Design Document

POWER Enjoy

Authors: Cannas, Castiglioni, Loiacono

POLITECNICO
MILANO 1863

SUMMARY

1. INTRODUCTION

A. Purpose

This is the Design Document for a digital management service required by Power EnJoy. The aim of this document is to give technical details omitted in the RASD document about the software to be. The document is addressed to developers and aims to identify:

- The high-level architecture of the System
- The main components, and interfaces provided to enable communication
- The Runtime behaviour of the System
- The design patterns used to develop the System
- The User Experience and Interface

B. Scope

The System under development is made of a mobile application and an application deployed directly on the Cars owned by Power EnJoy. A central server will let the communication between the two applications and with a database server.
As described in the RASD document, the main stakeholders for our System are Users previously registered with the mobile application. Through this mean, the System allows them to search and reserve a car within a selected geographical zone, using the car's GPS position; it also allows them to search for Charging Safe Area near their destination, to save money when they perform a ride. The Users are also allowed to have intermediate stops during their travelling, if they don't exceed the maximum time granted for the use of a Car.
Various discounts and overcharges are applied in the final computation of the payments, to encourage Users to have a virtuous behaviour. An Operator is contacted in the event of car accidents or empty battery.
The main purpose of the System is to increase the number of Users of the Power EnJoy Electric Car Service through a simple, powerful and efficient instrument, and to offer them a better service with a clear User Interface and User Experience.

C. Definitions, Acronyms, Abbreviations

- **RASD:** Requirements Analysis and Specifications Document.
- **DD:** Design Document.
- **API:** Application Programming Interface.
- **MVC:** Model View Controller pattern.
- **REST:** Representational State Transfer. An architectural pattern for client-server communications.
- **UX:** User Experience.

- **BCE:** Business Controller Entity pattern. A variant of the MVC pattern, used to express details of the User Experience in relation with the business goals.
- **Green Move System:** a series of software platforms used as interface between the Car's control unit and application. It is divided between the Green E-Box (on the car) and the Green Move Center (on the central server).
- **Stripe API:** an API which enables the communication with payment providers in a standardized way.
- **EUCARIS API:** an API which enables the communication with the European Driver Licence Database, to confirm the authenticity of Users' driver licenses.

Other Definitions are provided in the RASD document, in section 1.4 (Definitions, Acronyms, Abbreviations).

D. Reference Documents

- Assignments AA 2016-2017.pdf
- Slides and Lectures on Software Design By Di Nitto Elisabetta
- Sample Design Deliverable Discussed on Nov. 2.pdf
- Paper on the green move project.pdf
- PowerEnJoy_RASD.pdf

E. Document Structure

- **Introduction:** this section introduces the design document. It contains a justification of his utility and indications on which arguments are covered in this document, and the high-level approach to the Software-to-be.
- **Architecture Design:** this section highlights the reference architecture that will be used to develop the System. It is divided into several different subsections:
    - 1. **Overview:** this sections explains the division into different tiers of the System.
    - 2. **Component view:** this sections gives a detailed view of the components of the System, providing a reference for their functionalities.
    - 3. **Deployment view:** this section shows on which platform the components are meant to be deployed.
    - 4. **Runtime view:** this section provides different Sequence Diagrams, to highlight the runtime behaviour of the System, and the interaction between components.
    - 5. **Component interfaces:** the interfaces which allows the communication between components are showed in this section.
    - 7. **Selected architectural styles and patterns:** this section highlights the architectural choices taken during the design of the System.

    - 8. **Other design decisions.**
- **Algorithms Design:** this section describes the most critical parts of the System via several algorithms. Pseudo code is used to hide unnecessary implementation details.
- **User Interface Design:** this section shows mock-ups of the mobile application and of the Car application, which are also explained through UX and BCE diagrams, highlighting the interactions between the User and the System.
- **Requirements Traceability:** this section aims to explain how the decisions taken in the RASD are linked to the different design elements.

2. ARCHITECTURAL DESIGN

A. Overview

The architecture of the System is divided in multiple tiers.

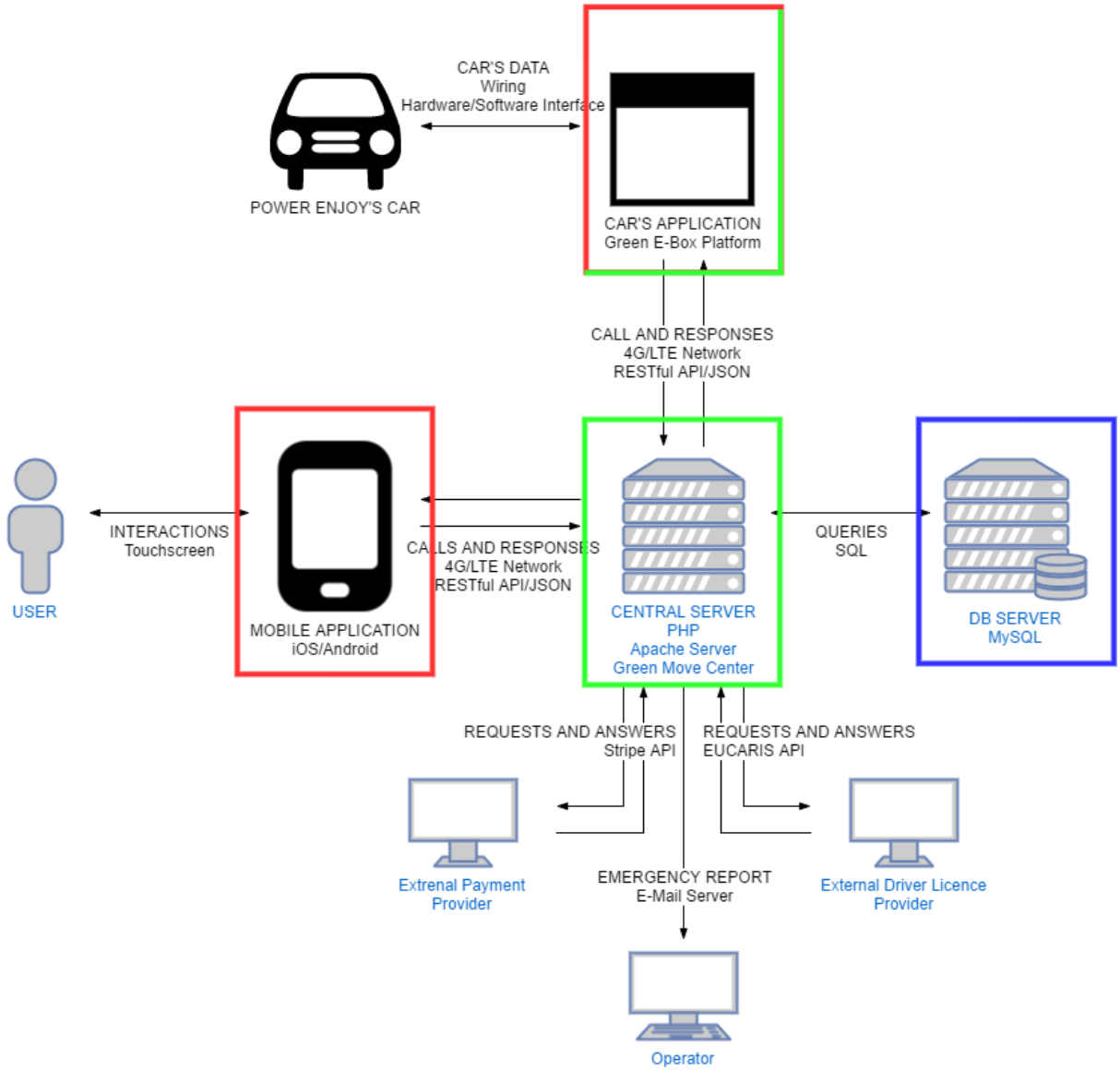

Figure 1: Architecture of the System

The Mobile Application hosts the presentation tier, while the GUI is generated partially dynamically, to show the processes computed by the Central Server. To ensure this happens, a module inside the Client communicates with the server using a RESTful API, expecting JSON responses.
The logic tier is mostly managed by the Central Server, which communicates with external providers via several different APIs.
Part of the logic tier is managed directly on the Car's Application, to ensure the Central Server is not overloaded with computation requests. Both logic and presentation tier of the Car's application are hosted on a Green E-Box platform, to ensure an easy deployment of the software.
With this architecture, the application tier and the database can easily be moved to a cloud system, for example to amazon AWS where it would have dedicated cloud servers with load balance for database and other for application logic on demand.

B. Component view

a. High-Level Component View

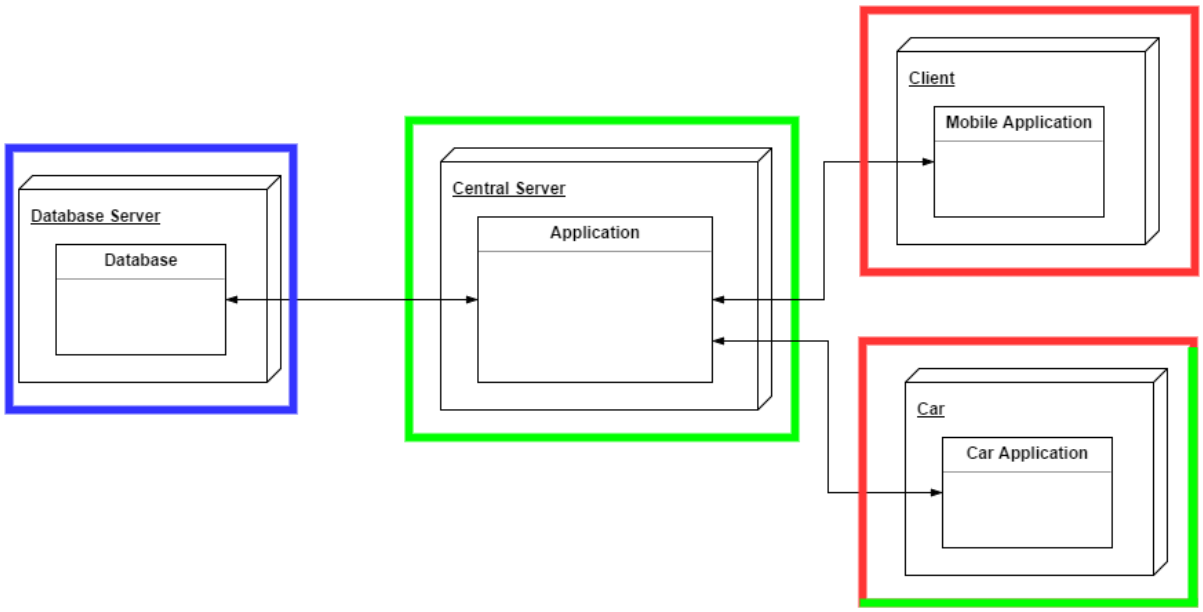

Figure 2: High-Level Component View

The high-level components architecture is composed of four different elements.
The Central Server receives search and reservation requests from the Client. The Client initiates this communication within the Mobile Application. The communication is managed synchronously: the Client waits for the answer of the Central Server that acknowledges him that his request has been taken into account. The Central Server sends asynchronous messages to the Client to inform him about payment bills, and expired reservations.
The Central Server also manages the Registration processes of Users, and Payment certifications, contacting external providers to ensure these functionalities.
The Central Server communicates with the Cars. It can send messages to Car Applications to inform them about Reservations and to request them their state data and position.
It also receives emergency messages, whose data are used to call external operators, responsible for retrieving and repairing the Car.
A Database is connected to the server. The Database stores every data about Users, Cars, Reservations and Safe Areas, and communicates with the Central Server through SQL queries.

b. Component View

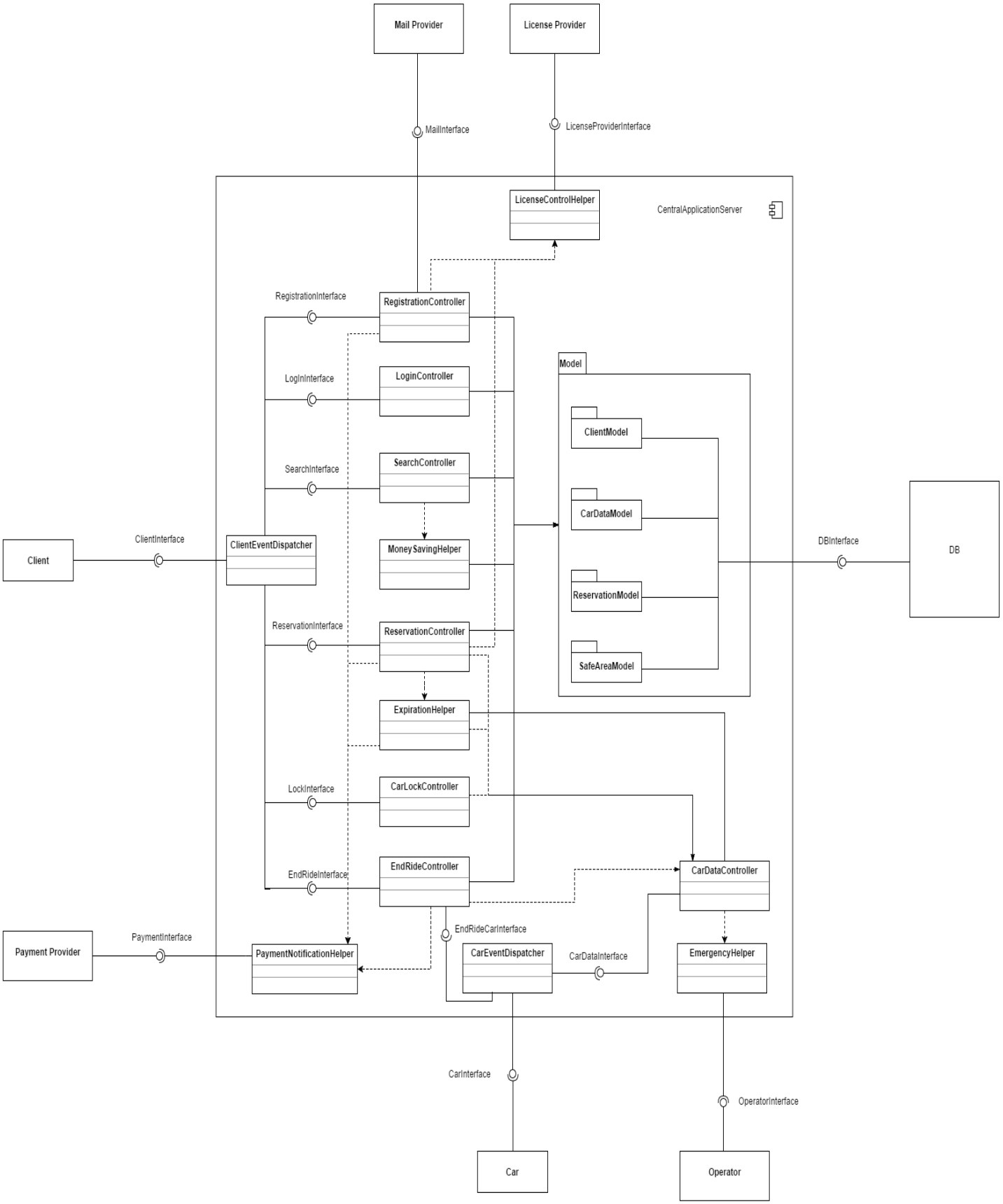

Figure 3: Server Component View

- **ClientEventDispatcher:** manages all Client requests and redirects them to other components.
- **CarEventDispatcher:** manages all Cars requests and redirects them to other components.
- **RegistrationController:** manages Registration request.
- **LoginController:** manages Login request.
- **SearchController:** manages Search request.
- **ReservationController:** manages Reservation request.
- **ExpirationController:** manages Expired Reservation Event.
- **CarLockController:** manages CarLock requests.
- **CarDataController:** manages Data Received from the Car.
- **MoneySavingHelper:** modifies the SearchController in order to manage the MoneySaveSearch request.
- **EndRideController:** manages payment computation, and EndRide process.
- **LicenseControlHelper:** uses the EUCARIS API to interface with the License Control Provider and check if the License is valid.
- **PaymentNotificationHelper:** uses the Stripe API to interface with the Payment Provider and ensure payments.
- **EmergencyHelper:** uses a standard email protocol to inform Operators of emergencies.

The Dispatchers receive requests from different Cars and Clients, and dispatch them to the appropriate controllers. Dispatchers also send messages and data back to Cars and Clients. Helpers expand Controllers functions, or are shared functions used by multiple Controllers. Communications with the Client and the Car will make use of a RESTful API and JSON. Different components adapt external APIs to interface with other components.
For what concerns the CarApplication, a diagram of its components is shown below.

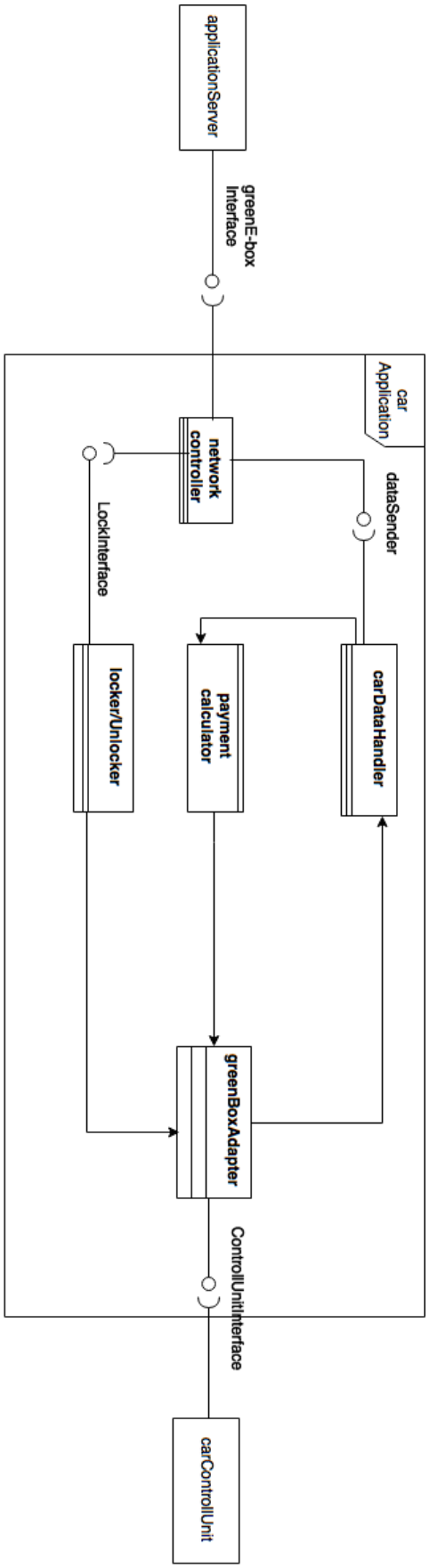

Figure 4: CarApplication Component View

- **networkController:** manages the communications between the CarApplication and the server.
- **carDataHandler:** receives the Car's data and analyses them, sending the information needed for the payment to the payment calculator and the server.
- **paymentCalculator:** computes the charge of the current ride, excluded the discounts and overcharges, notifying it to the carControllUnit in order to show it on the Car's monitor.
- **lock/Unlocker:** manages the lock and unlock request for the Car's doors.

The carDataHandler is in charge of providing all the necessary information (engine status and Car's position) to the paymentCalculator in order to compute and show to the User the correct charge for his/her rental of the Car without requiring communications with the server.
The only messages sent from the server to the car are Lock/Unlock requests messages that are dispatched to the locker/Unlocker.

C. Deployment view

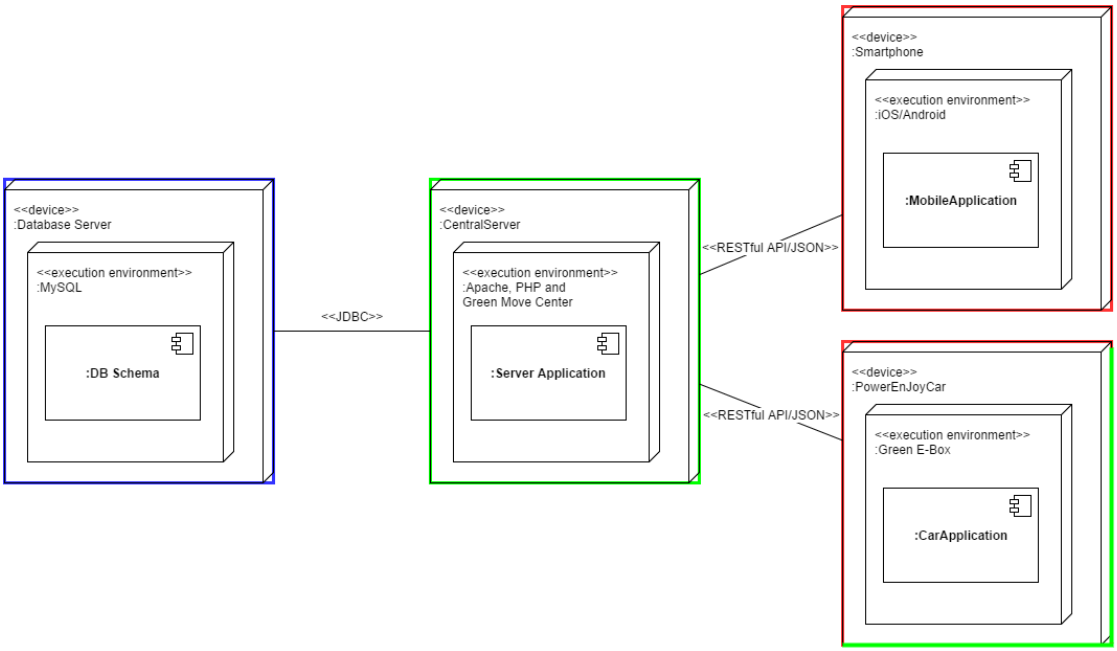

Figure 5: Deployment View, with tiers specified.

E. Runtime view

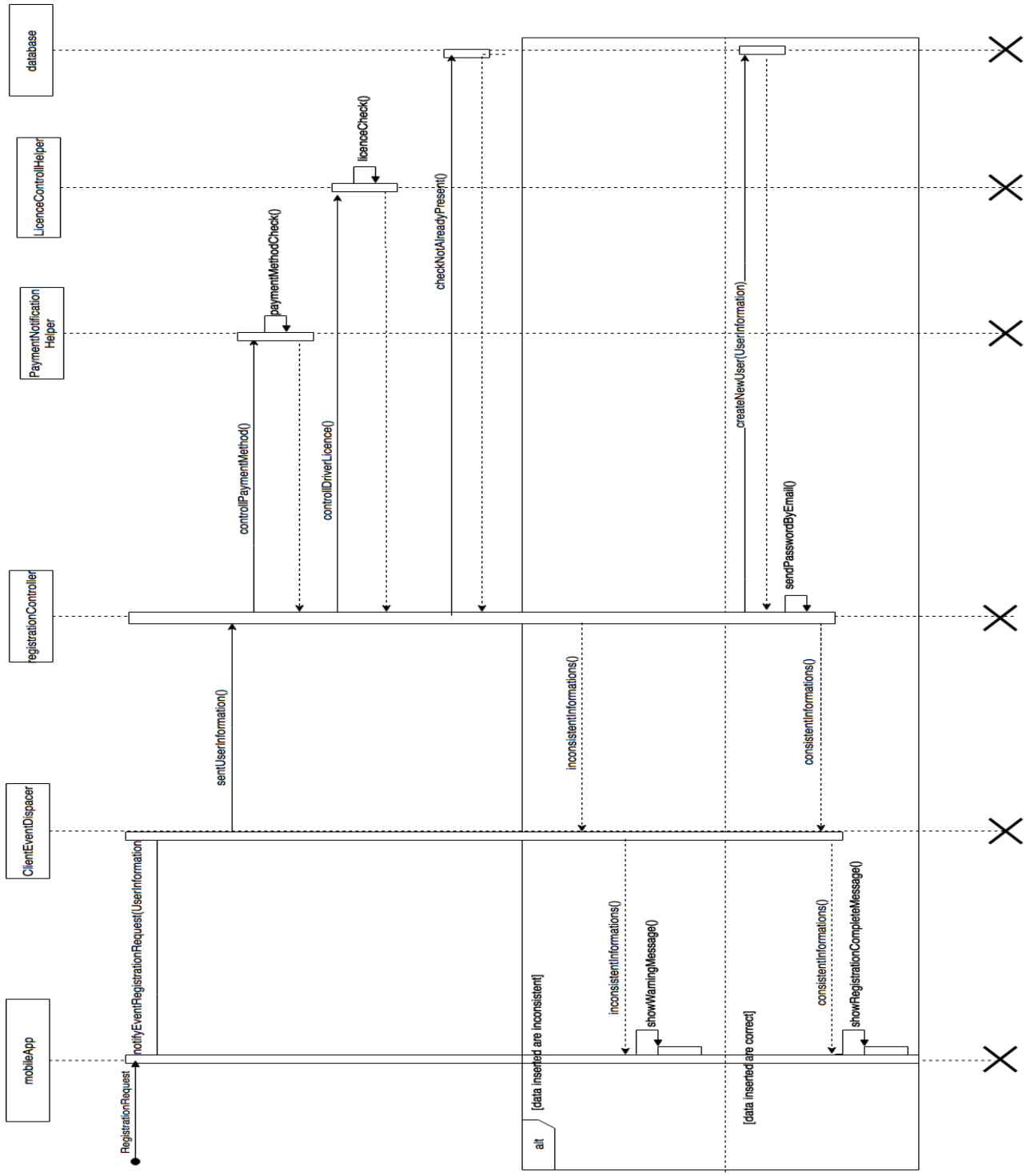

Figure 6: Registration Sequence Diagram

The sequence diagram above illustrates the registration process. The mobileApp sends to the ClientEventDispatcher the user information. Once received, the registrationController checks if the payment information and the drive license inserted by the User are correct and valid. The PaymentNotificationHelper and the LicenseControllHelper are used in this process to send the request to the external information providers.
If the information is correct the User is inserted in the database and a password is sent to his e-mail address. The mobileApp shows that the operation has been successfully completed, otherwise displays an error message.

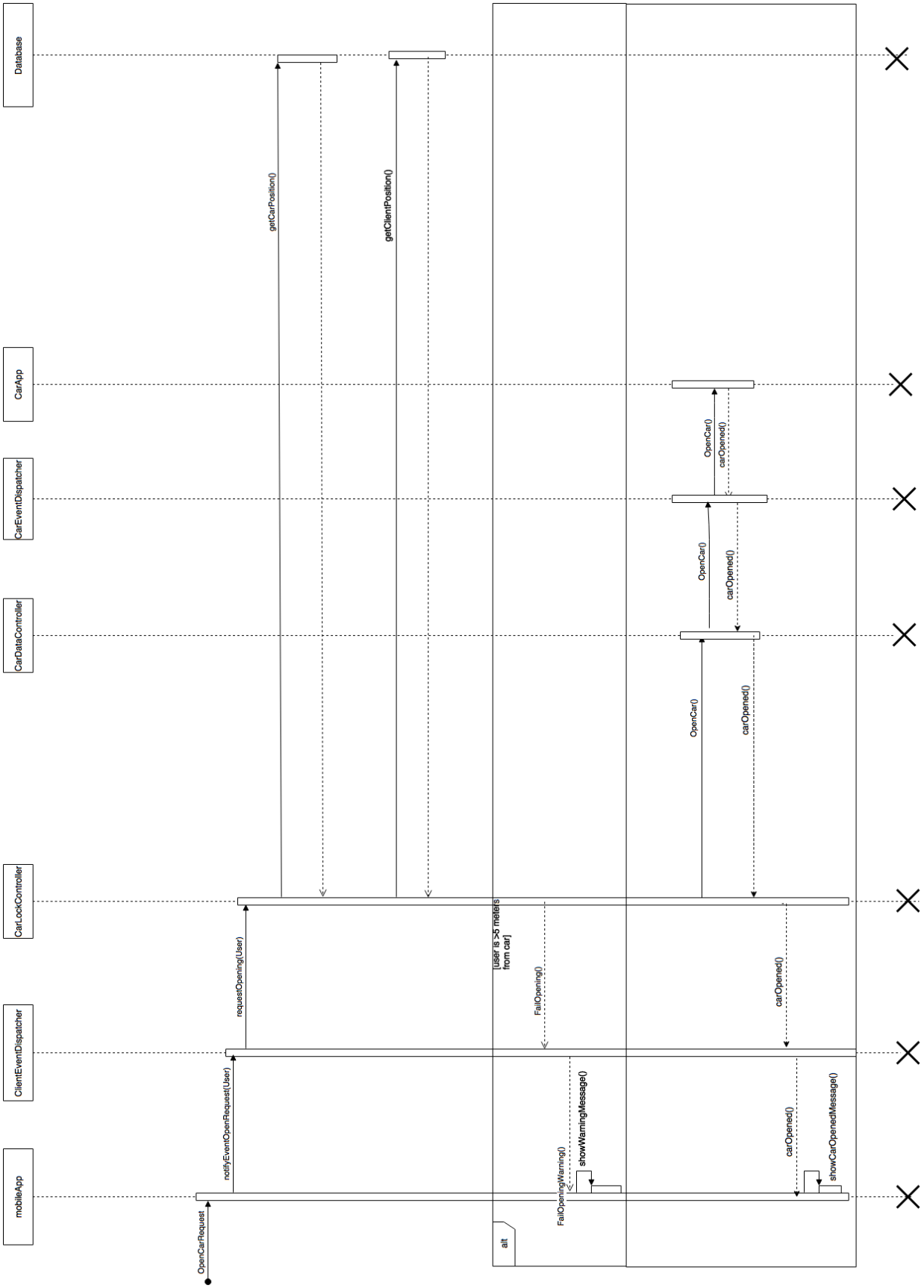

Figure 7: Car unlocking sequence diagram

The CarUnlockingSequenceDiagram illustrates how the user unlocks the car with his mobile App. Once the request gets to the carLockController, this component checks if the User's position is within 5 meters from the Car he/she has reserved. In this case the operation is successful, the carLockController sends a openMessage to the Car, while the mobileApp shows a carOpen message; otherwise, the mobileApp displays a warning Message asking the User to get closer to the Car.

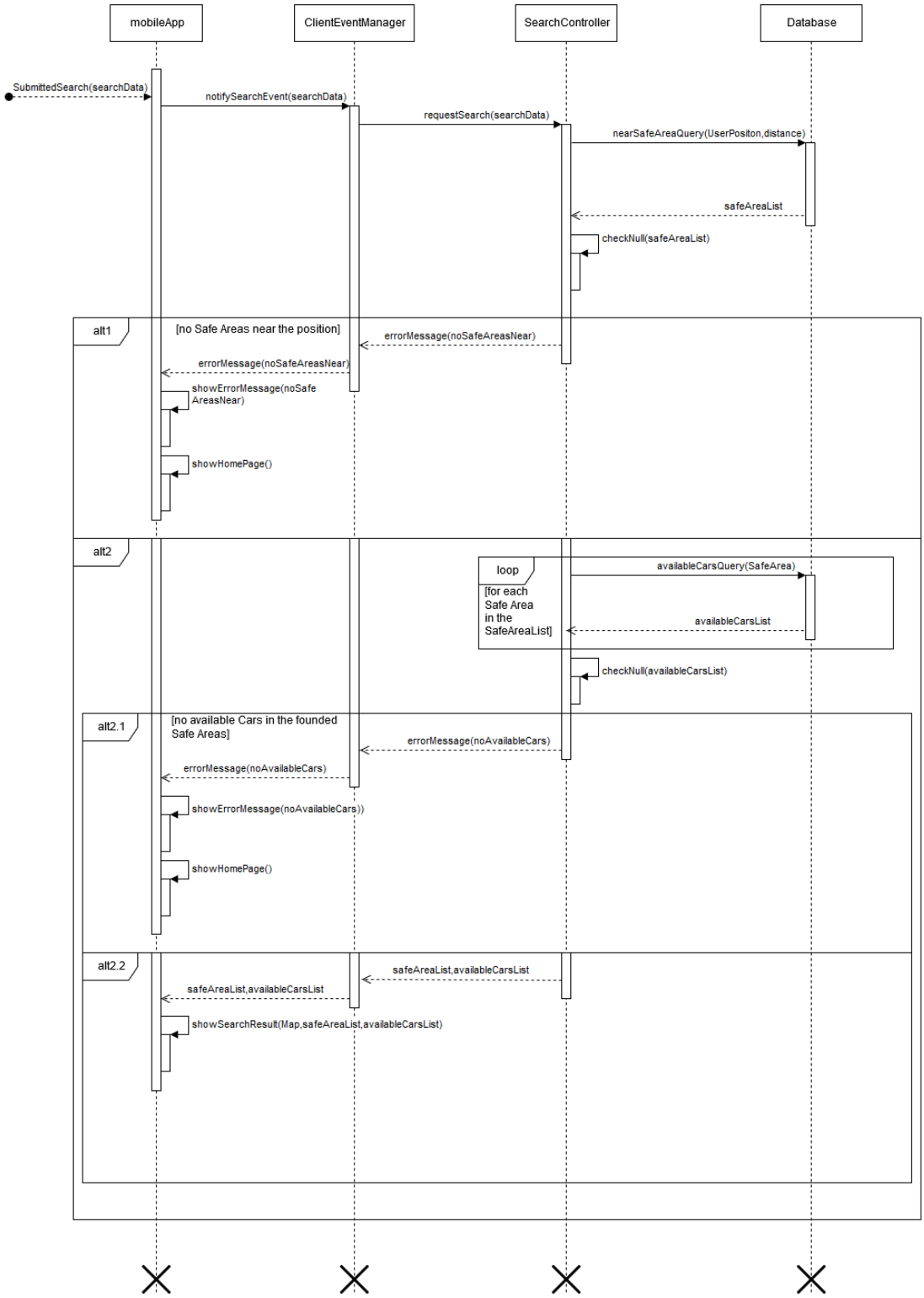

Figure 8: Search Car sequence diagram

The sequence diagram above illustrates the runtime view of the System executing a search for Available Cars.
The process starts when the User submits a search: the mobileApp notifies the event to the ClientEventDispatcher, which leaves the control of the process to the searchController.
This component is in charge of searching for SafeAreas near the GPS position inserted by the User, and for AvailableCars within the SafeAreas found. This happens through a first request to the Database for the SafeAreas near the communicated address, and then through a series of requests for the AvailableCars inside every SafeArea reported by the first query.
In case of successful search, all the searchData is sent back to the MobileApp, which presents them in a MapPage (see the Mockup section).
In both case of no SafeAreas found near the selected GPS position or no AvailableCars, an error message is sent to the mobileApp, which notifies the fact to the User through an appropriate feedback page.

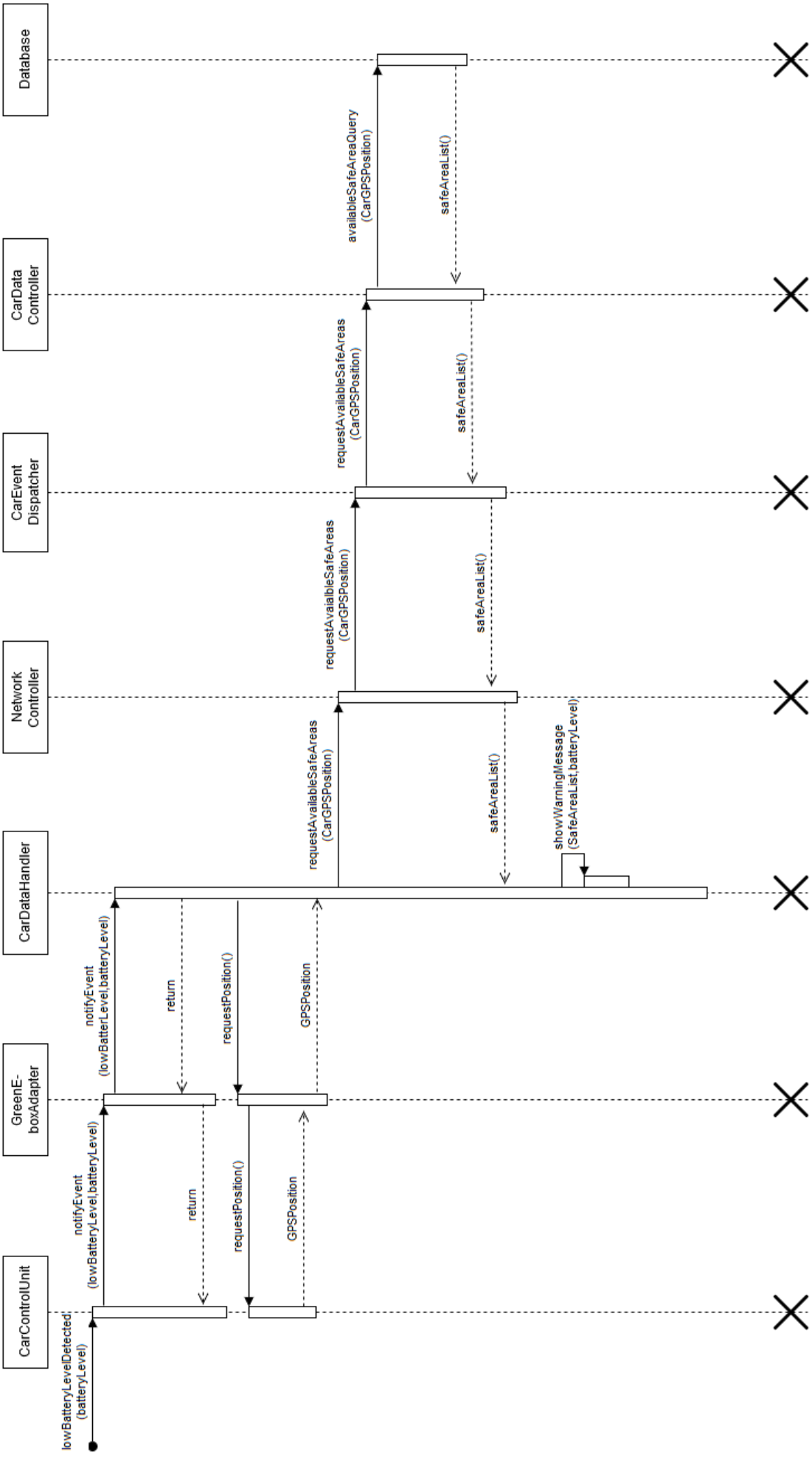

Figure 9: LowBattery Sequence Diagram

The lowBatteryEmergency Sequence Diagram shows how the emergency situations are handled by both the Car's application and Server. In this specific case, the emergency is a lowBatteryLevel detected by the CarControlUnit, which immediately notifies the event to the CarDataHandler through the Green-EboxAdapter.
The CarDataHandler then asks the Car's GPS position back to the CarControlUnit: this operation is needed to retrieve from the CarDataController (through a request sent via the networkController) the position of all the nearest Safe Areas with free power grids for plugging and recharging the Car.
Once the CarDataController returns back the lists of all the near and free Safe Areas for plugging the Car, the CarDataHandler displays them through the Car's screen in a warning message to the User (see the Mockups section).

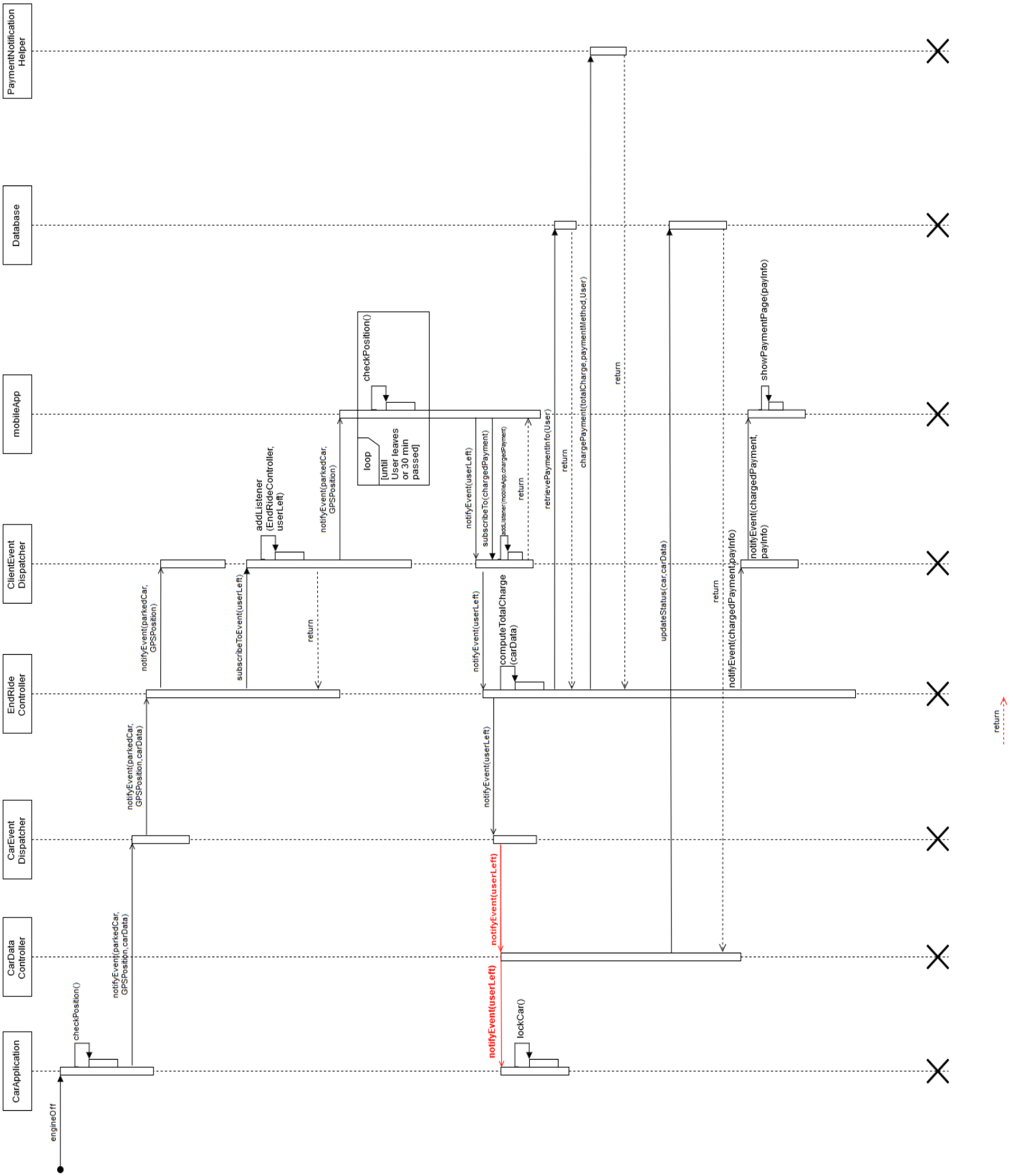

Figure 10: close Ride and Payment sequence diagram

The last sequence diagram shows the whole process of ending and closing of a Ride and Payment charging for the User's Car use.
Every time the User turns off the Car's engine, the CarApplication checks the actual Car's GPS position: if the Car is inside a SafeArea, then this component notifies the event that the Car has been parked to the CarEventDispatcher. The Dispatcher notifies the same event to the EndRideController, which in turn notifies the event to the ClientEventDispatcher: this operation is necessary in order to alert the User's mobileApp to check its GPS position and advise the EndRideController when the User leaves the SafeArea in which he/she has parked the Car.
In fact, the EndRideController subscribes itself to the userLeft event, which is notified by the mobileApp to the ClientEventDispatcher when the check for the User position gives a positive response. If the User does not leave the SafeArea after a certain amount of time, the System considers the Ride closed anyway, so the userLeft event is notified and the closing and payment operation are performed in the same manner.
After having announced the event to the Dispatcher, the mobileApp subscribes itself to the chargedPayment event, while the Dispatcher sends all the necessary information to the EndRideController in order to close the Ride and charge the payment to the User.
In fact, the component notifies successively the userLeft event to the CarDispatcher, which contemporarily notifies it to the CarApplication and the CarDataController in order to:

- automatically lock the Car's doors (operation executed by the CarApplication);
- update the Car's status to Available for further reservations (or to other states, depending on the way the User left the Car, operation executed by the CarDataController).

Please notice that the arrows notifying the event to those two components in the sequence diagram are purposefully in a red colour to highlight the parallel dispatching of the same event to both of them.
Meanwhile, the EndRideController computes the overall charging for the User's Car use (considering the total rent time, whether the User left the Car plugged to a power grid, etc…), retrieves the User's payment information from the Database and then commits the Payment to the PaymentNotificationHelper.
After the Payment has been charged, the EndRideController notifies the event to the ClientEventDispatcher, which subsequently advises the mobileApp of the charging.
Finally, this component shows a message to the User with all the payment information.

D. Component interfaces

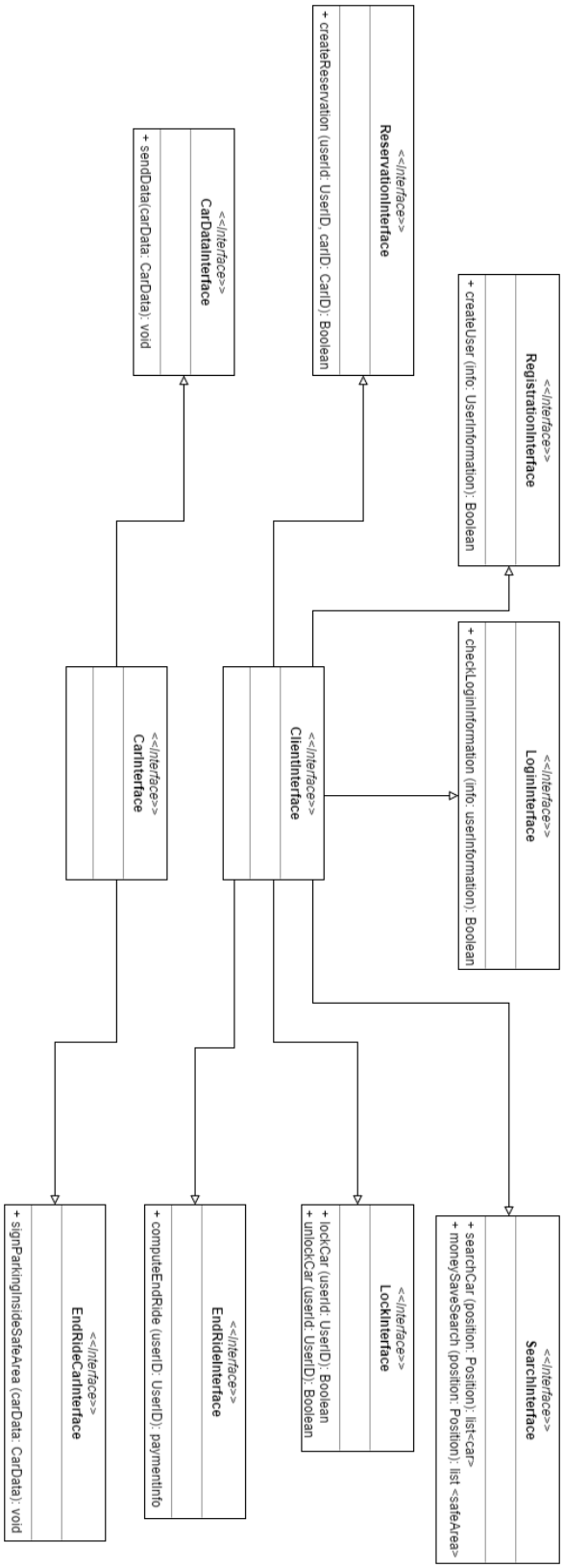

Figure 11: Component Interfaces Diagram

E. Selected architectural styles and patterns

The architecture is composed of different layer. A 3-tier logic has been used, but due to the need of not overcharging the Central Server with operations, some of the logic is hosted directly on the CarApplication.

Protocols

PDO: PHP Data Objects used by the Central Server to abstract Database Queries.
SQL: The protocol is currently used to communicate with the Database. (It is declined in order to interact with a MySQL Database).
RESTful API with JSON: used by clients (both Mobile Application and Car Application) to interact with the Central Server. API calls that need authentication are required to authenticate via HTTP basic authentication for each request. Exchanged data will be secured using SSL.
Stripe API, EUCARIS API: used by the Central Server to interacts with External Providers.

Architectural Patterns

MVC: Model-View-Controller pattern has been widely in our application.
Adapter: Adapters are also used widely to adapt Client Application and Server Application to the different APIs used to develop the System.
Client-Server: The application is strongly based on a Client-Server communication model. The Clients are the Mobile Application and Car Application. The clients are thin, thus to let the application run on low-resources devices. There is some logic inside the Car Application, but it doesn't access the Database, and manages only very simple operation with the data collected directly inside the Car.

F. Other design decisions

The application integrates different external APIs which ensure communication between the System and different External Providers. This decision makes the application easier to develop, because there is no need to develop dedicated interfaces for these External Providers.

3. ALGORITHM DESIGN

```
function calculateDiscountAndOvercharges( Car car){
        int discount=0; // percentage discount
        int overcharge=0; //percentage overcharge
        if(car.isCharging()){
                discount=30;
                return discount;
        }
         if(car.remainingBattery()>= 50%)
                discount=20;
        else if(car.HavePassengers())
                discount=10;
        if(car.remainingBattery()<20 || car.DistanceFromNearestPowerSafeArea()> 3km)
                overcharge=30;
        return discount - overcharge;
}

function moneySavingSuggestedSafeArea (Position position, Distance maximumDistance){
        List<SafeArea> nearestSafeAreas=List();
        foreach (s in SafeArea ){
                if(position.distance(s)<maximumDistance &&
                s.isChargingSafeArea &&
                s.AvaiableParks()>=1)
                        nearestSafeAreas.add(s);
        }
        if(nearestSafeAreas.size()=0)
                return NOSAFEAREAAVAILABLE;



        if(nearestSafeAreas.size()>1){
                foreach (s in nearestSafeArea ){
                        if(s.AvaiableParks()<nearestSafeArea.MaxNumOfAvaiableParks())
                                nearestSafeArea.remove(s);
                }
        }
        if(nearestSafeArea.size>=1){
                foreach (s in nearestSafeArea ){
                        if(position.distance(s)>nearestSafeArea.minDistance())
                                nearestSafeArea.remove(s);
                }
        }
        return nearestSafeArea.get(0);
}

function searchAvailableCars (Position position){
        List<SafeArea> nearSafeAreas=List();
        List<Car> reservableCars=List();
```

```
        foreach (s in SafeArea){
                if(position.distance(s) < maximumDistance &&
                s.availableCars.isEmpty() == False)
                        reservableCars.addAll(s.availableCars);
        }
   return reservableCars;
}

function reserveCar (Car selectedCar){
        foreach (s in SafeArea){
                if(s.availableCars.contains(selectedCar)){
                        s.availableCars.remove(selectedCar);
                        s.reservedCars.add(selectedCar);
                }
        }
}

function createRide (User driver, Car onRideCar){
        Ride newRide = new Ride();

        foreach (s in SafeArea){
                if (s.reservedCars.contains(onRideCar)
                        s.reservedCars.remove(onRideCar);
        }

        newRide.car=onRideCar;
        newRide.driver=driver;
        newRide.beginTime=LocalTime.now();
        newRide.beginPosition=onRideCar.getPosition();
        newRide.numberOfPassengers=onRideCar.getPassengers();
        ActiveRides.add(newRide);

        return newRide;
}

function closeRide (Ride endingRide){
        endingRide.endTime=LocalTime.now();
        int discount = calculateDiscountAndOvercharge(endingRide.car);
        float rentalCharging = (endingRide.endTime - endingRide.beginTime)*rentingFee;
        float totalCharging = rentalCharging + ((rentalCharging*discount)/100);
        Boolean committed = committPayment(totalCharging, endingRide.driver);
        if (committed)
                notifyUser(endingRide.driver);
        else
                contactProvider(error);
}
```

4. USER INTERFACE DESIGN

Mockups

Mockups of the Mobile Application have already been shown in the RASD document in Part 2. We present here some additional mockups of the Car Application, and another one of the Mobile Application, to provide a better understanding of the functionalities and information provided to the User.

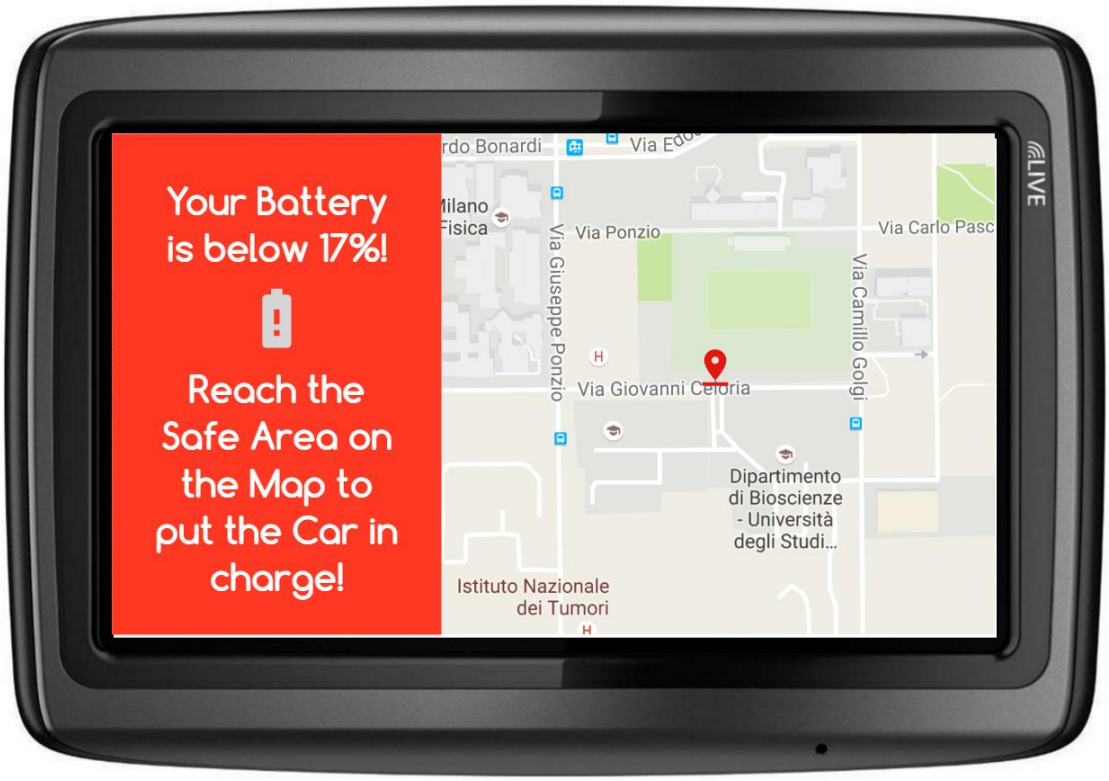

Figure 12: Low Battery Alert Mockup

Figure 13: Ride information on Car Mockup

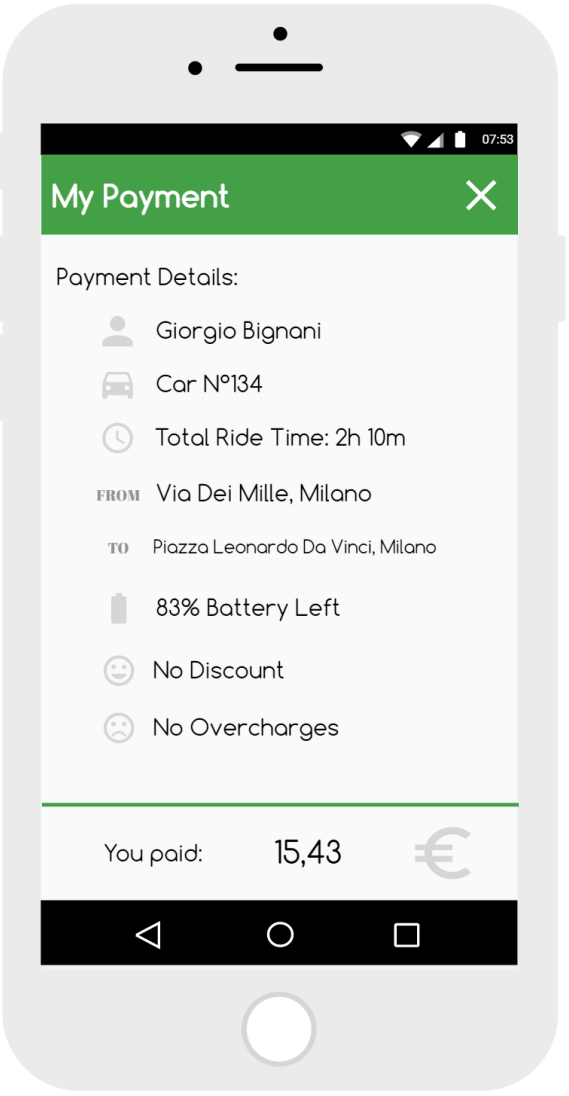

Figure 14: Payment bill screen Mockup

UX Diagram

A User Experience Diagram is inserted to better understand how the User can navigate throughout the Mobile Application. The Login and Registration pages are not shown, because they are trivial and well codified in many mobile applications.

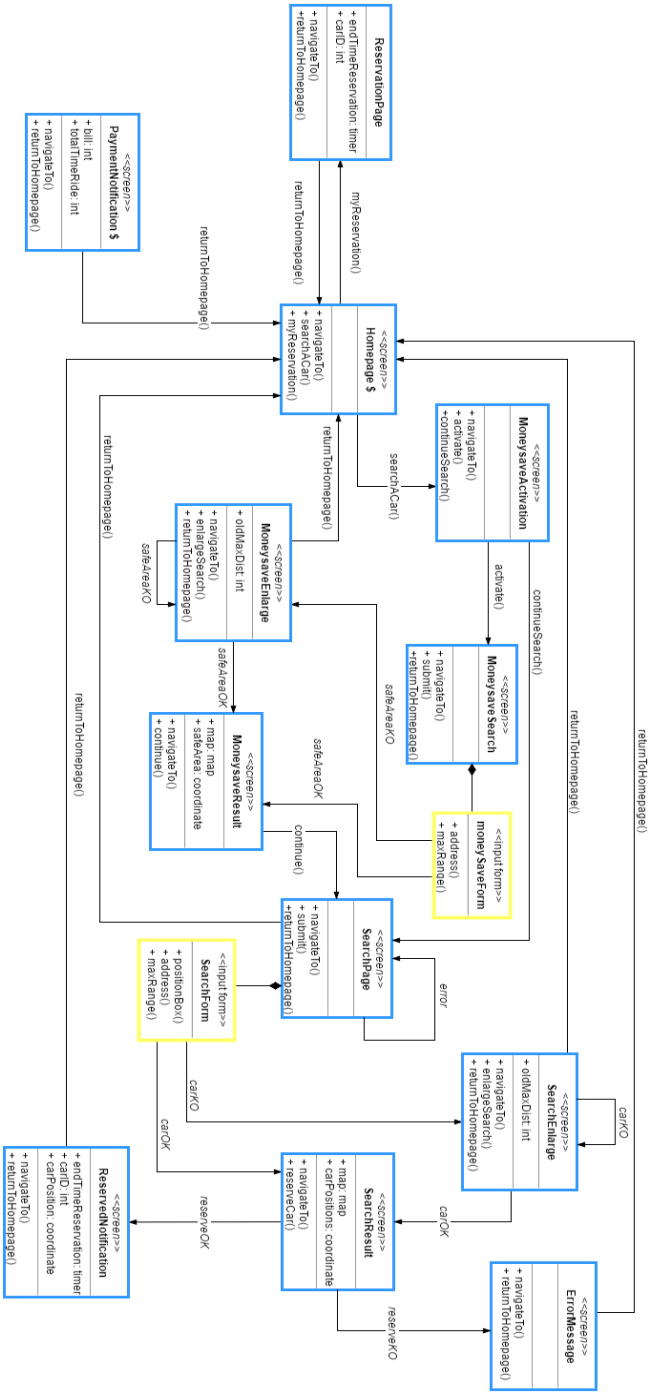

Figure 15: UX Diagram

BCE

A BCE diagram provides information on how functionalities are implemented by the various components, and how they interact with the database entities.

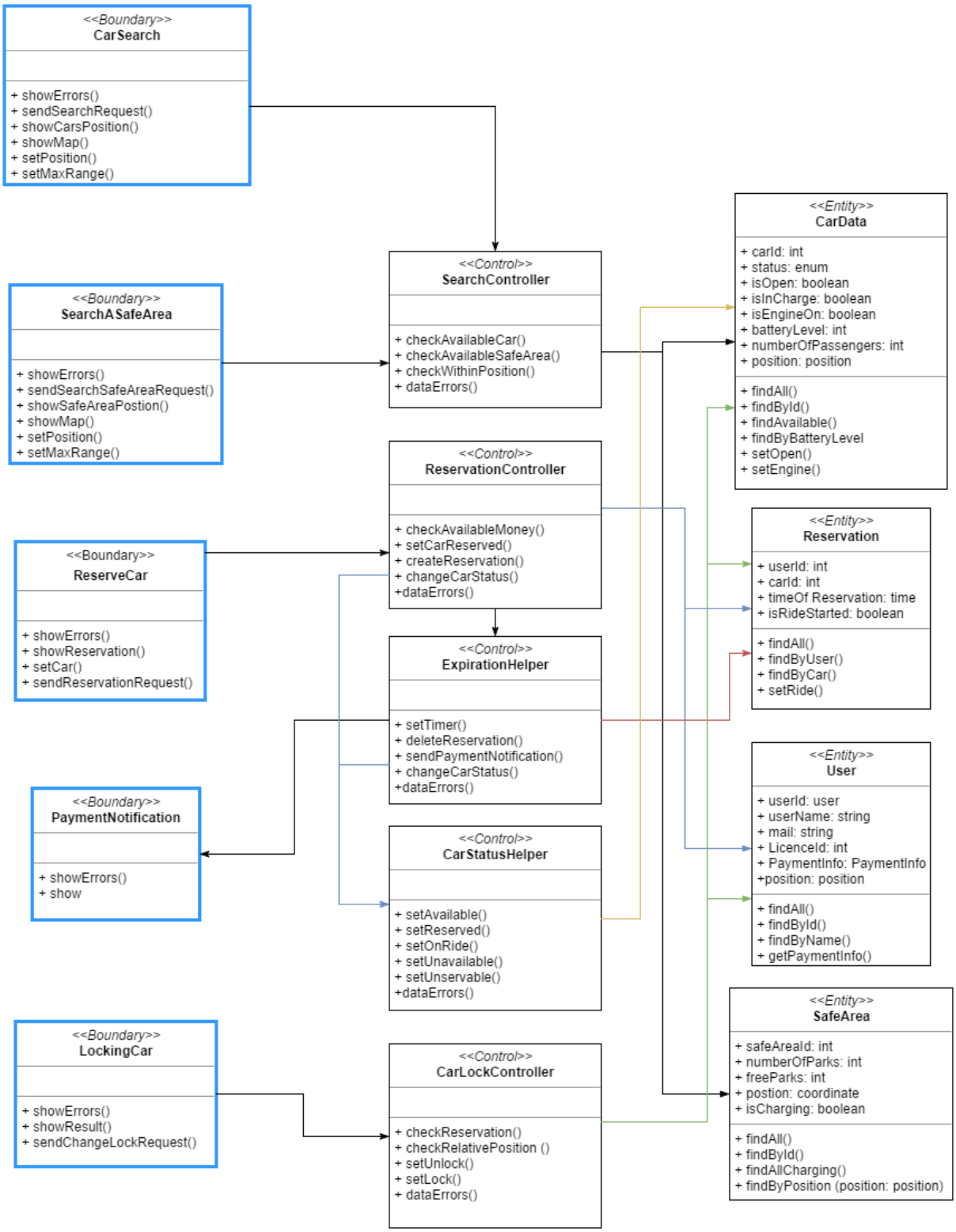

Figure 16: BCE Diagram

5.REQUIREMENTS TRACEABILITY

- **Goal1:** Users must be registered in order to access any of the Power EnJoy Services; they should provide their credentials every time they want to access the System Services.
    - RegistrationController
    - LoginController
    - LicenseControlHelper
    - PaymentNotificationHelper
- **Goal2:** Users must be able to find available cars within a certain distance from their current location or from a specified address.
    - SearchController
    - CarDataController
- **Goal3:** Users must be able to reserve one car at time, if there are any available in a specified area.
    - ReservationController
    - CarDataController
    - PaymentNotificationHelper
- **Goal4:** If a car is not picked-up within one hour from the reservation, the reservation expires and the user pays a fee of 1 EUR.
    - ExpirationHelper
    - CarDataController
    - PaymentNotificationHelper
- **Goal5:** Users should be able to save money if they leave the car in a Safe Area with Charging Station near to a destination suggested by the System; Suggested Safe Areas are selected by the System in order to ensure a uniform distribution of the cars.
    - MoneySavingHelper
- **Goal6:** Users should be able to open and access the car they have reserved; however, cars should not be opened in too much advance. From the moment Users access the Car, they should not be charged for the expiration of their reservation.
    - LockCarController
    - CarDataController
    - Locker\Unlocker
- **Goal7:** Users should be charged for the use of the Car from the moment they ignite the engine and they should be aware of how much they are spending through an interface device inside the Car.
    - CarDataController
    - CarDataHandler
    - PaymentCalculator

- **Goal8:** Users should be able to make stops during their rides. In any case, the time of these stops is still charged on the User, they can be outside a Safe Area, but the stop can last only for a determined amount of time (12 hours), otherwise they will be charged for the overall renting plus a fixed fine.
    - CarDataHandler
    - CarDataController
    - EndRideController
    - EmergencyHelper
    - PaymentNotificationHelper
- **Goal9:** Users should be supported in case of accidents and empty battery.
    - EmergencyHelper
    - CarDataController
    - CarDataHandler
- **Goal10:** Users should be aware of the level of charge of the Car's battery, in order to stop to the nearest Safe Area and plug the Car to a power station, and should pay a fixed fee if he leaves the battery empty.
    - CarDataHandler
    - CarDataController
    - PaymentNotificationHelper
- **Goal11:** Charging should be stopped from the moment Users park the Car in a Safe Area and shut down the engine.
    - PaymentCalculator
    - CarDataHandler
    - EndRideController
- **Goal12:** Users should be charged after the end of their ride. They should be advised of all the information about their ride and the payment in a short time after they have left the Car.
    - EndRideController
    - PaymentNotificationHelper
- **Goal13:** Cars automatically lock when the users leave them in the safe area. Then, they are available for other reservations.
    - CarDataController
    - EndRideController
- **Goal14:** At the end of every ride, users are granted with charge discounts or more fees: 10% discount for users who offer a ride to more than two passengers; 20% discount for users who leave the car with no more than 50% of the battery empty; 30% discount for users who park the car in a Safe Area with a power grid station, and take care of recharging the battery. If the user instead leaves the car at more than 3KM from the nearest power station, or the battery empty for more than the 80%, it should be charged of another 30% on the total bill.
    - EndRideController

6. EFFORT SPENT

	Cannas	Castiglioni	Loiacono
18.11	0	1.30	1.30
19.11	1	0	3
21.11	1	0	4
22.11	1	2	2
27.11	1	0	4
28.11	2		
29.11	1	0	2
30.11	4	0	0
03.12	0	2	0
05.12	3		
Sum	14	10	16.30