POLITECNICO
MILANO 1863

PowerEnJoy

Design Document

Luca Scannapieco - 877145
Andrea Pasquali - 808733
Emanuele Torelli - 876210

11/12/2016

Table of contents

1. Introduction

1.1. Purpose

The purpose of the Design Document is to provide more technical descriptions about the PowerEnjoy system, in particular the aim is to identify:

- The high-level architecture
- The structure of the software in the server
- The structure of the data in the database
- The physical distribution of the system
- The runtime behaviour
- The structure of the interfaces and the interaction among them
- The used design patterns

1.2. Scope

The system allows users to reserve a car via mobile app or via web app, using GPS to identify the position of the user and the position of the available cars around the city.

Obviously someone has to be successfully registered to the service before taking benefits from the service, so that the society can collect the information about who's driving the cars.

Users can reserve a car for up to one hour, if the reservation expires the user has to pay a fine.

Users can drive a car everywhere but they must park within safe areas accurately defined by the company.

The system calculates some eventual discounts to apply to the total fee of a ride, for example if a user shares the car with at least two other people or if he leaves the car with the battery charged or charging in a power station at the end of a ride.

An employee of the company, called the assistance coordinator, has also access to the service in order to exploit some tasks forbidden to the users.

The society has to build the whole system from scratch, so we don't have to deal with the problem of make the components interact with old pre-existent components.

1.3. Definitions, acronyms, abbreviations

- DD: this Design Document
- RASD: the Requirement Analysis and Specification Document provided before
- Client-server: the client-server model is a distributed application structure that partitions tasks between the providers of a resource or service, called servers, and service requesters, called clients
- MVC: model-view-controller, is a software design pattern for implementing user interfaces on a system.
- Three-tier architecture: it is a client-server architecture in which presentation, application processing, and data management functions are physically separated. In our system the three tiers are strongly related to the corresponding layers (DAL, BLL and presentation), defined in the next three definitions of this section.
- Data access layer (DAL): it is a layer which provides simplified access to data stored in a storage, for example a database.

- Business logic layer (BLL): the part of the application that encodes the logic and the rules that determine how data can be created, displayed, stored, and changed.
- Presentation layer: the top-most level of the application, containing the various user interfaces.
- TCP/IP: Internet protocol suite, it is the conceptual model and set of communications protocols used on the Internet and similar computer networks
- UX: user experience diagram
- BCE: business control entity

Further definitions can be found in the Glossary (section 1.5.) of the RASD.

1.4. Document structure

- Introduction: this section introduces the design document, defining its aims and structure.
- Architectural design: this section gives some information about the architecture of the system, and is divided into different parts:
    - Overview: this section gives an overview of the architecture and presents the division in tiers of the application.
    - High level components and their interaction: this sections gives a global view of the components of the system and how they communicate.
    - Component view for server: this section explains the structure of the server and how its software works.
    - Database structure: this section explains with a class diagram how the data of the application is structured.
    - Deployment view: this section shows how the components are physically deployed in the service.
    - Runtime view: in this section there are some sequence diagrams that show some of the main processes and tasks of the application and the interaction among the components in the system.
    - Component interfaces: the interfaces among the components and the methods provided by them are presented in this section.
    - Selected architectural styles, patterns and design decisions: this section shows a recap of the choices we made defining the architectures of the application.
- Algorithm Design: this section describes some of the most critical processes via pseudocode.
- User Interface Design: in this section are reported some UX and BCE diagrams to explain the user experience of the application.
- Requirements Traceability: this section aims to explain how the goals defined in the RASD are linked to the elements presented in the component diagrams.
- Other info: this section contains information like reference documents, used tools for the creation of the DD, effort spent and the changes made in the document.

2. Architectural design

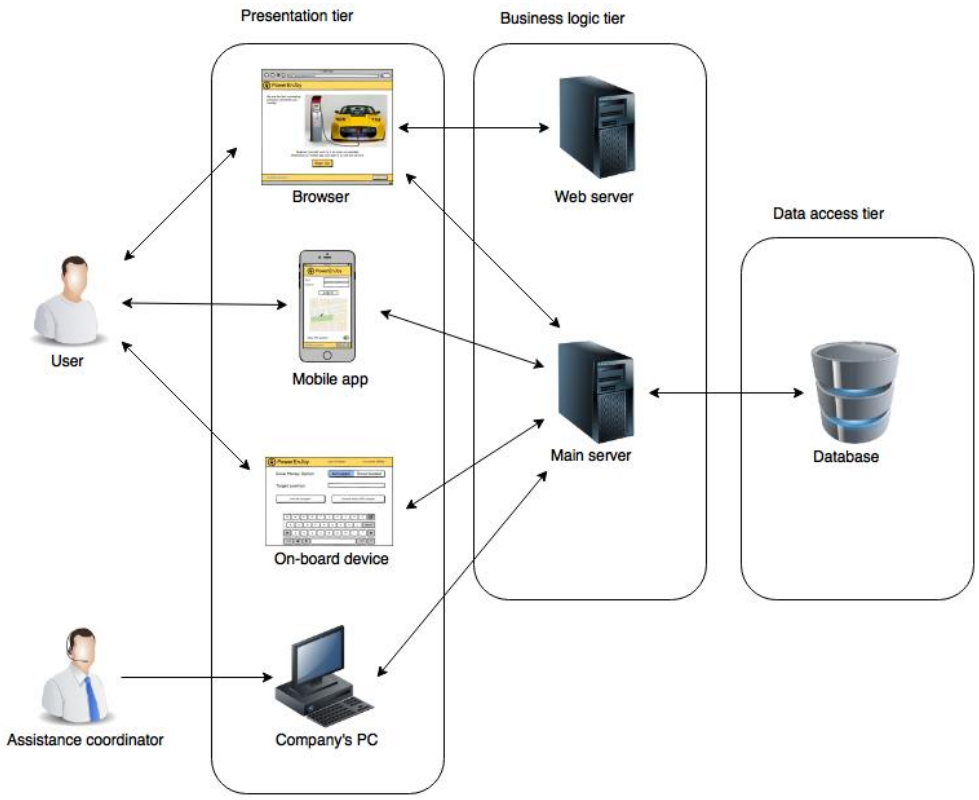
2.1. Overview

The PowerEnJoy service is implemented as a common client-server application, in which we can identify three logic layers:

1. Database (DAL: data access layer)
2. Application logic (BLL: business logic layer)
3. Thin client (a simple and easy interface to the BLL)

Each of these logic services is placed in the corresponding physic tier, the result is that we adopted a three-tier architecture. In the continuation of the document we refer to the tiers when we want to highlight the separation of the physical components in the system.

We provide an architectural overview schema to understand better the structure of the PowerEnJoy service:



The user interface has two different implementations, one is constituted by a web app that can be executed on a modern browser, and the other one is the PowerEnJoy mobile application. Furthermore, during a ride, there is another component with which the user interacts: the on-board device of the car. On the other hand, the assistance coordinator has an ad hoc interface to perform his work, because most of the coordinator's tasks are forbidden to the common users.

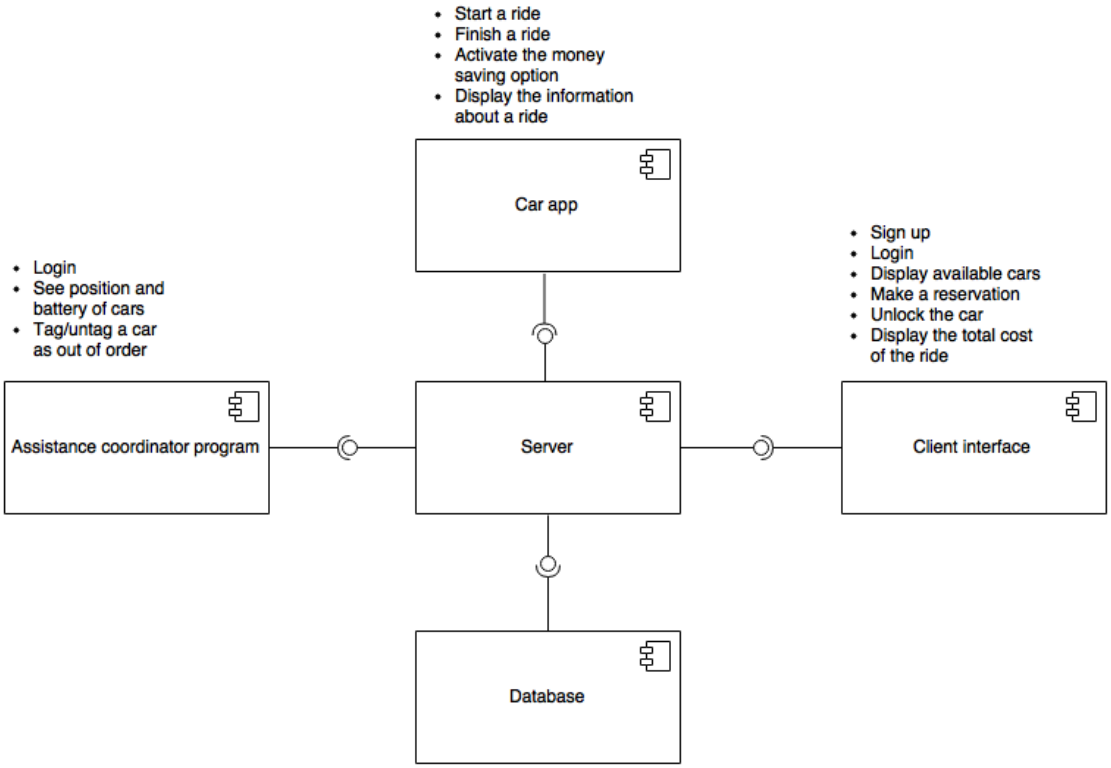
The core of our architecture is the main server, which provides the logic necessary to the system to run properly. In addition to the central server we need an external web server to run the PowerEnJoy web app. Due to ensure the highest level of security and to lighten the client apps, we decided to maintain the clients as thin as possible.

Separating the layer for the application logic and the database we can ensure a high modularity of the system, and for example the company can decide to move one of the layers (or both) to a cloud service, in fact there are some corporations like Amazon where it would have dedicated cloud servers with load balance for database and other for application logic on demand.

2.2. High level components and their interaction

In the following diagram there are shown the main components of the PowerEnJoy system, introduced in this section at the highest level possible.

For each of the three high level components of the presentation tier (including the client app, the car app and the assistance coordinator program), we list the functionalities that the system must ensure; we derived those functionalities from the use cases diagram presented in the RASD in section 3.2.



2.3. Component view for server

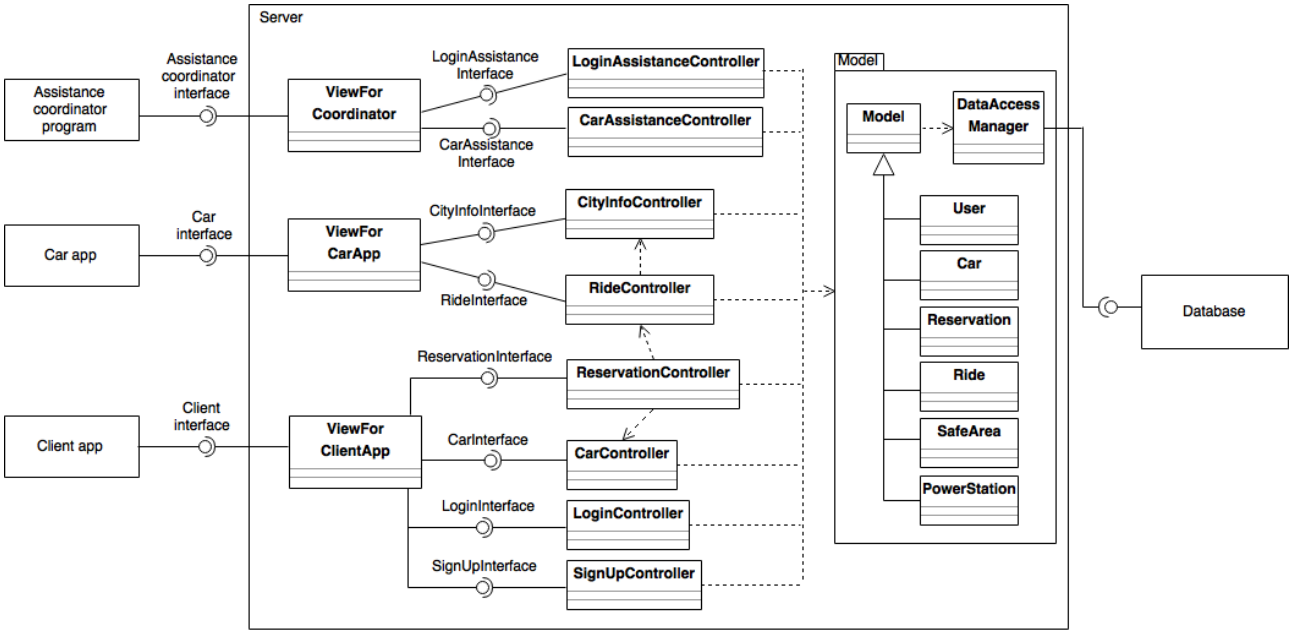
Now we can proceed with a further description of the main server, which is the core of the whole system, by analysing the structure of the software and how it works.

The various functionalities are carried out by multiple controllers, each of them implements the methods provided by the corresponding interface.

Each controller provides an interface to a view, in particular there are three views, one for each high-level component of the presentation layer. The views are important to guarantee the correct forwarding of the requests from the clients, and, thanks to the views, each client can invoke only the methods it needs.

In the server there is also a model that reflects the structure of the data in the database, this structure is described in the next section. In order to retrieve and manage the data required during the various processes, the model has also the logic necessary to query the database, which will be identified by the component called DataAccessManager.

Notice that for the names of the components of the server we will use the camel case notation (e.g. “RideController” instead of “ride controller”), with a remarkable similarity with the Java notation for classes names, in order to emphasize the fact that they are software components. The external components to the server, instead, don’t follow this notation.



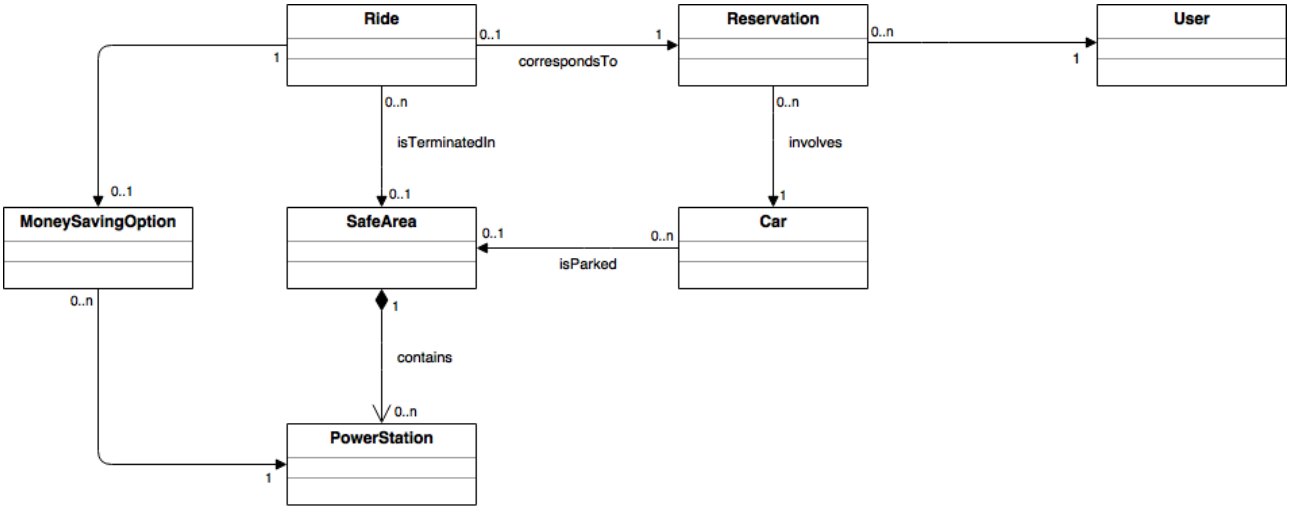
Here is a short explanation of the tasks performed by each controller:

- LoginAssistanceController: checks the info provided by the assistance coordinator during the login phase
- CarAssistenceController: collects all the information of the cars in order to display them to the assistance coordinator
- CityInfoController: collects all the information of the city (e.g. safe areas, power stations)
- RideController: collects and manages the information about rides, and has the power to create or delete rides in the database

- ReservationController: collects and manages information about reservations, and has the power to create or delete reservations in the database
- CarController: collects the information about the cars, changes the availability tag of the cars
- LoginController: checks the info provided by the users during the login procedure
- SignUpController: checks the info provided by the users during the sign up procedure

2.4. Database structure

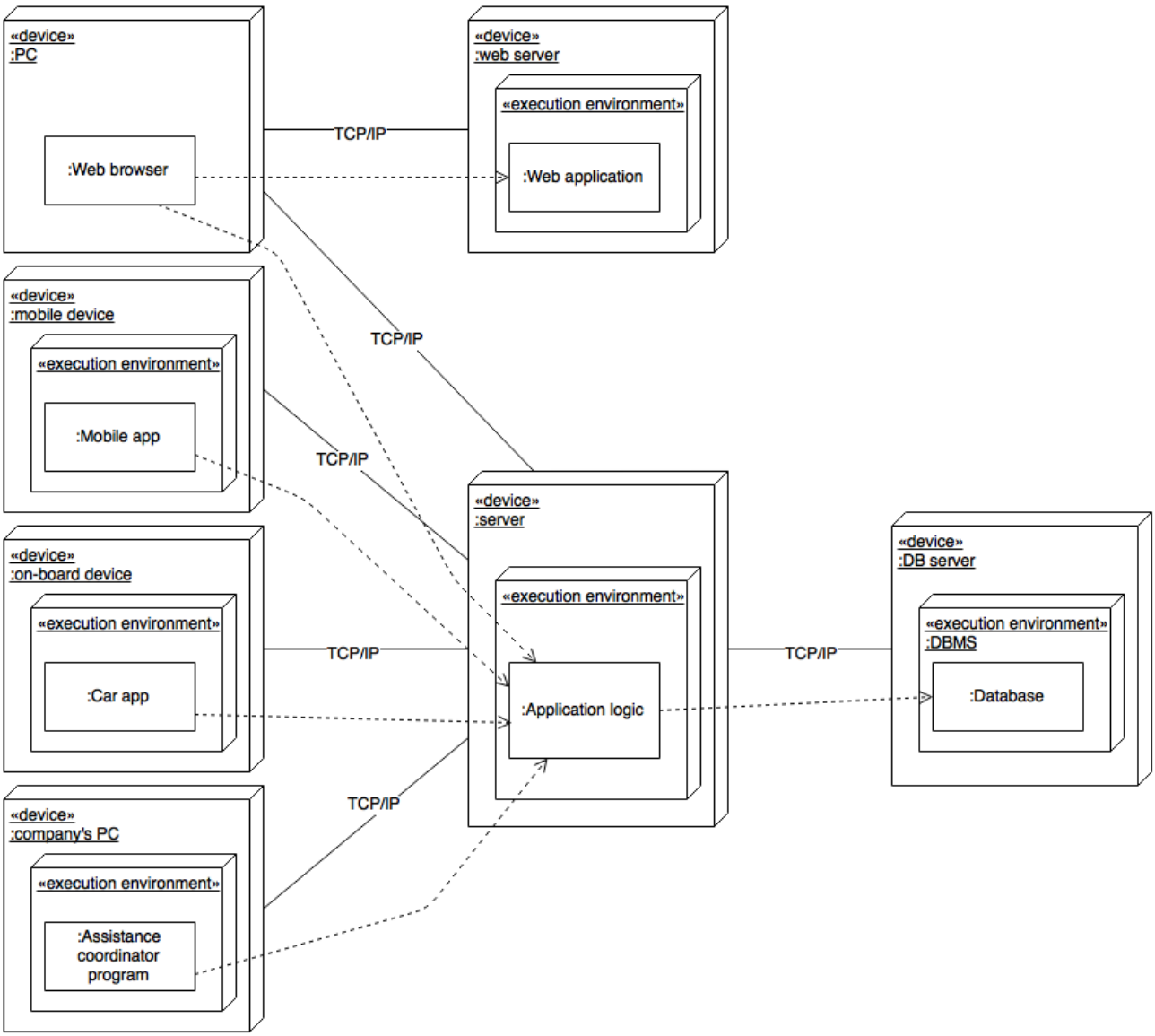
The following class diagram explains the structure of the data the system works with.



2.5. Deployment view

In this section there is a deployment diagram whose aim is to explain the distribution of the several instances of the components of the system. As mentioned before, for the presentation tier we have multiple devices in which the web app can run, as well as multiple mobile devices in which the PowerEnJoy mobile application is installed. There is also an on-board device in every PowerEnJoy car. The company features also a PC on which there is installed the program used by the assistance coordinator to do his tasks. All the components of the presentation tier communicate with the main server and, in addition to that, the web browser needs another server to run the application. The main server communicates with a database to collect and manage all the data required by the system.

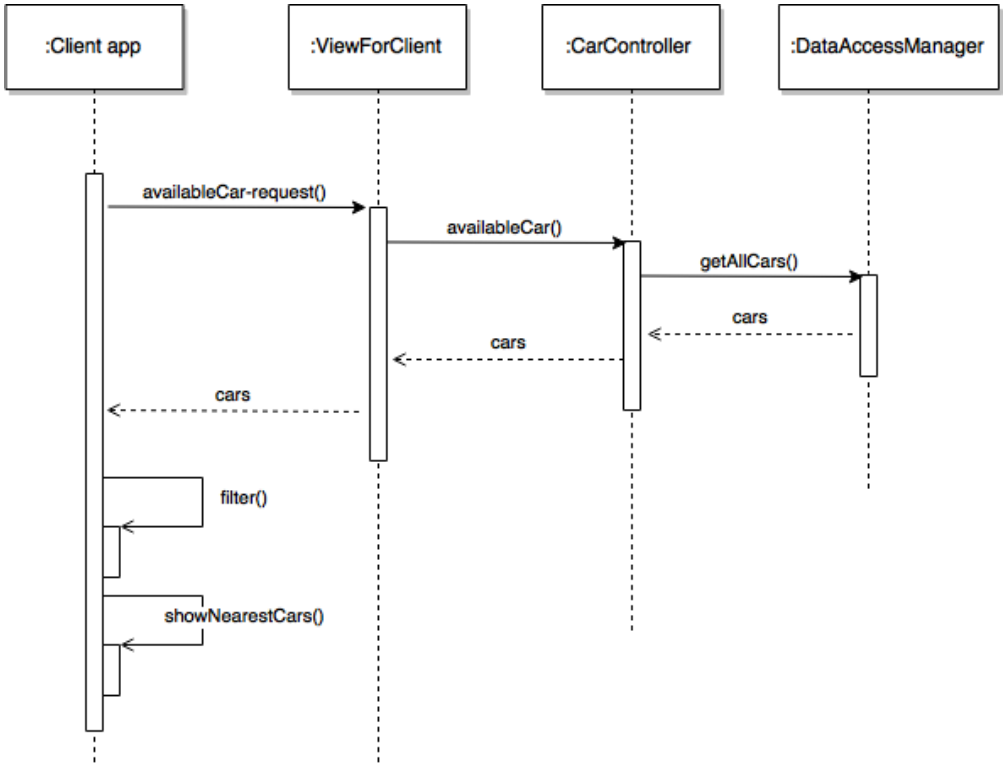
All the components of the system communicate among them by using the TCP/IP protocol.



2.6. Runtime view

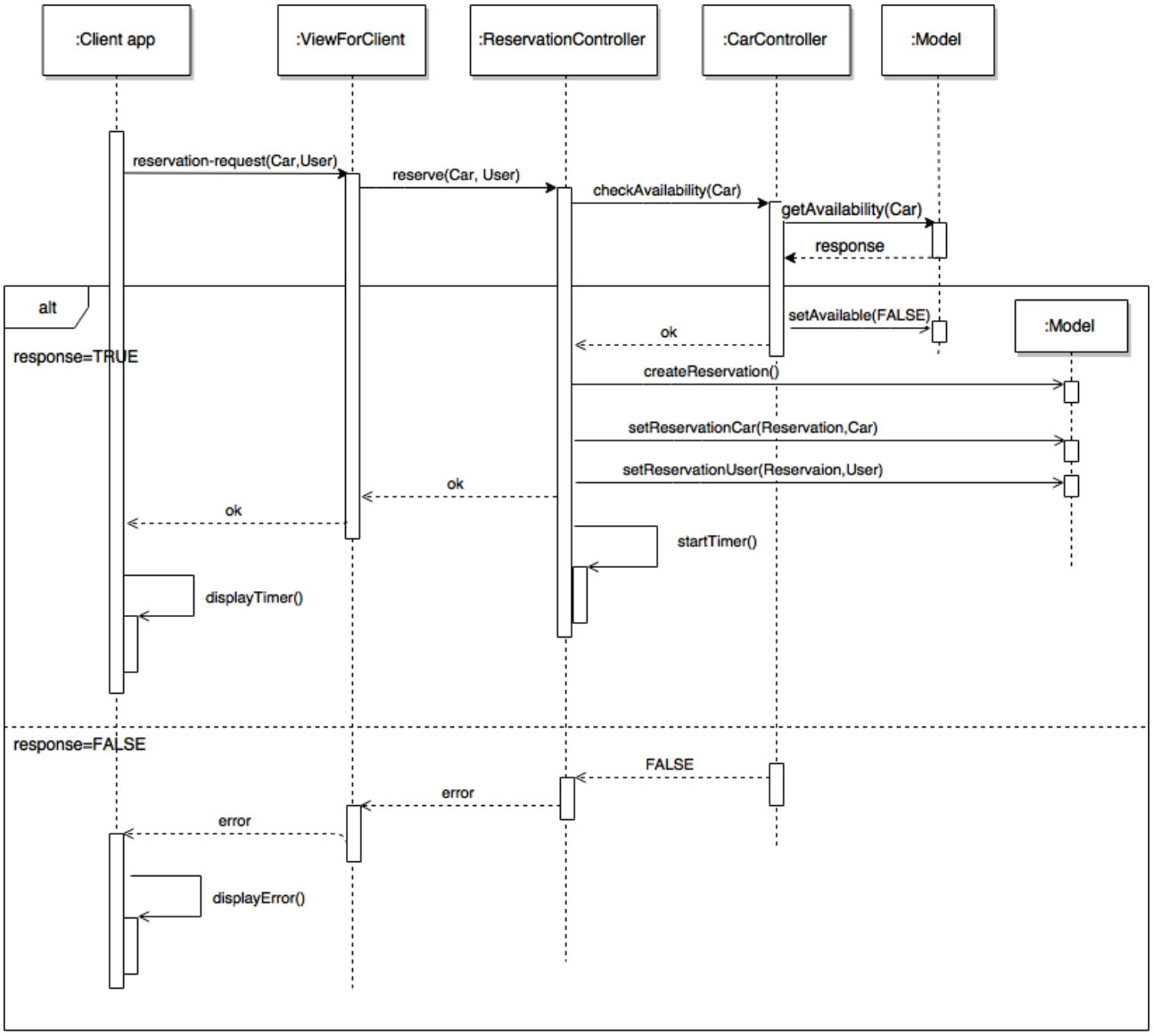
2.6.1. See available cars

The request is handled by the car manager who extracts all the cars from the database and sends them back to the client app. The client app has an internal render that is able to filter the cars both on the user position and on a given position inserted by the user.



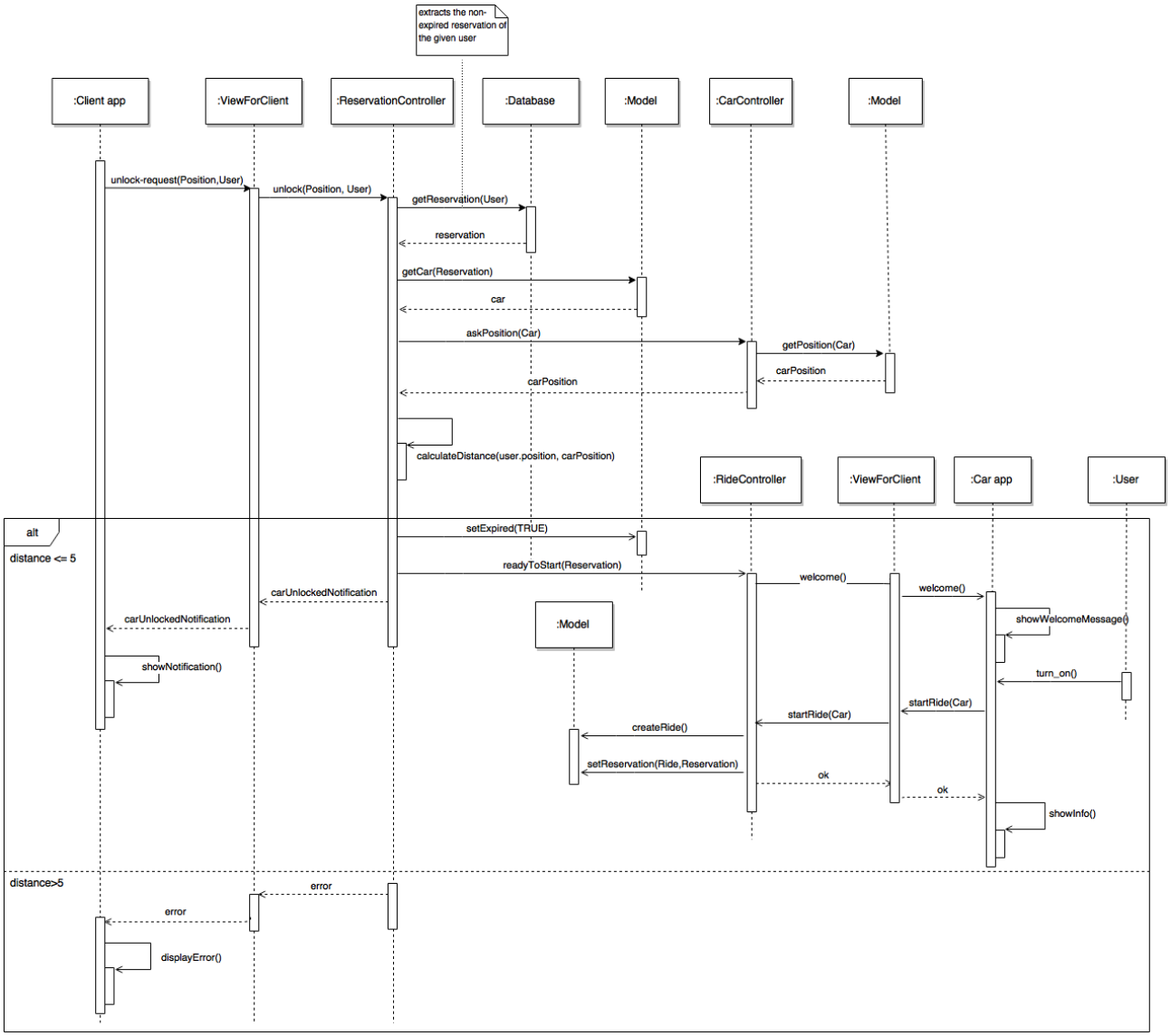
2.6.2. Make a reservation

The client sends the reservation request to the reservation manager through one of the method offered by its view. The resource manager needs to ask to the car manager whether the reserved car is still available: if it is, a new reservation will be instantiated by the reservation manager that will also set the car and the user of the new reservation and start the reservation timer. If the car is no longer available an error will be notified to the client app.



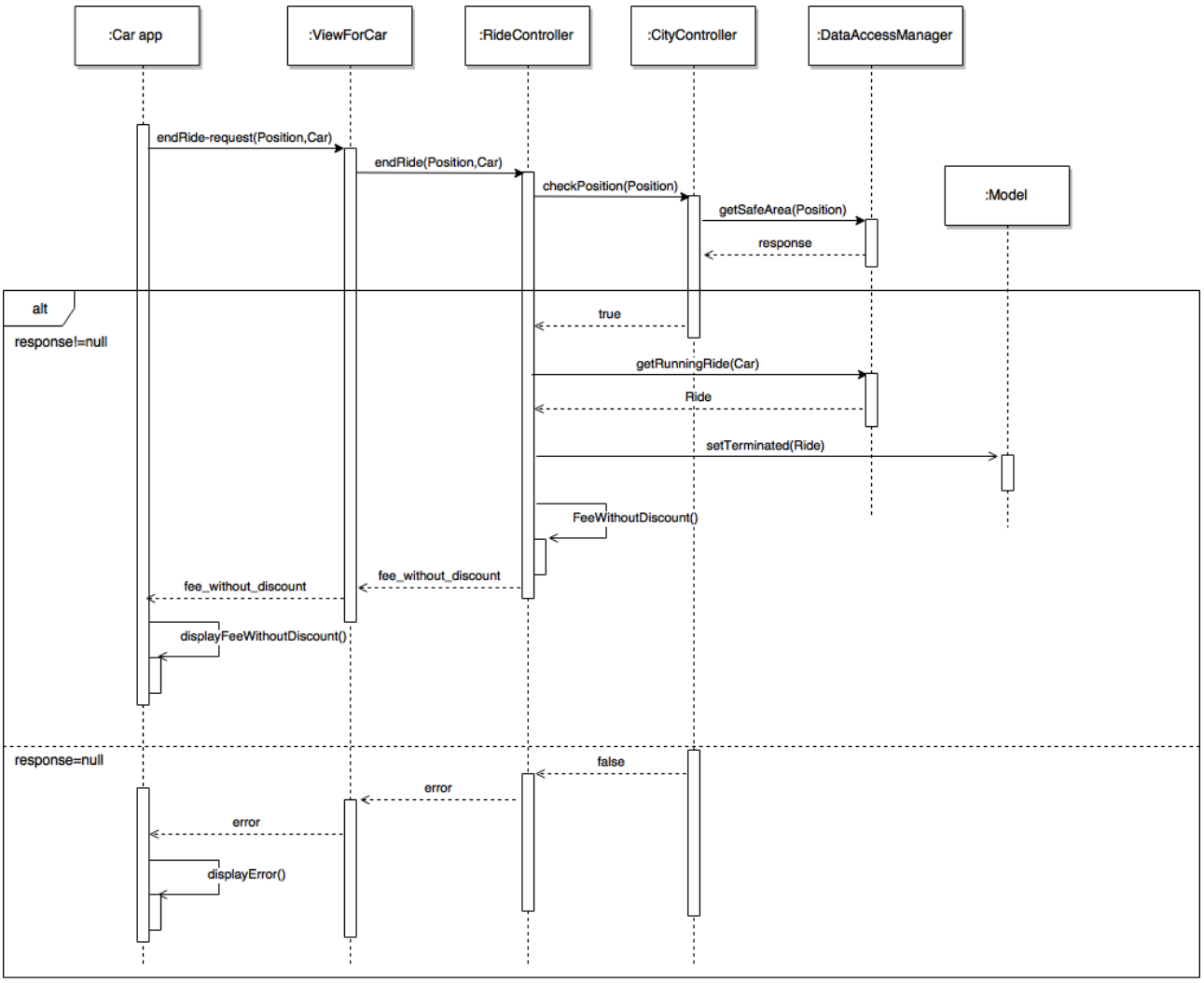
2.6.3. Start a ride

Once the client is at most five meters distant from the car he is reserved, he can send the request to unlock the car to the sever. This request is handled by the reservation manager who checks whether the client is actually close to the car. If this check goes well the reservation manager declares expired the reservation and unlocks the car. Then informs the ride manager that a reservation has just turned into a ride. The ride manager is in charge of instantiating the new ride and “waking-up” the car app on board the interested car. The new ride will be instantiated only once the user will turn on the car. If the reservation manager detects that the user is more than five meters far from the car an error will be notified to the client app.



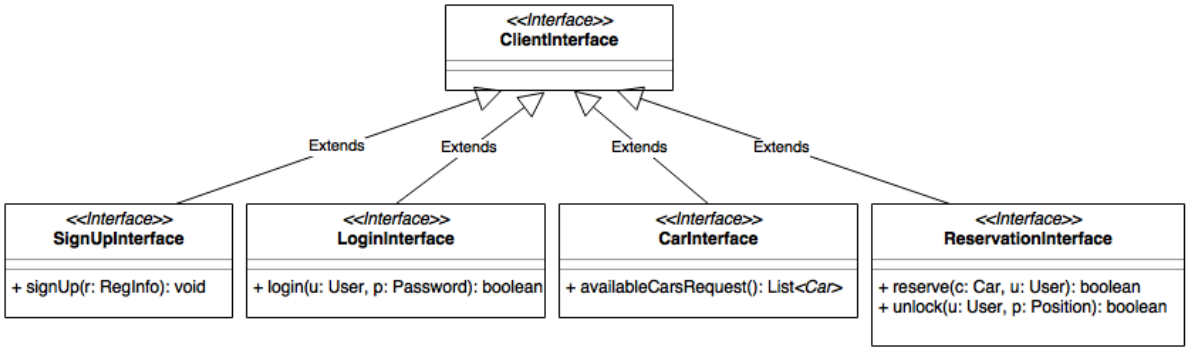
2.6.4. Finish a ride

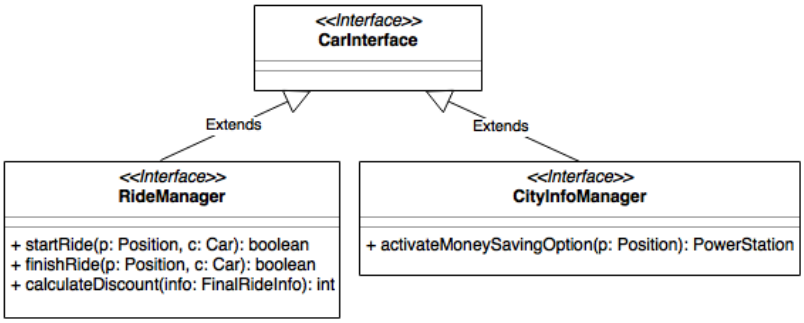
This procedure can start only once the user has turned off the car. He sends the request of terminating the ride through the on-board car app. The ride can terminate only if the car is parked in a safe area, so the ride controller handles the request asking to the city controller whether the car's position is actually in a safe area. The city controller, in turn, looks in the database for a safe area that matches the car's position. If this search goes well, then the ride manager queries the database in order to find the non-terminated ride of the given car and set the "terminated" attribute of this ride to TRUE. Then the ride manager calculates the preliminary fee, i.e. without taking into account eventual fines or discounts (the definitive total fee will be sent to the client app after three minutes from the ride termination), and sends it back to the car app.

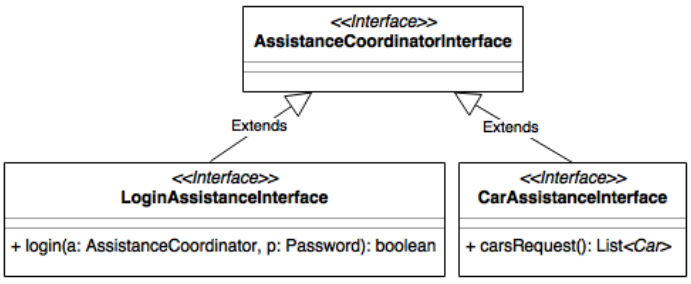


2.7. Component interfaces

In this section we explain the various relationships among the interfaces and there is the list of methods provided by each of them. The various methods are actually implemented by the controllers described in section 2.3.







2.8. Selected architectural styles, patterns and design decisions

In this section there is a recap of our decisions in the choice of the architecture and the pattern we used.

- Client-server: The application is strongly based on a client-server communication model. This approach has been chosen for different reasons:
    - Data synchronization: there is only one application that manages the data.
    - Having one unique server application improves the maintainability of our system.
    - The application is independent from the number of clients connected (it can be scaled up).
    - Improves the security between clients
- Three tiers: our application will be divided into 3 tiers, each one referring to the corresponding layer:
    1. Database (DAL: data access layer)
    2. Application logic (BLL: business logic layer)
    3. Thin client (a simple and easy interface to BLL)
- Thin client: we decided to spoil the client from as much logic as possible, in order to let the PowerEnJoy application run efficiently also in low-resources devices. Reducing the logic in the clients also ensures a more security of the system, because hackers will meet more difficulties to attack the logic if it is on a server instead of on applications.
- MVC: as mentioned before, the software of the server is structured according to the model-view-controller pattern. There are multiple controllers which implement the interfaces described in the previous section, as well as three views, one for client interface, whose purpose is to dispatch every request to the corresponding controller. There is also a model which represents the structure of the data of the database. Having designed one view per interface ensures that the software respects the open-closed principle, e.g. because if, in the future, the company wants to adopt another kind of device interfaced with the system, there is the possibility to write another view and the related controllers, without making changes that affect the pre-existent ones.

3. Algorithm design

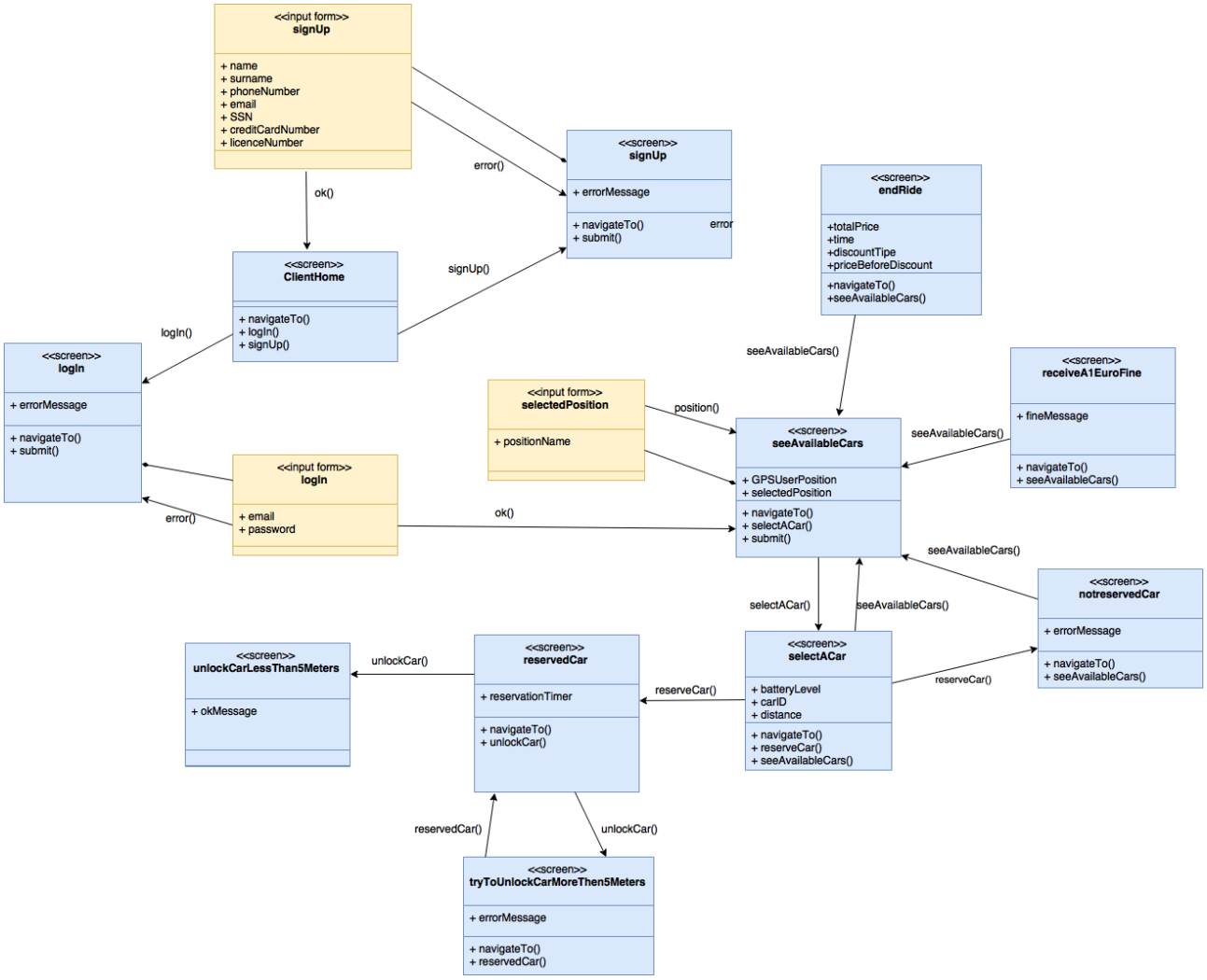
In this section we introduce the most relevant processes of the application using a pseudocode similar to Java.

3.1. Money saving option

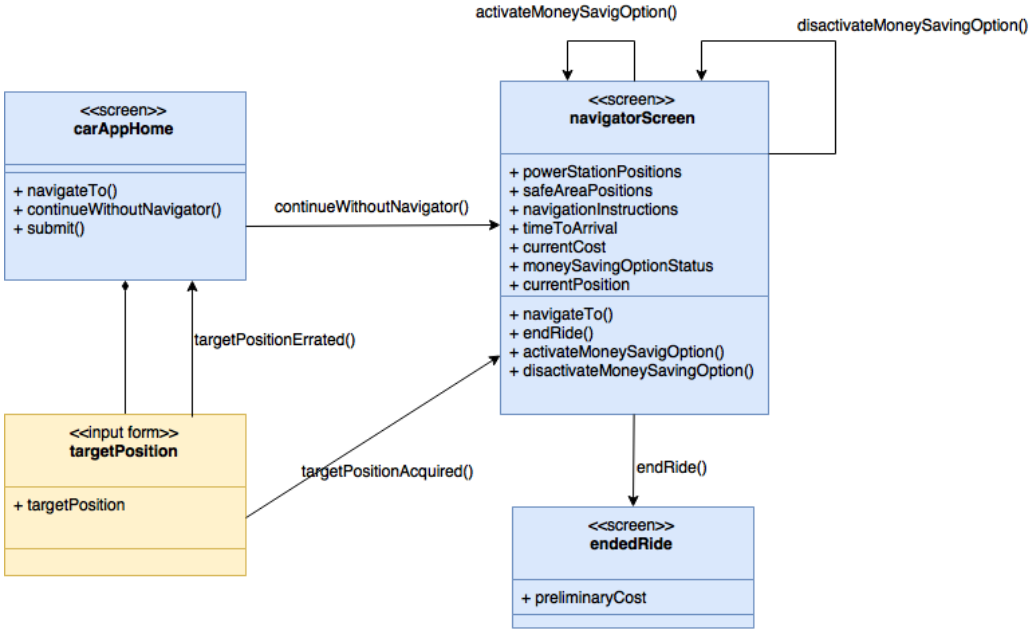
```
/*return the best power station where to plug in the car,
according to the distribution of the cars and the final destination.
Throws an exception if no suitable power station is found.*/
PowerStation moneySavingOption(Positon finalDestination, int r, SafeArea[] safeAreas) {
	SafeArea[] validSafeAreas;
	foreach(SafeArea safeArea in safeAreas)
	if (!safeArea.getAvailablePowerStations().isEmpty())
		validSafeAreas.add(safeArea);
	sortByParkedCars(validSafeAreas);
	foreach(SafeArea safeArea in validSafeArea)
	if (distance(safeArea, finalDestination) < r)
		return safeArea.getAvailablePowerStations.get(0);
	throw new Exception("no power station found");
}

void sortByParkedCars(SafeArea[] s) {
	//uses quickSort in order to sort the safe areas in O(NlogN)
}

int distance(SafeArea s, Position p) {
	//return the minimum distance in meters between a given position and a given safe
area
}
```

4. User interface design

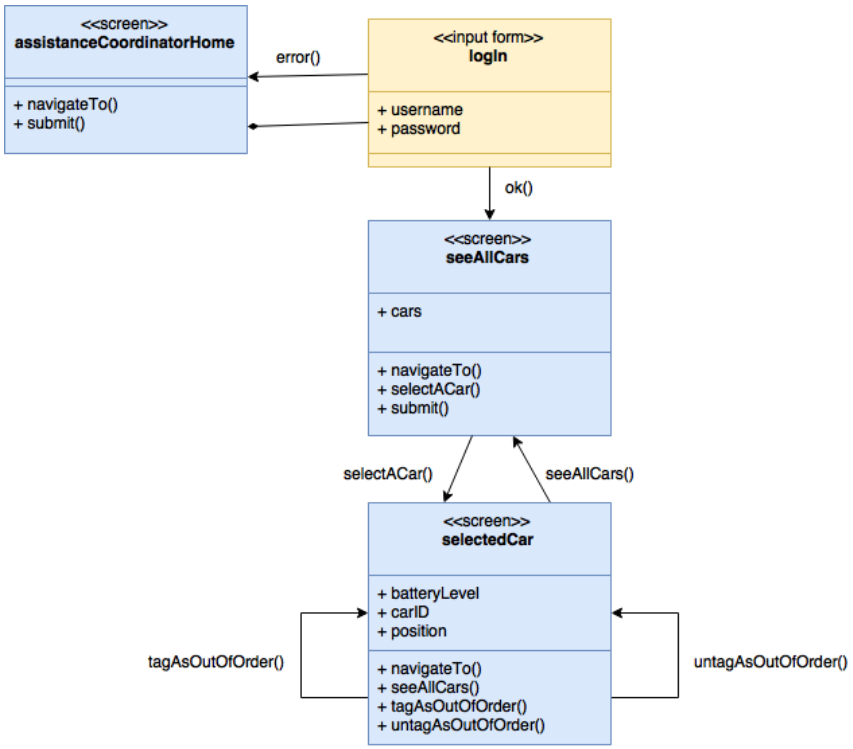
4.1. UX diagram client app



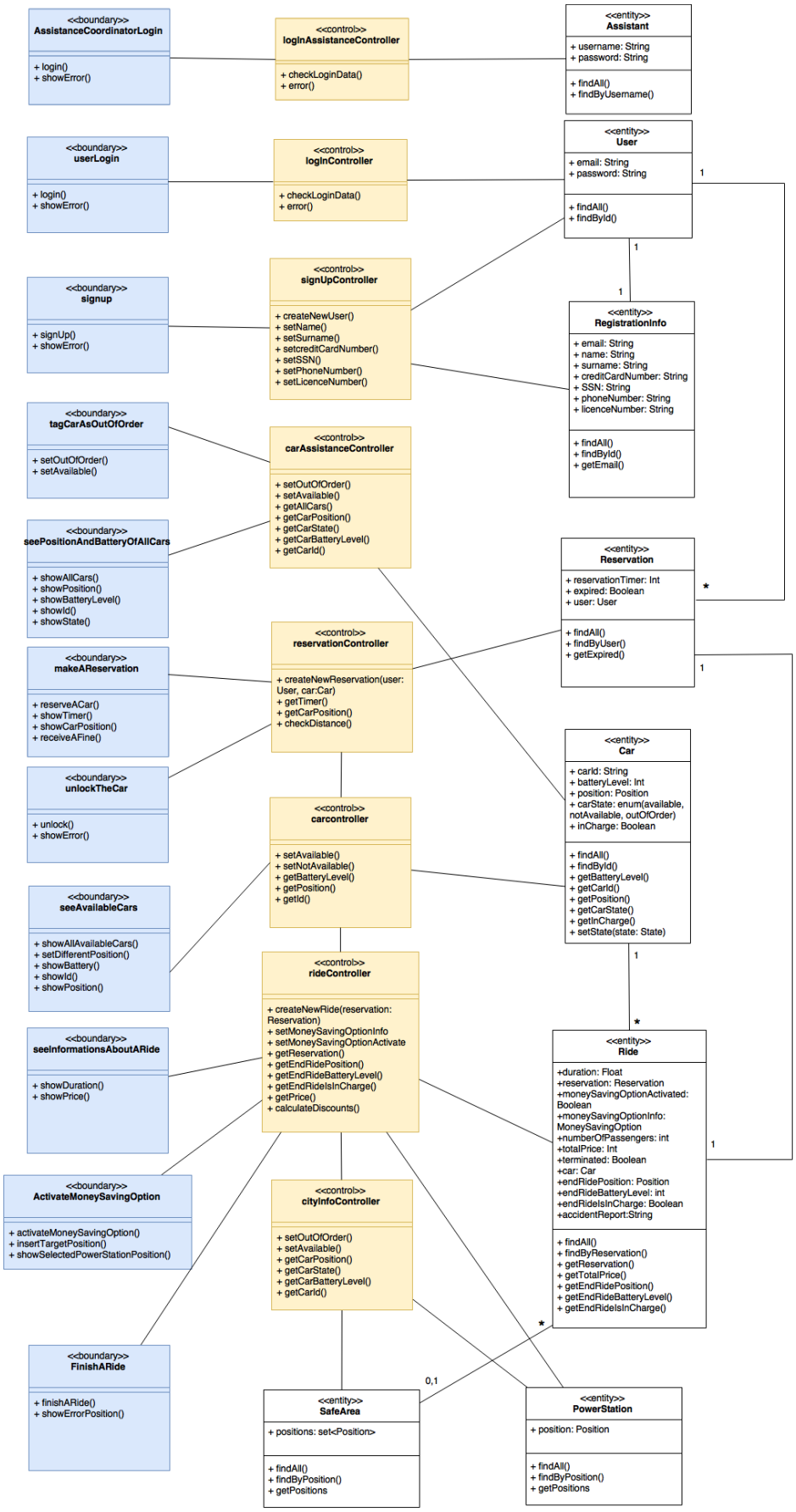
4.2. UX diagram car app



4.3. UX diagram assistance coordinator program



4.4. BCE diagram



5. Requirements traceability

The design of this project was made aiming to fulfill optimally the requirements and goals specified in the RASD. The reader can find here under the list of these requirements and goals and the designed component of the application which will assure its fulfillment.

1. Allow guests to sign up.
    - The LoginController
    - The ViewForClient and the client app
2. Allow users to sign in.
    - The SignUpController
    - The ViewForClient and the client app
3. Allow users to see the available cars (and their battery level) near them or near to a given address.
    - The CarController
    - The ViewForClient and the client app
4. Allow users to reserve an available car for up to one hour and to know if their reservation went successfully and eventually fine them if the hour expires.
    - The CarController
    - The ReservationController
    - The ViewForClient and the client app
5. Allow users to unlock and have access to a car if and only if they are close to that car and the car is reserved by them.
    - The CarController
    - The ReservationController
    - The RideController
    - The ViewForClient and the client app
6. Allow users to end a ride if and only if the car is in a safe area or the car has run totally out of battery or an accident happens.
    - The RideController
    - The CityInfoController
    - The ViewForCar and the car app
7. Allow users to receive a 10% discount from the total fee if they carry more than two people.
    - The RideController
    - The CarController
    - The ViewForClient and the client app
8. Apply a fine of 30% of the total cost to users if the car has been parked more than 3 km from the nearest power station or with less than 20% of battery.
    - The RideController
    - The CarController
    - The CityInfoController
    - The ViewForClient and the client app
9. Reward users with a 20% of discount if they leave the car with more than 50% of the battery.
    - The RideController
    - The CarController
    - The ViewForClient and the client app
10. Reward users with a 30% of discount if they leave the car charging into a power station.
    - The RideController
    - The CarController
    - The CityInfoController
    - The ViewForClient and the client app
11. Allow users to use the money saving option (see glossary)
    - The RideController
    - The CityInfoController

    - The ViewForCar and the car app
    - The ViewForClient and the client app
12. Allow users to know in real time all the information (cost, car's battery level, safe areas' location) about their ride.
    - The RideController
    - The CityInfoController
    - The ViewForCar and the car app
13. Allow assistance coordinator to login.
    - The LoginAssistanceController
    - The ViewForAssistanceCoordinator and the assistance coordinator program
14. Allow assistance coordinator to see the GPS position of all the available cars and their battery level in order to identify the cars in need of battery replacement.
    - The LoginAssistanceController
    - The ViewForAssistanceCoordinator and the assistance coordinator program
15. Allow the assistance coordinator tag a car/untag as out of order following an accident or damage report by a user.
    - The LoginAssistanceController
    - The ViewForAssistanceCoordinator and the assistance coordinator program

6. Other info

6.1. Reference documents

- Assignments AA 2016-2017.pdf
- Documents previously provided:
  - PowerEnJoy – RASD.pdf
- Sample documents:
  - Sample Design Deliverable Discussed on Nov. 2.pdf
- Course slides:
  - Design Part I.pdf
  - Design Part II.pdf
  - Architecture and Design in Practice.pdf
  - Examples of architectures.pdf
  - Reasoning on design through an example.pdf

6.2. Used tools

- Microsoft Word 2016, for the drafting of the DD
- Microsoft OneDrive, to allow concurrent editing
- GitHub, to store the project in a repo
- Draw.io, for the drawing of the diagrams

6.3. Hours of work

For redacting and writing the Design Document we spent approximately 25 hours per person.

6.4. Changelog

- Added component “DataAccessManager” in the component diagram of section 2.3.
- Changed the name of the object “Database” of the sequence diagrams in section 2.6. into “Model”.
- Minor changes in the drafting of the text and typing errors fixed.