POLITECNICO
MILANO 1863

"PowerEnjoy"

Design Document

Version 1.1 *(04/02/2017)*

Giorgio Marzorati (876546)
Aniel Rossi (877018)
Andrea Vaghi (877710)

INDEX OF CONTENTS

INTRODUCTION

Purpose

This document aims to provide design-level technical details on the architecture of PowerEnjoy, the mobile application of car sharing that we are going to implement.
It is addressed mainly to developers and stakeholders with at least some technical knowledge about architectural design and implementation.
The document will first analize high level components and give an overall description of the architecture. Those components will be then "unpacked" and described more precisely with a top-down approach, as well as the way they interact with each other.
The document will analize:

- High level architectures and the identification of the tiers
- Adopted design patterns
- Main components and their interaction
- Runtime behaviour with some UML diagrams & implementation pseudocode

Scope

PowerEnjoy is a car-sharing service thought for the city of Milan based on a mobile application with a single category of users.
The system allows clients to reserve, or directly identify, and use available electric-powered cars in the area around the client's GPS position or around an address inserted manually. In case of reservation, if the client does not identify the car within 1 hour from the reservation it is deleted and he is charged with a fee of 1€. Then the car becomes available again to other clients.
In order to use the application, clients have to register to the system, in particular they have to provide an e-mail address, biographical data and a valid driving license. On the other side, the system provides clients a personal PIN, with which they access to the system and are allowed to interact with car.
Clients are charged at the end of every ride and the payment is done with one of the supported payment methods that must be specified during the registration process and it can be modified in every moment.
In addition, the system defines different discounts and overcharges for every ride, as a result of particular client's behaviours (more informations are included in the Glossary session).

The system has the purpose of providing an efficient and environment-friendly alternative to public transportation to people who don't have to cover long-distance travels and don't want to (or cannot) use personal vehicles.

Definitions, Acronyms, Abbreviations

1. **API**: Application Program Interface, it exposes a set of public methods used to make two different systems communicating with each other
2. **BLL**: Business Logic Layer, is the central layer of the three-tier architecture. It's represented by the central server and its components
3. **CL**: Client Layer, is the first level of the three-tier architecture. It's represented by the Client, Cars and the Assistance Team
4. **DBMS**: Database Management System
5. **DD**: Design Document
6. **DL**: Data Layer, is the third and last level of the three-tier architecture. It's represented by the DBMS
7. **E-Mail Gateway**: it'a service used by the central server to send emails to clients
8. **JDBC Driver**: is a native Java connector used by the central server to comunicate with any relational DBMS
9. **HTTP**: HyperText Transfer Protocol, application-level protocol for exchanging information on the web, in a client-server arhitecture
10. **MQTT**: MQ Telemetry Transport, application-level protocol with a public-subscribe pattern
11. **RASD**: Requirements Analisys and Specification Document
12. **REST**: REpresentational State Transfer
13. **RESTful**: Web Service based on a REST architecture
14. **Top-Down approach**: is a descriptive model based on an iterative analisys, starting from a more generic and high level rappresentation of an architecture. At every iteration, the model is divided into more specific components which are also analized and divided.
15. **UML**: Unified Modeling Language, a graphic language used to represent different aspects (static, dynamic, architectural, behavioural..) of a specific software.
16. **URL**: Uniform Resource Locator
17. **UX**: User Experience

Reference Documents

- Assignments+AA+2016-2017.pdf
- Examples documents:
    - Sample Design Deliverable Discussed on Nov. 2

Document Structure

Introduction: the section explains the purpose of the document and underlines the main differents between DD and the RASD.

Architecture Design:

1. Overview: shows the overall architecture from an high level point of view.
2. Component view: provides an high level diagram and a more detailed one, explaining the purpose of the most important components and the interaction between tiers
3. Deploying view: this section shows the components that must be deployed to have the application running correctly.
4. Runtime view: sequence diagrams are represented in this section to show the course of the different tasks of our application
5. Component interfaces: this section presents the most important interfaces between components
6. Selected architectural styles and patterns: this section explain the architectural choices taken during the creation of the application

Algorithms Design: this section describes the most critical parts via some algorithms written in pseudocode.

User Interface Design: this section presents the user experience explained through an UX diagram.

Requirements Traceability: this section aims to explain how goals identified in the RASD are satisfied with the usage of design components.

ARCHITECTURAL DESIGN

Overview

PowerEnjoy features a three tier architecture, composed by a central server that communicates with different types of systems (represented by main end clients, cars and the assistance team) and a database that stores data collected and managed by the central server.
On the first level of the architecture we have clients, cars and the assistance. The central server acts as intermediary between these three elements (except for clients help requests, that are made directly to the assistance via phone calls).
The main client software representation is the mobile application, from which all client requests are made. Cars are equipped with embedded software that communicates with the central server for different purposes (explained in a more specific section of this document) and the assistance team gets and stores information about car problems and clients phone requests with software connected to the central server.

Component View

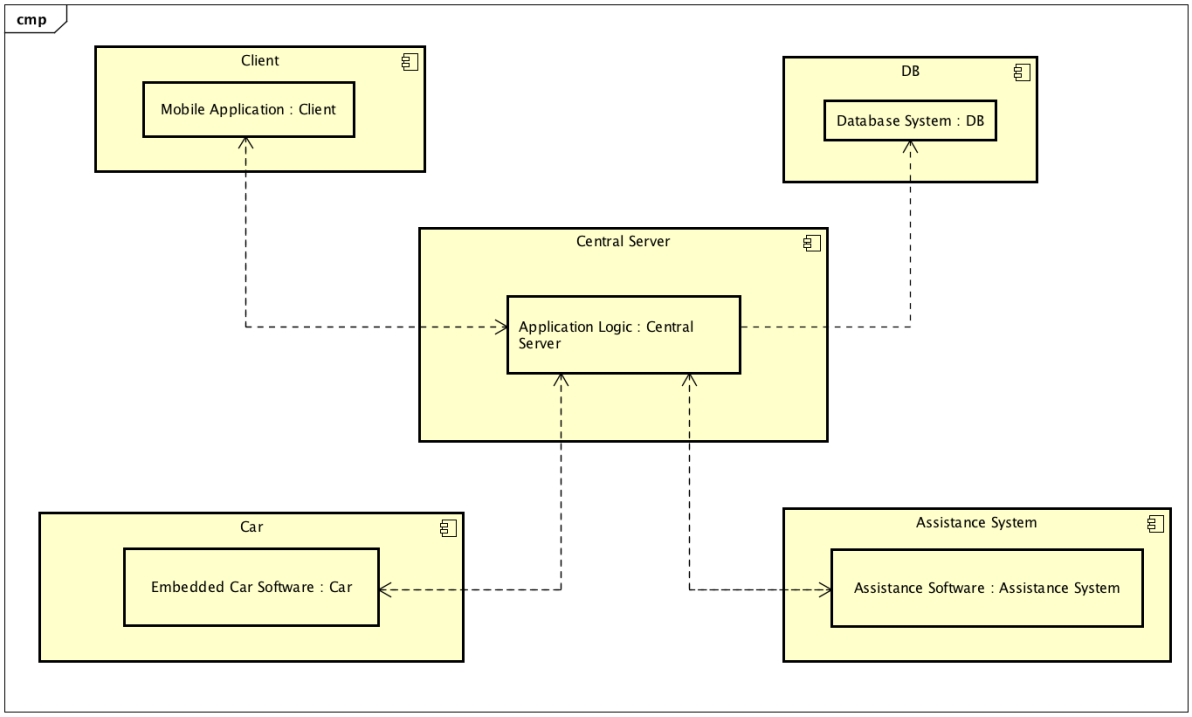

Figure 1 - High Level Component Diagram

In the High Level Component Diagram we have represented the overall architecture of our system showing the different element types. The main component is the central server, which communicates directly with every other one. It contains all the business logic of the application. The client interacts with the server from a mobile application with syncronous requests. The information in server's responses are used to update the user interface of the application, which is not static but dinamically generated. Every client's action and request is followed by an acknowledgement from the central server, in form of a popup on the mobile application (and an e-mail in the case of the first time registration to the system). Cars also exchange information with the central server, in particular it has to monitor and store all data from vehicles and their sensors (position, battery level, number of passengers, damaged components). Data are exchanged asynchronously. Cars are subscribed to a server specific component (more on this in the next paragraph), which perform specific operation on them (e.g. the unlocking operation).
The assistance team handles every help request, as well as cars malfunctions and recharge issues. This information is provided to the assistance team by the central server (except for client help requests made directly with phone call), and the team has also the responsibility of storing and managing data retrieved by clients about the specific issue. In order to achieve that goal, assistance team's terminals are equipped with dedicated software for central server interaction.
The last component the central server interacts with is the Database system, with which communicate synchronously to store and retrieve the whole managed data.

Component Diagram:

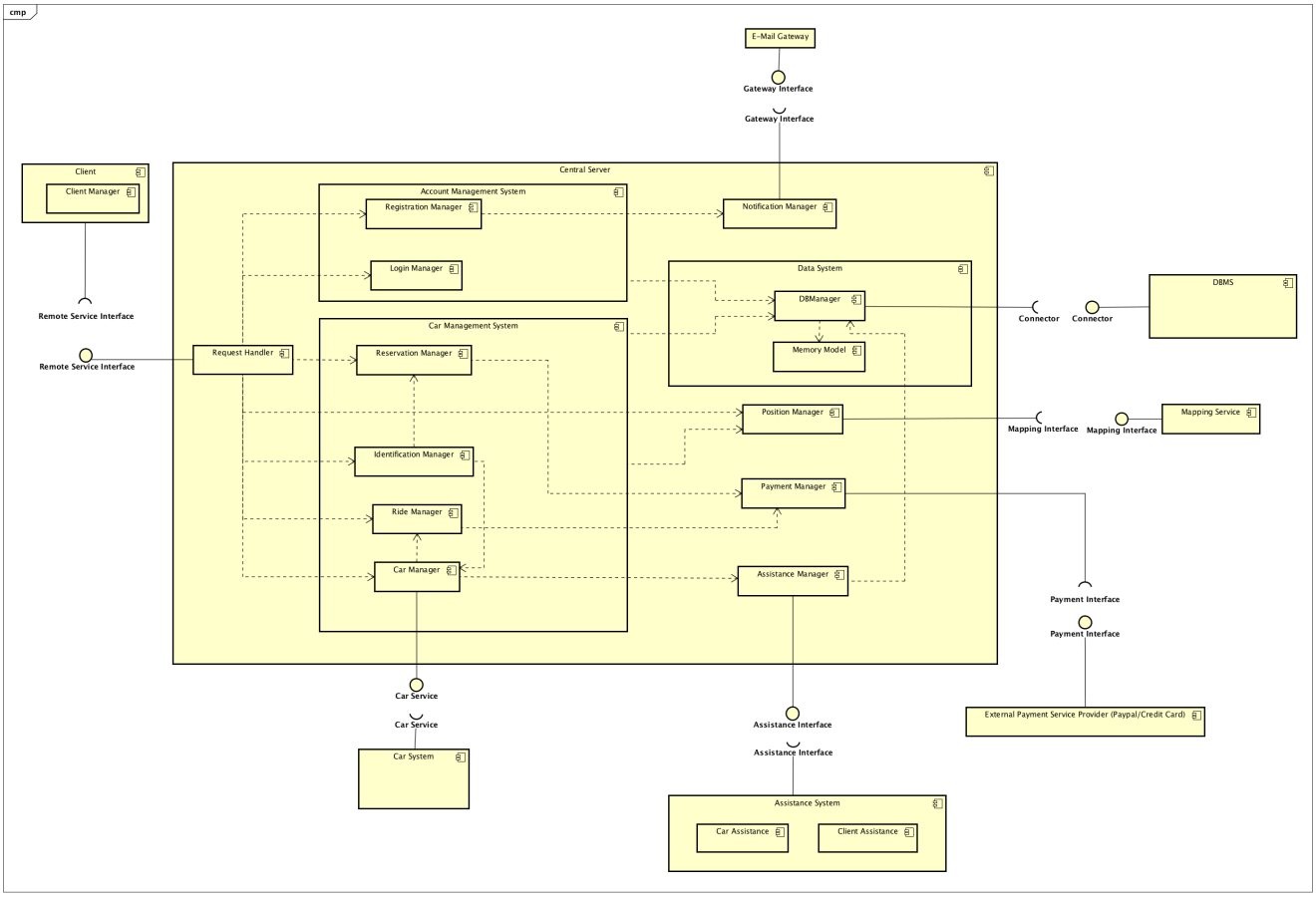

Figure 2 - Component Diagram

Component description:

- **Dispatcher**: routes Client's requests to the corresponding component
- **Account Management System**
  - **Registration Manager**: manages the Client's registration and the new PIN request processes
  - **Login Manager**: manages the Client's login procedure
- **Car Management System**
  - **Reservation Manager**: manages Client's car reservation requests and their validity
  - **Identification Manager**: takes care of the cars' identification by Client
  - **Ride Manager**: manages and calculates the total cost of every single ride, applying discounts or overcharges collecting data from other components
  - **Car Manager**: is the component directly interfaced with the car embedded software that handles the communication between cars and the Central Server
- **Data System**
  - **DBManager**: is the component that handles the operations on the DBMS
  - **Memory Model**: server side representation of the entities' model
- **Notification Manager**: manages the notifications mechanism from the Central Server to the Client
- **Position Manager**: manages the logic of determining Client and car's position and communicating it to other components, through API exposed by an external Mapping Service
- **Assistance Manager**: is the component with which the Assistance Team is interfaced to the Central Server for managing car issues and storing help requests
- **Car System**: is the embedded car software (provided by en external company) that reads sensor data and manages all physical elements and operations (e.g. unlocking). It's directly bound (through the Car Sevice Interface) with the Car Manager

DB ER-Model:

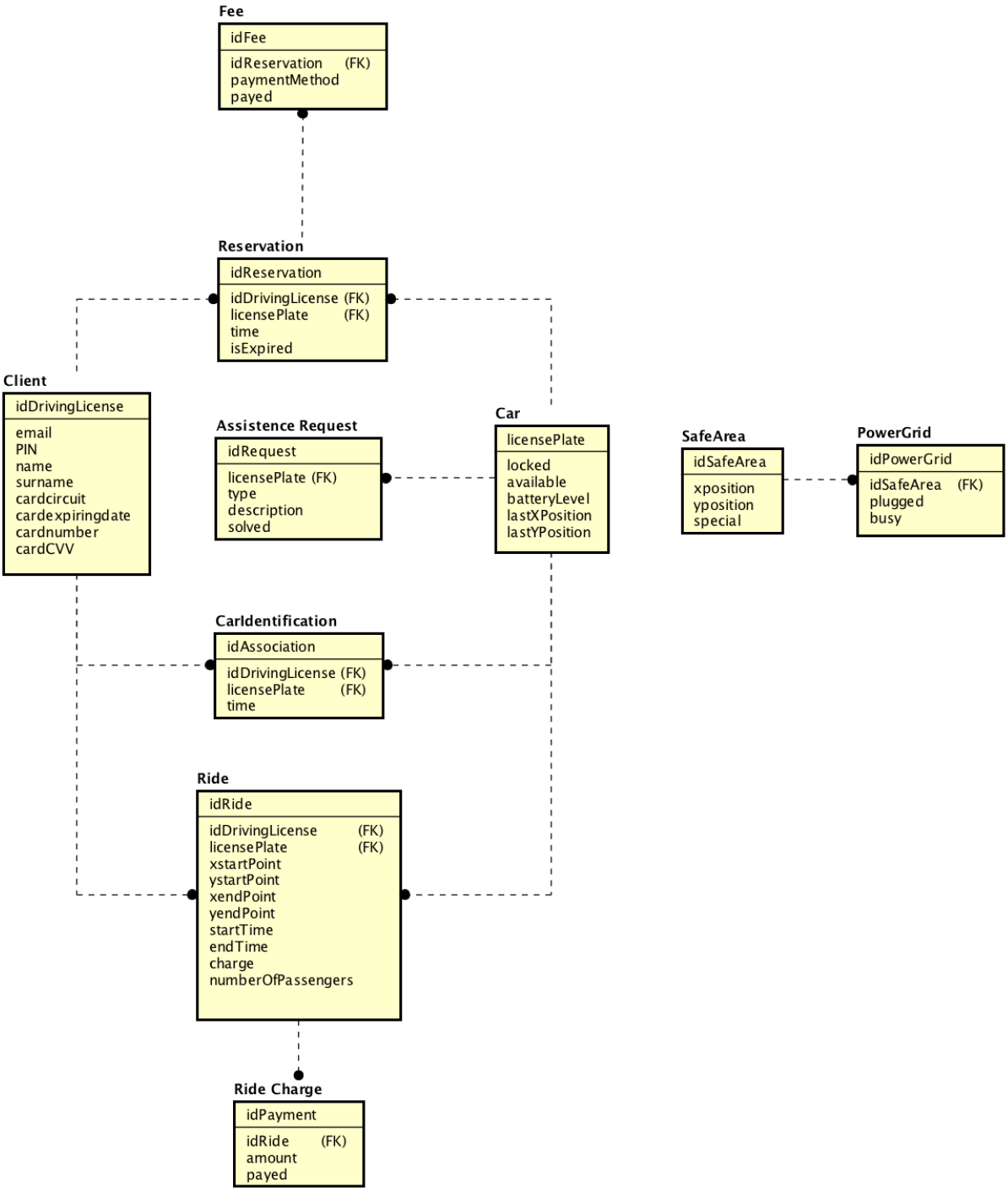

Figure 3 - ER Diagram

Communication within elements:

Client - Central Server:
The communication between these to elements is made possible by a client-server standard architecture supported by a REST API structure exposed to the client.

Car - Central Server:
Cars have two different roles towards the central server, with two dedicated access points (interfaces):

- It is subscribed to the BLL in a pub-server pattern, for the identification operation.
- It acts has a normal client when passing information to the BLL (at the end of every ride or when sensors detect issues and malfunctions)

Database - Central Server:
Database and the Central Server are connected together through a standard JDBC Driver

Assistance - Central Server:
Same as client - central server. The assistance team terminals are equipped with software components for performing specific operation:

- Monitor car issues (damages, low battery level) and mark those issues as open/resolved
- Store information about client help requests made by phone call

Client - Assistance:
By phone calls

Deployment View

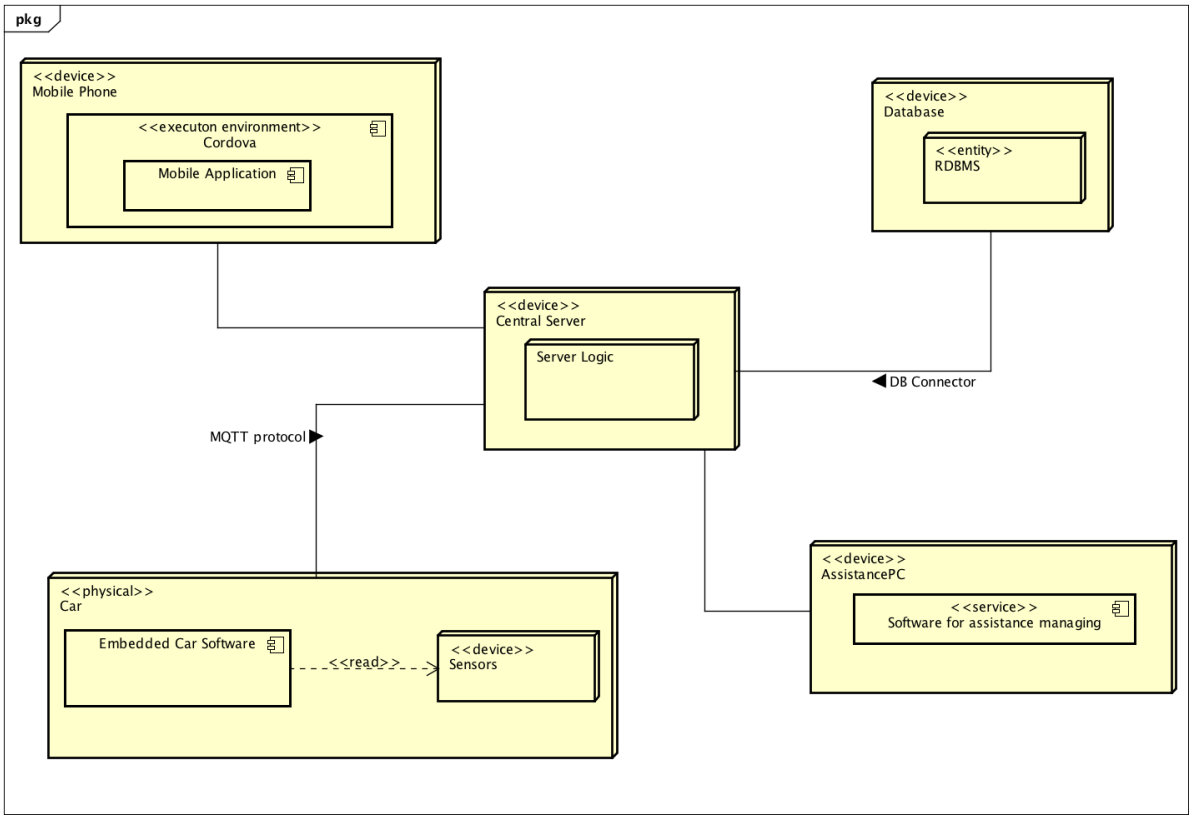

Figure 4 - Deployment Diagram

The deployment diagram highlights 5 nodes that reflect the structure shown in the first high level component diagram.
Important aspects:

- Our system will be a Hybrid Application, so it will be available for different OS (Android, iOS, Windows Phone) and we will take advantage of the Apache Cordova framework for the deployment
- We think that the Assistance Team needs a separated software running on separate machines, instead of a dedicated section inside the application
- The Central Server and the Database System will be replicated on more machines, in order to guarantee the availability of the service

Runtime View

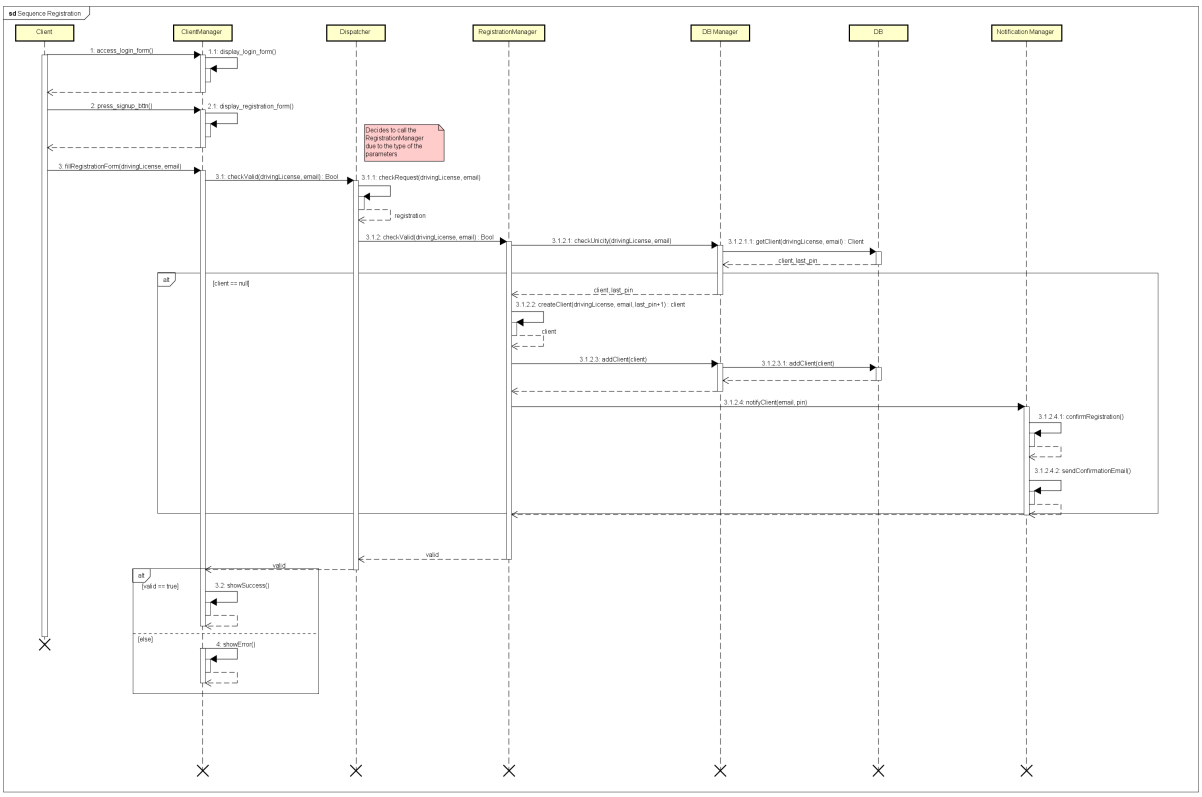

Figure 5 - Sequence Diagram (Registration)

This sequence diagram describes the procedure of registration into our application. Once opened the app from the mobile phone, the user is asked to insert his account credentials. If the user doesn't have an account yet, he has the possibility to enter in the registration section and insert his email address and driving license number. In the system both these information MUST be unique (a person can have only one account), so the system performs a verification in the database about the uniqueness of the data. If data are unique, the system generates a unique code (PIN) and sends it back to the user. From that moment the PIN is the only way to the user to effectively identify cars. In case the information is not unique, the user is notified that there is already an account using that email address or driving license number.

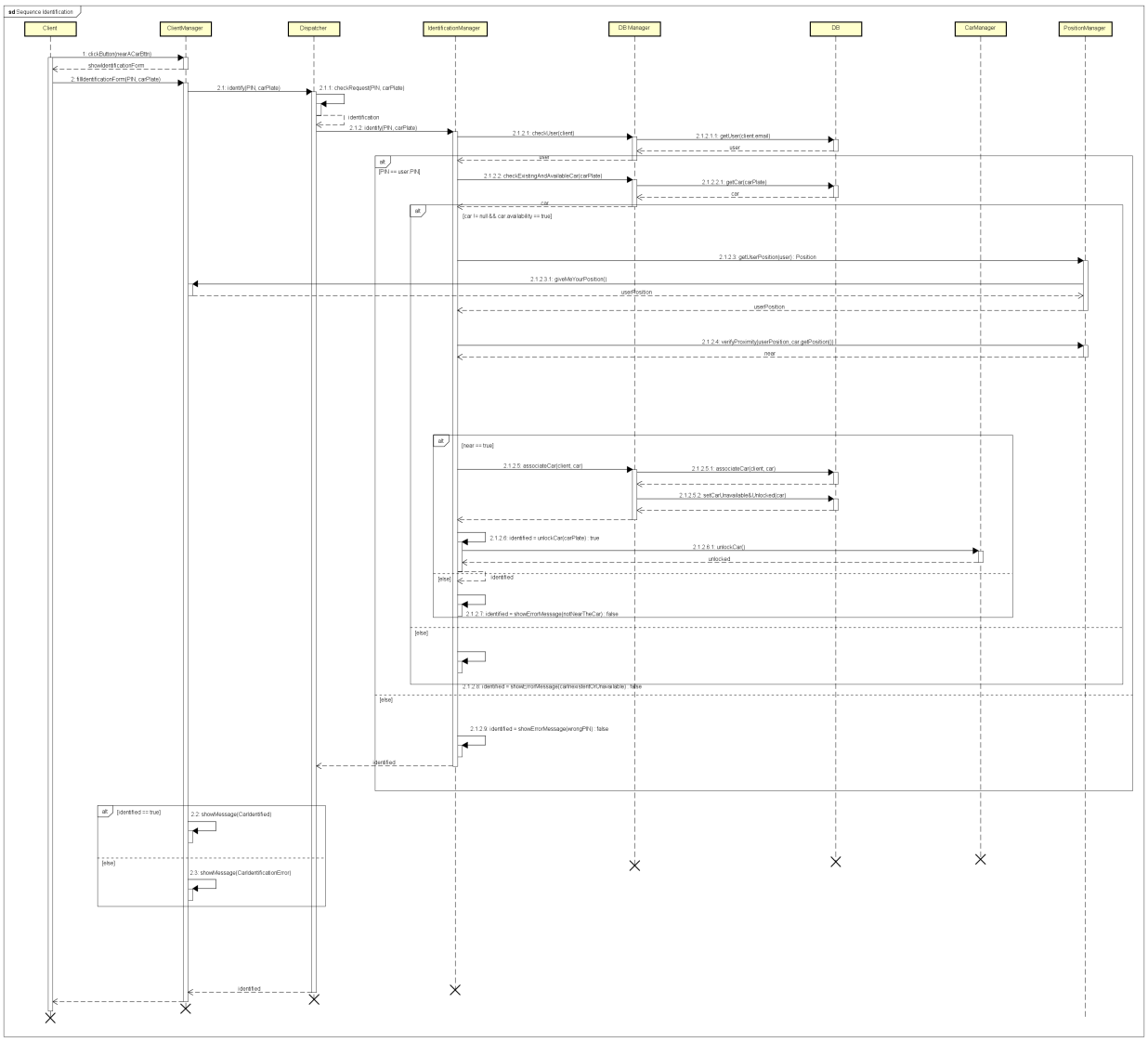

Figure 6 - Sequence Diagram (Identification)

This sequence diagram describes the procedure of identification of a car in the moment in which the user reaches it.
Once the user opens the app and logs in, he has the possibility to push a button called "I'm near a car". Doing this, the app answers opening the identification form in which the user has to insert his personal PIN and the license plate of the car. The verification process is divided in more phases in a precise order:

1. Right PIN -> comparison between the provided PIN and the PIN linked to the user into the system
2. Existence and availability of the car from the license plate
3. Proximity of user and car -> the system gets the user and car positions and verifies if the positions are near

In case these controls pass the car is associated to the user, unlocked by the system and the user notified of the success of the identification process.
In the other cases the system notifies the user about the error that occurs in this process and the car is not unlocked and not associated to the user.

Component Interfaces

We included 2 class diagrams that describe the interfaces' structure between:

- *Client and Central Server*

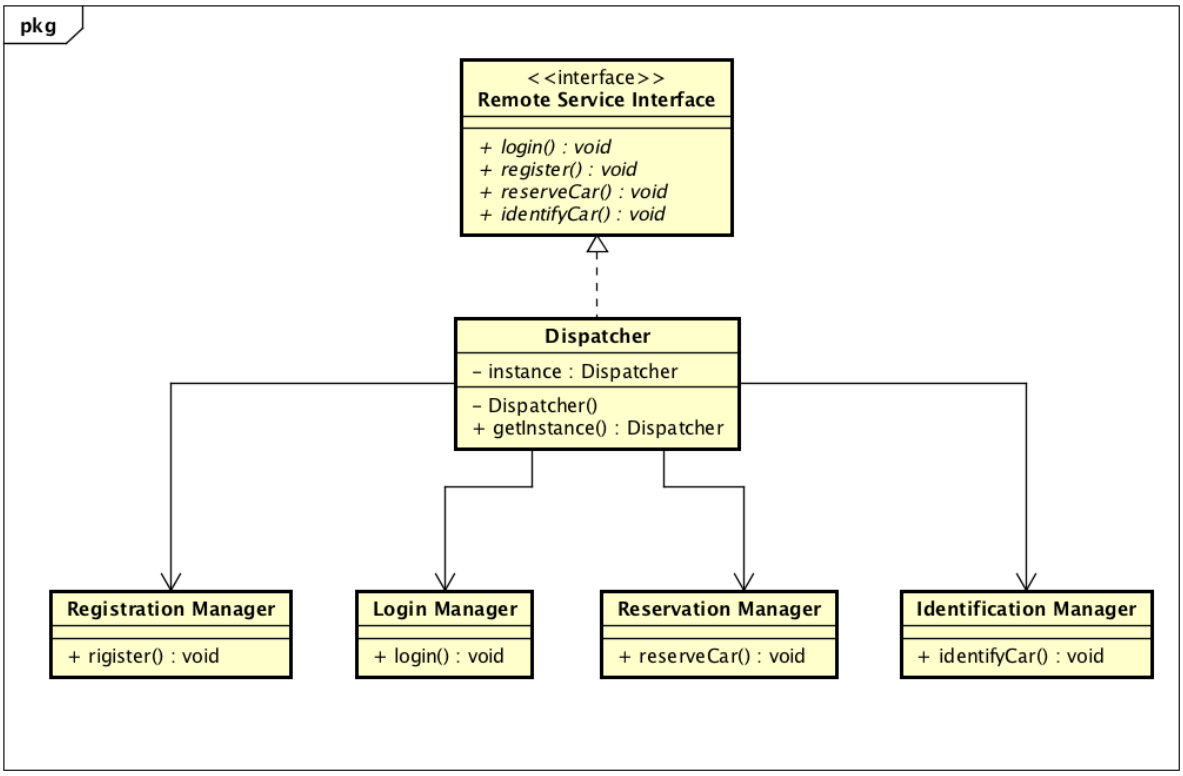

Figure 7 - Client/Server Interface

- *Car and the Central Server*

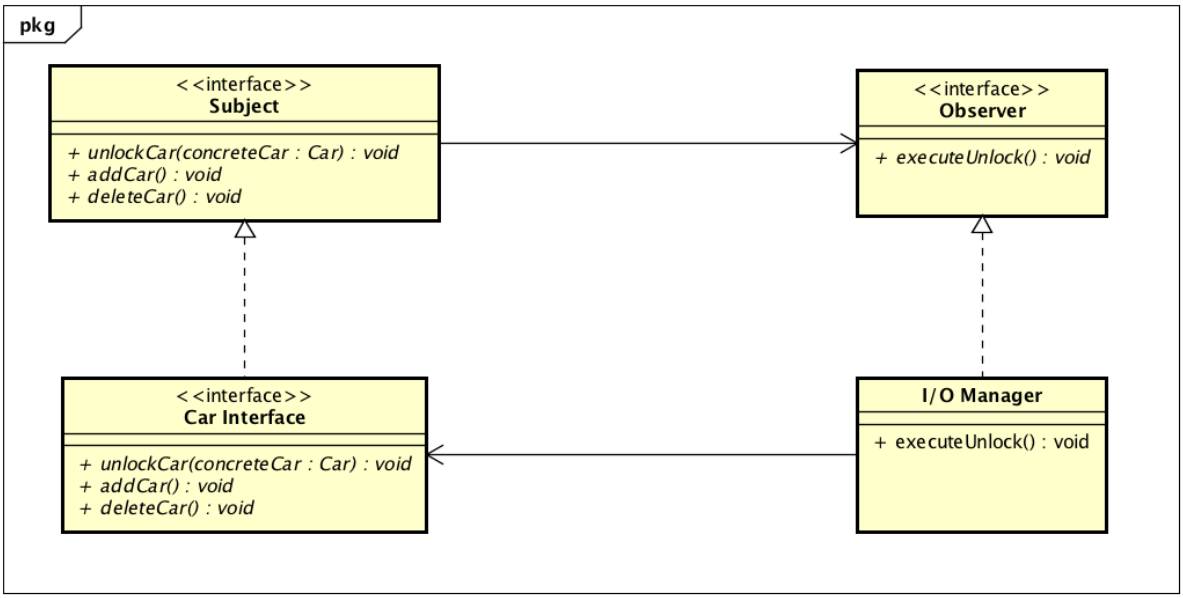

Figure 8 - Car/Server Interface

Selected Architectural Styles and Patterns

Overall Architecture:

Our system is based on a three-layers architecture:

- Client Layer [CL], with basic operations and simple interfaces to central server services. This level is represented by main end client, cars and the assistance team
- The Business Logic Layer [BLL], written in Java
- The Data Layer [DL]

Protocols and technologies:

In the next section we will show the protocols used to make tiers communicate with each other.

JDBC Driver: used to allow the BLL to access to the DL and execute queries and updates. The choice is made considering the fact that JDBC Driver is already included in the Java platform and supports every type of relational DBMS.

RESTful API: used to let the Client and Assistance Team making synchronous request to the Central Server (throw different and dedicated interfaces). Central Server responses are provided in JSON format. The web service is REST-based because of:

1. Simpler implementation, taking full advantage of the HTTP protocol
2. Loose coupling between Client and Central Server

MQTT: protocol used to connect cars and the Central Server, useful in particular to monitor vehicles' sensors and collect data.

Design Patterns:

Singleton: used for some particular instances that needs to be unique in our system:

- DB Manager
- Dispatcher
- Managers

Observer: used to create the pub-sub relation between the car node and the central server (we included a diagram in the "component interfaces" section)

Facade: used to create a unique interface that exposes all the services provided by different server components to the client

ALGORITHM DESIGN

In this section we show the pseudocode of some important function calls:

Verify if the user is near a car:

Position Manager:

```
bool verifyProximity(Position userPosition, Position carPosition){
      bool isClose = false;
      if(MappingService.calculateDistance(userPosition, carPosition) <
            IdentificationManager.MAX_CLOSENESS_BOUND)
            //upper bound parameter for the identification {
      isClose = true;
      }
      return isClose;
}
```

Money Saving Mode:

Ride Manager:

```
Position moneySavingMode(Position endPosition){
      DBManager dbm = DBManager.getInstance();
      PositionManager pm = PositionManager.getInstance();
      List<SpecialSafeArea> specialList = dbm.getSpecialSafeAreas();
      Float min=pm.calculateDistance(endPoint, specialList[0].getPosition);
      Position parkingPos=specialList[0].getPosition;
      for (SpecialSafeArea special : specialList){
            if(calculateDistance(endPoint, special.getPosition) <= min){
                  min=calculateDistance(endPoint, special.position);
                  parkingPos = special.position;
            }
      }
      return parkingPos;
}
```

Car reservation (from a GPS client position):

Reservation Manager:

```
List<Car> getAvailableCars(Position clientPosition){
      Double bound = ReservationManager.CAR_SEARCH_BOUND;
      DBManager dbm = DBManager.getInstance();
      PositionManager pm = PositionManager.getInstance();
      List<Car> availableCars = dbm.getAvailableCars();
      List<Car> availableCarsInBound = new ArrayList<Car>();
      for(Car c: availableCars){
            if(pm.calculateDistance(c.getPosition, clientPosition) >=
            bound){
                  availableCarsInBound.add(c);
            }
      }
      return availableCarsInBound;
}

boolean reserveCar(Strig carPlate, int PIN){
      DBManager dbm = DBManager.getInstance();
      if(dbm.getCar(carPlate).isAvailable==true){
            dbm.setUnavailableCar(carPlate);
            Reservation res = new Reservation(carPlate, PIN, timeStamp);
            dbm.addReservation(res);
            return true;
      }else{
            return false;
      }
}
```

Position Manager:

```
int calculateDistance(Position p1, Position p2){
      int distance = 0;

      //this method return an int representation of the street distance
      //between 2 pairs of GPS coordinates, using a dedicated library for
      //this purpose

      distance = MappingService.calculateDistance(p1,p2);

      return distance;
}
```

USER INTERFACE DESIGN

We included user interface mockups in the RASD document.
Here we provide a Class Diagram representation, showing the paths according to user operations.

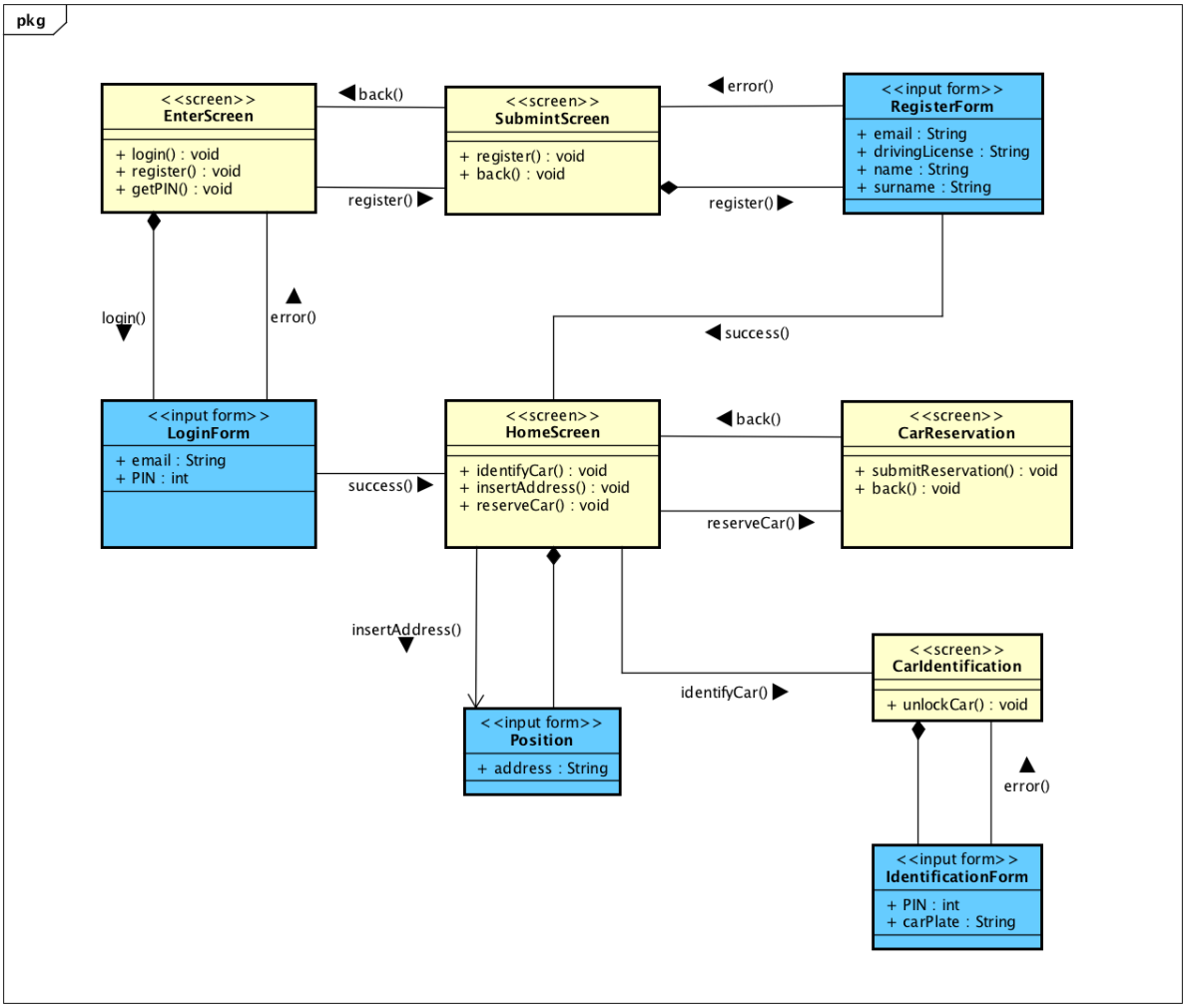

Figure 9 - UX Diagram

REQUIREMENTS TRACEABILITY

- *Client*
  - [G1] - Register to the system:
    - Registration Manager
    - Notification Manager
    - E-Mail Gateway
  - [G2] - Log into the system:
    - Login Manager
  - [G3] - Request a new PIN:
    - Registration Manager
    - Notification Manager
    - E-Mail Gateway
  - [G4] - Find available cars from the current position:
    - Position Manager
    - Car Manager
  - [G5] - Find available cars from a specific address:
    - Position Manager
    - Car Manager
  - [G6] - Reserve a car:
    - Reservation Manager
    - Car Manager
    - Notification Manager
  - [G7] - Identify a car:
    - Identification Manager
    - Car Manager
    - I/O Manager
  - [G8] - Drive a car:
    - Car Manager
    - Ride Manager
    - I/O Manager
    - Display Manager
    - Position Manager
  - [G9] - Monitor current charging during a ride
    - I/O Manager
    - Display Manager
    - Ride Manager

- [G10] - Enable “Money Saving Mode” for a ride
  - Position Manager

- [G11] - Host Passengers for a ride
  - I/O Manager

- [G12] - Park the car in a Safe Area
  - Car Manager
  - Position Manager
  - Display Manager
  - I/O Manager

- [G13] - Plug the car in a Power Grid
  - Display Manager
  - I/O Manager

- [G14] - Pay a ride
  - Ride Manager
  - Payment Manager

- [G15] - Pay a fee
  - Reservation Manager
  - Payment Manager

- [G16] - Get a discount on a ride
  - Display Manager
  - I/O Manager
  - Car Manager
  - Position Manager

- [G17] - Get an overcharge on a ride
  - Display Manager
  - I/O Manager
  - Car Manager
  - Position Manager

- [G18] - Ask for assistance
  - Notification Manager

- *Assistance*

  - [G19] - Get notified about car’s low battery level
    - I/O Manager
    - Car Manager
    - Assistance Manager

  - [G20] - Get notified about car malfunctions
    - I/O Manager

        - Car Manager
        - Assistance Manager

    - [G21] - Receive client's assistance request
        - Assistance Manager

EFFORT SPENT

Giorgio Marzorati

29/11/2016 - 3h
04/12/2016 - 3h
05/12/2016 - 2h
08/12/2016 - 3h
09/12/2016 - 4h
10/12/2016 - 3h
11/12/2016 - 2h

Aniel Rossi

29/11/2016 - 3h
04/12/2016 - 3h
05/12/2016 - 2h
08/12/2016 - 3h
09/12/2016 - 5h
10/12/2016 - 3h
11/12/2016 - 1h

Andrea Vaghi

29/11/2016 - 3h
01/12/2016 - 2h
02/12/2016 - 3h
04/12/2016 - 1h
05/12/2016 - 2h
06/12/2016 - 1.30h
07/12/2016 - 1h
08/12/2016 - 3h
09/12/2016 - 1h
10/12/2016 - 5h
11/12/2016 - 3h

CHANGELOG

V1.0 - First release

V1.1

- Added subsystem components in Component Diagram
- Algorithm Design and Sequence Diagram improvement due to a previous method call (verifyProximity now is in Position Manager)
- ER Diagram correction due to fields regarding payment method added in Client entity
- Adjustments in changes regarding the V1.2 of the RASD