Design Document

Document Version 3.3

Matteo Frosi (mat. 875393)
Luca Costa (mat. 808109)

Contents

1 Introduction

1.1 Purpose

This document, addressed mainly to developers, aims to go into further details regarding the specification of the project, described in the RASD. Here more technical features will be described, including:

- The high level architecture
- Some possible design patterns
- The description of the main components and their interaction (Runtime View)
- A brief description of the algorithms on which the software relies
- A more detailed overview of the user interfaces

Although these are the main topics of the document, other minor details will be touched and discussed, such as the architectural style and a brief comparison with other styles and the mapping between the requirements defined in the RASD and the designed elements described in this document.

1.2 Scope

PowerEnJoy is an electrical car sharing service, based on a mobile application. The targets of the service, intended as users, are people that needs to move from a place to another within a city and requires a conveyance to move (because they don't have their own or simply can't use it).
A user can make a reservation for a car, using the mobile app and his/her account, and check for the availability and status of all the cars within his/her position, identified using GPS localization, or a specific one, inserted manually by the user. As stated before, to access the service, the user must possess a private account, so a registration is needed.
The system provides the users a safe way (identification code) to access the cars, and the riding service and keeps trace of the status of all the cars.
Moreover, the system prizes or punishes a respectively good or bad behavior from the users, applying a discount or an overcharge on the cost of a ride. As example, if the user leaves the car without much battery, he/she will have to pay more than the standard cost of the ride, because the car will need to be charged and this operation has a cost. On the other hand, if a user plugs the car before ending the service, it receives a discount.
The system includes other functionality, such as GPS based maps available in every car, an emergency procedure in case an accident occur during a ride and the notification of a car status if the user requested it.

1.3 Definitions, acronyms, abbreviations

- RASD: document about the requirements analysis of the project.
- DD: document about the design choices and the components description of the project.
- GPS: global navigation satellite system that provides location and time information in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.

- SMS: short message service; it is a notification sent to a mobile phone, we need a GSM gateway to use it.
- GMS gateway: device that allows SMS text messages to be sent and/or received by email, from Web pages or from other software applications by acquiring a unique identifier from the mobile phone's Subscriber Identity Module, or SIM card.
- Push notification/ push message: it is a notification sent to a smartphone using the mobile application.
- API: application programming interface; it is a common way to communicate with another system.
- REST: representational state transfer, it's one way of providing interoperability between computer systems on the Internet.
- RESTFul: REST with no session.
- UX: user experience design
- URL: uniform resource locator.
- MVC: model view controller, it's a design pattern.
- SOA: service oriented architecture, it's a style of software design where services are provided to the other components by application components, through a communication protocol over a network.
- Layer: way of organizing code in sections sharing a common goal. The highest partition includes Presentation Layer, Business or Logic Layer and Data Layer.
- Tier: physical deployment of layers.
- Safe area: it is a specific area where the electric cars of PowerEnJoy service can park. The set of safe areas is predefined and owned by the company/society that requested the management system for the service.
- Special safe area: it is a safe area where power grid stations are installed.
- Power grid station: it is an installation that allows the recharge of an electric car.
- Communication primitives: set of instructions and procedure that allow a communication between machines and devices over a network.
- Sensor data retrieval: procedure that consists in getting all the information collected by the sensors of the car.
- ECU: electronic control unit of a car.
- VI: vehicle interface.
- OBD-II: it is a sort of computer which monitors emissions, mileage, speed, and other useful data.
- CAN bus: the controller area network bus is a system that allows the interaction between a processor and the other peripherals of the device in which it is (usually, a vehicle).

1.4 Document structure

- Introduction: this section introduces the design document. It gives an overview on what topics will be covered and what aspects described in the RASD will be improved here.
- Architecture Design: this section is divided into different parts, each describing an aspect of the software design.
  - Overview: brief description of the division in tiers and layers of the application.
  - Overview of the car system: brief description of the interaction between car hardware (electronics) and software
  - Components and interaction: high level and detailed view of the components of the application and the way they communicate.
  - Deployment: view of the physical devices and the way they contains components
  - Runtime diagrams: describe the interaction between the components of the system, using some cases described in the RASD, under the use case description section (which also refers to some scenarios).
  - Component interfaces: describe the interaction between the interfaces of the system.
  - Architectural styles and patterns: describes in detail the architecture chosen for our software and the structural patterns supporting it.
- Algorithm Design: this section describes some of the algorithms that the application will rely on. To focus on the algorithm idea and not the fine grained implementation, each one will first be described using pseudo-code and only then will be shown using real code.
- User Interface Design: this section presents mockups and user experience explained via UX diagrams.
- Requirements Traceability: this section aims to explain how the decisions taken in the RASD are linked to design elements.
- Document further description: containing useful information about changelog, the references and the used tools.

2 Architecture design

2.1 Overview

2.1.1 General description

PowerEnJoy relies on a three tier architecture. Referring to the proposed system architecture presented in section 2.3 of the RASD, the following figure represents the tier division of the system.

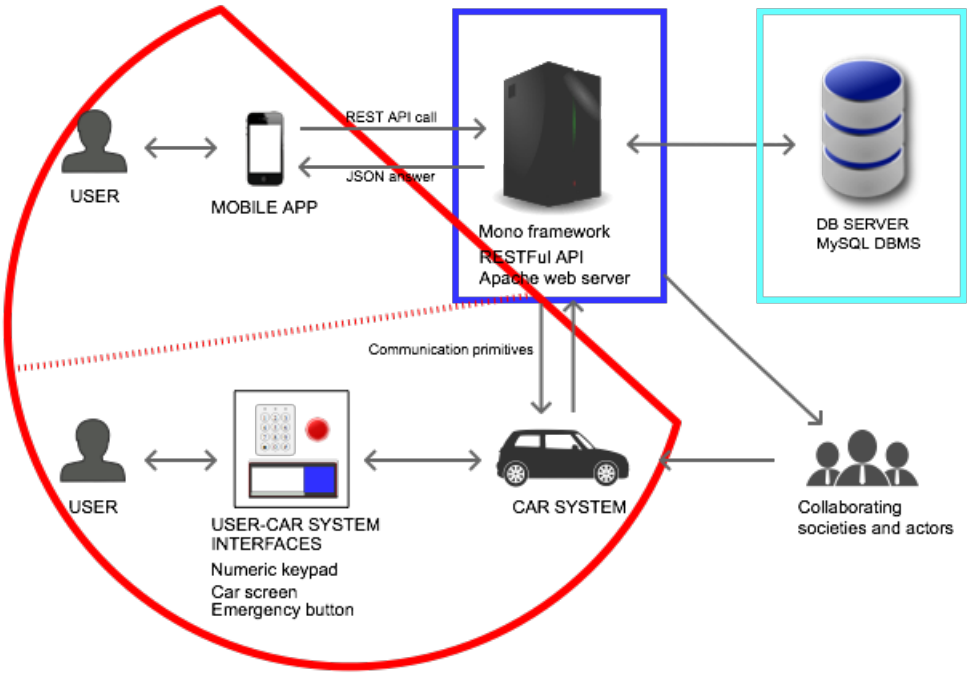


Figure 1: Proposed architecture, tier division - red: client, blue: server, azure: database

The client side includes all the possible ways by which a user can interact with the server (and the service), that are the mobile application and the car itself. The first represents the classical idea of client tier, including a GUI and a minimal amount of processing power that allows the user to have a connection with the rest of the tier, shown in a graphical way, that is more user friendly. The latter is, instead, a particular client tier that acts as a mini server, because it contains more logic than the mobile application. Here there is not a real graphic interface but multiple mechanical interfaces (MUI) can be detected, such as the numeric keypad that make the user able to unlock the car or end the service, the emergency button and the car screen, that is thought more towards an informative interface than an interactive interface. To better understand this concept, follow the same figure, this time divided in high level layers.

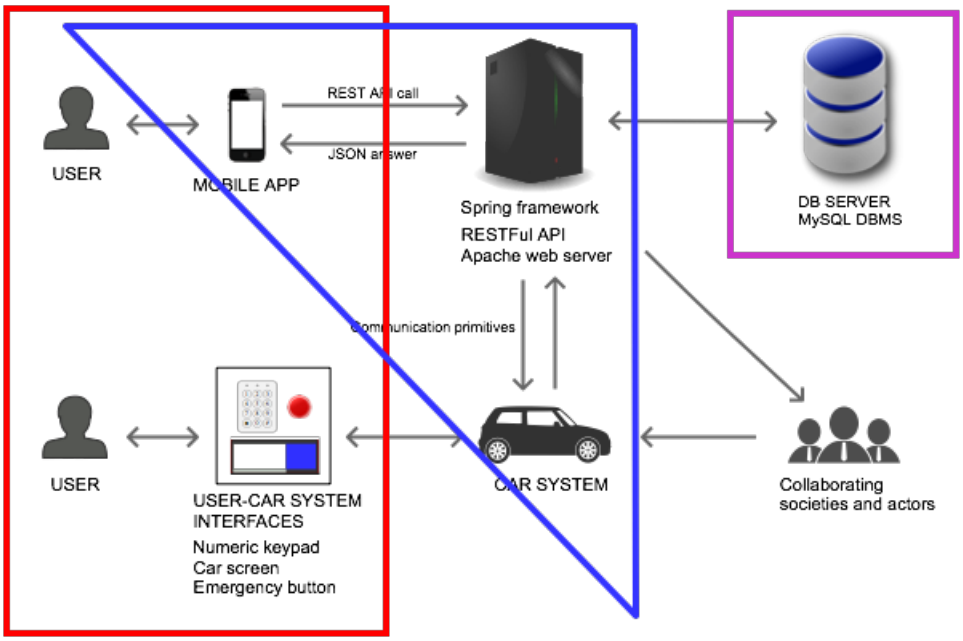


Figure 2: Proposed architecture, layer division - red: presentation, blue: logic/business, purple: data

The mobile application is also part of the logic layer because it can hold some immediate processes, instead of delegating them to the main server. An immediate example could be the elaboration of the GPS based map during the research of an available car.

2.1.2 Layers

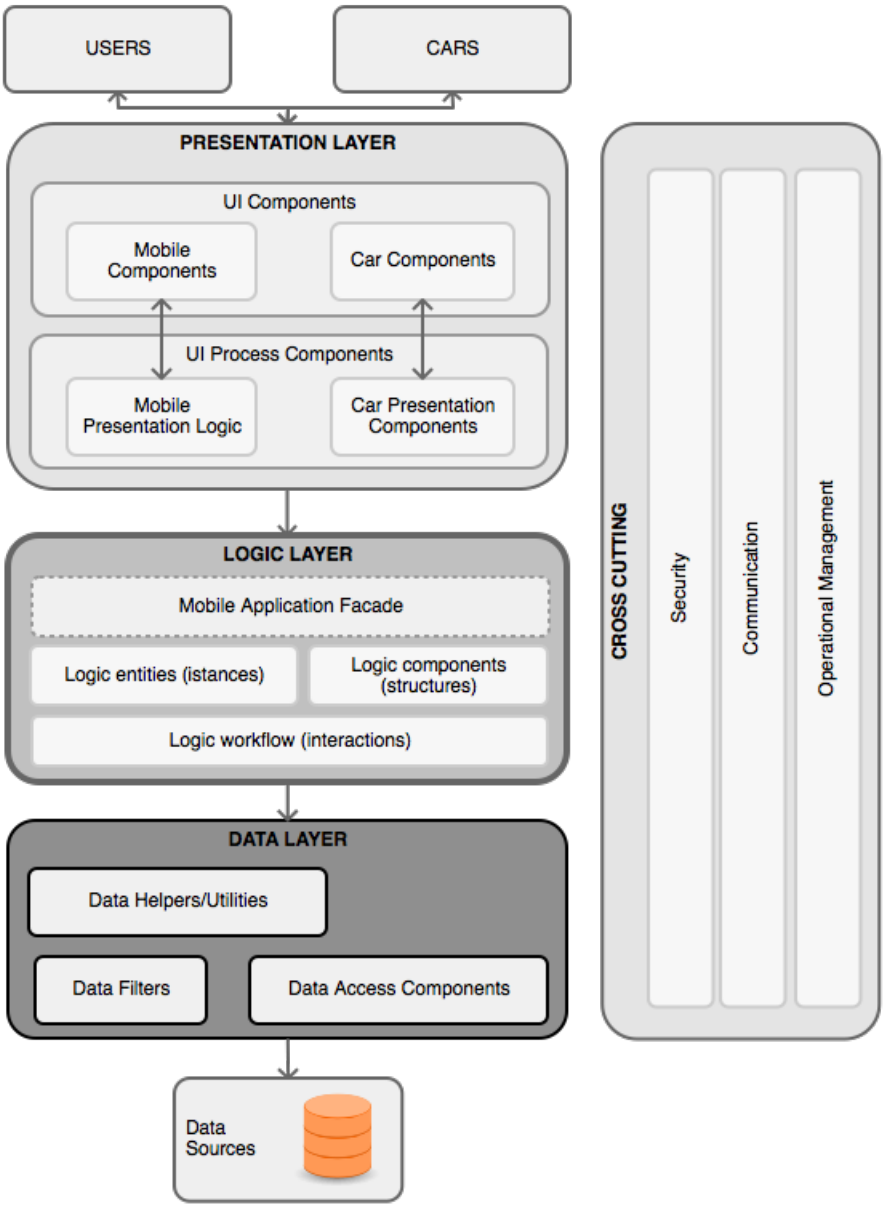
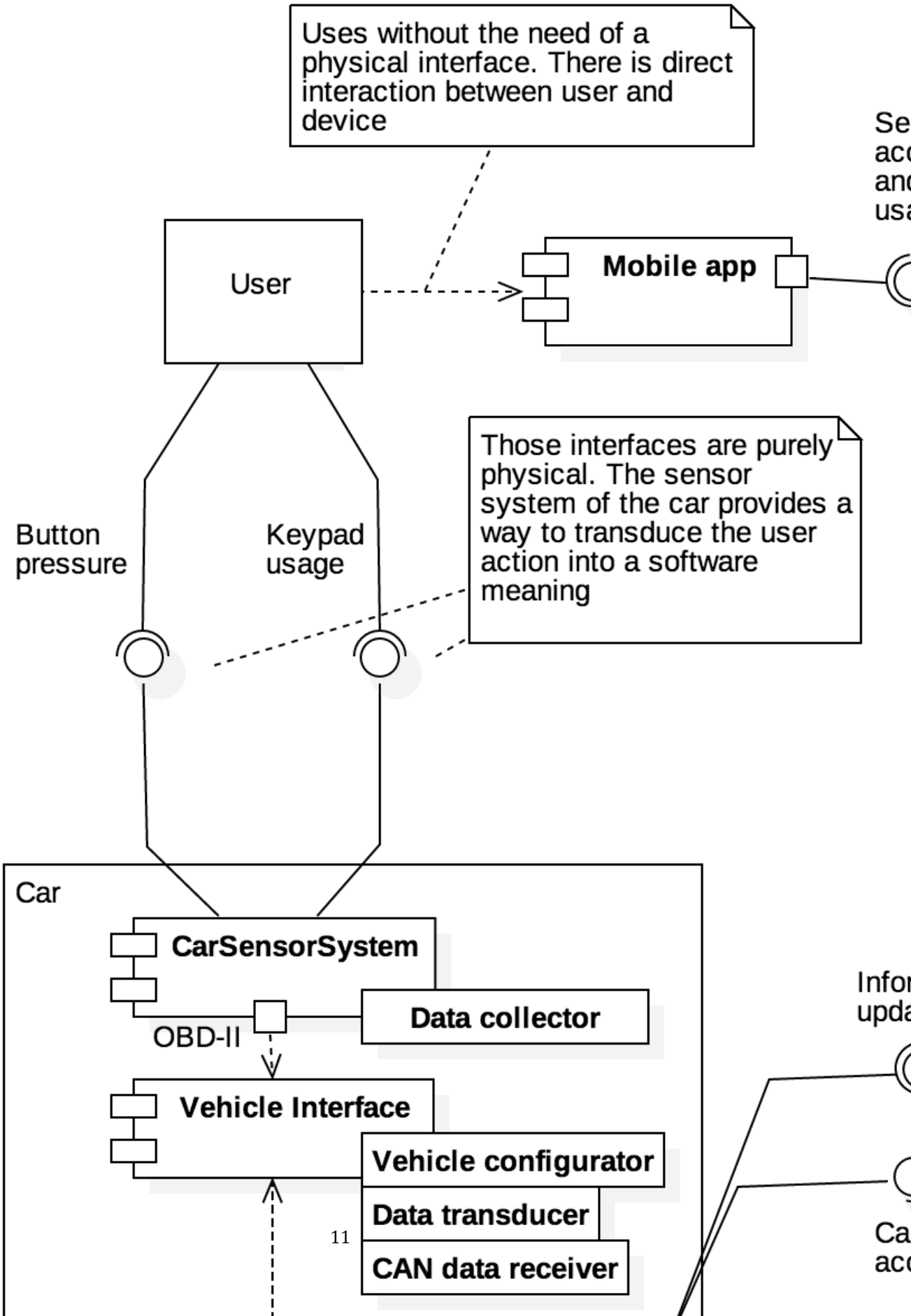


Figure 3: Layer representation

The figure is quite self explanatory but it is worth to spend some words about each layer.
The presentation layer can be found in the mobile application and in each car, however with a major difference. In the car the presentation layer consists of the devices that allows an interaction with the user, that are not just the screen and the way the information is displayed to the user. Obviously, the car screen works in the same way, elaborating and showing the information of the car and the ride, but other objects are already "presenting" something useful for the user: the numeric keypad to insert the access code and the emergency button.
The logic layer is divided into two minor parts, that are located on the mobile device and on the car, and a major section, that is on the server(s). The first parts have the goal to elaborate small chunk of information (car data) or to simply remodeling the view presented to the user (mobile app and car screen) , while the latter represents the real core of the software, where all the decisions and interactions are taken.
Lastly, the data layer involves the data filtering and interaction with the server, when requested.
It is noticeable that all the layers share critical components, mostly regarding the security and communication of the software.

2.2 Overview of the car system

It should be useful to spend some words to discuss more in detail the structure of the car and the way the user can interact with it, and consequentially with the server. Using multiple sensors, information about the car status can be retrieved, such as battery charge, number of passengers, degradation level of the car components, or even the localization data, obtained with GPS. Such data pool is interfaced with the car system, that resembles the shape of an application. The car app can be considered as a cluster of procedures and interfaces that allows user, car hardware and system to communicate and "know" about each other. Speaking in pattern terms, the car app emulates the controller entity in the MVC pattern, even if only partially. As said in the previous subsection, while the emergency button and the numeric keypad make the user interact with the car in an active way, the car screen acts as a passive element and is simply needed to inform the user of the car status and the service info. We can consider such feature as a way to diminish the interaction user-system to a very strict level, without changing the user perspective of the service. Such reduction lightens the logic needed in the car, making the project more achievable from a budget and complexity point of view.


Uses without the need of a physical interface. There is direct interaction between user and device
User
Mobile app
Button pressure
Keypad usage
Those interfaces are purely physical. The sensor system of the car provides a way to transduce the user action into a software meaning
Car
CarSensorSystem
Data collector
OBD-II
Vehicle Interface
Vehicle configurator
Data transducer
CAN data receiver

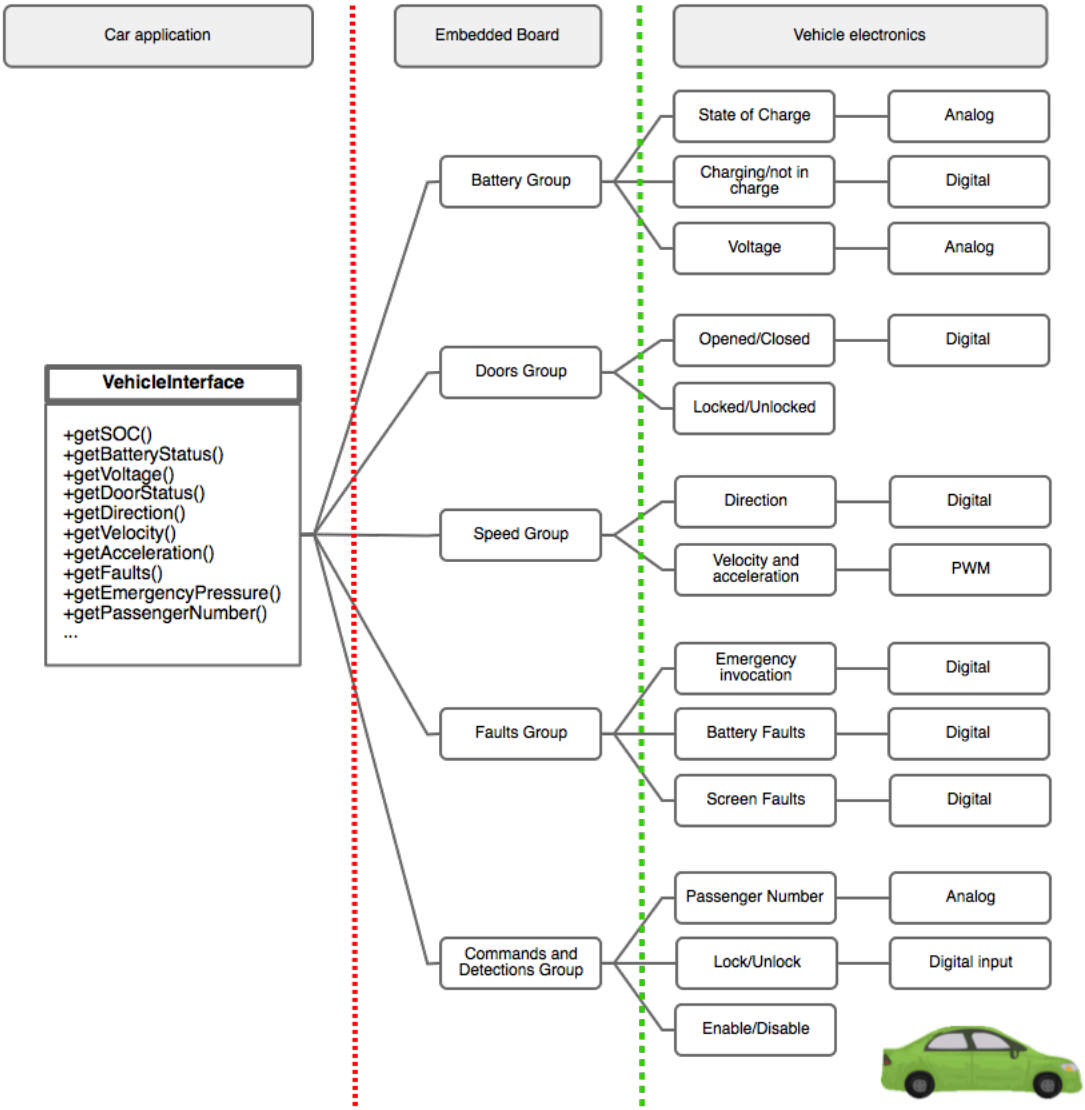


Figure 5: Car structure, including electronics

2.3 Components of the system

2.3.1 High level view

From an high point of view, the system can be divided into four elements. The users can interact with the server, that is the main central where all the logic is concentrated, directly or with the support of the car. A user can initiate the interaction with the server using his/her mobile application. To proceed in using the service, the user must first be logged into the system and for every action he/she does (monitoring request, reservation request, take back of a reservation, code request and so on), he/she has to wait for the server response, informing the user that the request has been successful or not. Because of this sequentiality of actions, the communication directed from user to server has to follow synchronous rules when proceeding in a certain sequence and leave all the other non-interactive options as to be implemented in an asynchronous way. As example the user shouldn't be prohibited to search for an available car even if he/she already made a reservation, but he/she cannot made a code request if there is no reservation

made. A clever way to implement the user-server relationship is to made all the communications asynchronous but thanks to particular patterns, MVC among them, denying the usage of particular services if certain conditions are not met.
The user can also interact indirectly with the server using the car - or it can be said that the car interacts with the server -. Some physical interfaces, such as the emergency button or the numeric keypad, allow the user to modify the condition of a car and, consequentially, of his/her service.
The server/central acts mainly as an elaborator, or referring to the chosen architecture, as a monolithic logic processor. Its job is to coordinate the requests from the users and the change of status of all the cars. As said before, the interaction between user and server can be described with a simple MVC pattern. From a very high view, the user makes an action, communicates it to the server, the server elaborates the request and confirms/rejects the will of the user. The server-car-user relationship is a bit more complex. Firstly, there is a continuous interaction between every car and the server, that follows this sequence:

1. Data retrieval
2. Car status update (car to server)
3. Database update (server to database)
4. [Optional] Car update (server to car), as example the lock following the end of the service

Moreover, to this sequence must be added the interaction with the user, that is both passive and active as said in the previous section.
The server communicates with all the other components using asynchronous messages, because it does not depend on their status but acts as the independent core of the software. Just referring to the user behavior, it would be unthinkable to have the server wait for a user decision, stated that the thinktime can be to long, and each communication would become a bottleneck for the whole system.

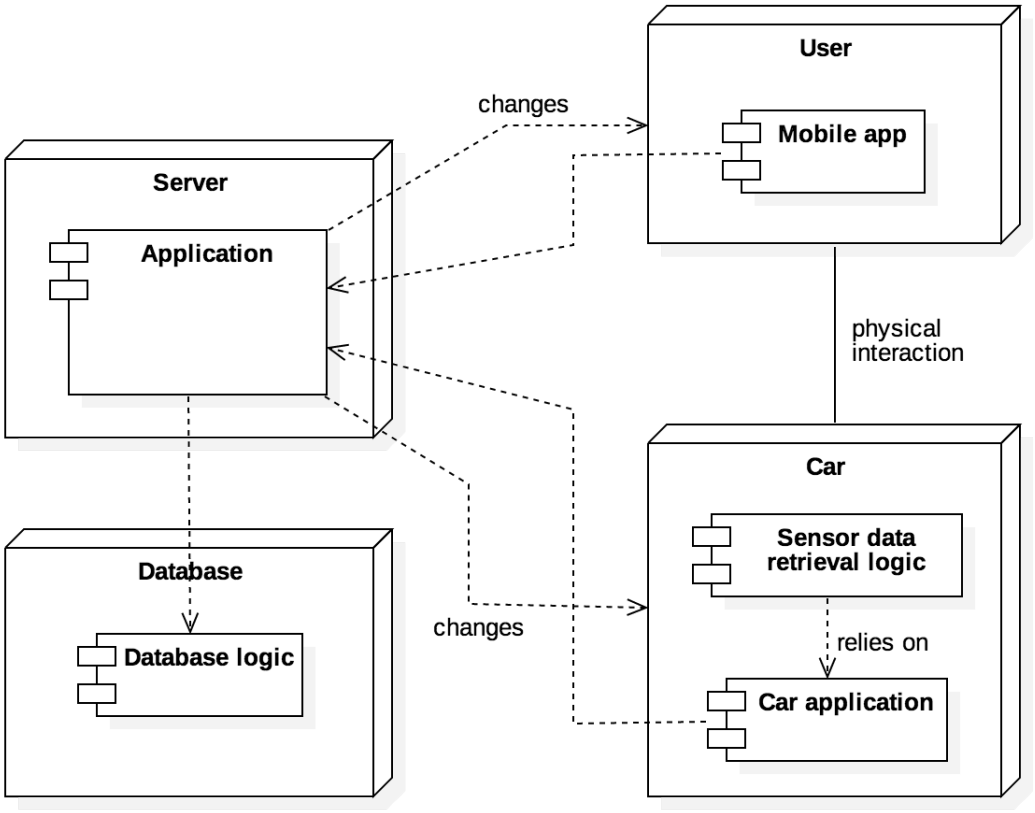


Figure 6: High level view of the components of the system

2.3.2 Detailed views

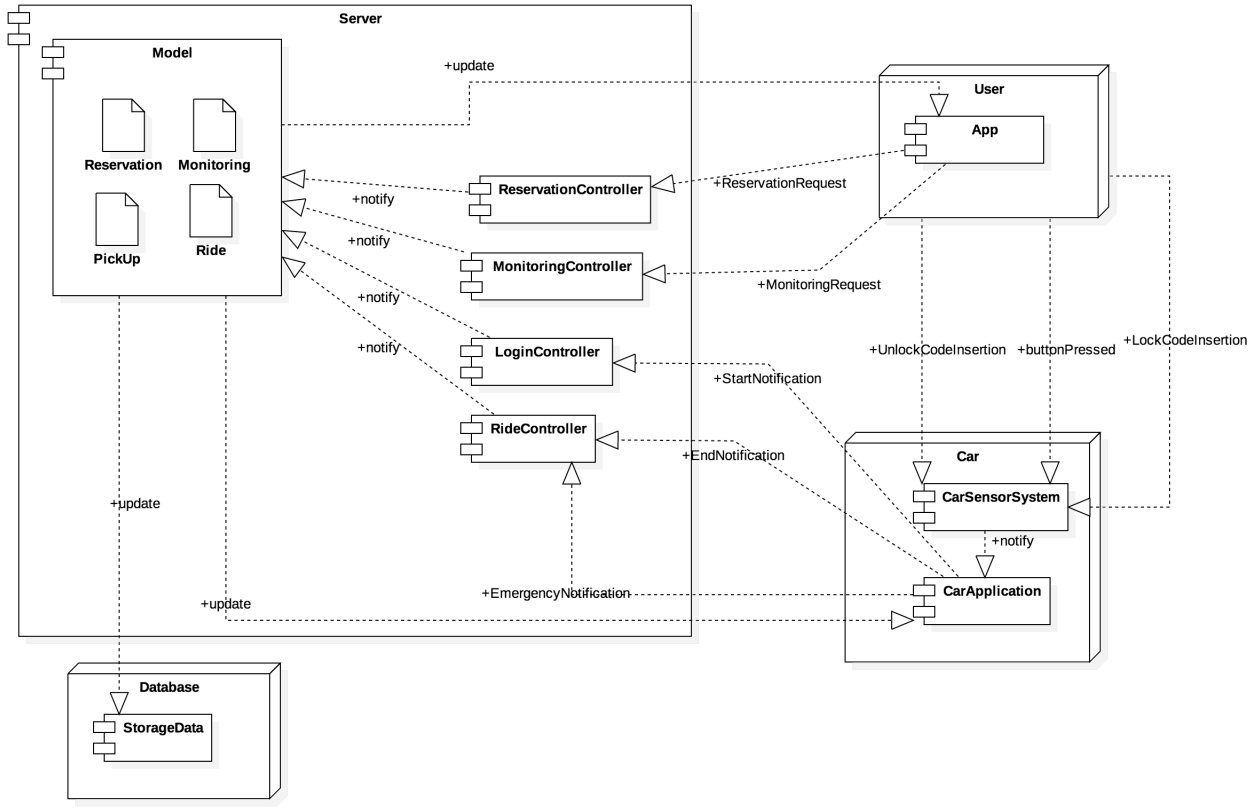


Figure 7: Deeper high level component diagram

Deepening the high level component diagram, each big node can be fragmented into smaller parts, each characterized by its functionality. As previously said, The user can interact directly with the main server/central using his/her mobile application, or, indirectly, using some of the car physical devices (the button and the numeric keypad). Focusing more on the server composition, multiple components to handle and manage every part of the service are needed. Purely from a topological point of view, the server consists of two big areas. One refers to all the structures and components that handles and addresses incoming data from and to the external world, while the other refers to all the functionality and procedures that allow the elaboration of an event and the consequent updates on the server itself, but also on the external world (database, clients or cars).

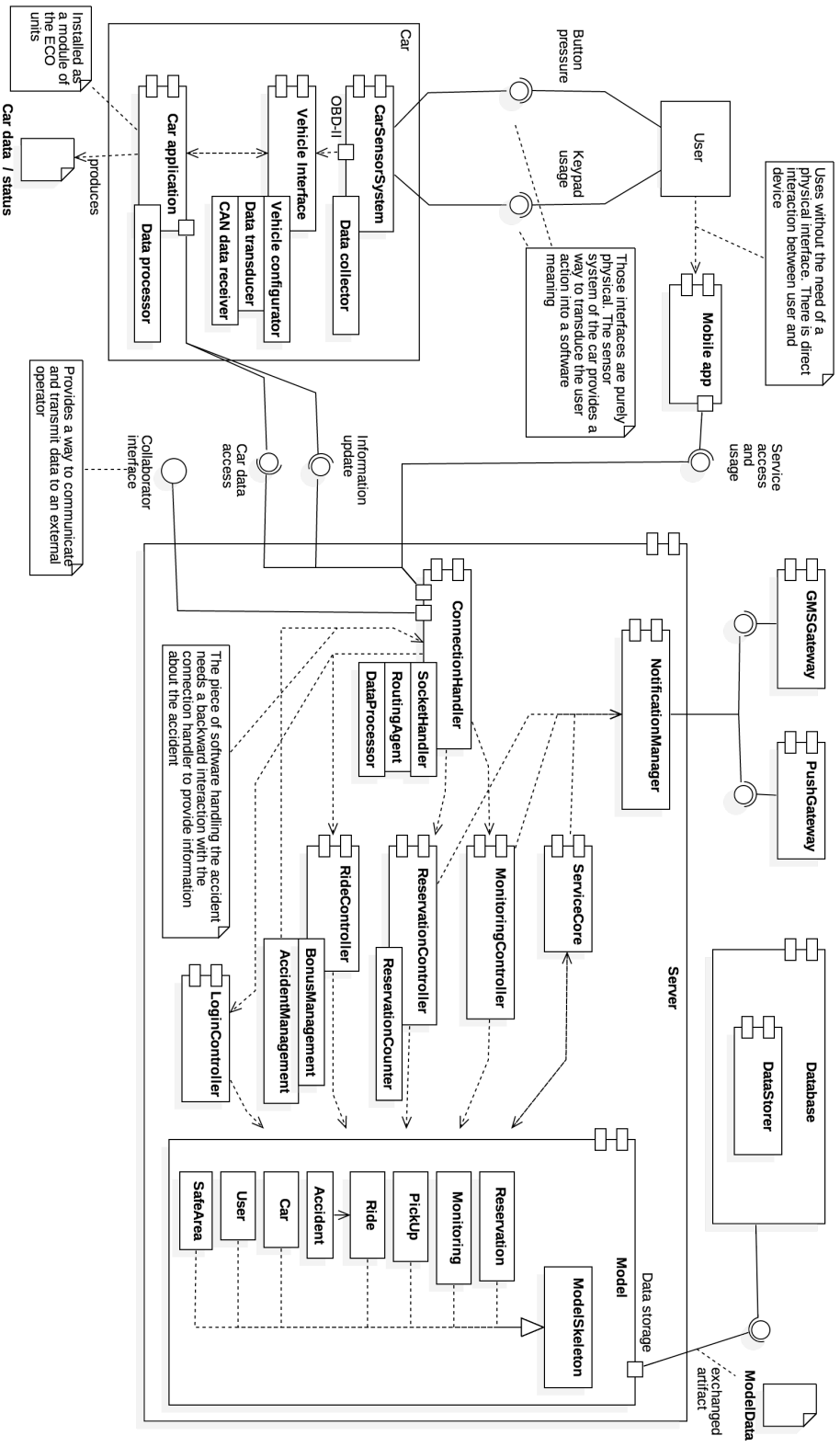


Figure 8: Detailed component diagram

Figure 8 represents an even more detailed component diagram, including sub components, interfaces, products (artifacts) and ports. One of the two area previously described can be seen as a mail office. The component ConnectionHandler, as the name suggests, handles the incoming and outgoing data, sorting it to the specific controller. The other area, composed by the Model and the ServiceCore interacts with the Database, providing it the information about the Model itself. The NotificationManager component takes care of the communication and interaction needed with the user's device. In detail:

- GMSGateway: device that allows SMS text messages to be sent and/or received by email, from software applications.
- PushGateway: device(s) that manages push notifications.
- NotificationManager: component that takes care of the interaction with the user's device.
- ConnectionHandler: component that sorts incoming and outgoing messages (data transfer).
- ServiceCore: contains all the procedures supporting the software (algorithms, functions and so on).
- ModelSkeleton: representation of the analyzed world
- Vehicle Interface: allows an interaction between the sensor systems of the car and the application installed on a module of the ECUs. It contains different modules, including:
  - Vehicle configurator: depends from the car, contains information about the hardware of the vehicle, that will be used by the VI component.
  - CAN data receiver: interacts with the sensor systems of the car, connecting the VI hardware to the OBD-II port. It receives data in CAN format (figure 9, sub subsection 2.3.3).
  - Data transducer: takes the CAN data and traduces it into a readable format for the application. In this case it will be a CAN to Java transduction.
- CarApplication: contains all the procedures that elaborates the data retrieved by the sensor system of the car and interacts with the server.
- Database: device used to store the model data

Lastly, the LoginController handles all the login/logout requests and the registration event. Notice that the logout can happen voluntarily (the user selects it) or by expiration (fixed time of log in expires).

2.3.3 Car components description and considerations

After having described the components of the whole system, it's now time to discuss further how exactly do they work. This sub subsection, as the title implies, describes how the car data is retrieved and how the communication within its modules (both hardware and software) works.
An ECU is a computer with internal pre-programmed and programmable computer chips that is not much different from a home computer or laptop. The vehicle's engine computer ECU is used to operate the engine by using input sensors and output components to control all engine functions. The ECU needs inputs from vehicle sensors to compute the information using a program that has been stored in the ECU on a programmable memory chip. The ECU program will use the inputted sensor information to compute the needed output like the amount of fuel injected and when to spark the coil in order to start the engine. Some vehicles may incorporate more than one ECU into a single unit called a powertrain control module (PCM). These units can be an advantage by having more modules in one location but may be a disadvan-

tage by adding longer wires to reach the component it operates.
As stated multiple times in this document and in the RASD, the interaction between user and car will be mostly passive and there will be no real modification of the vehicle features (speed, breaks, and so on). A standard ECU will be then more than sufficient to the purpose of the thought system. However, more logic is needed, to analyze the retrieved data in site (directly on the car) and communicating it to the user by means of a screen. For this purpose an ECU-line module can be created and incorporated into the PCM of the vehicle.
But how exactly do the data retrieval components of the car and the ECU modules communicate? The vehicle interface (VI) is a device that plugs into the OBD-II port (and thus to the CAN bus), reads and translates OBD-II requests and CAN messages into a standard cross-vehicle format. The translated messages can be sent over USB or Bluetooth, so they can be read by any computer or device. The host device connects to the vehicle interface and reads the translated vehicle data.
The idea is then to connect, using an inner OBD-II port, to a vehicle interface and then send the data directly to the car application, to elaborate it.

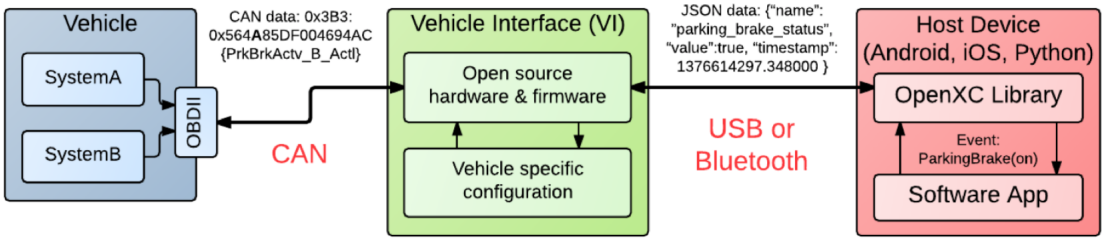


Figure 9: Data communication concept

2.4 Deployment

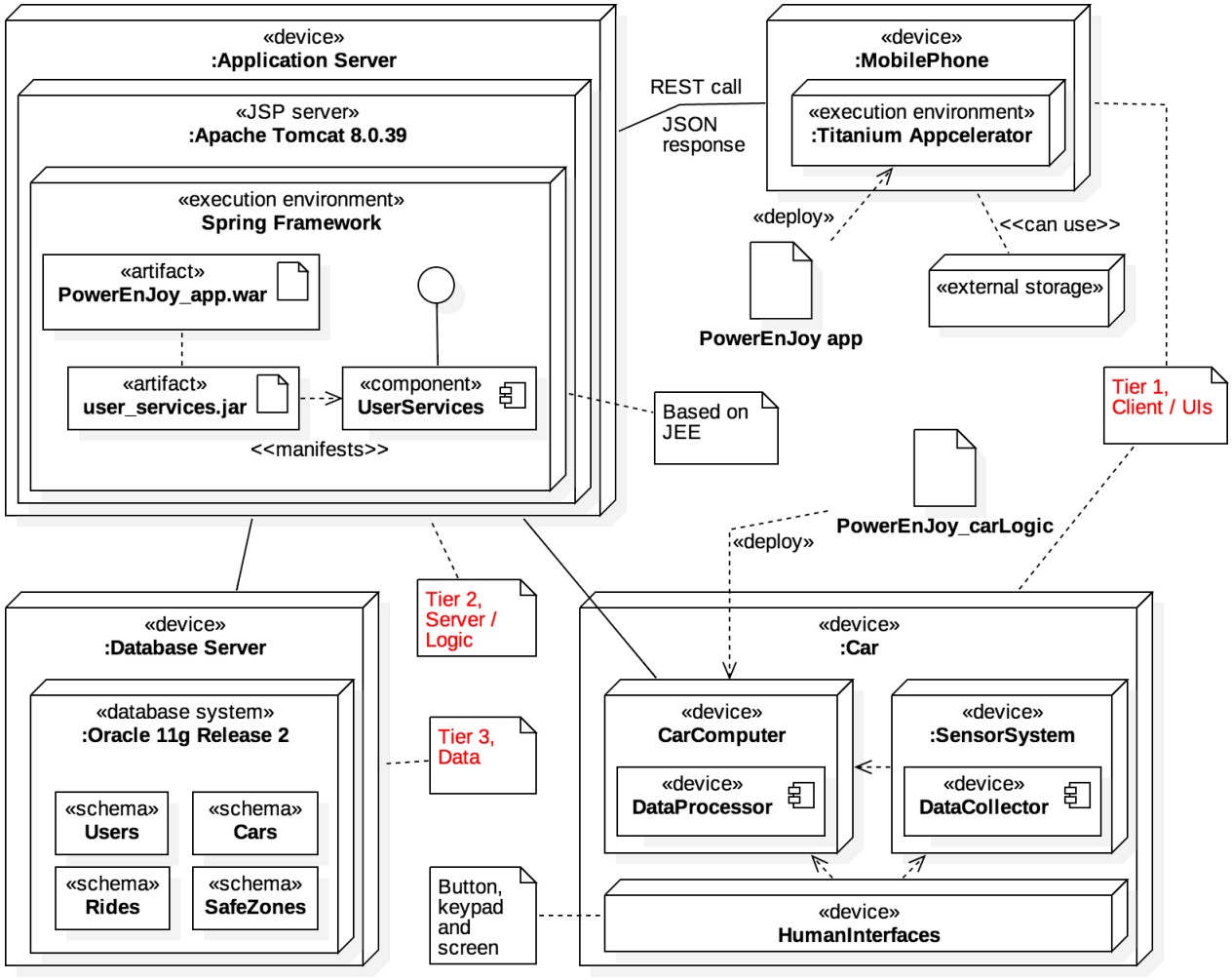


Figure 10: Deployment diagram

The PowerEnJoy software can be divided into three cores. The user-related core is deployed on the mobile device and in the same way the car-related core, which slightly differs from the first because it mainly takes care of the car logic and data elaboration, is deployed on the car processor / computer. The last core is obviously related to the logic tier of the whole software, deployed on the server machine(s).
The Apache Tomcat server runs on the application server and the Spring framework, that provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level. Moreover it provides foundational support for JDBC, allowing a simple and easy interaction with the main database.

2.5 Runtime diagrams

2.5.1 Login runtime diagram

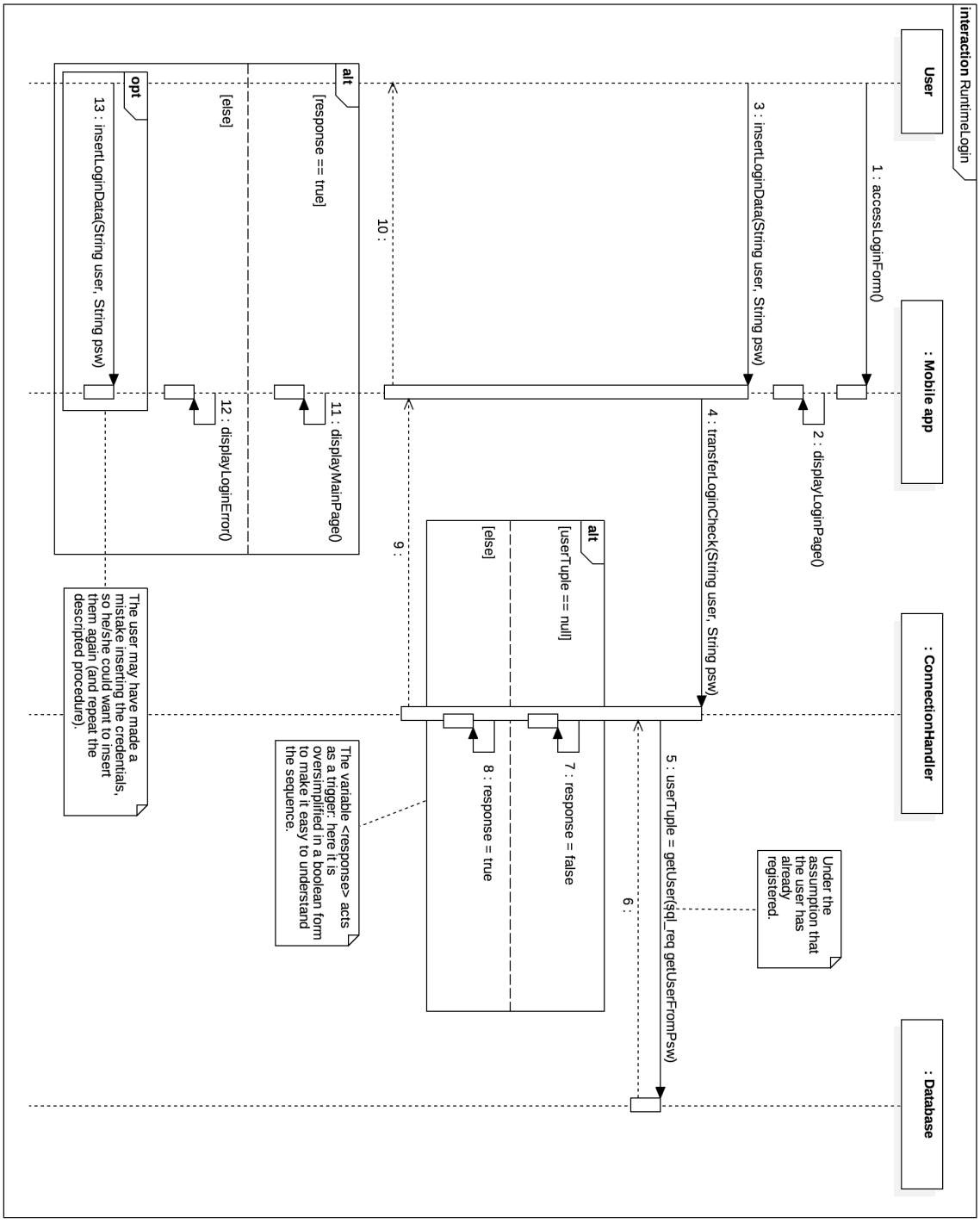


Figure 11: Login runtime sequence

This diagram describes, as the name implies, the interaction of some components during the login phase. After having accessed the login section on the mobile application, the user has to insert the login credentials. The app communicates with the server, thanks to the ConnectionHandler component, that makes a request on the database to retrieve the couple user/password. As the diagram represents the *use case description 1.1*, described in the RASD, it holds the assumption that the user has already registered. Otherwise a double check is needed: before searching for the couple user/password the server has to check if the user is in the database, meaning he/she already registered. The user may have inserted a wrong password, so he/she sees an error message on the screen, instead of the main page of the app.

2.5.2 Reservation runtime diagram - purpose of model -

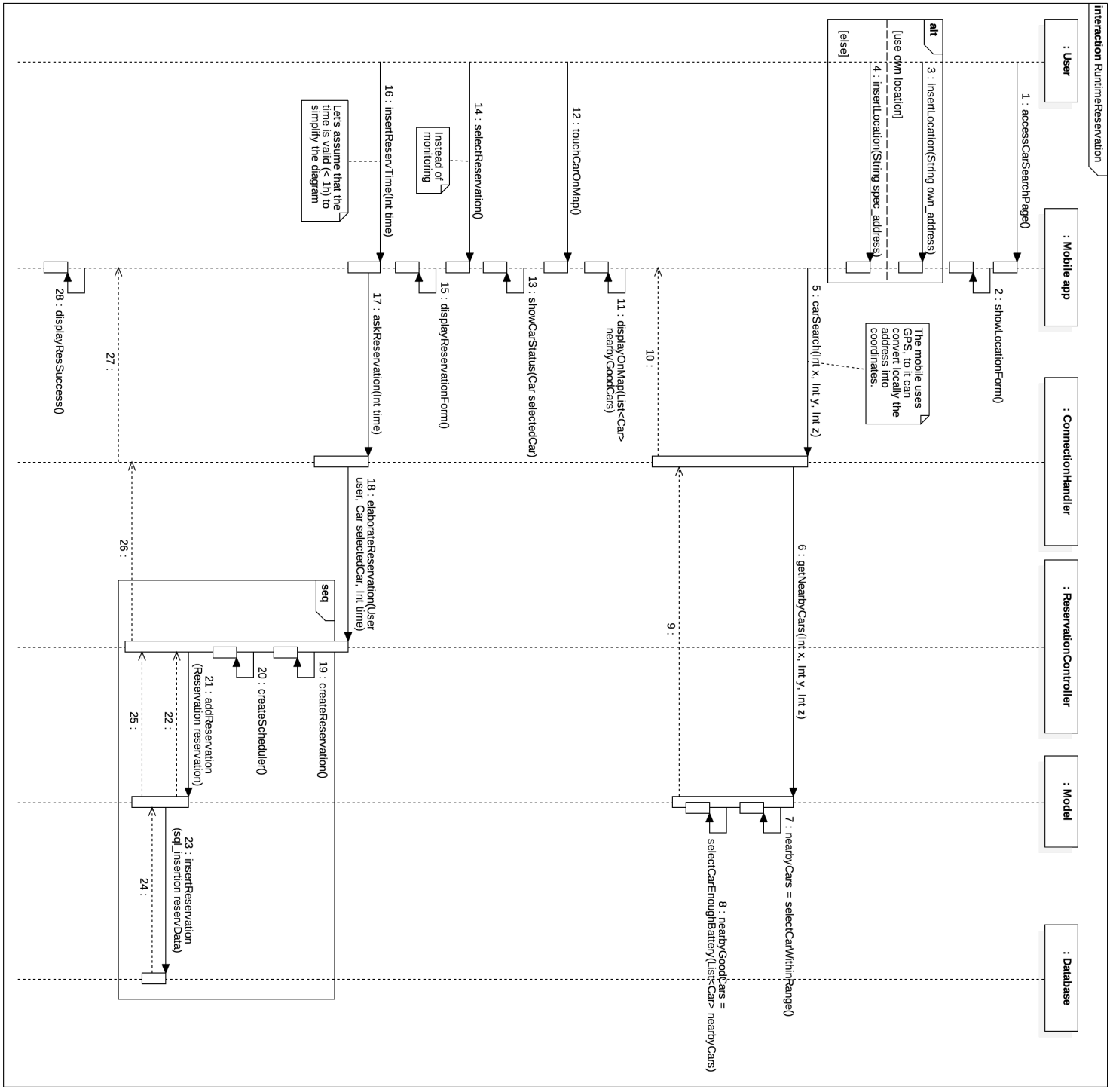


Figure 12: Reservation runtime sequence

The diagram represents the first part of the *use case description 1.2* in the RASD, not considering the pickup phase. The user accesses the car search section on the mobile app and can decide if using his/her own location or a specific address. The mobile app converts the address into coordinates (and this denotes a part of logic in the app, to lighten the server logic core) and communicates them to the server. The ConnectionHandler component interacts directly with the model. The model searches for the nearby cars, in a 5km range zone, and selects the ones with more than 20% of the battery.
Why the model and not the database? After having delayed the description of the model from the components description, it is necessary to explain the purpose of the model. It acts as a sort of cache: instead of interacting continuously with the database, the server contains, in the model, a bit chunk of information, such as car location and minimal status, user status of login and so on.
The server communicates then the list of cars to the mobile app, that displays it to the user. He/she can now select a car and make a reservation. After having informed the server of the reservation request, the ReservationController component handles the request, creating a timer (to count the expiring time before the reservation expires), the reservation object and updating Model and Database.

2.5.3 End of service runtime diagram

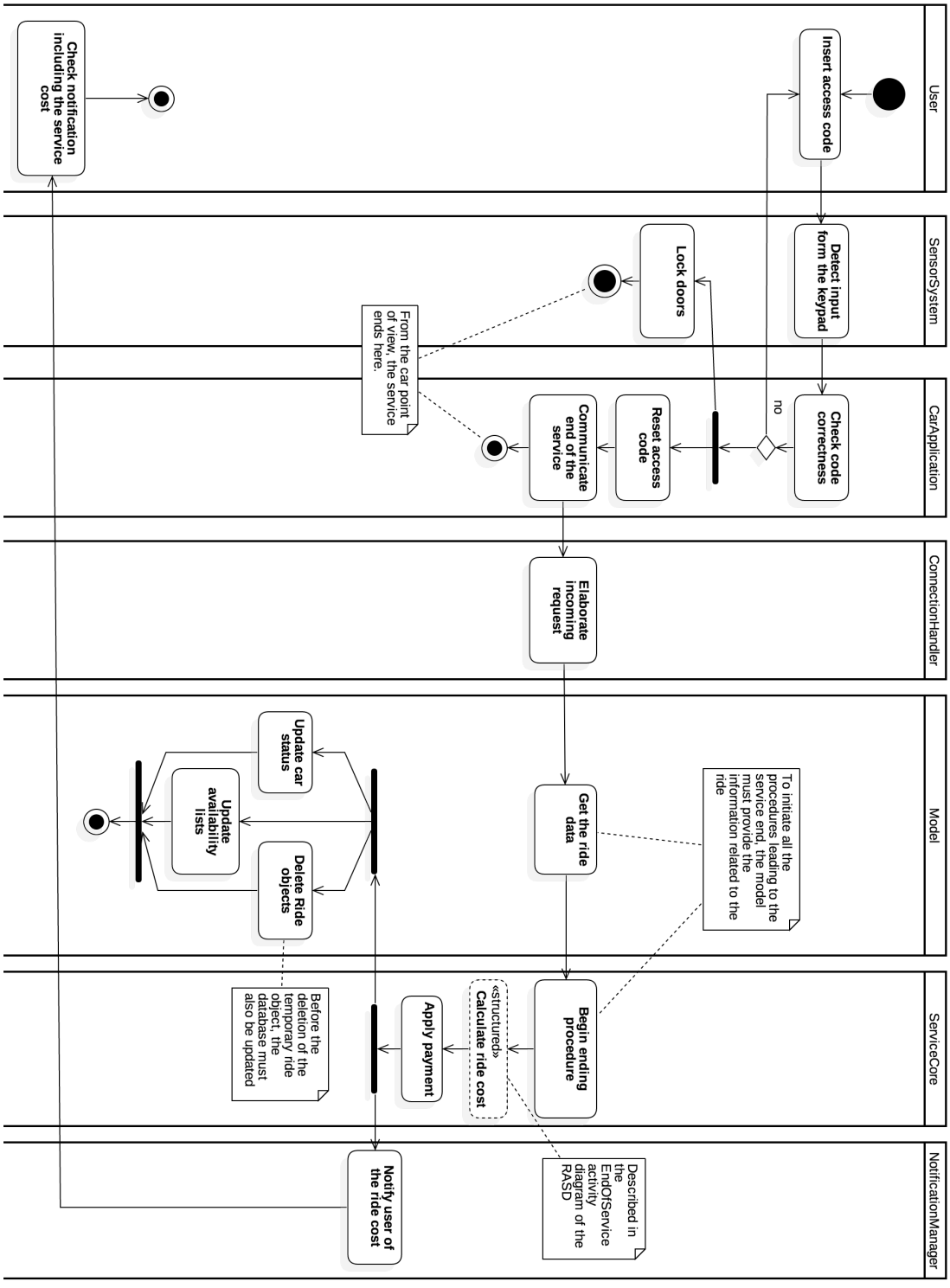


Figure 13: End of service runtime activities

The diagram describes, with some simplifications, the relations of the components during the end phase of the service, referring to the *use case description 5* of the RASD. The phase begins when the user inserts the access code of the service using the numeric keypad of the car. The latter detects that the service is going to end, so the code is reset (to avoid further uses of the car) and all the doors are locked. Meanwhile the server is informed of the end of the ride, so it collects from the model all the data about the ride and starts all the core functionality, in the ServiceCore component, concerning this phase, such as ride cost calculation, payment, and so on. The Model and the Database are updated (the model doesn't need to know the ride anymore, but the database does, for future usage of such data). The ServiceCore will also use the NotificationManager to notify the user of the service cost, through the PushGateway described in section 2.3.2.

2.5.4 Temporary stop by runtime diagram

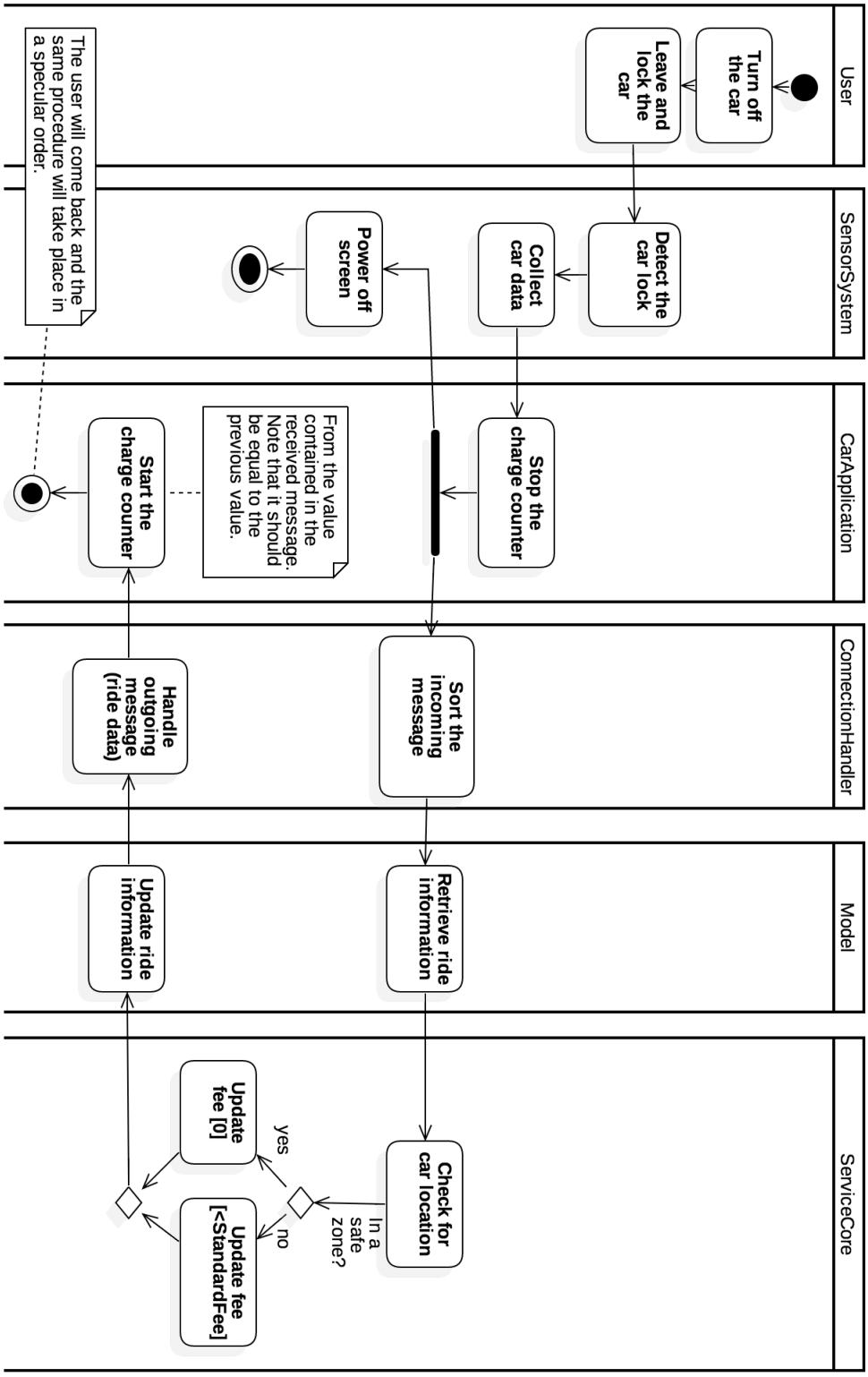


Figure 14: Temporary stop by runtime activities

The diagram describes the sequence of activities of the components of the system involved in the temporary stop by action, described partially in the *use case description 4* of the RASD. After having turned off and locked the car, the CarApplication stops the ride cost counter and powers off some of the car components, such as the dashboard, the internal lights and the screen - the user is gone, there is no need to waste energy -. The server receives information about the stop by action and calculates the new fee, operation done by the ServiceCore, using the data contained in the Model about the ride and the received data. This to check for the coherence of information, in fact the cost on the model should be the same of the cost calculated by the car - logic replica -. After the application of the new fee, the Model is updated and CarApplication is informed of it, ready to begin once again charging the user. Last thing to notice, and it's also written in the diagram, is that the value when the counter stopped and when it begins again, must be the same.

2.5.5 Emergency handling runtime diagram

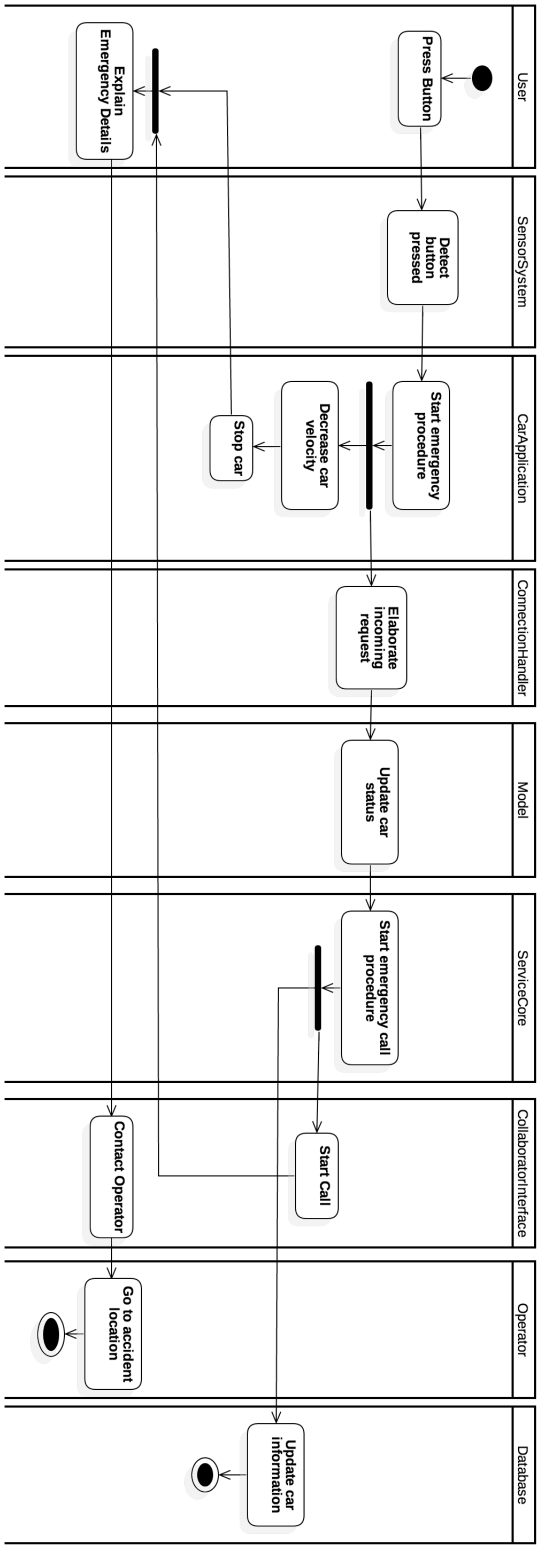


Figure 15: Emergency handling runtime diagram

The diagram describes the sequence of activities of the components of the system involved in the emergency handling, described in the use case description 6 of the RASD. During the ride an accident or a health problem to the driver can suddenly happen, making him/her press the emergency button. The car system detects the button pressed and calls the car application that starts the emergency procedure. This procedure will slow down the car together with notifying the Model wrought the ConnectionHandler. The CarStatus is changed to Emergency by the Model and the ServiceCore starts the emergency call procedure which notifies the Database to update the old data and use the CollaboratorInterface to call to the user in the car and to know what is the problem. Finally the sequence ends when the external operator called by the collaborator, arrives on site.

2.6 Component interfaces

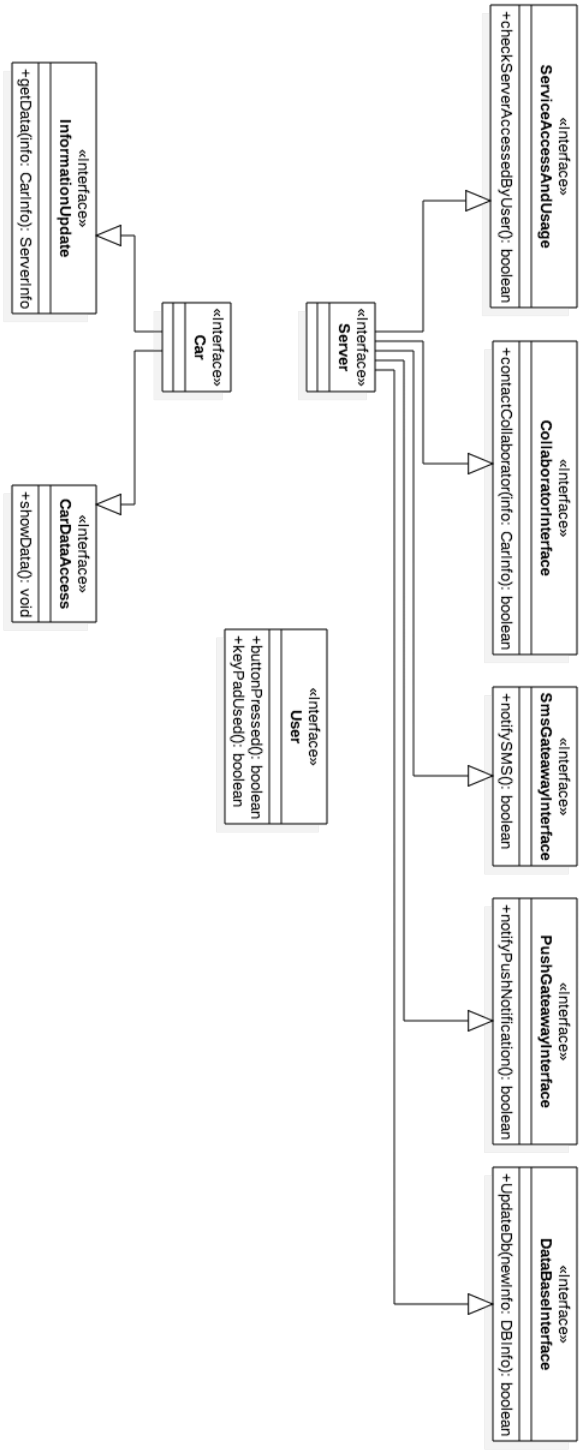


Figure 16: Component interfaces

2.7 Architectural styles and patterns

2.7.1 Overall architecture and comparison with other architectures

As described in the introduction (Overview 2.1), PowerEnJoy software will be deployed following a 3-Tier architecture, composed of a database, a server and a client. The client consists of both the mobile app and the car (that is the other main actor, along with the user). Here some features of this architecture:

- Specialized (there is 1 tier dedicated to a certain task).
- Flexible: the server and the database are single points of failure of the system but acts independently. If a client "brakes" the other tiers will still continue to work.
- Secure: the client will not have direct access to the database, but each tier acts as a proxy to the next level.
- Deployed as a Monolith, so easier implementation of module interactions.
- Simple to test and deploy.

Before making a paragon with other architectures, here is made a compartmentalization between layers, already described in the previously said Overview:

- Client: contains an interface to the Business Logic Layer (BLL).
- Server, containing the biggest chunk of software logic: Business Logic Layer, that is separate from other tiers or layers, such as the data access layer or service layer.
- Database, containing the Data Access Layer (DAL).

3T vs Hexagonal The core of the hexagonal architecture is logic implemented by modules with attached several services using specific adapters. As the 3T it is also deployed as a Monolith and due to the modularity it is simple to test and deploy. However hexagonal architectures are more suitable for big and complex systems, dealing with multiple services and actors. The developing complexity is also higher, due to the implementation of more modules and adapters/interfaces than a 3T.

3T vs SOA The SOA, service oriented architecture, is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online. SOA is:

- Scalable
- Reusable
- Flexible

SOA would not be a good idea for PowerEnJoy due to the heavy data exchange and the lack of the underlying idea of services: in the studied problem all the "services" are fragment and functionality provided by the same provider, such as the reservation request or the monitoring request.

3T vs Microservices The Microservices architecture is a specialization of and implementation approach for service oriented architectures (SOA) used to build flexible, independently deployable software systems. Services in a microservices architecture are processes that communicate with each other over a network in order to fulfill a goal and services should have a small granularity.
For the same reasons described in the paragraph, 3T vs SOA, also the Microservices architectures is not a good option in describing and developing the PoerEnJoy software, despite the better efficiency and the very low dependence factor.

2.7.2 Protocols

The tiers of the PowerEnJoy software communicate through the network and exchange data in respect of the following protocols.

JDBC Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented towards relational databases. It is used by the BLL to communicate with the DAL. Considering all the drivers, the supported database servers are:

- MySql 5.1
- Java DB 10.5.3.0
- Oracle 11
- PostgreSQL 8.4
- DB2 9.7
- Sybase ASE 15
- Microsoft SQL Server 2008

The value-add provided by the Spring Framework JDBC abstraction is perhaps best shown by the sequence of actions outlined in the table below. The table shows what actions Spring will take care of and which actions are the responsibility of the application developers.

Action	Spring	Developer
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

RESTful API with JSON The word REST refers to web services that are one way of providing interoperability between computer systems on the Internet. REST-compliant web services allow requesting systems to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations. They are used by clients to interact with the BLL.

2.7.3 Design patterns

Most of the applicable design patterns are already described in the RASD, especially in the Class diagram subsubsection 4.2.4. Here follows some detailed information about some of them.

MVC Spring is a big framework, that contains a lot of components. One of these is Spring MVC, a module that lets the developers implement the web application according to the model-view-controller design pattern. Inside the application the following objects will be implemented:

- Models: represented by the classes that in turn represent the handled objects and the access classes to the database.
- Views: represented by various JSP files (compiled in HTML) and some classes related to the exportation of files which format differs from HTML (PDF, XLS, CSV...).
- Controllers: represented by classes that keep listening from incoming messages on a specific URL each, and thanks to Models and Views handle the user's requests.

Adapter Adapters are used in our mobile application to adapt the Driver interface to the RESTful API interface.

SOA and Microservices SOA and Microservices may be software architectures, but the underlying idea can be also applied to pieces of software. Along with the controllers and models, those "simil-patterns" can improve the modularity of the software, disjointing the model data (so that there is not a monolithic model but multiple fragmented ones) and the business logic (proven by the fact that multiple controllers coexist an the ServiceCore component could be broken into minor cores, each containing functions and procedures related only to a specific event, such as the logic of the service end rather than the login phase).

Client - Server As already described in previous sections, the software development will follow the client - server paradigm, where the large part of network communication happens between the users and the server. The client-server paradigm has the following advantages:

- High maintainability
- Independence from the number of clients connected
- Good level of security, regarding data transfer

3 Algorithm design

Here follows some simple algorithms that will be needed as logic parts in the implementation of the PowerEnJoy software. These algorithms are implemented and tested with JUnit4 over a small fragment of the code describing the server piece of software. Code and tests can be found at `https://github.com/MatteoF94/Software-Engineering-II-Project/tree/master/CodeFragments`.

3.1 Monitoring request

3.1.1 Pseudo code

check if the user has already made a reservation
    if the Model contains a reservation made by the user
        communicate error and handle it
    else continue with the algorithm
check if there is already someone monitoring the car which request is made about
    if the Model contains a monitoring with the same car the user wants to monitor
        if the user is already monitoring that car
            communicate error and handle it
        else add user to the monitorers list on that car
    else create a new monitoring to add in the Model

3.1.2 Possible simplified implementation

```
void monitoringRequest(User user, Car car) throws ReservationAlreadyDoneException, AlreadyRequestedException {

    /**
     * Check if the user has already made a reservation
     */
    if(model.reservationExistance(user)) return;

    /**
     * Server procedure to add the monitoring of the user
     */
    // Check if the system already contains a monitoring on that car
    if(!model.monitoringExistance(car))
        model.addMonitoring(new Monitoring(car, user));
    else
        model.getMonitoringByCar(car).addUser(user);
}
```

Figure 17: Algorithm, contained in the MonitoringController component

```java
public boolean monitoringExistance (Car monitoredCar) {
    for (Monitoring mon: monitorings) {
        if (mon.getCar().equals(monitoredCar)) return true;
    }
    return false;
}

public boolean reservationExistance (User monitoringUser) throws ReservationAlreadyDoneException {
    for (Reservation res: reservations) {
        if (res.getUser().equals(monitoringUser)) throw new ReservationAlreadyDoneException();
    }
    return false;
}

public Monitoring getMonitoringByCar(Car monitoredCar) {
    Monitoring monitoring = null;
    for (Monitoring mon: monitorings) {
        if(mon.getCar().equals(monitoredCar)) {
            monitoring = mon;
            break;
        }
    }
    return monitoring;
}
```

Figure 18: Algorithm parts inside the model

Even though the pseudo code is clear in the description of the sequence of procedures and actions to be invoked, the possible implementation that has been thought using Java, given the architectural decisions described in the document, needs some explanations. Only the essential parts are shown in the figures, and all the support methods in various classes are omitted. One software design idea is to reduce the information usable by the controllers (MonitoringController in this case) making the model expose only the needed methods, which differs for each controller. Moreover, to improve modularity, for each controller the respective procedure is divided into the controller itself and the Model, and possibly all the classes involved. However the Model contains only utility functions, such as the ones described in Figure 15, where some search are made on the list of reservations and monitoring of the server.
The time complexity of the algorithm is mainly due to the checks that involve the cycling of the elements in the model. If there are n monitoring, u users and m reservations, in the worst case scenario (where for each monitoring there are already u/n users) the time complexity is O(max{u, m}).

3.2 Ride cost calculation

3.2.1 Pseudo code

calculate standard cost of the ride
    get ride time
    get stop by time
    subtract the stop time to the ride time
    multiply effective ride time with standard fee
calculate stop cost of the ride
    get stop by time
    multiply stop by time with stop fee
calculate bonus/overcharge multipliers
    if the time of multiple passengers > half of the ride total time
        apply passenger bonus multiplier

if the car battery >= 50
    apply high battery bonus multiplier
else if the car battery < 20
    apply low battery overcharge multiplier

if the car is plugged into a charger
    apply plugged bonus multiplier

check if the car is more 3km from the nearest safe area
    for all the safe areas
        for all the positions in each safe area
            if the euclidean distance from the car and the safe area is less than 3km
                don't apply any multiplier (0)
    apply distance overcharge multiplier
apply all the multipliers to the previously calculated cost of the ride

3.2.2 Possible simplified implementation

```
private final float NORMAL_FEE = (float) 2.99;
private final float STOPBY_FEE = (float) 1.39;
private final float PASSENGER_BONUS = (float) -0.1;
private final float HIGH_BATTERY_BONUS = (float) -0.2;
private final float LOW_BATTERY_OVERCHARGE = (float) 0.3;
private final float PLUGGING_BONUS = (float) -0.3;
private final float LONG_DISTANCE_OVERCHARGE = (float) 0.3;


public float calculateRideCost(Ride ride) {
    float standardCost = (ride.getDurationOfRide()-
                          ride.getDurationOfStopBy()-
                          ride.getDurationOfStopBySpecial())
                         *NORMAL_FEE;
    float stopByCost = ride.getDurationOfStopBy()*STOPBY_FEE;
    /* There is no need to consider the special stopby,
    because the fee is 0 due to the car being in a special area */

    float passengerBonusContribute = calculatePassengerBonus(ride);
    float batteryContribute = calculateBatteryBonus(ride);
    float pluggingContribute = calculatePluggingBonus(ride);
    float distanceContribute = calculateDistanceBonus(ride);

    return (standardCost+stopByCost)*
                        (1+passengerBonusContribute+batteryContribute+pluggingContribute+distanceContribute);
}
```

Figure 19: Ride cost algorithm, part including fees and bonus multipliers

```
private float calculatePassengerBonus(Ride ride) {
    if(ride.getDurationPassengers() > ride.getDurationOfRide()/2) return PASSENGER_BONUS
    else return 0;
}

private float calculateBatteryBonus(Ride ride) {
    float battery = ride.getCar().getBatteryCharge();
    if(battery >= 50.0) return HIGH_BATTERY_BONUS;
    else if(battery < 20.0) return LOW_BATTERY_OVERCHARGE;
    else return 0;
}

private float calculatePluggingBonus(Ride ride) {
    if(ride.getCar().isPlugged()) return PLUGGING_BONUS;
    else return 0;
}

private float calculateDistanceBonus(Ride ride) {
    Position carPosition = ride.getCar().getPosition();
    if(ServiceCore.isMoreThan3kmDistant(carPosition)) return LONG_DISTANCE_OVERCHARGE;
    else return 0;
}
```

Figure 20: Ride cost algorithm, part including the calculation sub-functions

```
public static boolean isMoreThan3kmDistant(Position position) {
    Set<List<FixedPosition>> safeAreas = model.getSafeAreas();
    for(List<FixedPosition> safeArea : safeAreas) {
        for(FixedPosition pos : safeArea) {
            double gradX = Math.pow((pos.getX() - position.getX()),2);
            double gradY = Math.pow((pos.getY() - position.getY()),2);
            double gradZ = Math.pow((pos.getZ() - position.getZ()),2);
            if (gradX + gradY + gradZ <= 9) return false;
        }
    }
    return true;
}
```

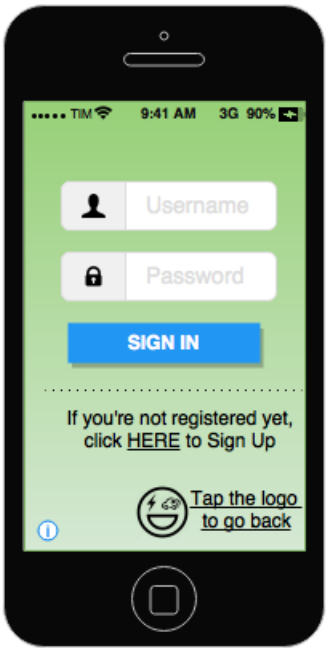
Figure 21: Ride cost algorithm, function that checks the distances

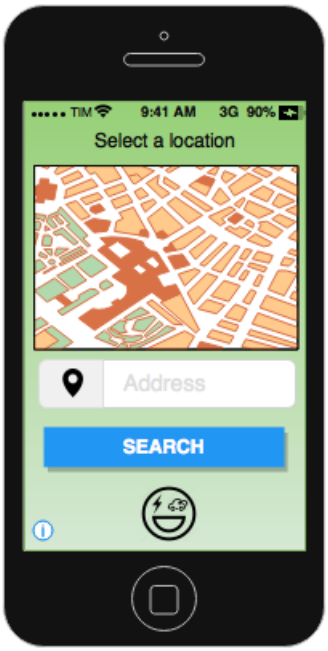
It is important to stress the fact that the algorithm is improvable in many way. As example, if we consider that there are n safe areas and m positions dedicated, the algorithm has a time complexity (worst case where each safe area contains m/n positions) of O(m), given by the linear time for all the checks added with the exception of the distance multiplier calculation (n time m/n).
Moreover, there are many simplifications. Considering again the distance check, it is calculated in the euclidean way, but in reality it is likely to follow the road lines and it does not follow a straight line from car to safe area.

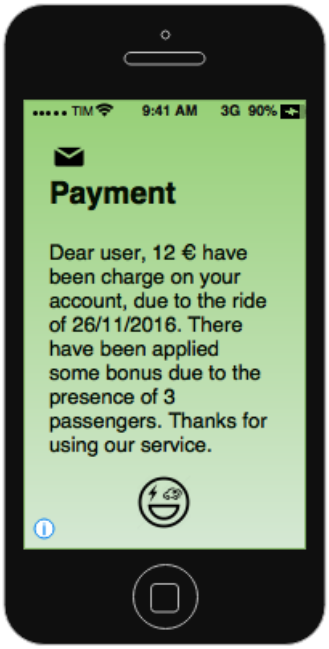
4 User Interface design

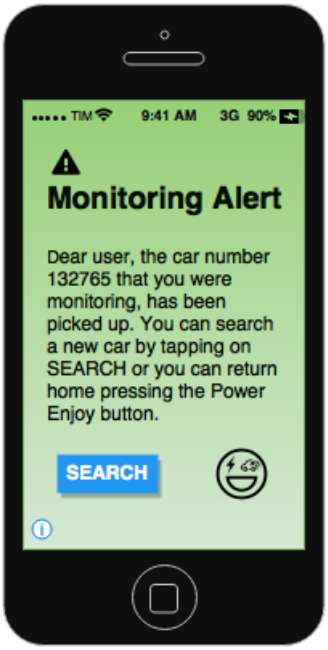
4.1 Interfaces mockups

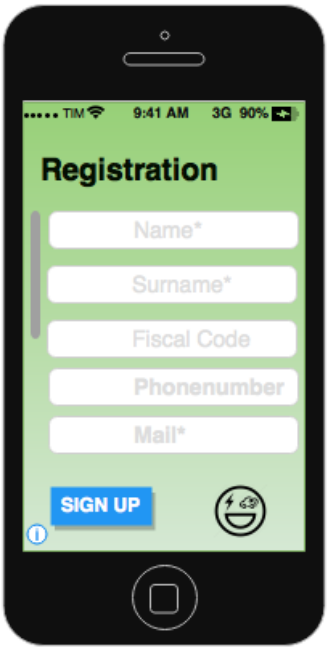
4.1.1 Mobile dashboard



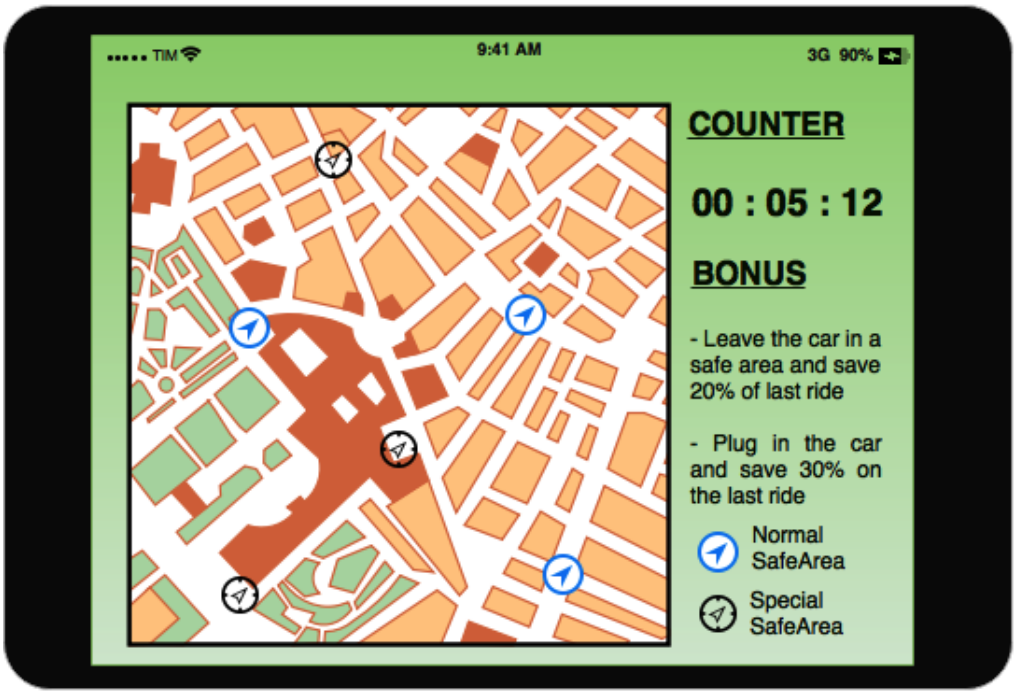


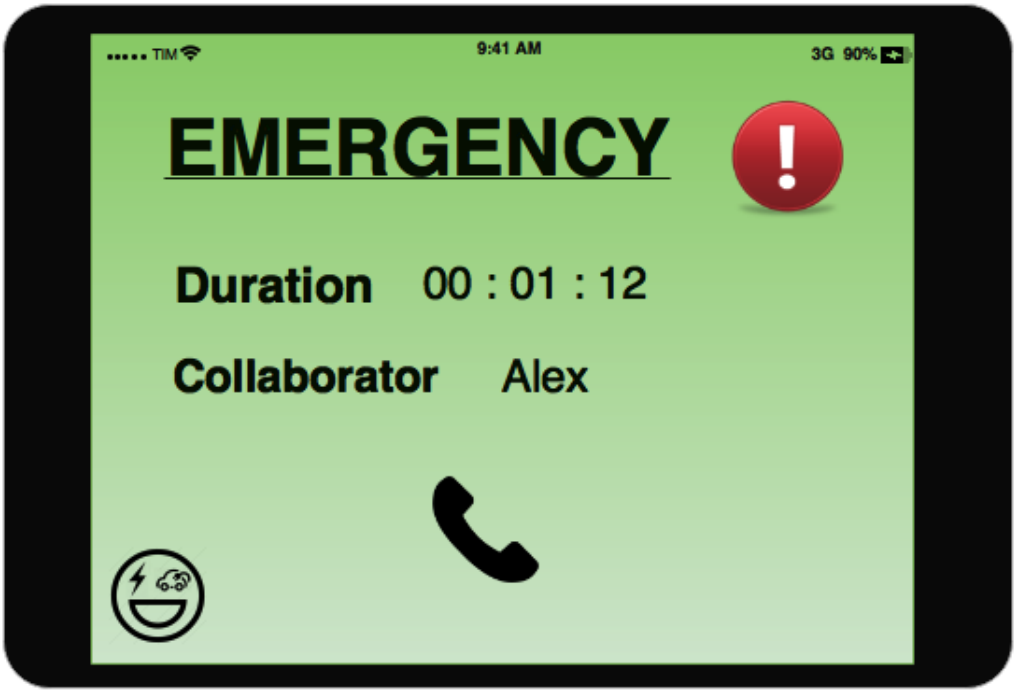

TIM 9:41 AM 3G 90%
Payment
Dear user, 12 € have been charge on your account, due to the ride of 26/11/2016. There have been applied some bonus due to the presence of 3 passengers. Thanks for using our service.


TIM 9:41 AM 3G 90%
Monitoring Alert
Dear user, the car number 132765 that you were monitoring, has been picked up. You can search a new car by tapping on SEARCH or you can return home pressing the Power Enjoy button.
SEARCH


TIM 9:41 AM 3G 90%
Registration
Name*
Surname*
Fiscal Code
Phonenumber
Mail*
SIGN UP

4.1.2 Car screen





4.2 User experience diagrams

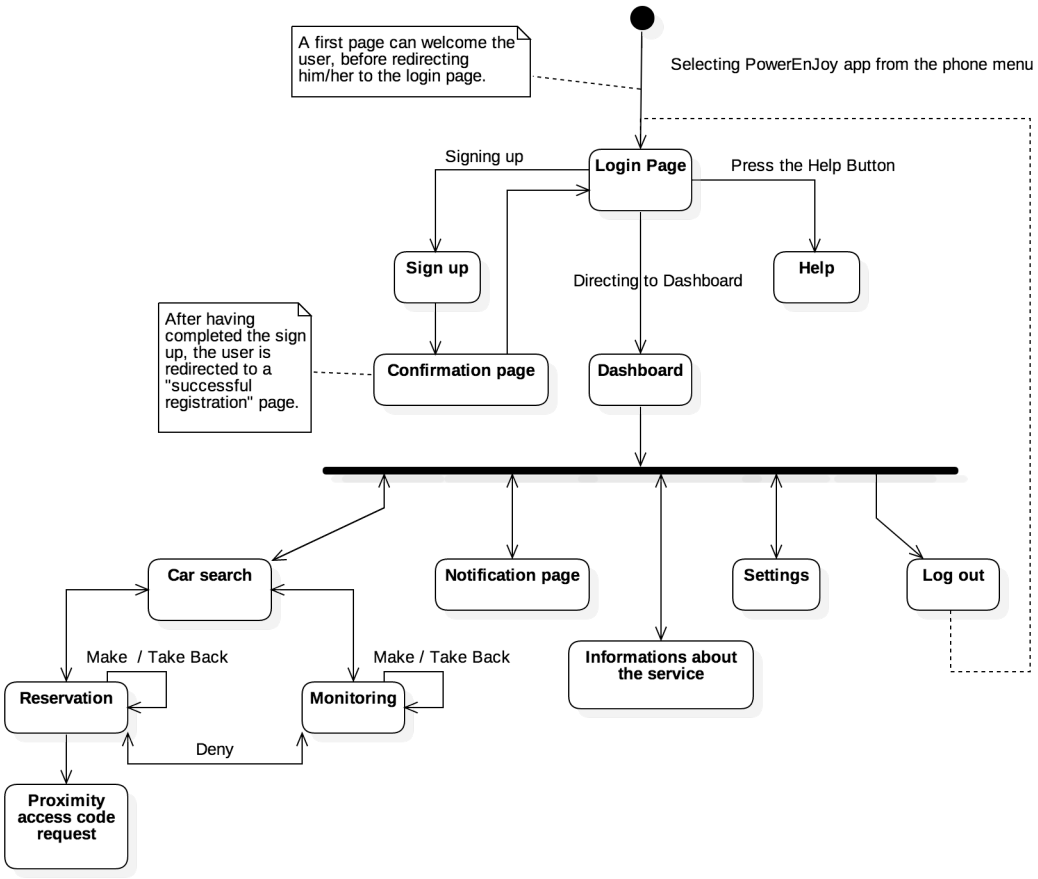


Figure 22: Schematic vision of the PowerEnJoy application

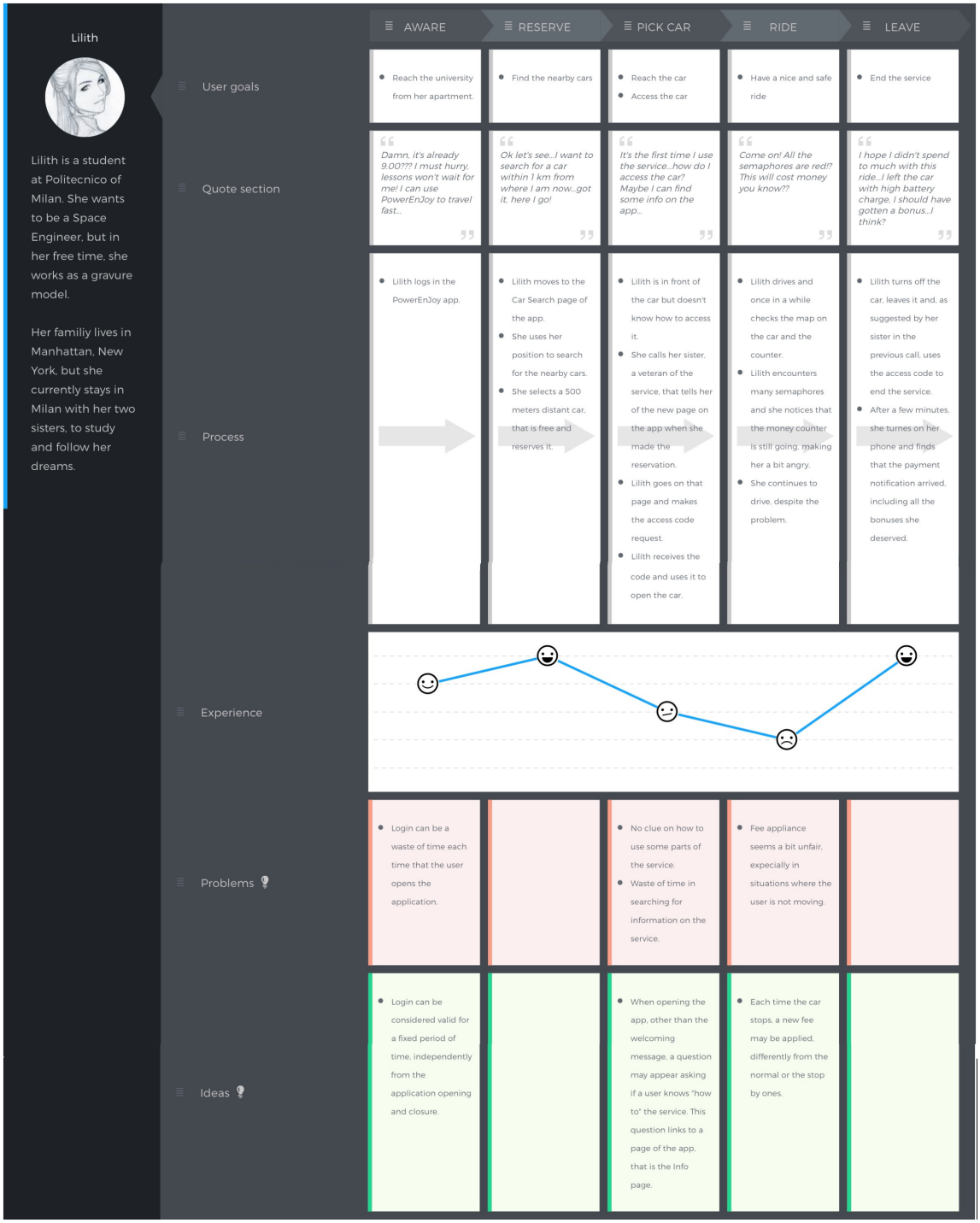


Figure 23: Consumer Journey Map of the service use

The reader may wonder why such diagrams, concerning about a user point of view of the service, are in the design document. For Figure 14 the reason is obvious: the diagram represents both how the user will find the page arrangement of the app pages (sections), so it also implies a design choice. Figure 15 is more subtle. Apparently the journey map tells of the experience of a particular user, Lilith in the case - that is also one protagonist of the scenarios described in the RASD -. However, a bad experience of the user or whichever point of disappoint corresponds to an improvable design choice, or a better assumption. The first problem in the diagram clearly represents what just mentioned: the user found difficult to proceed in the usage of the service because she didn't know what to do. It could be better to inform the user on the "how to" as soon as possible, maybe adding a link on the welcome screen, asking whether the user know how to proceed or not, linking to a page of the app where instructions on the service are described (see also Figure 14).

5 Requirements traceability

In this section it is defined list of the requirements and goals described in the RASD and how they map with the designed components of the application.

- [G1] Ensures that only the account owner can access his/her own account, if he/she is already registered.
  - [R1.1] Mobile app, ConnectionHandler, ServiceCore (for code generation), interface to Database (to store the registration data), NotificationManager and GMSGateway (to notify the user).
  - [R2.2] Mobile app, ConnectionHandler, Database.
- [G2] Allows the user to reserve an available car for up to one hour before the service starts and to take back such reservation before the picking up time expires.
  - [R2.1] Mobile app, ConnectionHandler, Model, Database.
  - [R2.2] Mobile app.
  - [R2.3] Mobile app, ConnectionHandler.
  - [R2.4] ServiceCore, Model, NotificationManager, PushGateway.
  - [R2.5] MobileApp, ConnectionHandler, ReservationController, Model, Database (for the reservation).
  - [R2.6] MobileApp, CarSensorSystem, CarApplication, ConnectionHandler, Model, Database (for the pick up event).
  - [R2.7] MobileApp, CarSensorSystem, CarApplication, ConnectionHandler, ReservationController.
- [G3] Allows the user to look for the available cars within an area that he/she specified, or in the nearby areas, for a possible reservation. If the car has a low battery charge, such as less than 20\%, it does not appear on the user interface.
  - [R3.1] Mobile app.
  - [R3.2] Mobile app, ConnectionHandler, ServiceCore.
  - [R3.3] ServiceCore, Model.
  - [R3.4] ServiceCore, Model, Database.
- [G4] Ensures that the user is able to trace/check the status of a car, available or already reserved, and receive push notification about it.
  - [R4.1] ServiceCore, Model.
  - [R4.2] Mobile app
  - [R4.3] ReservationController, Model, NotificationManager, PushGateway.
  - [R4.4] Mobile app, MonitoringController, Model.
  - [R4.5] Model, Database, NotificationManager, PushGateway.
- [G5] Ensures that the user is able to access, and open, the car he/she reserved, if he/she is in time.
  - [R5.1] Mobile app, ConnectionHandler, ServiceCore, NotificationManager, PushGateway.
  - [R5.2] Mobile app, ConnectionHandler, ServiceCore, CarApplication, CarSensorSystem.

  - [R5.3] ServiceCore, Model, ConnectionHandler, CarApplication, CarSensorSystem.

- [G6] Ensures that the service, and the charge on the user, starts only when the engine of a car is ignited.
  - [R6.1] CarSensorSystem, CarApplication.
  - [R6.2] CarSensorSystem, CarApplication, ConnectionHandler, RideController.
- [G7] Ensures that the user is informed real-time on the current cost of the service, when using it, and the possible discounts and overcharges.
  - [R7.1] to [R7.5] CarSensorSystem, CarApplication.
- [G8] Ensures that the user knows the safe areas around him/her during the ride.
  - [R8.1] CarApplication.
  - [R8.2] CarSensorSystem, CarApplication.
  - [R8.3] CarApplication, ConnectionHandler, Model.
- [G9] Allows the user to press a button on the car to contact an operator in case of emergency, such as a car accident, a mechanical problem or sudden illness.
  - [R9.1] to [R9.4] CarSensorSystem, CarApplication, ConnectionHandler, RideController, Model, Database, CollaboratorInterface.
- [G10] Allows the user to leave the car but continue to use the service.
  - [R10.1] CarSensorSystem, CarApplication, ConnectionHandler, RideController, Model, ServiceCore.
  - [R10.2] RideController, NotificationManager, PushGateway.
- [G11] Ensures that the user can end the service at any time.
  - [R11.1] and [R11.3] CarSensorSystem, CarApplication, ConnectionHandler, RideController, Model.
  - [R11.2] CarSensorSystem, CarApplication.
  - [R11.4] CarSensorSystem, CarApplication, ConnectionHandler, RideController, ServiceCore, Model, Database.
- [G12] Ensures that the user receives a discount or overcharge when specific conditions are met.
  - [R12.1] to [R12.4] CarSensorSystem, CarApplication, ConnectionHandler, RideController, Model, ServiceCore, Database.
- [G13] Ensures that the user is notified of the applied discount and overcharges once the service ends and payment is applied.
  - [R13.1] ServiceCore, NotificationManager, PushGateway.
- [G14] Ensures that when the user picks up a car, it has no mechanical or electrical problem.
  - [R14.1] and [R14.2] CarSensorSystem, CarApplication, ConnectionHandler, Model, CollaboratorInterface.

6 Document further information

6.1 References

- RASD, before Version 4.5
- Assignments AA 2016-2017.pdf
- https://msdn.microsoft.com/en-us/library/ee658116.aspx, containing some information about layers design
- http://alistair.cockburn.us/Hexagonal+architecture, contains some information about the Hexagonal architecture
- https://www.nginx.com/blog/introduction-to-microservices/, contains some information about the Microservices architecture
- https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html, contains some advices over the architecture choice and implementation
- http://www.uxbooth.com/articles/8-must-see-ux-diagrams/, contains some examples of user experience diagrams
- https://projects.spring.io/spring-framework/, contains some information about the Spring framework
- http://www.agilemodeling.com/artifacts/componentDiagram.htm. contains a brief description of the component diagrams from an AGILE point of view
- Sample Design Deliverable Discussed on Nov. 2.pdf
- Paper on the green move project.pdf
- Second paper on the green move project.pdf

6.2 Used tools

In creating the DD document, the following tools have been used:

- Github, for version controller
- LyX, to write the document and converting in .pdf format
- StarUML, for UML diagrams and image exporting
- Pencil, for the architecture logic and physical design
- https://uxpressia.com, for the Consumer Journey Map
- https://www.draw.io, for the interface mockups
- Hunspell, for the spell check of the document
- Atom, for the project markdown
- IntelliJ IDEA, for the algorithms implementation and testing

6.3 ChangeLog

- [21/11/2016] [**Version 0.1**] :: Added first part of the document introduction, including preface, Purpose, Scope and Definitions. ChangeLog and Hours of work sections also introduced.
- [21/11/2016] [**Version 0.2**] :: Added the References and Document structure sections.
- [23/11/2016] [**Version 0.3**] :: Added the overview of the architecture of the project and a brief description of the car system, so subsections 2.1 and 2.2.
- [23/11/2016] [**Version 0.4**] :: Added the description of the different architectural styles that were considered and their brief comparison.
- [24/11/2016] [**Version 1.0**] :: Added high view of the components of the system with a detailed description, subsubsection 2.3.1.
- [24/11/2016] [**Version 1.1**] :: Added two detailed component diagrams, subsubsection 2.3.2.
- [25/11/2016] [**Version 1.2**] :: Added the deployment diagram and its description, subsection 2.4.
- [26-27/11/2016] [**Version 1.3**] :: Added three runtime views with the respective description (login, reservation and end of the service), subsubsections 2.5.1, 2.5.2, 2.5.3.
- [28/11/2016][**Version 1.4**] :: Added runtime diagram of the service temporarily stop by, subsubsection 2.5.4.
- [28/11/2016][**Version 1.5**] :: Added runtime diagram of an emergency handling, subsubsection 2.5.5 and rewriting of section 2.7, including the addition of Protocols subsubsection, 2.7.2.
- [29/11/2016] [**Version 1.6**] :: Added Spring/developer matrix in the Protocols subsubsection 2.7.2 and added the Architecture patterns, subsubsection 2.7.3.
- [30/11/2016] [**Version 2.0**] :: Added Requirement traceability, section 5. Incomplete sections about algorithm design and user interfaces (respectively 3 and 4) also were added.
- [30/11/2016] [**Version 2.1**] :: Added ComponentInterfaceDiagram
- [1/12/2016] [**Version 2.2**] :: Added mockups in subsection 4.1 and two User eXperience diagrams in subsection 4.2. Also added subsection about used tools, 6.3.
- [2/12/2016] [**Version 3.0**] :: Added the monitoring algorithm, section 3.1, both in pseudo code and a possible implementation in Java.
- [6-7/12/2016] [**Version 3.1**] :: Added the ride cost calculation algorithm, section 3.2, both in pseudo code and a possible implementation in Java.
- [27-28/12/2016] [**Version 3.2**] :: Checked name coherence between the structure diagrams and added the LoginController component in the component diagrams.
- [15/01/2017] [**Version 3.3**] :: Added major description to the Car components, going into further detail.

6.4 Hours of work

Matteo Frosi

[Before 21/11/2016]: 3.00 hours (spent in analyzing well known architectures, to study and apply them to our problem).
[21/11/2016]: 2.00 hours (further analysis of an applicable architecture to our problem, alleging to the classic 3 Tier architectures).
[21/11/2016]: 1.00 hours (writing of the first part of the document).
[23/11/2016]: 2.00 hours (writing of the overview section of the document and drawing of architectures images)
[24/11/2016]: 2.00 hours (drawing up of the component view and writing of the component high view section)
[24/11/2016]: 2.30 hours (drawing of the component in a more detailed view)
[25/11/2016]: 1.00 hours (writing of the section about the detailed component diagrams)
[25/11/2016]: 3.00 hours (thinking, drawing and deploying the deployment diagram)
[26-27/11/2016]: 4.00 hours (thinking, drawing and describing three runtime views, of Version 1.3)
[28/11/2016]: 1.30 hours (addition of the fourth runtime diagram)
[28/11/2016]: 1.00 hours (rewriting of subsection 2.7 and addition of sub subsection 2.7.2)
[29/11/2016]: 2.30 hours (writing of sub subsection 2.7.3 about architecture patterns and addition of concepts to sub subsection 2.7.2)
[30/11/2016]: 1.30 hours (writing of the requirement traceability section, with some modifications of the RASD)
[1/12/2016]: 2.30 hours (drawing and insertion of the first two user experience diagrams)
[2/12/2016]: 3.00 hours (writing and testing the monitoring algorithm, creating a fragment of code)
[6-7/12/2016]: 3.00 hours (writing and testing the ride cost algorithm, improving the previously created fragment of code)
[15/01/2017]: 1.00 hours (writing sub subsection 2.3.3, about a more detailed description of the car components and interaction between them)
TOTAL: 36.30 hours

Luca Costa

[Before 21/11/2016]: 2.30 hours (spent learning about the various architectures and apply them to our problem)
[22/11/16]: 1.00 hours (spent learning about a possible implementation of the different architecture styles)
[23/11/16]: 2.30 hours (spent writing comparison between architectures and informing about DD document in general)
[24/11/2016]: 2.00 hours (learning about how to draw a component diagram and its effective first drawing)
[25/11/2016]: 1.00 hour (informing on runtime and deploy diagram)
[26/11/2016]: 1.30 hours (thinking and drawing architecture schema, not part of the DD)
[28/11/2016]: 2.00 hours (drawing emergency handling runtime diagram and its description)
[30/11/2016]: 2.00 hours (thinking and drawing componentInterface diagram)
[1/12/2016]: 4.30 hours (drawing the mockups)
TOTAL: 19.00 hours