. 2

1.5 Document Structure

1. **Introduction**: this section introduces the design document. It contains the purpose and the scope of this document and the parts that are more specified with respect to the RASD document.
2. **Architecture Design**: this section is divided into different parts:
   2.1. Overview: this part explains the main tiers of our application;
   2.2. High level components and their interaction: this second part gives a high-level view of our architecture and its main components and the interaction between them;
   2.3. Component view: this sections gives a more detailed view of the components of our applications;
   2.4. Deploying view: this section explains which components must be deployed to let the application running in the correct way;
   2.5. Runtime view: “architectural” sequence diagrams are represented in this section to better define the process of the different tasks of our application;
   2.6. Component interfaces: this section shows the main interfaces between our components and existing software;
   2.7. Selected architectural styles and patterns: this section explains the architectural choices taken during the creation of the application;
   2.8. Other design decisions
3. **Algorithms Design**: this section describes the main algorithms and the most critical ones that are needed for the application. We use Pseudo code in order to show the main idea of the algorithm and at the same time to hide unnecessary information and keep the algorithm “light” as much as possible.

4. **User Interface Design**: this section presents the mockups that have been already seen in the RASD document and present the user experience explained via UX.
5. **Requirements Traceability**: this section aims to explain how the decisions taken in the RASD are linked to design elements.

2 Architectural Design

2.1 Overview

The overview of our architecture is represented in the diagram below. In particular, we consider the traditional client-server architecture based on three-tier.

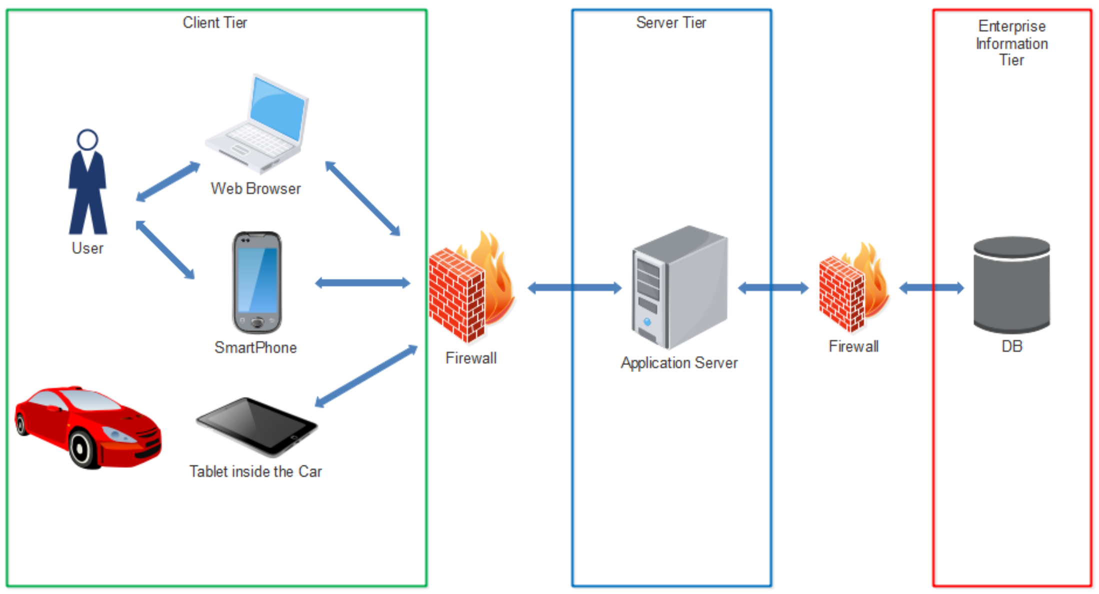


Figure 1 Overview of the Architecture of the system

As we can see from the diagram this architecture reflects the basic one of the java EE standard, in fact also that one has 3 tier and allows us to abstract from the complexity of the application by organizing it into components and containers. This allows us also to abstract from many issues related to a complex architecture since a lot of services are provided directly from the containers and also the deployment time is reduced (due for example to the use of the annotations). We report also the architecture of the J2EE in order to show the parallelism with our system.

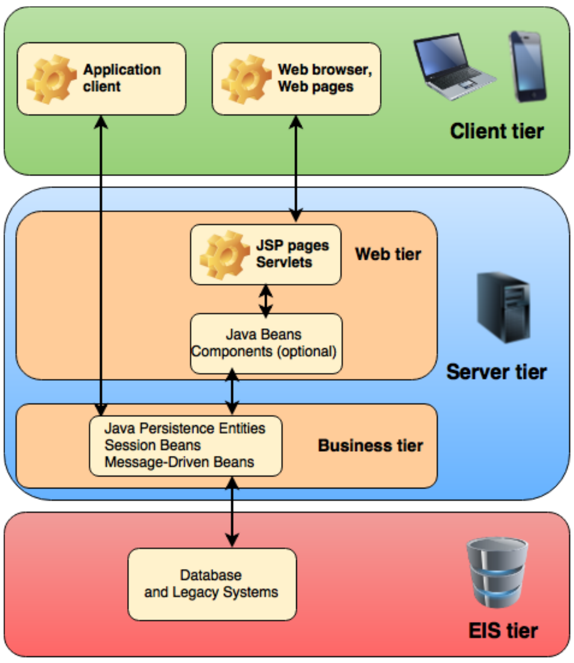

Application client
Web browser, Web pages
Client tier
JSP pages Servlets
Web tier
Java Beans Components (optional)
Server tier
Java Persistence Entities Session Beans Message-Driven Beans
Business tier
Database and Legacy Systems
EIS tier

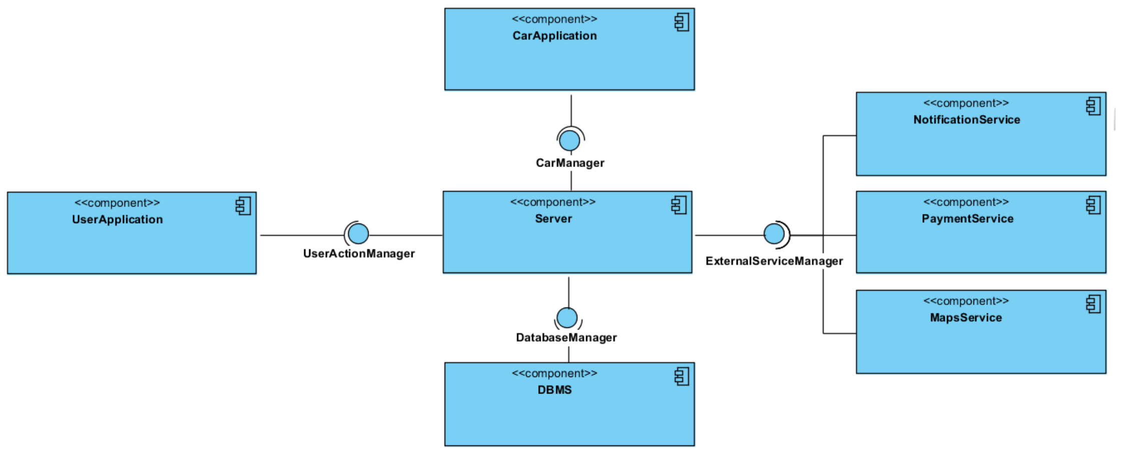
Figure 2 J2EE Architecture

We show the high-level components and their functionalities:

- **Client-tier**: run on the client machine, both on the web browser and mobile application and also on the machine's tablet;
- **Server-tier**:
  - **Web-tier:** run on the Java EE server; they are organized in Servlets and JSP pages and they will be used to manage the interaction with the web application;
  - **Business-tier**: run on the Java EE server; these components manage the internal logic of the system and they will be analysed later in the Component Diagrams;
- **Enterprise Information System (EIS)-tie**: run on the database server and handle enterprise infrastructure systems, database systems and legacy systems.

2.2 COMPONENT VIEW

First, we give an overview of the main components of our system and how they interface to each other, then we will present the components, each one in higher level of details. The figure below shows the high-level view of our system's components.



<<component>> CarApplication
CarManager
<<component>> UserApplication
UserActionManager
<<component>> Server
ExternalServiceManager
<<component>> NotificationService
<<component>> PaymentService
<<component>> MapsService
DatabaseManager
<<component>> DBMS

Figure 3 Main Components Diagram

Main components and their functionalities:

1. **User Application**:
   1.1. Research a car on the map
   1.2. Reserve a car
   1.3. Unlock a car and use it
2. **Notification, Payment, Maps Services**:
   2.1. Make a payment;
   2.2. Send a notification (email or sms);
   2.3. Use the maps' service;
3. **Car Application**:
   3.1. Visualize the car's status;
   3.2. Monitor the position of the car;
   3.3. End a started trip;

The high-detailed component diagram is shown below.

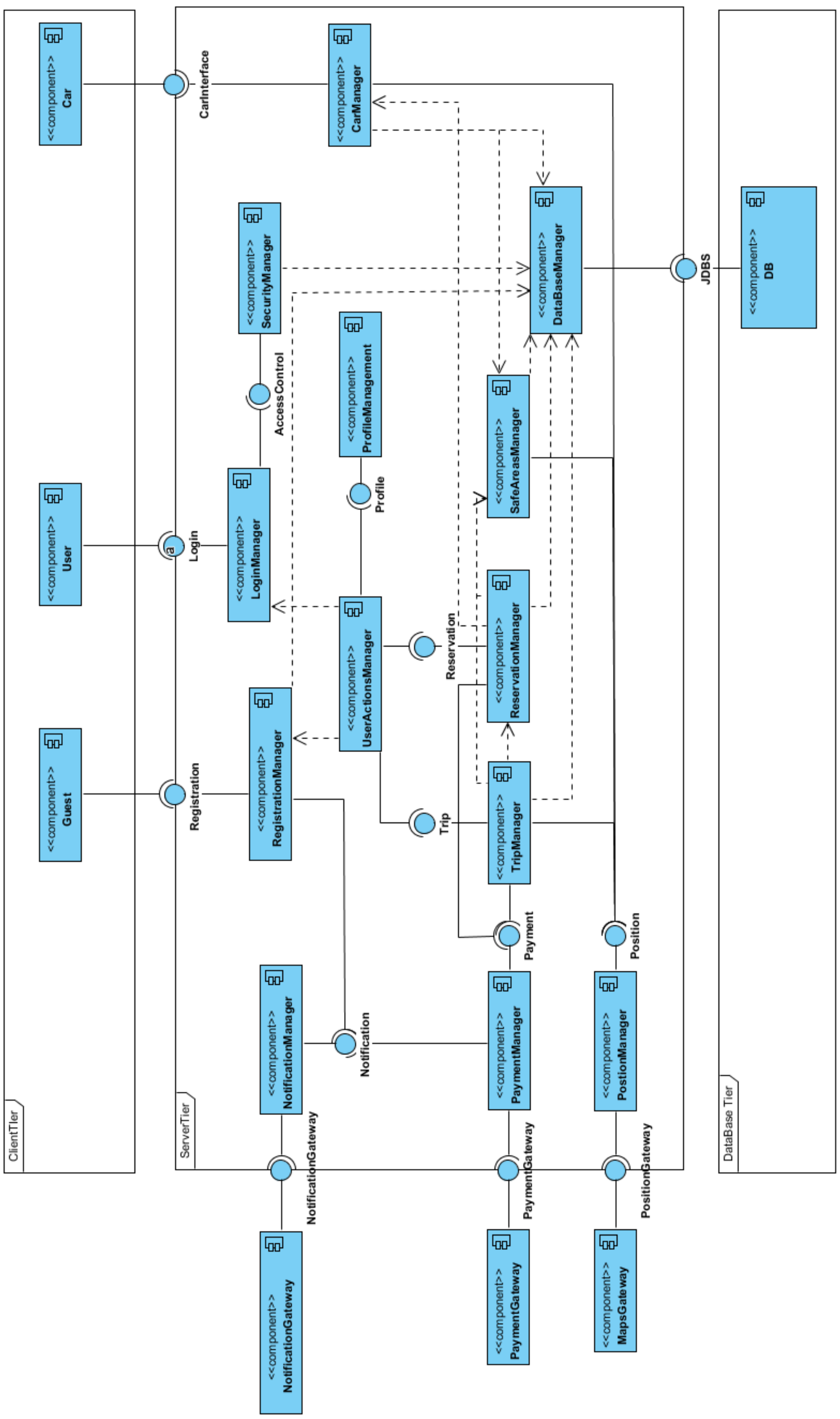

ClientTier
<<component>> Car
<<component>> User
<<component>> Guest
ServerTier
CarInterface
Login
Registration
<<component>> CarManager
<<component>> SecurityManager
AccessControl
<<component>> LoginManager
<<component>> ProfileManagement
Profile
<<component>> RegistrationManager
<<component>> UserActionsManager
Reservation
Trip
<<component>> DataBaseManager
<<component>> SafeAreasManager
<<component>> ReservationManager
<<component>> TripManager
Payment
Position
<<component>> NotificationManager
Notification
<<component>> PaymentManager
<<component>> PostionManager
NotificationGateway
PaymentGateway
PositionGateway
<<component>> NotificationGateway
<<component>> PaymentGateway
<<component>> MapsGateway
JDBS
DataBase Tier
<<component>> DB

Figure 4 High-Detailed Component Diagram

The main components of our system are listed below:

- **RegistrationManager**: this component is responsible for the registration of a new user to the system. It uses some services offered by the securityManager to handle the registration of a new user into the system, using some interfaces offered by the Security component.
- **LoginManager**: this component is responsible for the login process of a user into the system. It will use some interfaces offered by the Security component in order to allow only registered users to use the system;
- **SecurityManager**: this component is responsible for the security of the system, it can recognize authorized users and let them use properly the system and denied its usage to unregistered users;
- **ProfileManager**: this component can manage the information associated to a logged user, in fact it can change some of them such as the card that he uses for the payment or his driving license in case it expires.
- **CarManager:** this component is responsible for the interface of the system with the cars of the system, in particular with this component we can acquire information related to the status of the car such as the battery level, position or if it is plugged or not.
- **SafeAreasManager**: this component coordinates all the areas of the city, in particular it distinguishes between:
  - SafeAreas
  - ChargingStationAreas
  - Other Areas
- **TripManager**: the goal of this component is to manage a trip and all the data associated to it, so it can perform different operations:
  - Make a new trip;
  - Apply the discounts to a trip;
  - Order the payment for the user who ends the trip;
- **ReservationManager**: this component can manage a reservation and all the information associated to it, so it can also perform different operations:
  - Make a new reservation;

    - Delete a reservation;
    - Know if a reservation has expired and ask the PaymentManager to add the fee to the user associated to that reservation;
    - Mark a reservation as complete if the user picks up the car in time;
- **UserActionsManager**: the scope of this component is to manage all the possible actions that the user can perform, for example it allows the user to manage his account, reserve a car or start a trip according to the state of the user. It can perform all this actions using the different operations that other components provide through the interfaces.
- **PaymentManager**: can manage a payment since it can interface with the PaymentGateway who manages the effective payment.
- **PaymentGateway**: is the component that is directly interfaced with another software that manage all the payments of the system.
- **MapsGateway**: the scope of this component is to get the GPS positions of the devices that uses it. This component after using it sends the data to the other components in the system;
- **NotificationManager**: this component can manage all the types of notification that must be sent to the user:
    - TripReview and Cost;
    - Fee;
    - Confirmation Email;
    - Sms message to confirm the smartphone association;
- **NotificationGateway**: is linked with the external software that can send sms and email;
- **DatabaseManager**: this component manages the interaction with the database; it is responsible for providing all the information asked by all the others components and also to send new data information or modifications to the DB.

  In order to complete the specification of the components and in particular of the DatabaseManager we need to explain which is the model of the system and so his main classes for the objects that it manages. The data structure of the system is represented in the UML diagram below.

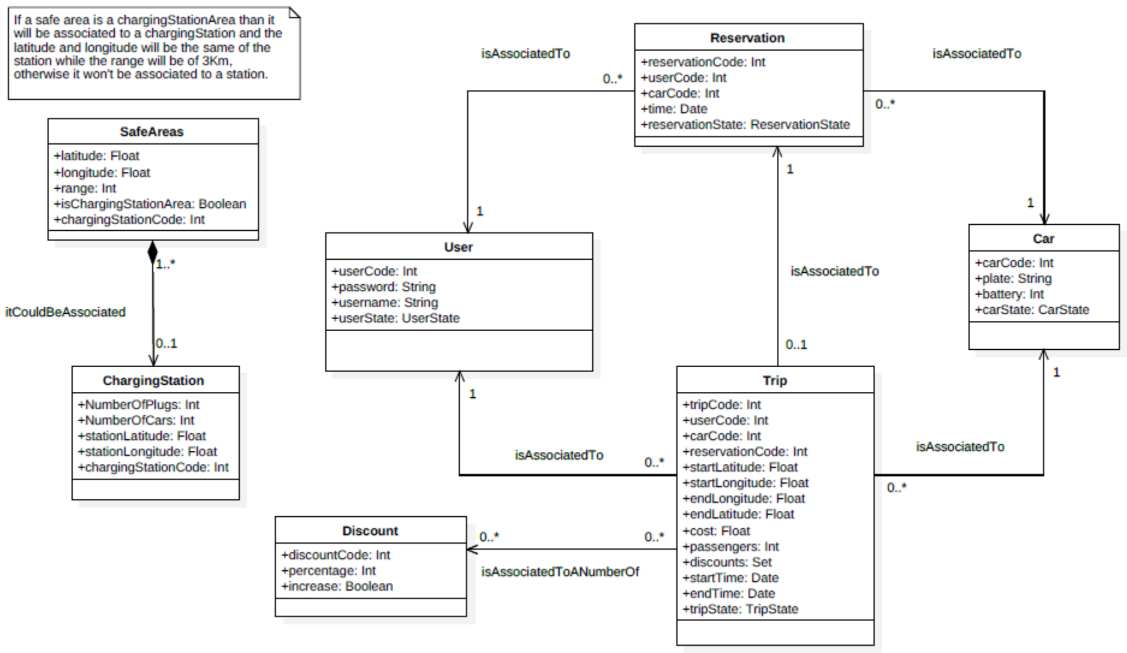
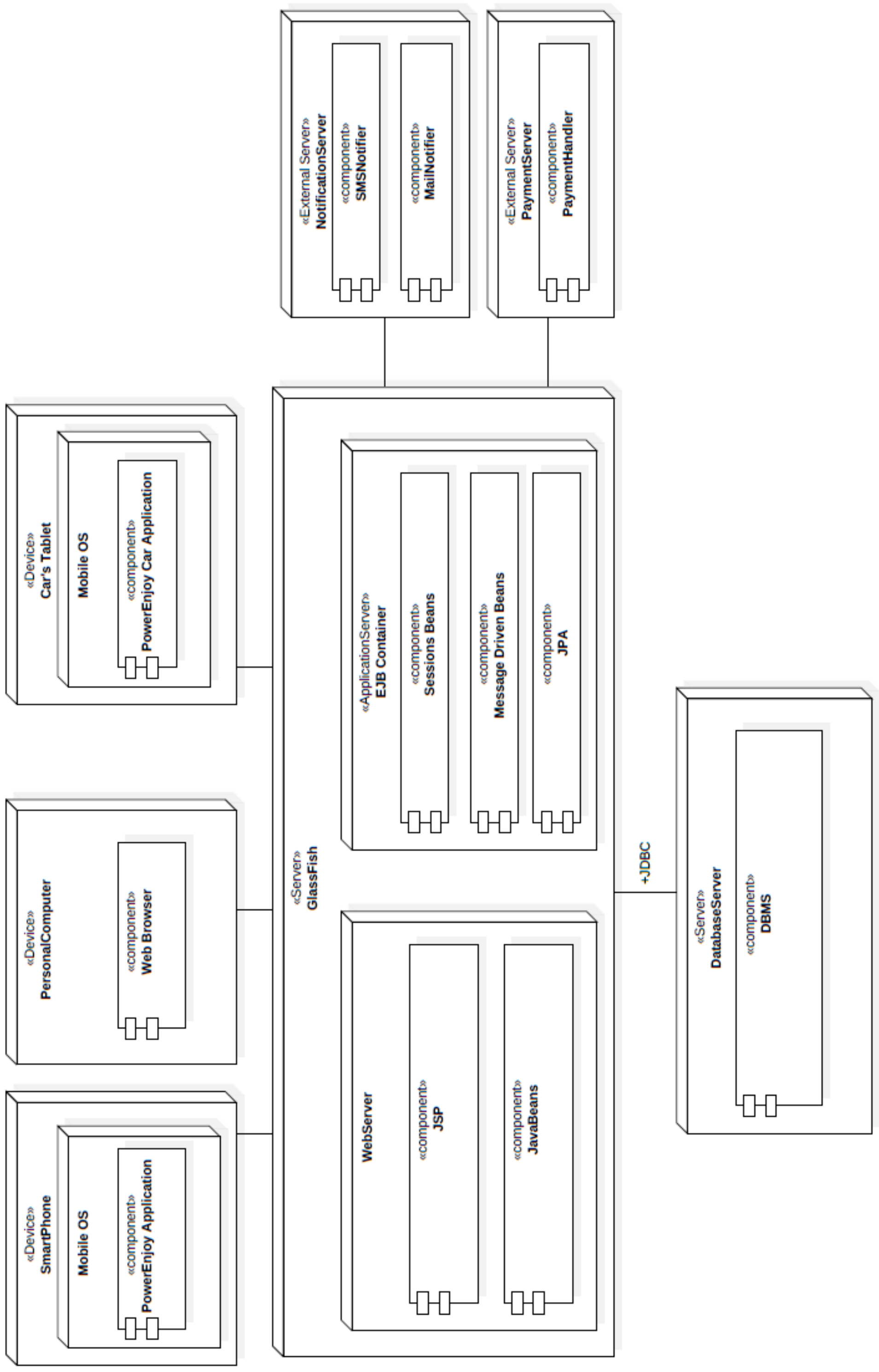


Figure 5 Data Structure of the system

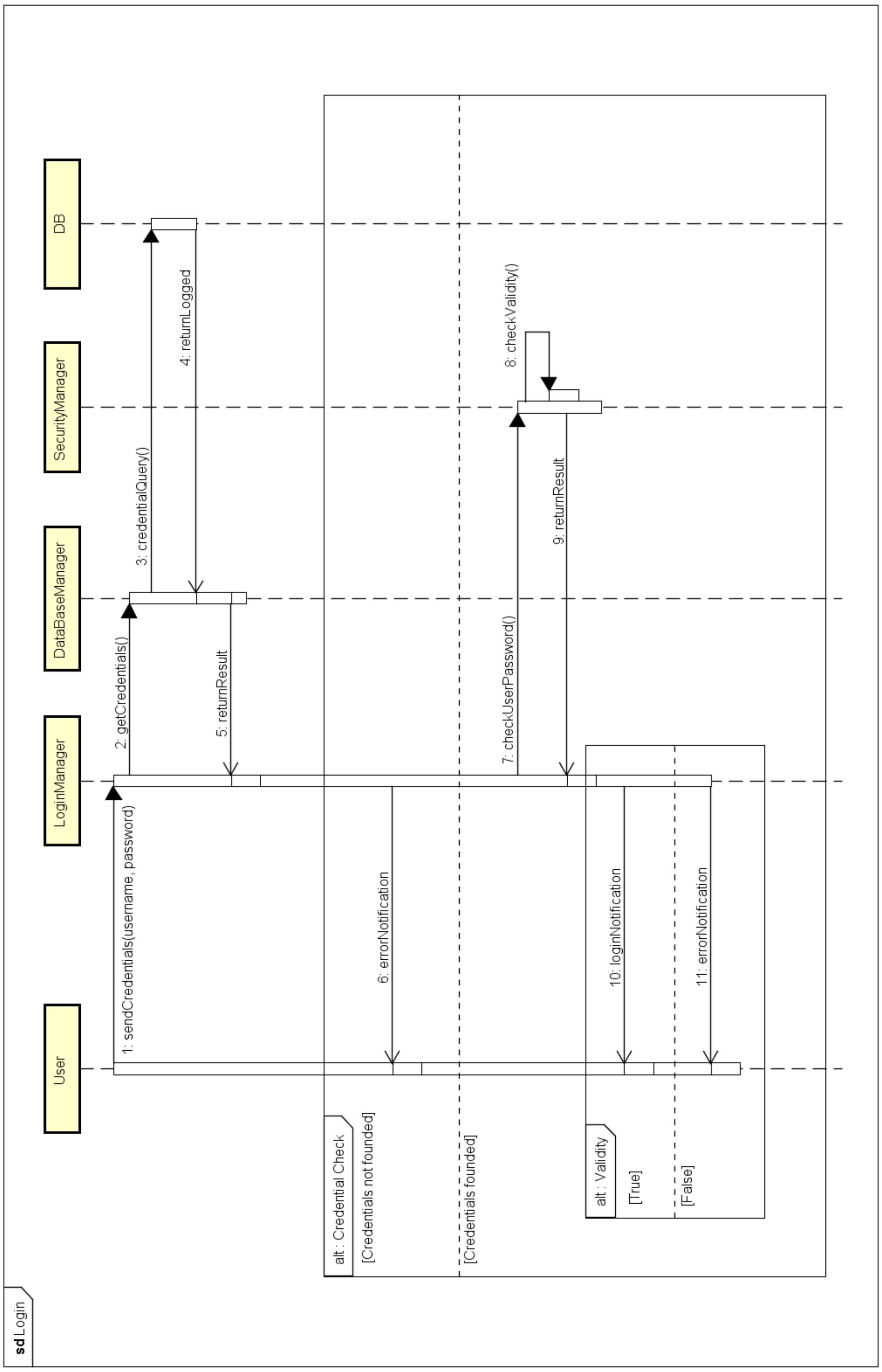
2.3 Deployment view

In order to offer a deployment view, we have decided to use the Deployment Diagram. This diagram is strictly related to the Component Diagram and it shows how software components, previously described, are deployed with respect to the hardware.

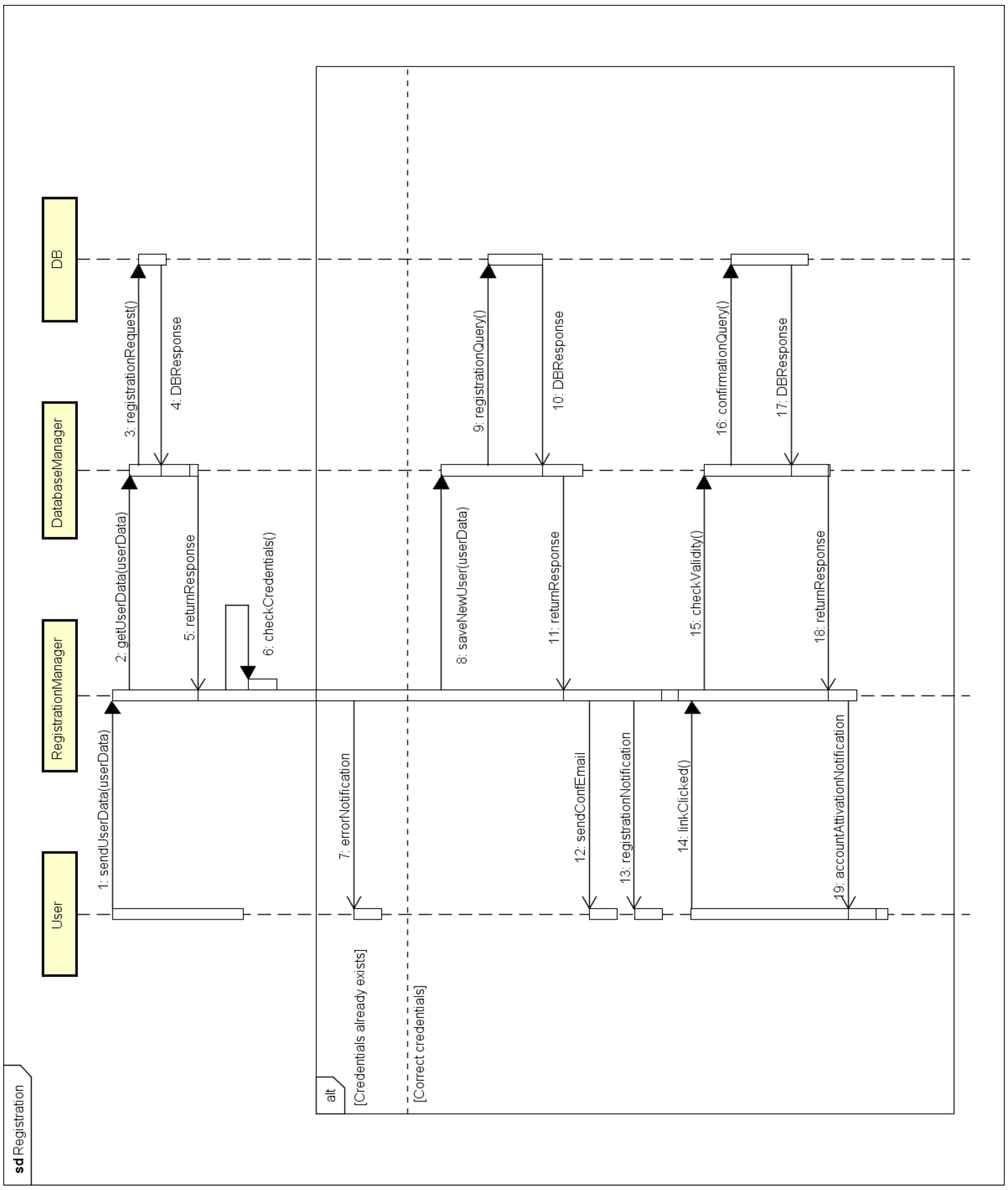


2.4 RUNTIME VIEW

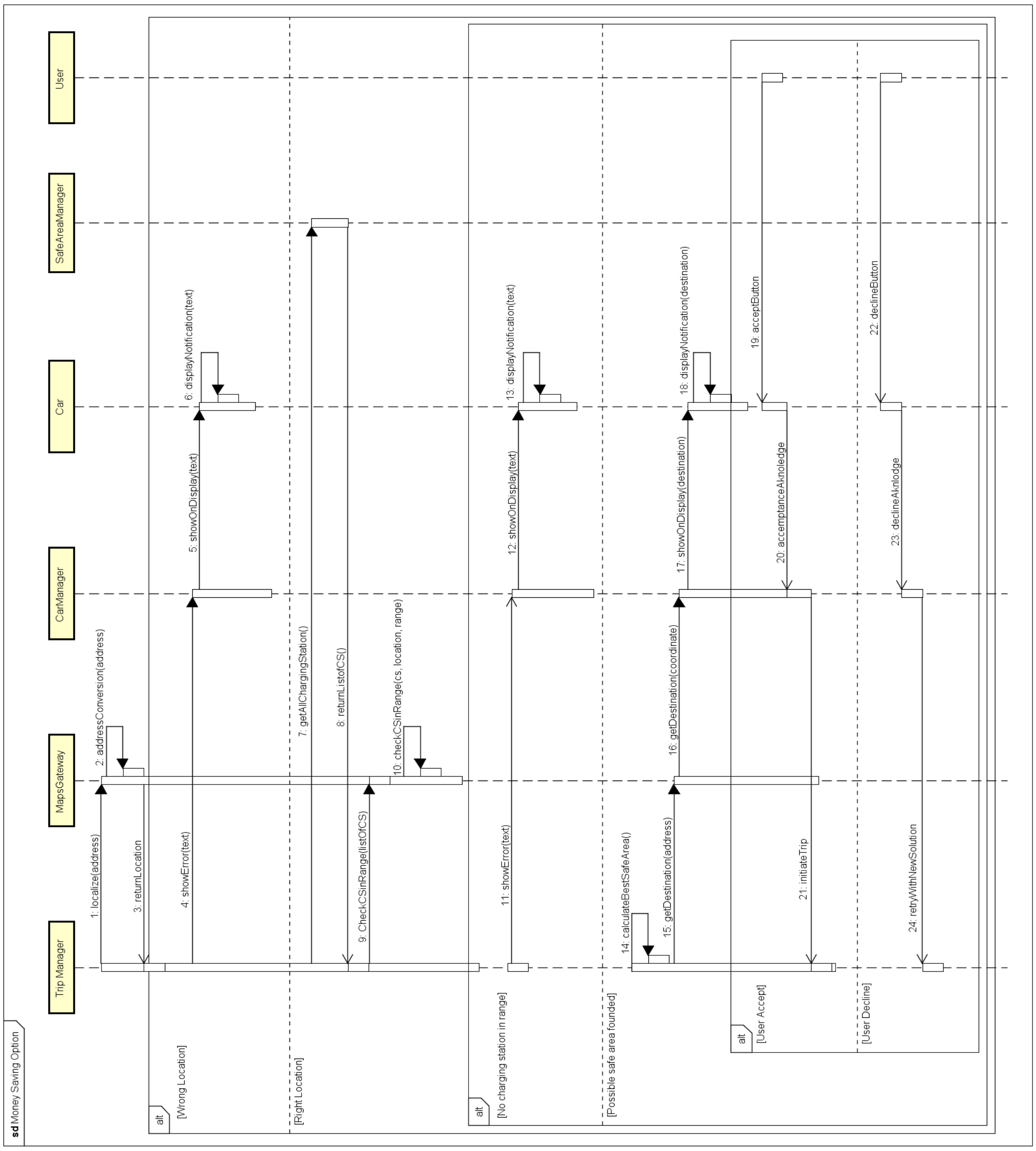
2.4.1 Login



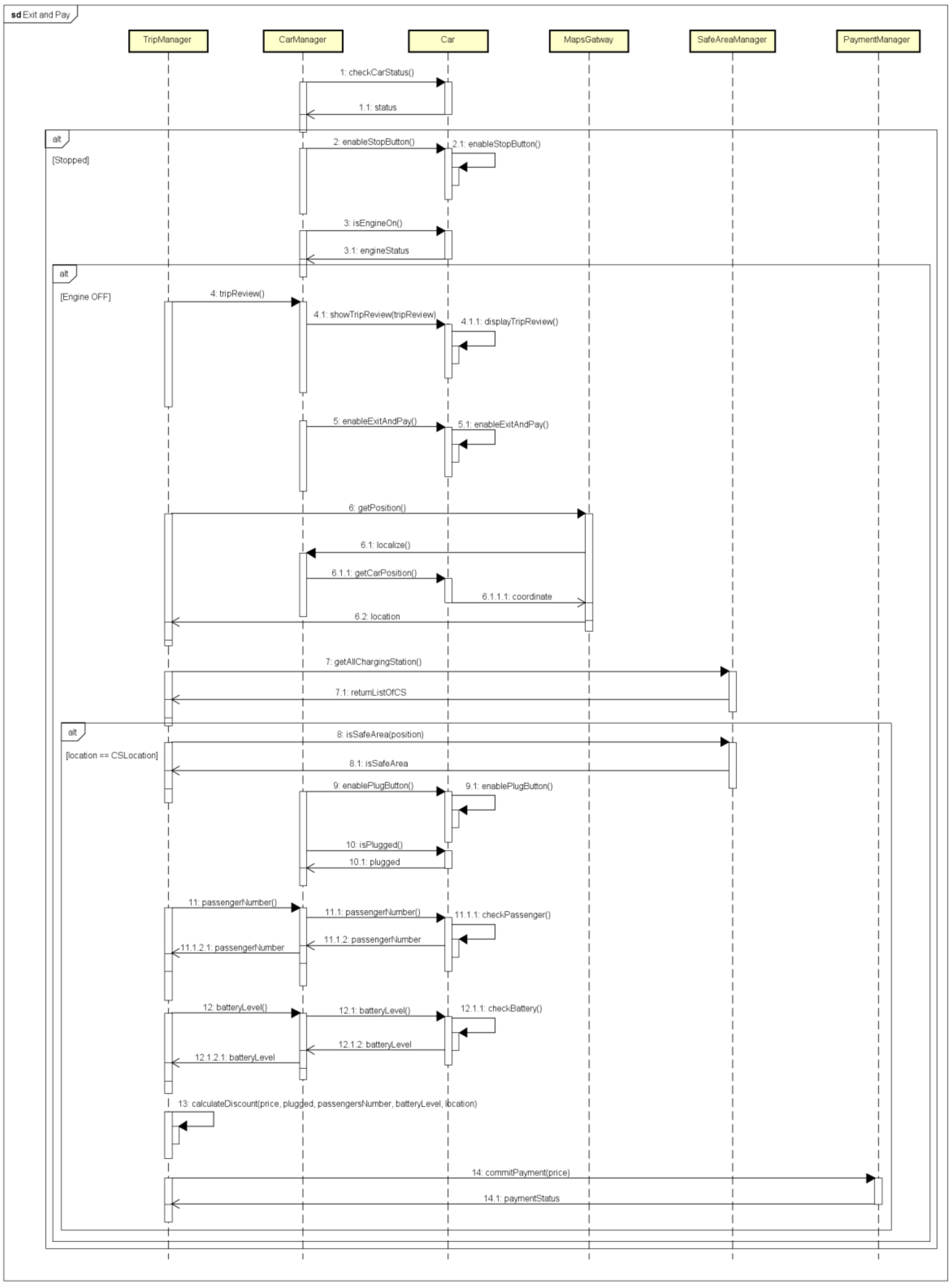
2.4.2 Registration



2.4.3 Money Saving Option



2.4.4 Exit and Pay



2.5 COMPONENT INTERFACES

In this section, we will describe and discuss the interfaces between different components in our system, explaining what each component provides and what it requires.

- **LoginManager**
  - Provides: method to log in users.
  - Requires: access control method to valuate credentials
- **RegistrationManager**
  - Provides: method to register guests.
  - Requires: notification method to send notification once a registration is successfully concluded.
- **SecurityManager**
  - Provides: access control method to valuate log in credentials
- **NotificationManager**
  - Provides: method to send notification, such as email or sms
  - Requires: an external system which effectively send notification
- **UserActionsManager**
  - Requires: methods to manage the profile, one to make a reservation and one to start a trip.
- **ProfileManagement**
  - Provides: methods to manage a profile
- **PaymentManager**
  - Provides: methods to manage, control and handle payment. Calculate discount and fees.
  - Requires: an external system that concretely conclude the payment
- **PositionManager**
  - Provides: map with current position of the car, and its coordinates. It offers also method to calculate a path and another method to detect if the car is stopped in a safe area or not.
  - Requires: an external system that calculate cars' position and pathway.
- **TripManager**
  - Provides: method to start and stop the trip and to inform users about its state.
  - Requires: methods provided by PositionManager for the path and its final position and methods from PaymentManager to apply discount and final charge.
- **ReservationManager**
  - Provides: methods to make a reservation, decline it or decide whether a fee must be charged or not.

    - Requires: methods offered by PaymentManager to handle fees and by PositionManager to inform user about car’s starting position.
- **SafeAreasManager**
    - Requires: method provided by PositionManager to define an area and detect if it’s safe or not.
- **DataBaseManager**
    - Provides: information to all other components in the system.
    - Requires: methods from an external database system to update, delete or add data in the database.
- **CarManager**
    - Requires: information taken by the car and methods offered by PositionManager.

2.6 SELECTED ARCHITECTURAL STYLES AND PATTERNS

2.6.1 Selected Architecture

As we said before on the document the architecture of our system is divided into three tiers:

1. **Client tier**: where there are the users of the system with the application and the web browser and the cars of the system with their application;
2. **Server tier**: that is composed by both the Application Server and the Web Server;
3. **Database tier**: where is hosted the database system of our application;

2.6.2 ModelViewController (MVC)

Our user interface is based on the ModelViewController that is a software design pattern which divides a given software application into three interconnected parts, to separate internal representations of information from the ways that information is presented to or accepted from the user. Furthermore, also the complexity of the application is divided into these components since rather than having one that must manage all these aspects, we may have multiple components that performed different actions such as the business logic of the system (servlet), the information of the system (EJB component) or the presentation of these information (JSP). The schema of this design pattern is shown below:

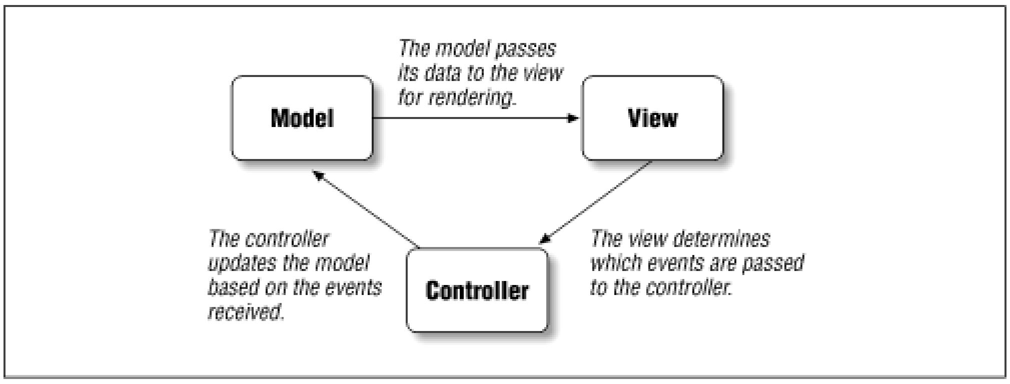

The model passes its data to the view for rendering.
Model
View
The controller updates the model based on the events received.
Controller
The view determines which events are passed to the controller.

Figure 6 MVC Pattern Schema

2.6.3 Pattern State

We decide to use this design pattern to manage the behaviour of some entities such as the users or the cars of the system. In fact, these entities have a status which allow them to perform different actions, if we consider for example the user we know that he can be in a “DRIVING” status and so the only possible action is to stop the trip; or for example if the user is in a “LOGGED” status he can reserve a car. In the same way, a car is associated to a car state and so for example if it is in the state “RESERVED” first the colour of the car in the map will be different from another one of state “AVAILABLE” and furthermore this car cannot be reserved since it is already reserved by a previous user. Also, the trip and the reservation have different state, the first one for example has no final destination and final position while it is in a “RUNNING” state, on the other hand if the trip is “COMPLETE” then it will have final destination and position and also it could be associated to some discounts. In the end, also the reservation can have a state as for example a reservation that is in a “WAITING” status will have a counter that consider the time that is passing in order to check if it last for more than one hour and so it must charge the user associated with a fee. The schema of this pattern is shown below:

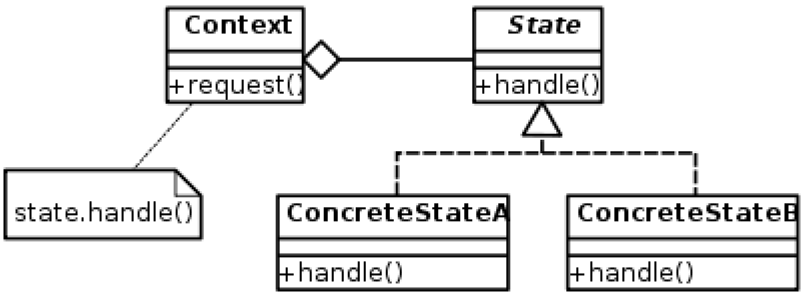


Figure 7 Pattern State Schema

2.6.4 Pattern Strategy

We decide to use the pattern strategy for our system in order to have flexibility of our application in the future implementation. In fact, we do not give a unique implementation of the algorithm used to calculate the discounts of a trip since these could change during the time (as for example new promotion or special discount could be applied in the future). So, we decide to have an interface of the algorithm that can be modified over the time and so we can change it without modifying the architecture or the implementation of the classes of the system. The first implementation of this algorithm will satisfy all the constraints express in the assignment document related to the discounts. The pattern schema is shown below:

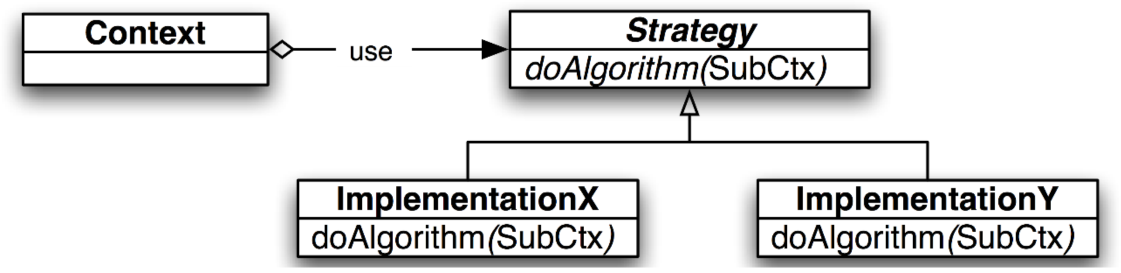


Figure 8 Patter Strategy Schema

2.6.5 Client-Server

The application is strongly based on a Client-Server communication model. We use this type of design for the communication between the client (both on the web browser and the application) and the application server and also in the communication and the exchange of the information between the application server and the cars of the system.

2.6.6 Gateway Pattern

We use this kind of pattern in our system to abstract from the complexity of interfacing an external system such as the payment system, the notification system and the maps system using some classes that are defined as gateway (as shown in the component diagram). These classes provide some methods that the system can use without considering the complexity of the external system because they already manage this interaction and offer some simple methods that can be used. The schema of this design pattern is shown in the figure below:

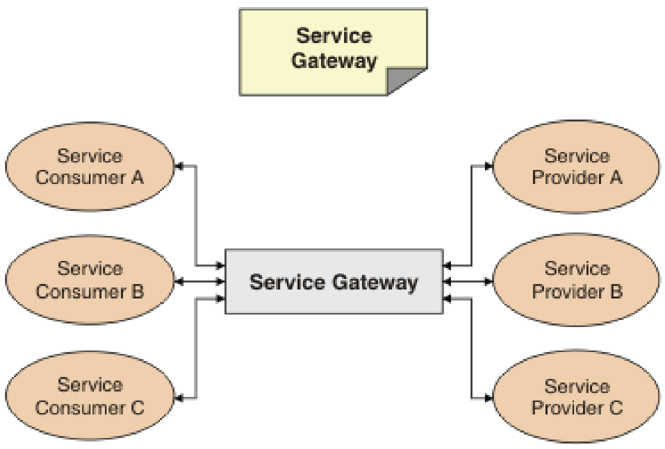


Figure 9 Pattern Gateway Schema

3 ALGORITHM DESIGN

3.1 ALGORITHM 1

3.1.1 Code

```
import java.util.ArrayList;

public class MoneySavingOption{
 public Location MoneySavingOption(String address, int range){

   CarManager carManager = new CarManager();
   Location location = new Location();
   location = MapsGateway.localize(address);
   ChargingStation bestChargingStation = new ChargingStation();

   ArrayList<ChargingStation> chargingStation = new
ArrayList<ChargingStation>();
   ArrayList<ChargingStation> chargingStationinRange = new
ArrayList<ChargingStation>();

   if (location.isEmpty()){
     carManager.showError("We cannot find your destination, retry!");
     return null;
   }

   if (checkRange(range)){
     carManager.showError("Your range is invalid, please type a new one");
     return null;
   }

   chargingStation = SafeAreaManager.getAllCS();

   for(ChargingStation cs : chargingStation){
     if (MapsGateway.checkCSinRange(cs, location, range)){
      chargingStationinRange.add(cs);
     }
   }

   if (chargingStationinRange.isEmpty()){
     carManager.showError("No charging station nearby your destination");
     return null;
   }

   for(ChargingStation cs : chargingStationinRange){
     if ((cs.getNumberPluggedCar()) <
(bestChargingStation.getNumberPluggedCar())){
      bestChargingStation = cs;
     }
   }
   return bestChargingStation.getLocation();
 }
}
```

3.1.2 Description

This first algorithm manages the “Money Saving” option in our system. We created a specific class for the location, to provide coherent and consistent information about car’s and charging station’s position. So, this function receives an address, the one that user has typed on the car’s display, and a range, also typed on display. CarManager plays an important role during this situation, because permits to interact with the display and consequentially with the user in the car. We created also a specific class for the charging station, which are provided by SafeAreaManager during the computation. MapsGateway is a third part service and can calculate if a specific charging station is or not in range, from the location desired by the user. In the end, before the user can see the result on display, the system checks which charging station has less car and it advices its position to the user.

3.2 Algorithm 2

3.2.1 Code

```
public boolean ExitAndPay(){

    boolean carStopped = false;
    CarManager carManager = new CarManager();
    boolean tripStatus = false;
    TripManager tripManager = new TripManager();
    ArrayList<ChargingStation> chargingStation = new
ArrayList<ChargingStation>();
    boolean parkingSafeArea = false;
    float price;
    boolean payment = false;
    boolean plugged = false;
    int passengers = 0;
    int batteryLevel = 0;

    carStopped = carManager.checkCarStatus();
    if (carStopped){
        carManager.enableStopButton();
    }

    tripStatus = TripManager.tripStatus();
    if (!tripStatus){
        if(!carStatus.isEngineOn()){
          carManager.showTripReview(tripManager.tripReview()));
          carManager.enableExitandPayButton();
          Location location = new Location();
          location = MapsGateway.localize(carManager.getPosition());

          chargingStation = SafeAreaManager.getAllCS();
          for(ChargingStation cs : chargingStation){
              if (location == cs.getLocation()){
               parkingSafeArea = true;
               carManager.enablePlugButton();
               plugged = carManager.isPlugged();

               PaymentManager paymentManager = new PaymentManager();
               price = tripManager.currentCharge();
               passengers = CarManager.passengers();
               batteryLevel = CarManager.batteryLevel();

               price = TripManager.calculateDiscount(price, plugged, passengers, batteryLevel, location);

               payment = paymentManager.commitPayment(price);
               if (payment){
                 return true;
               }else{
                 CarManager.showError("Payment unsuccesful. Retry!")
               }
              }
          }
        }
    }
  }
}
```

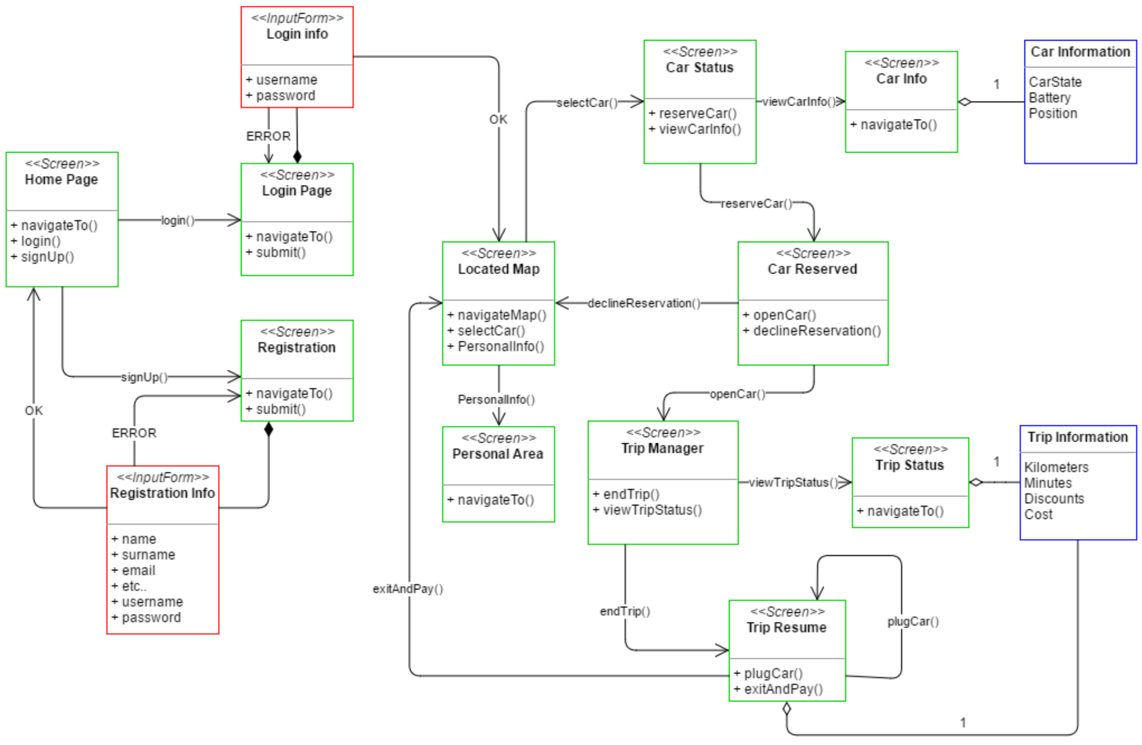
3.2.2 Description

This second algorithm has the aim to explain the conclusion of the rent. This difficult situation has lot of possible alternatives so we should check more details in order to guarantee coherence between the trip and the payment. After variable's declaration, this algorithm begins to check the status of the car (precisely, if it stopped or not) and then if the user asks to stop its trip. As soon as the engine is stopped, the system stops charging the user. Then car's tablet shows the review and the details of the trip, calculated thanks to TripReview. The user can now touch the "Exit and Pay" button on the display, that is enabled coherently after some checks. The algorithm saves current location of the car thanks to CarManager that sends coordinates to MapsGateway, and then it processing received data in order to retrieve a location. The system downloads all the charging station, and this choice is justified by the fact we want to keep always updated the list of active charging station (with this method, system's administrator only should update the central database after, for example, a charging station's braking). If the user stops near a charging station, and then he is in a safe area, CarManager enables the possibility to plug the car as well as enables "plug" button. After TripManager calculates right discount of the trip (considering also passengers and battery level), PaymentManager tries to commit the payment, and the system must check if it successful. If it is unsuccessful, we provide the possibility to retry the payment manually and change some information about the payment's method through car's display. After three attempts, the user will be charged to discourage this event.

4 USER INTERFACE DESIGN

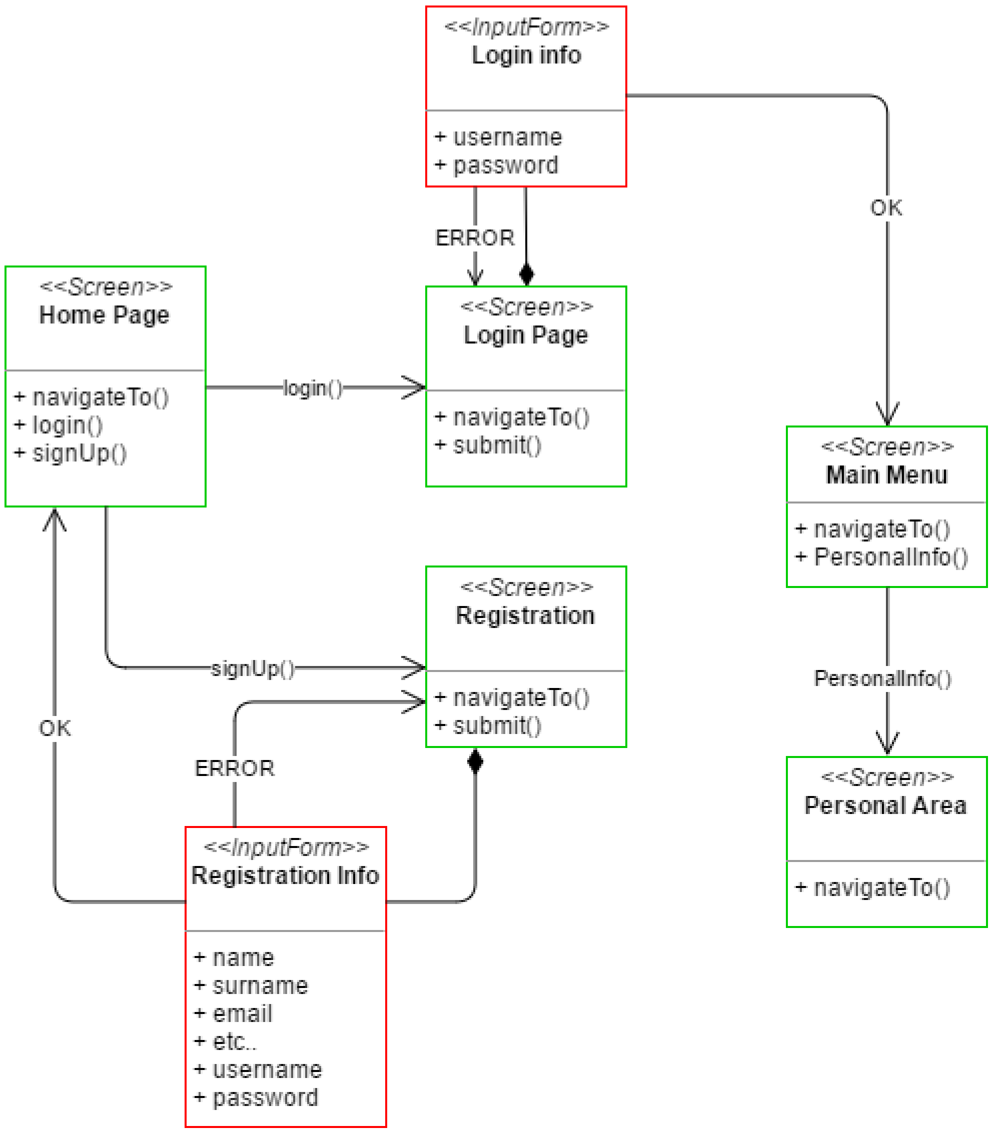
In user interface design section, we will present both mobile application and web browser user experience. We are going to show how user can move from one screen to another, which functionalities are available from each screen and we will briefly describe them.

4.1 UX MOBILE APPLICATION



The diagram above represents user experience with mobile application. User can sign up or login from Home Page, interacting with two different input forms according to what has been selected. Then he/she is redirected to Located Map, which is the main screen of our application. Here user can check personal information, navigate the map looking for available car near his/her position and eventually select it. Since user selected a car, it's possible to view car information such as battery and position, and even reserve that car. User is then redirected to Car Reserved screen: here he/she can decline the reservation, and that will bring him/her back to Located Map screen; while if user is near the reserved car, it's possible to open it. By choosing this option, all information about driver and location are transferred to the tablet inside the car. So in mobile's screen is not possible to do anything else, regarding our application, until trip will be concluded. Now from car's screen user can instantly check trip status, that reports information such as kilometres, minutes, eventual discount and total cost, or end the trip. By clicking this button, a trip review is showed on the screen: if the user is near a charging station and he/she feels like to plug the car to get some extra discount before concluding the trip, he/she is requested to press "plug the car" button so the system can better detect this action and elaborate new total cost according to new discount. If user just wants to exit the car, by pressing related button he/she officially ends the trip, and again the Located Map screen is shown on mobile device so user is again allowed to reach other cars.

4.2 UX Web Browser



As show in the picture above, user experience with web browser is much more limited. That’s because we decide to just let user essentially sign in or sign up through the web. User can sign up to service by compiling Registration info input form, and if data are correct he/she is redirected to Home Page from where he/she is allowed so sign in with new credentials. Once user logged in, he/she can check personal information.

4.3 Scenario example

Simone is a new user. From home page he clicks on sign up button, he inserts all requested personal information and, since it’s correct, he is redirected to home page. Now he can log in into the system using his freshly created credentials. He wants to find a car near his house, so navigating the map he can find one just in the street behind his position. He reserves the car and reaches it in few minutes: he opens it through “open the car” button placed in mobile screen, enters the car and starts his trip on the way for his girlfriend’s house. When Simone is pretty near the destination, he finds a charging station, so he parks car there, ends the trip, selects button to plug the car so he can get extra discount, he physically plugs in the car, then he finishes his trip closing the car.

5 REQUIREMENTS TRACEABILITY

In this section we want to match RASD functional requirements with their components in the DD.

#Requirement in RASD	Requirement	Component in DD
3.2.1	Sign up	RegistrationManager DatabaseManager
3.2.2	Login	LoginManager, DatabaseManager, SecurityManager
3.2.3	User Account Management	UserActionsManager, ProfileManagement
3.2.4	Research Car	UserActionsManager, ReservationManager and CarManager
3.2.5	Select car, view car status, reserve car	UserActionsManager, ReservationManager, CarManager, PositionManager
3.2.6	Delete reservation	ReservationManager
3.2.7	Unlock the car	UserActionsManager, TripManager
3.2.8	View charge during trip	TripManager
3.2.9	Enable “Money Saving” option	TripManager, PositionManager, CarManager
3.2.10	Plug-in the car to get discount	CarManager, TripManager
3.2.11	Visualize “trip review”	TripManager, PaymentManager
3.2.12	Conclude the rent and pay	TripManager, PaymentManager, NotificationManager, CarManager, PositionManager

6 EFFORT SPENT

We report approximately how many hours each member has worked on this document.

- Simone Bruzzechesse: 30 hours
- Luca Franceschetti: 30 hours
- Gian Giacomo Gatti: 30 hours

7 References

7.1 Tools used

- **Network Diagram Maker**: for the architecture overview;
- **Visual Paradigm Modeler**: for the component diagram;
- **StarUML**: for the data structure of the system and the deployment diagram;
- **Astah**: for UML diagrams