PowerEnJoy
Design Document

Stefano Brandoli (mat. 875633)
Silvia Calcaterra (mat. 874887)
Samuele Conti (mat. 875708)

December 11, 2016

Version 1.0

Contents

1 Introduction

1.1 Purpose

The Design Document (DD) follows the redaction of the RASD.

The main purposes of this document are: record the main design information and decisions to communicate them to some key stakeholders; identify the main architectural components of the system together with their functional description and interactions; highlight the interfaces of the system and their impact on the architecture; describe the main algorithms to be implemented; provide a view of the user interfaces.

The intended audience of this document are: developers, designers, project managers, quality assurance people and testers.

This document tries to keep an high level of abstraction and to provide a top down approach. It can be used as a basis to design future refinements of the system architecture through "Detailed Design Documents" and as a basis for the developers who want to obtain a general idea on how to start the implementation phase.

1.2 Scope

The aim of the system is to manage the PowerEnjoy car-sharing service, which only employs electric cars.

The system will interact with some external interfaces to provide part of its functionalities, in particular regarding the interaction with the cars' hardware and sensors; the handling of user's payments and payment informations; the interaction with Google Maps APIs.

During the design of the system, architectural styles and design patterns will be used, when reasonable, to provide a structure of the system based on well known solutions. This may also to reduce the development and maintenance costs during the implementation phase.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Tier = a tier can be seen as a synonym of software layer. It is a level of software responsibility, a software functional area. It's important to notice that this definition doesn't deal with the number of physical machines on which the software will actually be running on.

Component = a component is a design unit that represents a piece of software that will typically be implemented using replaceable modules. A component encapsulates behavior and implements specified interfaces.

Artifact = An artifact is a physical unit, such as a file, executable, script, database etc, which is deployed on physical machines.

1.3.2 Acronyms

- DD = Design Document.
- RASD = Requirements Analysis and Specification Document.
- UI = User Interface.
- API = Application Programming Interface.
- ACID = Atomicity, Consistency, Isolation, Durability.
- HTTP = HyperText Transfer Protocol.
- HTTPS = HyperText Transfer Protocol over Secure Socket Layer (SSL).
- DBMS = Database Management System.
- RESTful = Representational State Transfer.
- JAX-RS = Java API for RESTful web services.
- JPA = Java Persistence API.
- EJB = Enterprise Java Bean.
- CRUD = Create, Read, Update, Delete.
- JDBC = Java DataBase Connectivity.
- JSF = Java Server Faces.
- JSP = Java Server Pages.
- SDK = Software Development Kit.
- SoC = State of Charge.
- DMZ = Demilitarized Zone.

1.4 Reference Documents

- Specification Document of PowerEnjoy.
- PowerEnjoy RASD.
- Danny Coward - Java EE 7: The Big Picture: this book has been used to get a general and precise overview of the Java EE 7 platform.
- "API First" Architecture: https://www.leaseweb.com/labs/2013/10/api-first-architecture-fat-vs-thin-server-debate/
- Concept of "Component" in UML2: https://www.ibm.com/developerworks/rational/library/dec04/bell/
- iOS MVC pattern: https://www.raywenderlich.com/132662/mvc-in-ios-a-modern-approach

1.5 Document Structure

This document has been structured in the following sections:

- **Section 1: Introduction**: this first section provides the introductory information about the purposes of the DD and its structure.
- **Section 2: Architectural Design**: this section defines the core concepts of the whole document.
- **Section 3: Algorithm Design:** this section will be used to present the most important algorithms designed to fulfill the goals of the S2B.
- **Section 4: User Interface Design**: this section will provide a refinement of the mockups presented in the RASD and a UX diagram.
- **Section 5: Requirements Traceability**: this section will show how the design decisions taken in this document and the components identified are related with the requirements provided in the RASD.

2 Architectural Design

2.1 Overview

This section aims to present an overview of the architecture of the PowerEnjoy system, starting from the main software layers (also called abstraction layers or logical components) and their interconnections.

The software layers can be considered the main high-level components of the PowerEnjoy system. Then the high-level components will be exploded in their sub-components, also by presenting some static and dynamic behavioural views of the system using UML diagrams.

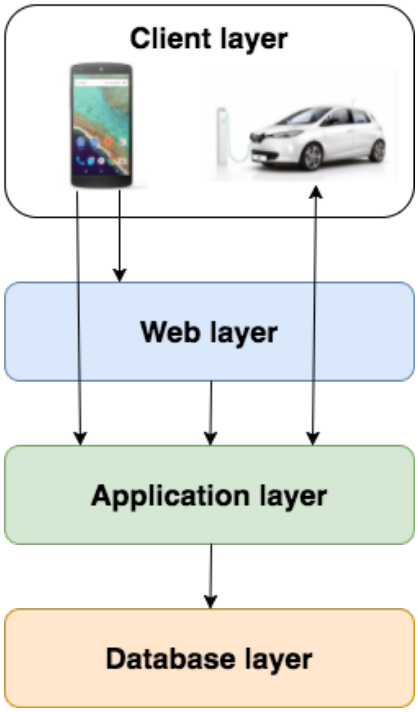
The end of this section will provide information about the main architectural styles and patterns used to design the system and their rationales.

2.2 High Level Components and Their Interaction

2.2.1 High Level Components

The system will be constituted by the following four high-level components (logical layers):

- **Database layer:** this layer is responsible for storing and retrieving the persistent data of the PowerEnjoy company. It doesn't implement any business logic and must guarantee the ACID properties during transactions. This layer is accessed through the Application layer.
- **Application (Business) layer:** the application layer implements the whole business logic and its algorithms. It is also responsible for the interaction with the cars (management and connection handling). It interacts with the database layer. Part of the business functionalities can be provided through service-oriented interfaces.
- **Web (Presentation) layer:** the web layer implements the web presentation. This layer can be used partially for the presentation on the mobile application. This layer does not involve any business logic.
- **Client Layer**
  - **Mobile application:** the mobile application used by the users and the operators, which runs on an Android or iOS device. The mobile application communicates directly with the application layer to access the main services offered by the PowerEnjoy system, but also with the web layer to delegate part of the presentation.
  - **Car application:** Android application running on a car owned by the PowerEnjoy company; it communicates directly with the application layer, since the car application will have its own native UI and there's no need to interact with the Web Layer for the presentation.



High level view of the layers of the system

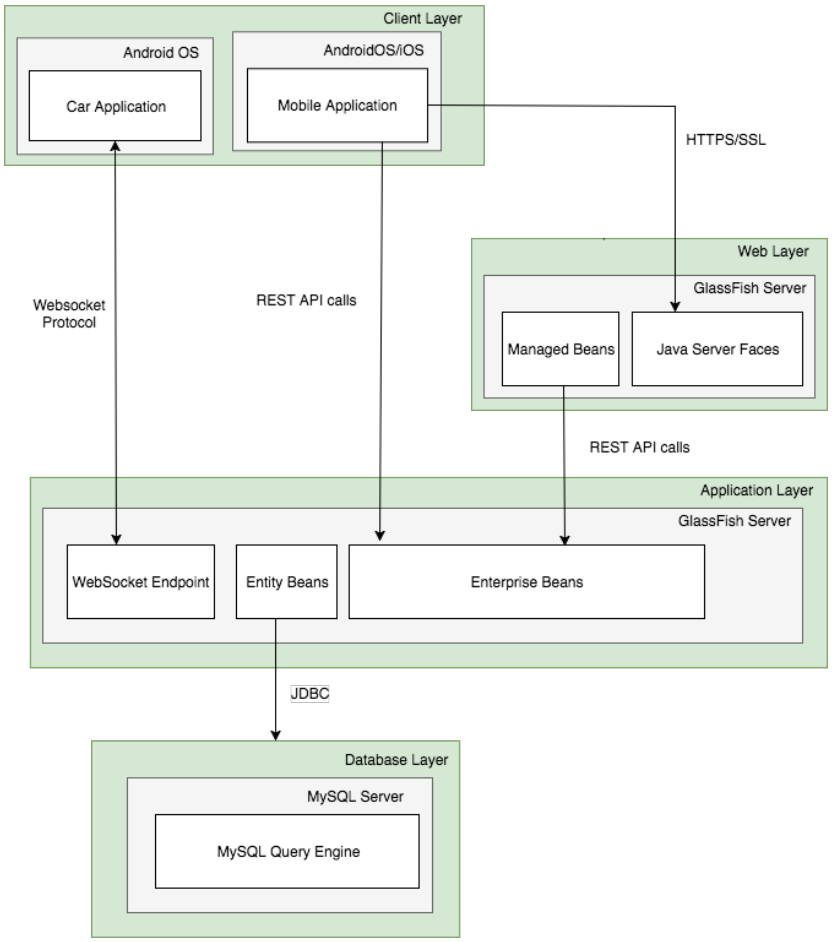
This design separates well the business logic and the web presentation, allowing more easily the extension of a layer and thus providing an adequate degree of scalability. In fact, as stated in the RASD, the architecture should be developed having in mind a future access to the PoweEnjoy services through a web browser and in this case the web presentation layer will become more important.
The four logical layers identified define a 4-tier architecture.
The architecture provided is not fully layered, this is mainly due to the interaction with the cars.

All the interactions between the layers can be considered synchronous, except the interaction between the Car Application and the Application layer, which is asynchronous.

2.2.2 Technological Viewpoint

The Mobile Application will be developed as an application on Android OS and on iOS.
The Car application will be developed as an application on Android OS.
The Web Layer and the Application Layer will be developed with Java EE 7. In particular the vendor implementation chosen is GlassFish.
For the Database Layer we will use MySQL Server.
The 4-layers previously presented, can be designed to follow an "API First" ar-

chitecture, where the focus is put on the services exposed by the system, thus on mobile and web devices. The following diagram shows the relations between the high-level components identified and a possible set of technologies to be used for the future implementation phase.

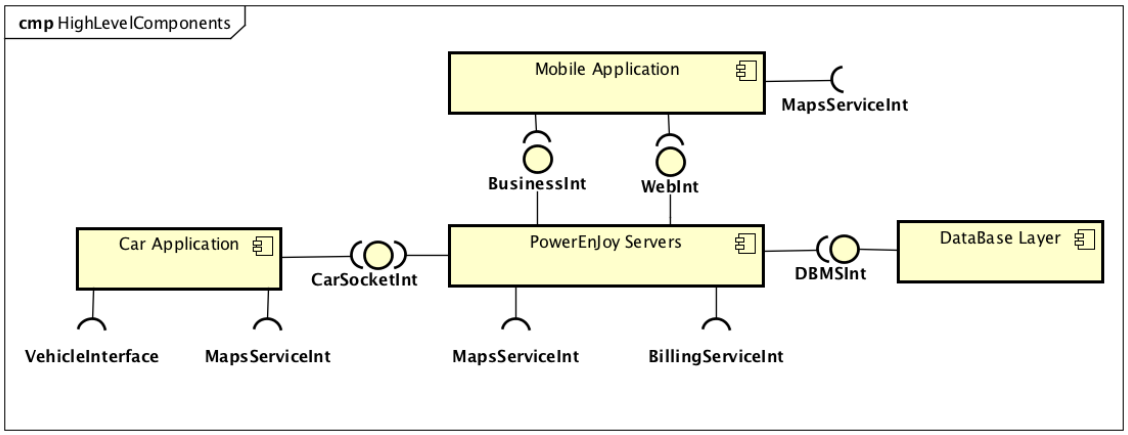


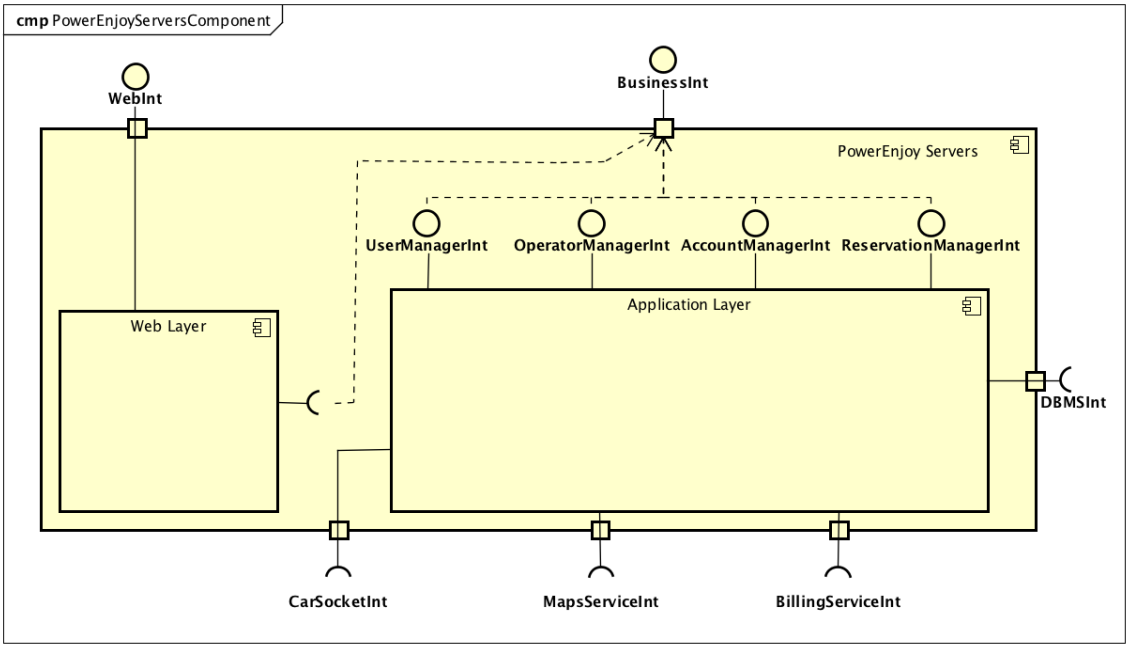
The interaction between the Web Layer and the Application Layer may use RESTful APIs or injection mechanism, depending whether these two layers are deployed on the same machine or not.

2.3 Component View

This section aims to provide more information about the sub-components identified from every high-level component previously mentioned.
Some UML Component diagrams will be provided in order to show the modular decomposition of a component into separate modules and the decoupling of the architecture in functional areas.

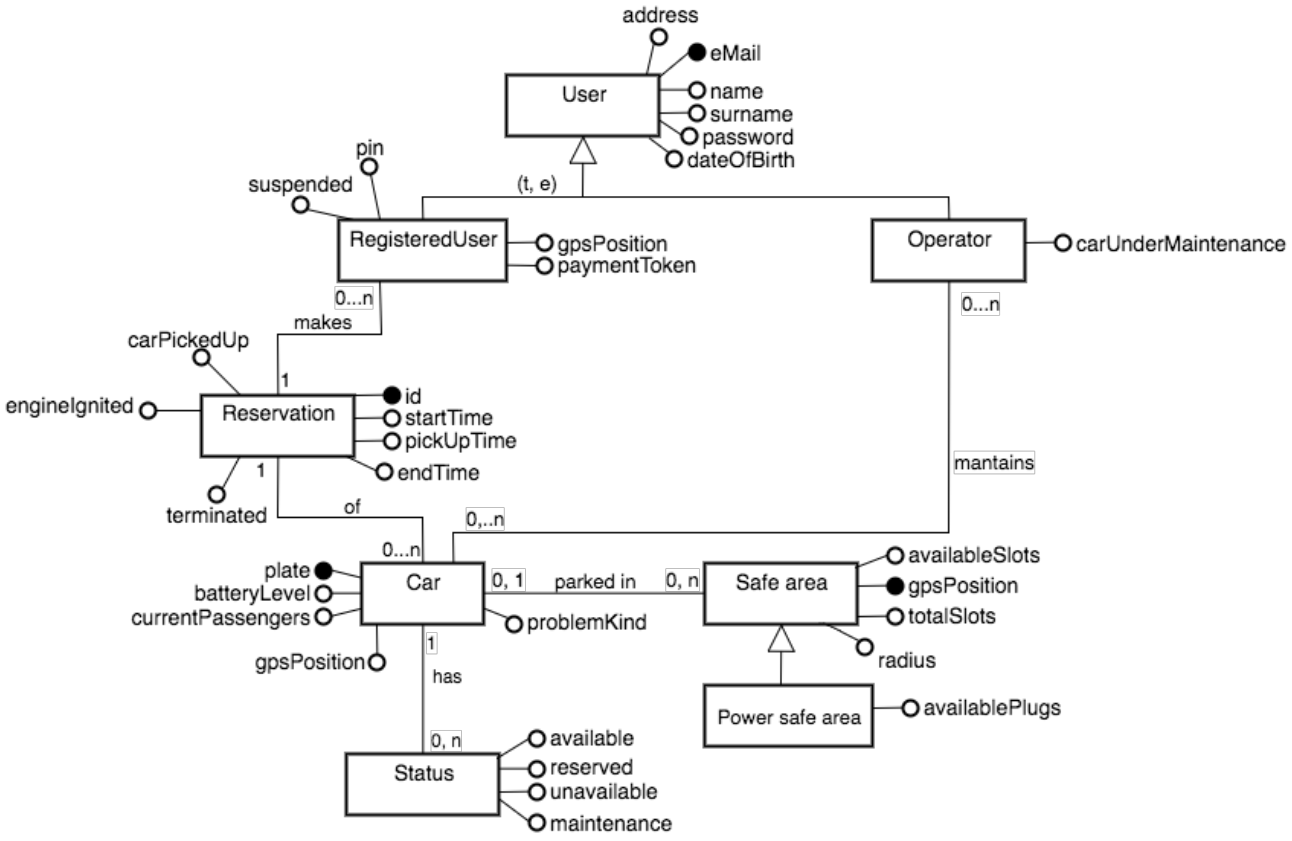
This section will follow a descriptive top-down approach. In designing the component diagrams we have tried to minimize the risks of desynchronization of the different areas of the architecture. The Web Layer and the Application layer are included for simplicity in an abstract component called "PowerEnjoy Servers".



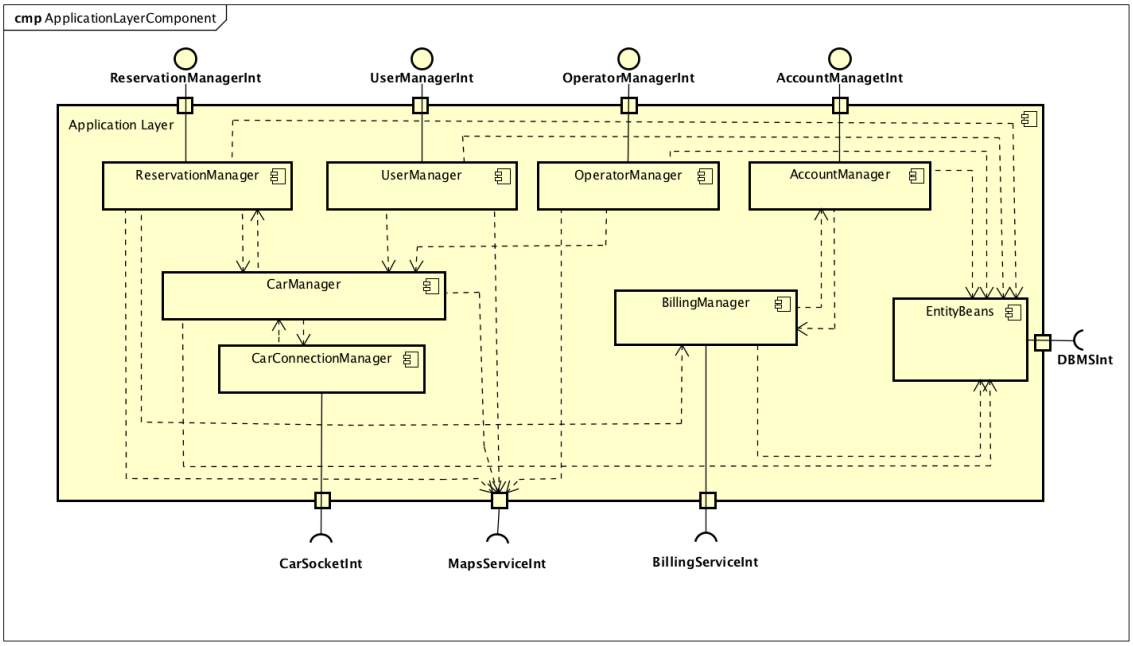


2.3.1 Database Layer

The database layer uses MySQL Server; for every transactions it must guarantee the ACID properties. It is interacted from the application layer using the Java Persistence API (JPA) instead of the classic JDBC. JPA provides a convenient shortcut to many of the steps that JDBC has to follow in order to store and retrieve the application data into the DBMS and to make the application layer synchronized with the database layer. JPA deals with the O/R Mapping and allows to store data by converting the application objects into persistence entities in the relational database, using annotations in Java classes. JPA manages the transition of entities from the application layer to the database layer through an API called the EntityManager. JPA abstracts JDBC which is used "under the hood". The following diagram shows the E/R Model of the PowerEnjoy company.



2.3.2 Application Layer



The main sub-components of the application layer are implemented using stateless Enterprise JavaBeans (Session Beans).

These sub-components provide the majority of the business logic of the system and are designed to follow the design principles of high cohesion and loose coupling, by offering functionalities strictly related and also by reducing the dependencies between each other.

Some of these beans implement the RESTful principles and so they offer some external services. The external services provided will be designed to follow the CRUD principles, so that each HTTPS request can be mapped to an appropriate service offered.

- **AccountManager**

This EJB is responsible for all the operations related to users' and operators' accounts. This bean is enough self contained to be easily extended in future with other operations related to account management, like: editing of account informations; deletion of a user account; providing a more complex policy to manage the suspension of a user from the PowerEnjoy system; modify the payment informations associated to an account.

- **UserManager**

This EJB manages the most used functionalities that are available through the mobile application to the users registered to the system. Among these function-

alities the most relevant are: locate all the available cars by GPS or address; insert the personal PIN to unlock the reserved car. This bean can be extended in future with functionalities like: contact an operator of the PowerEnjoy service to ask questions regarding the usage of the service.

- **OperatorManager**

This EJB provides all the features needed by the operators in order to accomplish their maintenance tasks: obtaining a list of all the unavailable cars; the possibility to set one of the unavailable cars as under maintenance; the possibility to set the fixed car back to an available state.

- **ReservationManager**

This EJB manages the features related to the management of a reservation of a car performed by a user, like: reserve an available car; understanding when to start counting the reservation time; evaluate if a reservation can terminate; evaluate if the user can make a stop; evaluate if a reservation has been missed; handle the reservation according to a car failure. This EJB may be extended in future to retrieve the list of all the reservations made by a user since its registration to the system, or to visualize the remaining time until the pickup of a reserved car will expire.

- **BillingManager**

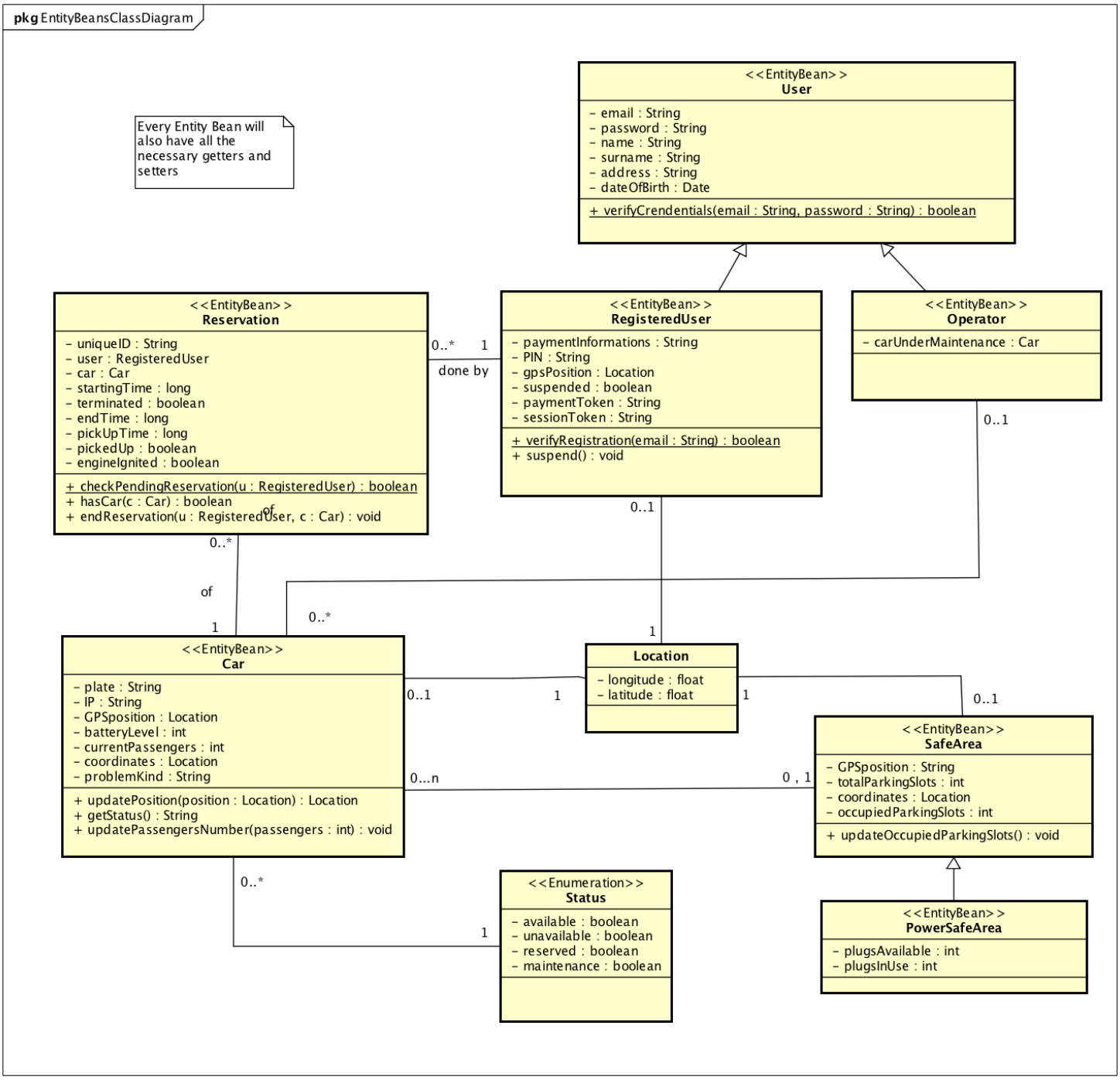
This EJB manages all the functionalities related to the user's billing, like: the verification of the payment information submitted by the user during the registration procedure; the computation of the user's bill, discounts and penalties at the end of a ride; the charge of a given amount of money to a user. In order to accomplish part of the billing functionalities, this EJB interacts with the BillingService.

- **CarManager**

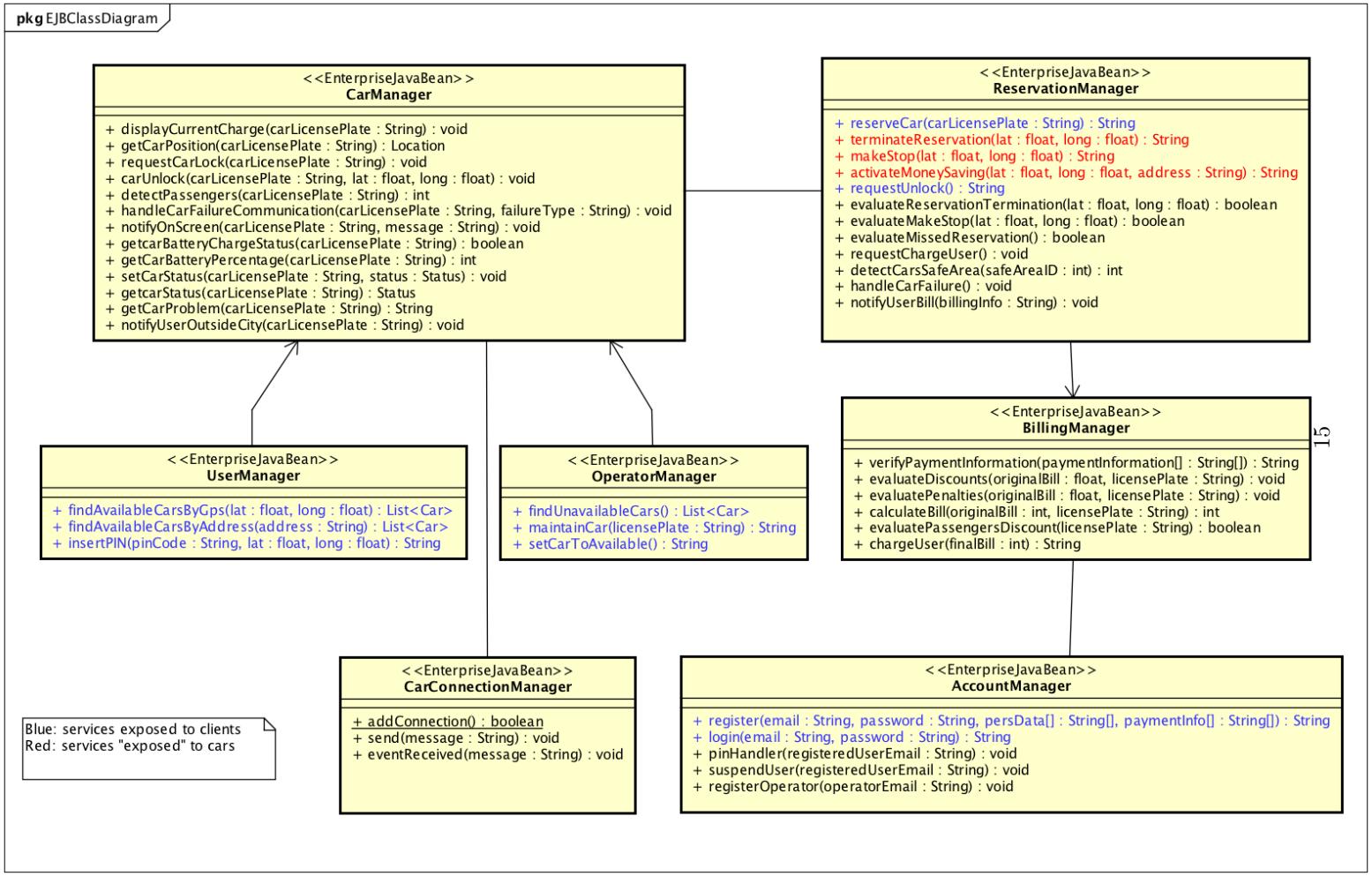
This EJB manages all the logic that determines the requests that need to be sent to the cars and handles all the logical consequences of the events that a car forwards to the the Application Layer. Any modification of a car property will be forwarded to the related entity bean in the EJB container, and JPA will handle the O/R mapping.

- **CarConnectionManager**

This EJB handles all the requests that the Application Layer needs to send to the Car Application and deals with the connection established with each car. This bean will be able to request any actuation of a command on a specified car; in particular, the communication with the car is done through a WebSocket endpoint.



Class diagram of the Entity Beans (up) and of the Enterprise Beans (down).


pkg EJBClassDiagram
<<EnterpriseJavaBean>>
CarManager
+ displayCurrentCharge(carLicensePlate : String) : void
+ getCarPosition(carLicensePlate : String) : Location
+ requestCarLock(carLicensePlate : String) : void
+ carUnlock(carLicensePlate : String, lat : float, long : float) : void
+ detectPassengers(carLicensePlate : String) : int
+ handleCarFailureCommunication(carLicensePlate : String, failureType : String) : void
+ notifyOnScreen(carLicensePlate : String, message : String) : void
+ getcarBatteryChargeStatus(carLicensePlate : String) : boolean
+ getCarBatteryPercentage(carLicensePlate : String) : int
+ setCarStatus(carLicensePlate : String, status : Status) : void
+ getcarStatus(carLicensePlate : String) : Status
+ getCarProblem(carLicensePlate : String) : String
+ notifyUserOutsideCity(carLicensePlate : String) : void
<<EnterpriseJavaBean>>
ReservationManager
+ reserveCar(carLicensePlate : String) : String
+ terminateReservation(lat : float, long : float) : String
+ makeStop(lat : float, long : float) : String
+ activateMoneySaving(lat : float, long : float, address : String) : String
+ requestUnlock() : String
+ evaluateReservationTermination(lat : float, long : float) : boolean
+ evaluateMakeStop(lat : float, long : float) : boolean
+ evaluateMissedReservation() : boolean
+ requestChargeUser() : void
+ detectCarsSafeArea(safeAreaID : int) : int
+ handleCarFailure() : void
+ notifyUserBill(billingInfo : String) : void
<<EnterpriseJavaBean>>
UserManager
+ findAvailableCarsByGps(lat : float, long : float) : List<Car>
+ findAvailableCarsByAddress(address : String) : List<Car>
+ insertPIN(pinCode : String, lat : float, long : float) : String
<<EnterpriseJavaBean>>
OperatorManager
+ findUnavailableCars() : List<Car>
+ maintainCar(licensePlate : String) : String
+ setCarToAvailable() : String
<<EnterpriseJavaBean>>
BillingManager
+ verifyPaymentInformation(paymentInformation[] : String[]) : String
+ evaluateDiscounts(originalBill : float, licensePlate : String) : void
+ evaluatePenalties(originalBill : float, licensePlate : String) : void
+ calculateBill(originalBill : int, licensePlate : String) : int
+ evaluatePassengersDiscount(licensePlate : String) : boolean
+ chargeUser(finalBill : int) : String
<<EnterpriseJavaBean>>
CarConnectionManager
+ addConnection() : boolean
+ send(message : String) : void
+ eventReceived(message : String) : void
<<EnterpriseJavaBean>>
AccountManager
+ register(email : String, password : String, persData[] : String[], paymentInfo[] : String[]) : String
+ login(email : String, password : String) : String
+ pinHandler(registeredUserEmail : String) : void
+ suspendUser(registeredUserEmail : String) : void
+ registerOperator(operatorEmail : String) : void
Blue: services exposed to clients
Red: services "exposed" to cars

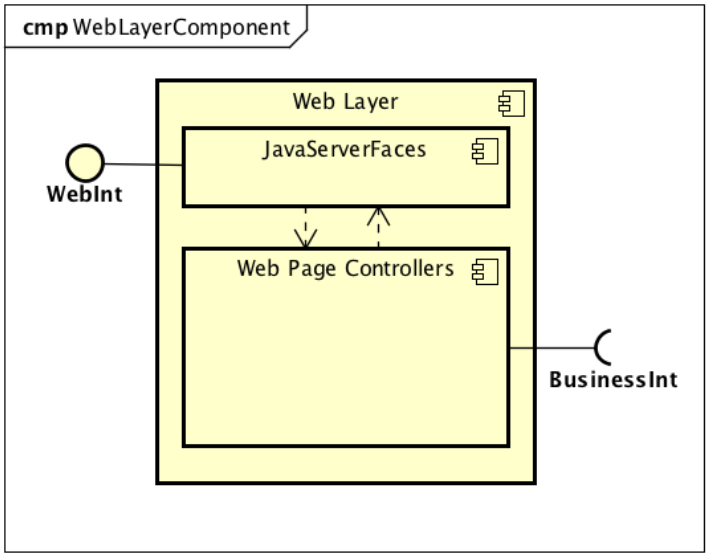
2.3.3 Web Layer

This layer is responsible to handle presentation functionalities. Among the multiple available web components in Java EE 7 specification (servlets, JSP), we have decided to use the JSF framework (JavaServer Faces) because they are designed based on the MVC pattern in order to separate precisely the roles of the View and the Controller. The view can be implemented using static or dynamic HTML web pages, the controller can be implemented using stateless session beans. The model is the core of the PowerEnjoy system and it's accessed from the Application Layer.

For the first release of the PowerEnjoy system, we won't provide a web application accessible completely through a web browser. However this layer will be immediately useful for the Mobile Application, since for some functionalities, like registration, we can use this layer instead of creating twice (Android, iOS) the native UIs.

This layer may also be useful in future to provide with ease, also on Mobile Applications, some "FAQs", "Q&A", "Billing Guides" and basic informations about the PowerEnjoy service.

The following is a component diagram exploding the internal of the Web Layer at a very high level.



2.3.4 Client Layer

Mobile Application

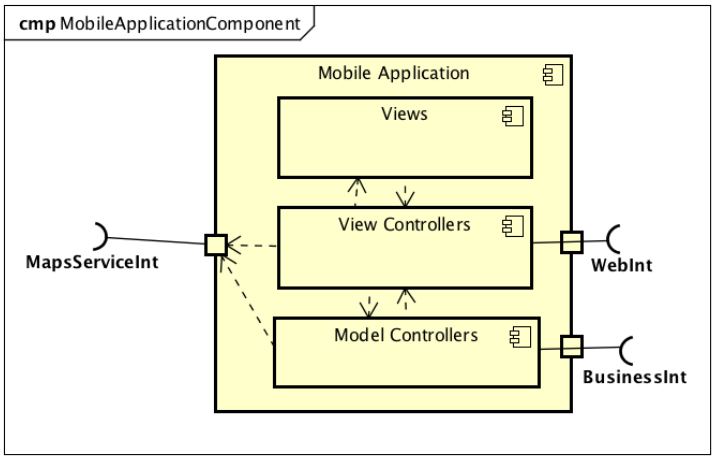
The Mobile Application will be natively developed using Android SDK and iOS SDK respectively for Android OS and iOS. The language used for the Android Application is Java, the language used for the iOS application is Swift.

In general, for both platforms we will use the MVC pattern because:

- The distinction between view and controlled is already implicitly present in both the above mentioned SDKs, since layouts, images, assets and media are considered view resources, which get coupled at runtime with the view controllers code.
- The distinction between controllers and model (logic) can be done more easily since we access remote services and we keep all the business logic separately on the Application Layer.

We've made a distinction between two types of Controllers:

- **View Controllers**: controllers responsible to handle view events. This controllers will also deal with the interaction with the MapsService, since the MapsService uses specific views, and with the WebInt through Web Views.
- **Model Controllers**: controllers responsible to:
  - interact with the model offered by the Application Layer through external interfaces.
  - do some necessary elaborations, like parsing the response obtained by the Application Layer.



Some Web Views will be used to offer part of the PowerEnjoy functionalities (like the registration), without having to write them natively on each platform. See Web Layer for more informations.

- **Android**
  - **View Controller**: we can use Activity from *android.support.v7.app.AppCompatActivity*. We use the AppCompatActivity to deal more easily with multiple Android OS versions, in order to provide similar experiences to users.

- **Model Controller**: we can use again *android.support.v7.app.AppCompatActivity*.
- **View**: the native UI view layout is defined with image assets and with XML layout files, one for every View Controller. We can use all the UI elements from *android.view.View*, *android.app.Dialog* and *android.webkit.WebView*, like: buttons, pickers, dialogs, switches, web views etc.
- **GPS positioning**: in order to deal with GPS positioning, we can use the APIs provided by the LocationManager from *android.location.LocationManager*. Special permissions "ACCESS_COARSE_LOCATION" and "ACCESS_FINE_LOCATION" will be needed in the application manifest file.
- **Internet connection**: can be used immediately by adding "INTERNET" and "ACCESS_NETWORK_STATE" permissions in the application manifest file.
- **MapsService**: the MapsService is already part of the Android SDK, it should only be configured properly.

- **iOS**
  - **View Controller**: we can use UIViewController.
  - **Model Controller:** for this part we can use custom Swift classes using specific functionalities provided by iOS SDK.
  - **View**: we can use UIView subclasses, and also UIWebView.
  - **GPS Positioning**: we can use the CoreLocator APIs.
  - **Internet Connection**: we don't need special permissions to use internet connection.
  - **Maps Service**: can be used by integrating separately the Google Maps SDK for iOS.

Car Application

The Car Application will be developed with the Android SDK. The whole code will be written using the Java language.

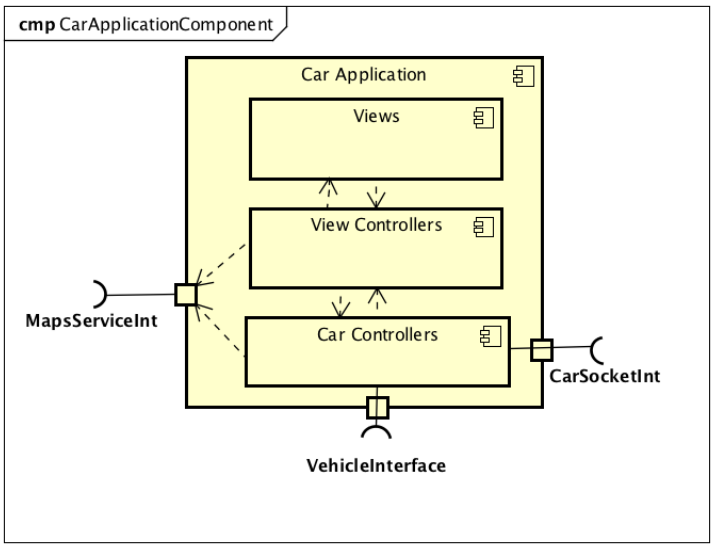
The general concepts about Views and View Controllers are the same as the ones provided for the Android Mobile Application in the previous section, however, there are differences in terms of the interaction with external interfaces and in the communication with the Application Layer.

The Car Controllers are responsible to interact with the car's hardware and sensors and to manage the network communication with the Application Layer. They also partially interact with the MapsServiceInt for some path discovery and map navigation functionalities.

The network communication between the Car Application and the Application Layer is based on a full-duplex communication channel. The Application Layer plays an active role in this communication, meaning that it can push messages to the Car Application. See other design decisions.

The Car Application will interact with:

- **VehicleInterface**: this interface will be used to get informations from the car's hardware and sensors (like battery SoC) and to actuate commands on the car hardware itself (like locking the car).
- **MapsServiceInt**: this interface (or set of interfaces), will be used to interact with Google Maps services, for functionalities like: map visualization; find a feasible path from an origin GPS position to a destination GPS position; car navigation functionalities; visualization of safe areas on the map view etc.
- **CarSocketInt**: the interface used in the full-duplex network communication with the Application Layer. This communication is based on the WebSocket protocol. //aggiungi che libreria usi per usare websocket su Java.

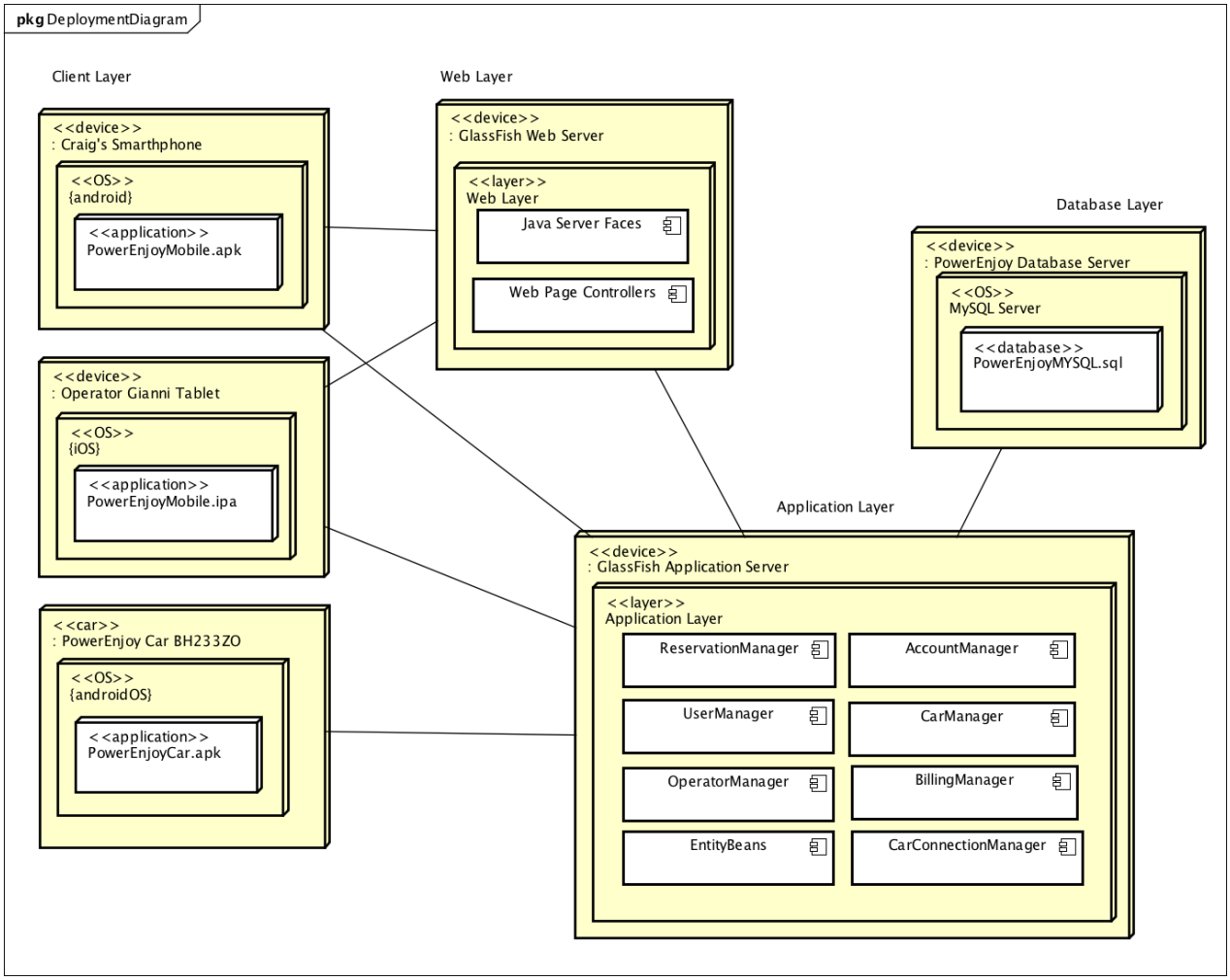


2.4 Deployment View

This section aims to show the topology of the system architecture, by underlining anything that exists in its operational context.

Usually the mapping between tiers and physical machines is not 1:1. However we have decided to deploy our 4 tiered architecture on 4 different machines: client device, Glassfish Web Server, GlassFish Application Server, mySQL Server.

This diagram shows a possible deployment of our artifacts and components on physical machines.



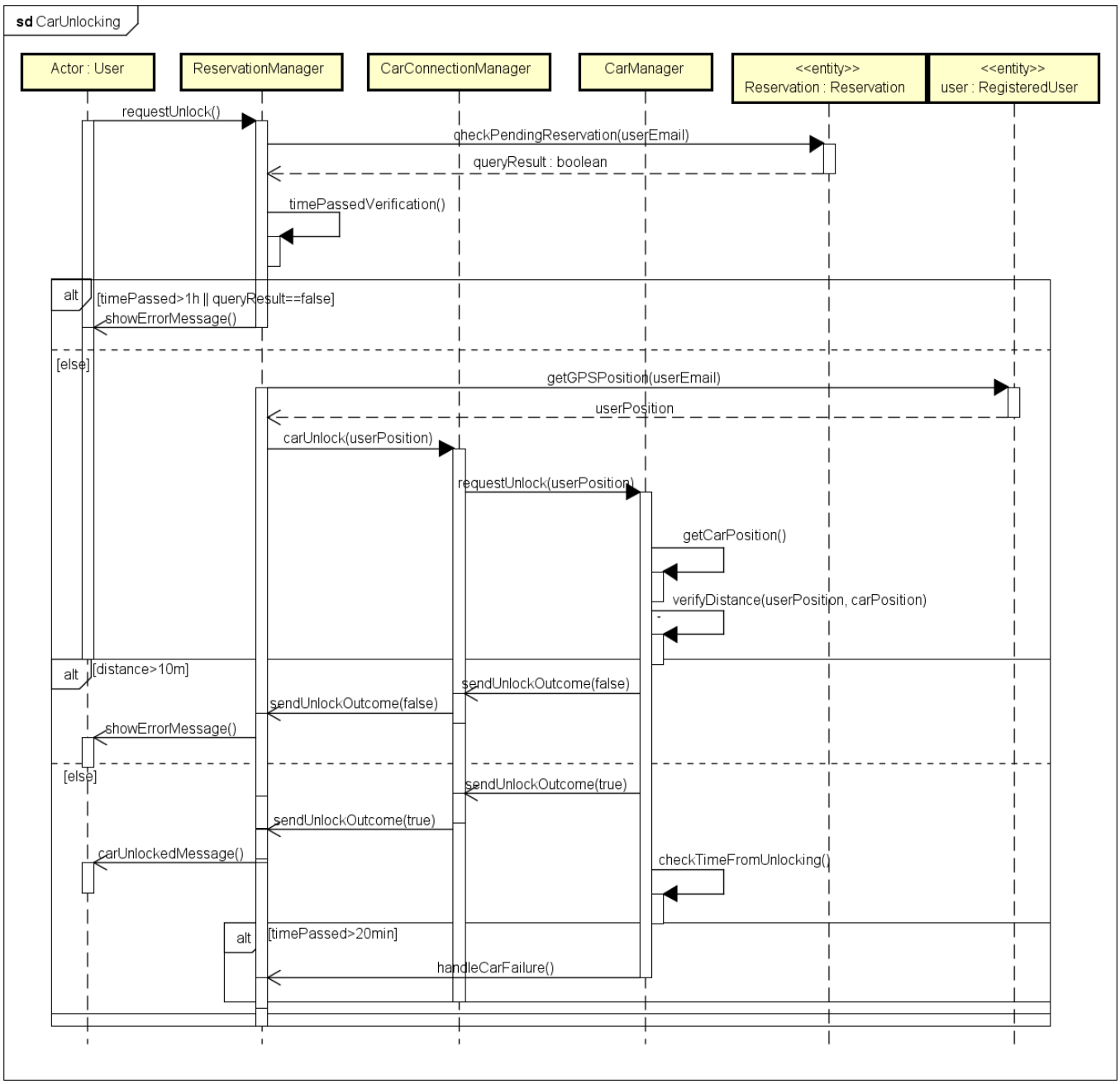
2.5 Runtime View

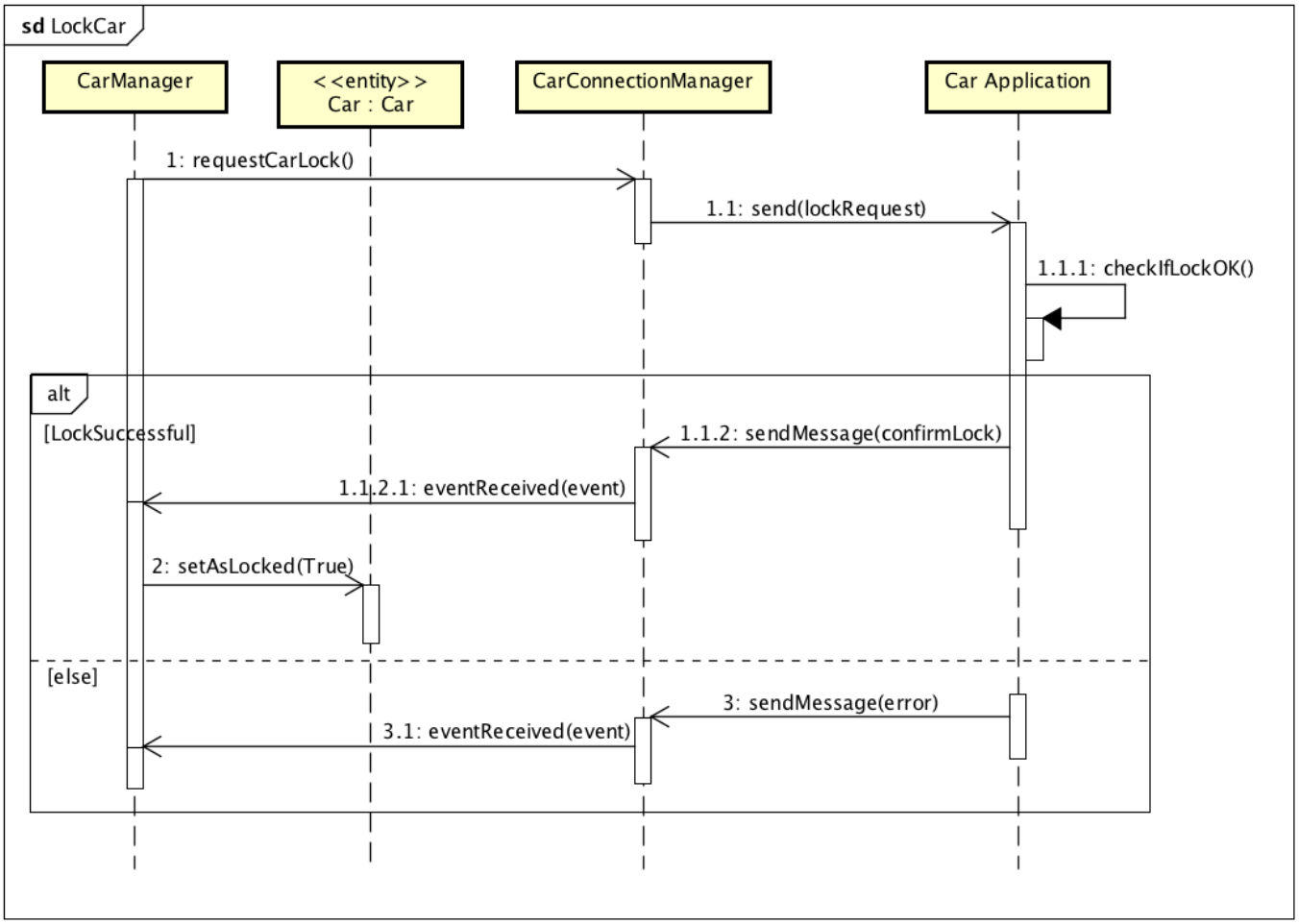
This section aims to show part of the runtime behaviour of the system mainly through sequence diagrams. These diagrams represent the most significant interactions between components.

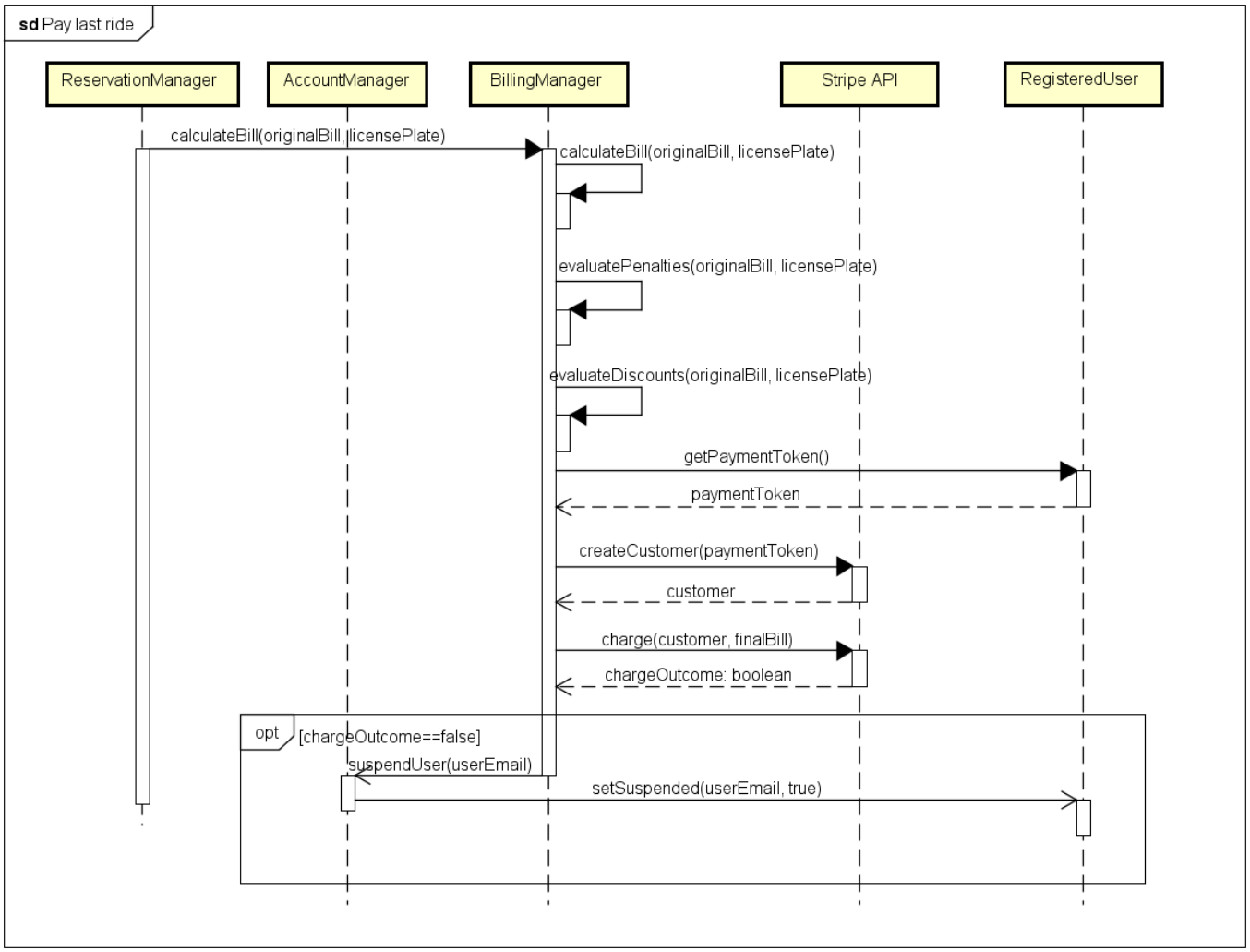
Many interactions have already been shown in the RASD document, using high level sequence diagrams where the PowerEnJoy system was treated like a "black box". Here, instead, we want to focus more on the internal interactions between components that occur in the PowerEnJoy system.

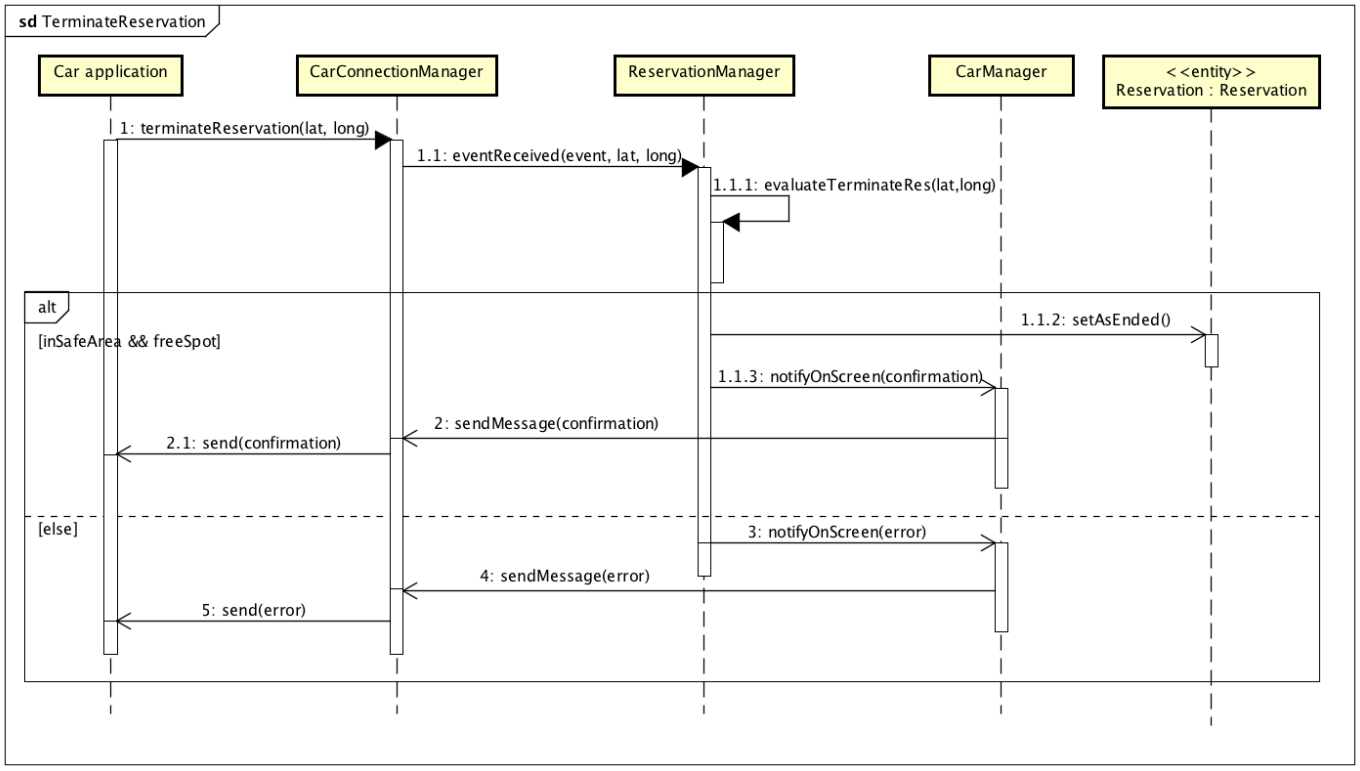
At the end of the section we will provide a state chart diagram regarding the evolution of the status of a car and the events triggering its transitions.

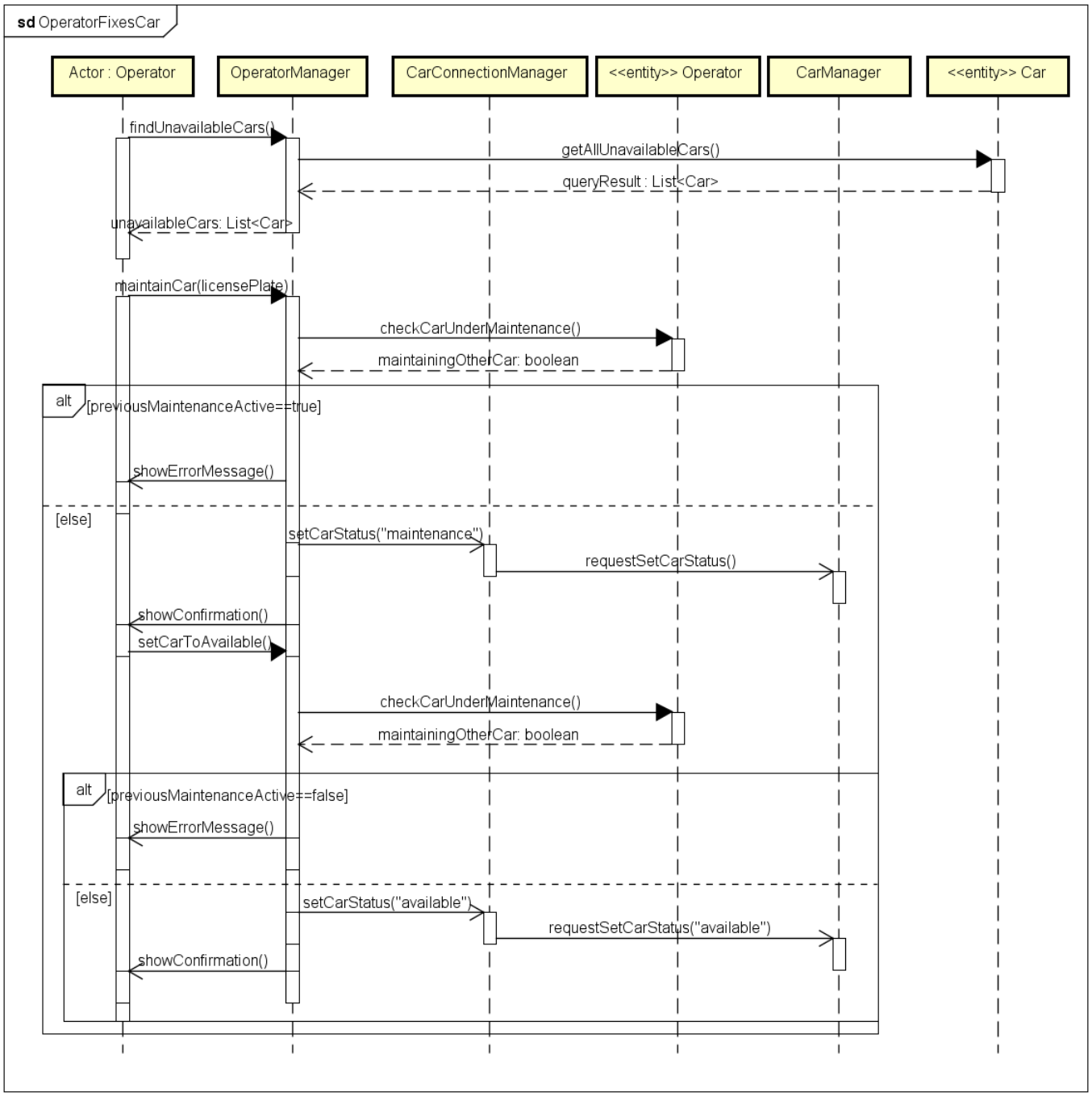
2.5.1 Sequence Diagrams



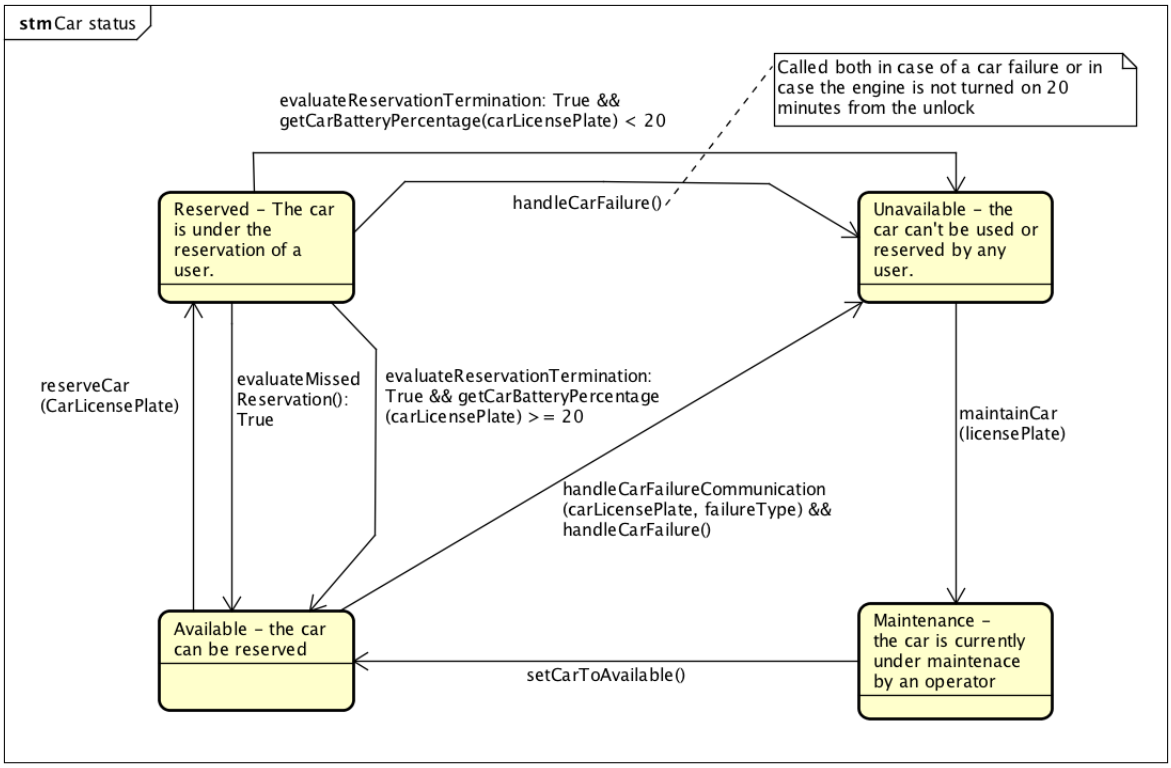

sd LockCar
CarManager
<<entity>>
Car : Car
CarConnectionManager
Car Application
1: requestCarLock()
1.1: send(lockRequest)
1.1.1: checkIfLockOK()
alt
[LockSuccessful]
1.1.2: sendMessage(confirmLock)
1.1.2.1: eventReceived(event)
2: setAsLocked(True)
[else]
3: sendMessage(error)
3.1: eventReceived(event)


sd Pay last ride
ReservationManager
AccountManager
BillingManager
Stripe API
RegisteredUser
calculateBill(originalBill, licensePlate)
calculateBill(originalBill, licensePlate)
evaluatePenalties(originalBill, licensePlate)
evaluateDiscounts(originalBill, licensePlate)
getPaymentToken()
paymentToken
createCustomer(paymentToken)
customer
charge(customer, finalBill)
chargeOutcome: boolean
opt
[chargeOutcome==false]
suspendUser(userEmail)
setSuspended(userEmail, true)


sd TerminateReservation
Car application
CarConnectionManager
ReservationManager
CarManager
<<entity>>
Reservation : Reservation
1: terminateReservation(lat, long)
1.1: eventReceived(event, lat, long)
1.1.1: evaluateTerminateRes(lat,long)
alt
[inSafeArea && freeSpot]
1.1.2: setAsEnded()
1.1.3: notifyOnScreen(confirmation)
2: sendMessage(confirmation)
2.1: send(confirmation)
[else]
3: notifyOnScreen(error)
4: sendMessage(error)
5: send(error)


sd OperatorFixesCar
Actor : Operator
OperatorManager
CarConnectionManager
<<entity>> Operator
CarManager
<<entity>> Car
findUnavailableCars()
getAllUnavailableCars()
queryResult : List<Car>
unavailableCars: List<Car>
maintainCar(licensePlate)
checkCarUnderMaintenance()
maintainingOtherCar: boolean
alt
[previousMaintenanceActive==true]
showErrorMessage()
[else]
setCarStatus("maintenance")
requestSetCarStatus()
showConfirmation()
setCarToAvailable()
checkCarUnderMaintenance()
maintainingOtherCar: boolean
alt
[previousMaintenanceActive==false]
showErrorMessage()
[else]
setCarStatus("available")
requestSetCarStatus("available")
showConfirmation()

2.5.2 StateChart



2.6 Component Interfaces

2.6.1 Interface between Car Application and Application Layer

The Car Application communicates with the Application Layer by sending string messages over the WebSocket protocol: it is necessary then to define a common interface composed by a shared list of possible types of messages, so that both the parties will be able to communicate using an agreed message protocol. The services listed below are considered public, since they are accessed by the car Application, even if they are not exposed as RESTful services.

- terminateReservation
  - INPUT:
    - carLatitude
    - carLongitude
  - OUTPUT:
    - terminationRequestOutcome

This function allows a registered user to terminate his reservation by pressing on the related button on the reserved car's screen. It receives as parameter

the GPS location of the car (characterized by latitude and longitude). It calls the local method evaluateReservationTermination(latitude, longitude). Only if the latter method returns a true value the reservation can actually terminate. It returns a terminationRequestOutcome containing the outcome, that will be displayed on the car's screen: reservation ended correctly or error message.

- makeStop
  - INPUT:
    - carLatitude
    - carLongitude
  - OUTPUT:
    - stopRequestOutcome

This function allows a registered user to make a stop during his ride by pressing on the related button on the reserved car's screen. It receives as parameter the GPS location of the car (characterized by latitude and longitude). It calls the local method evaluateMakeStop(latitude, longitude). Only if the latter method returns a true value the user can actually make a stop. It returns a stopRequestOutcome: success or error_message, that will be displayed on the car's screen.

- activateMoneySaving
  - INPUT:
    - carLatitude
    - carLongitude
    - destinationAddress
  - OUTPUT:
    - destinationPowerSafeArea or errorMessage

This function allows a registered user to activate the money saving option from the reserved car's screen. It receives as parameters the current GPS position of the car (characterized by latitude and longitude) and the destination of the user (specified as an address). The function returns in the parameter destinationPowerSafeArea the GPS location of the selected power safe area where to park the car at the end of the ride in order to possibly get a discount. The safe area selected guarantees a uniform distribution of cars in the city area and depends on both the available parking slots and on the distance from the user's destination. If no power safe area is found satisfying the above criteria, it returns an errorMessage.

2.6.2 Interface exposed by the Database Layer to Application Layer

The Application Layer interacts with the Database Layer by using the interfaces offered by the Java Persistence API: JPA, in particular, enables the status of the Entity Beans of the Application Layer to be persisted in a relational database. In particular we will use the Hibernate implementation of the JPA specification.

2.6.3 Interface exposed by the Web Layer to Mobile Application

The Mobile Application interacts with the Web Layer by using the interfaces offered through the HTTPS protocol, allowing a secure communication between the clients and the Web Layer. For the initial release of our system, this interface will be used only for the registration procedure. However, this interface could assume a more significant role in future updates of the system: by already adapting this interface, in fact, an eventual expansion to desktop browsers would be more easily feasible.

2.6.4 Interfaces exposed by Application Layer to Mobile Application and Web Layer

The following are the public APIs exposed by some components of the Application Layer. Not all the components of the Application Layer offer public APIs. These APIs can be exposed using a RESTful approach.

The output of each exposed functionality can be wrapped in an XML file, an HTML page or a JSON. We think JSON is the most versatile response format, however the choice is at discretion of the development team. Since this is not particularly relevant, we will only show the outcome at a high level.

Every service whose functionality is associated and accessed by a particular user, requires also as input a token, which identifies the session of that user. This is necessary since the RESTful approach requires the user session to be stateless.

AccountManagerInt:

- register
  - INPUT:
    - userEmail
    - userPassword
    - personalData[]
    - paymentInformation[]
  - OUTPUT:
    - registrationOutcome

This function allows guests to register to PowerEnJoy. It creates a new user with the information provided as parameters and saves it as an entity in the database of the system. It returns a registrationOutcome representing the outcome of the registration procedure: registration successful or failed. The paymentInformation[] are not stored in the database. More informations about the management of payment information is shown in the Other Design Decision section.

- login
  - INPUT:

        * userEmail
        * userPassword
    - OUTPUT
        * session_identifier_token or error_info

This function allows a user to log into the PowerEnJoy system using his email and password. If the credentials are correct, it returns a session_identifier_token that identifies the session of the user during his interaction with the system; otherwise it returns an error_info message.

UserManagerInt:

- findAvailableCarsByGPS
    - INPUT:
        * userLatitude
        * userLongitude
    - OUTPUT:
        * available_cars_list
        * available_cars_info

This function allows a registered user to find available cars starting from the GPS position of his mobile device (characterized by the attributed latitude and longitude) and to see their info. It returns a list containing all available cars in the city area and their related informations.

- findAvailableCarsByAddress
    - INPUT:
        * address
    - OUPUT:
        * available_cars_list
        * available_cars_info

This function allows a registered user to find available cars starting from an address in the city area and to see their info. It returns a list containing all available cars in the city area and their related informations.

- insertPIN
    - INPUT:
        * uniquePIN
        * userLatitude
        * userLongitude
    - OUTPUT:

        * PINvalidationOutcome

This function allows a registered user to insert his unique PIN code on his mobile device in order to unlock his previously reserved car. It receives the PIN code and the user's GPS position (latitude and longitude, retrieved through the mobile device's GPS signal). This function checks that the PIN is correct. It returns a PINvalidationOutcome containing the message that the PIN insertion was successful if the above conditions are valid, otherwise containing an error message.

OperatorManagerInt:

- findUnavailableCars
  - OUTPUT:
    * unavailable_cars_list
    * unavailable_cars_problems
    * unavailable_cars_info

This function allows an operator to find all unavailable cars and to see their info and the kind of problem they have. It returns a list containing all unavailable cars in the city area, with related informations and problem occured.

- maintainCar
  - INPUT:
    * carLicensePlate
  - OUTPUT:
    * maintenanceEnabled

This function allows an operator to set as under maintenance an unavailable car, identified by its license plate. It returns a maintenanceEnabled containing the request's outcome: successful or unsuccessful, depending if the operator had another car currently under his maintenance.

- setCarToAvailable
  - OUTPUT:
    * requestOutcome

This function allows an operator to set as available a car he maintained. It returns a requestOutcome containing the outcome of the request: successful or unsuccessful, depending if the operator did not have any car under his maintenance.

ReservationManagerInt:

- reserveCar
  - INPUT:
    - carLicensePlate
  - OUTPUT:
    - reservationOutcome

This function allows a registered user to reserve an available car, identified by its license plate. It also asynchronously enables the verification of the time passed from the reservation, in order to set the car back to available if the user doesn't unlock it within one hour. It returns a reservationOutcome representing the outcome of the reservation request: reservation successful or failed.

- requestUnlock
  - OUTPUT:
    - unlockRequestOutcome

This function allows a registered user to unlock his latest reserved car. It verifies that a reservation for the user that requested the unlock actually exists and that the time passed from the reservation is no more than one hour. If so, it calls the method carUnlock inside CarManager, getting the user's GPS position from his mobile device. It also starts an asynchronous verification of the time passed from the unlocking, so that, if the engine is not ignited within 20 minutes, the method handleCarFailure inside reservationManager is called. This function returns a unlockRequestOutcome representing the outcome of the unlocking request performed by the user.

2.7 Selected Architectural Styles And Patterns

This section aims to list all the architectural styles and patterns applied during the design phase of the system, underlining also the rationales behind the main design decisions.

2.7.1 Architectural Styles

- **Client/Server:** if we abstract away some details, our architecture is implementing a Client/Server architecture, even if we don't have only 2 tiers. This is mainly due to the fact that the most computationally intensive business logics are located inside the Application (Business) Layer, while the Client Layer has only to deal with presentation functionalities (completely, or partially if it is interacting with the web layer).
  The client layer is therefore left "thin" regarding the business logic, while the Application Layer is "fat".
  This is advantageous for multiple reasons:

- Avoid platform desynchronization:
The main devices accessing the PowerEnjoy services are mobile devices. The SDKs used to develop Android or iOS applications change frequently and this needs to be accounted to guarantee the best possible user experience. If the business logic is located too much on the Client Layer, this may cause desynchronization between the platforms, ending up privileging updates to a platform instead of another.

- Device support flexibility:
With a "thin" Client Layer and a "fat" Application Layer it is easier to add support to new client devices, for example web browsers.

- Maintenance:
A "fat" Application Layer improves maintenance, since the changes in the business logic are encapsulated in the Application Layer.

- **4-Tier Architecture:** the separation of our system in 4 software layers (tiers) can be useful for a number of reasons:

  - Process Flexibility:
  Process flexibility is enhanced since during the implementation phase, each layer can be assigned to different teams of people, which can proceed the development in parallel.

  - Device support flexibility:
  Same rationales as in the Client/Server explanation.

  - Maintenance and testing:
  Every software layer can be tested and maintained more or less separately from the others. Therefore it will be easier to identify and correct eventual bugs.

  - Scalability:
  The number of machines on which a particular layer is deployed can be incremented or decremented based on demand loads and budget.

  - Modularity and extendability:
  Since each software layer as a specific functionality to achieve, it is easier to add modules and components to it.

  - Security:
  If in the future the security of the system becomes a bigger concern, then it will be easier to create multiple DMZ and add firewalls in order to protect more efficiently the business data.

- **Active Server (Server Push):** in the HTTP protocol the server is passive and it is able to send data to the client only in response to a client's request.
  In the PowerEnjoy system, though, it is necessary from the Application Layer to request different commands to be actuated on the cars when needed.
  The HTTP protocol offers some workarounds to this issue, such as HTTP

polling, by basically putting the client in a situation of periodically sending requests to the server to know if there are some new updates destinated to him.
This technique though is pretty expensive in terms of time and resources, so it seems preferable to use another communication protocol which guarantees a full-duplex communication channel, such as the WebSocket protocol.
In addition, JEE offers the WebSocket APIs for creating and managing easily WebSocket components.

2.7.2 Design Patterns

- **MVC:** this design pattern has been followed for both the Mobile Application, Car Application and the Web Layer. It is well known for the separation of concerns between the roles of model, view and controller.
- **Publisher/Subscriber:** this design pattern can be used for the communication between the Application Layer and the Car Applications over the WebSocket protocol.

2.8 Other Design Decisions

- **System configuration**
  Since we are not providing any particular interface for administrators, the configuration of the system will be done using various files. Every bean needing a configuration setting will read from an appropriate file. In these configuration files we have at least:
  - A list of tuples <safeArea or powerSafeArea, GPS position, safeAreaRadius>
  - All the emails and passwords of the operators.
  - All the settings needed by each tier.
- **BillingService: Stripe API**
  We will use Stripe APIs to validate the payment informations provided by the users and to charge the users when needed. The main reason for this choice are:
  - Security:
    By using Stripe APIs we will limit the storage of personal payment information in our database. After checking the payment informations provided by the user through Stripe APIs, Stripe APIs return us a payment token that identifies that payment informations inside their database. We just need to store this token in our database and retrieve it when we will need to charge the user using Stripe APIs.
  - Simplicity and Good Documentation:
    The Stripe APIs are simple to use and very well documented.

The following sample code can be used to charge a user:

```
// Create a Customer

Map<String, Object> customerParams = new HashMap<String, Object>();
// here we use the payment token retrieved after the validation
// of the user's payment information performed by Stripe
customerParams.put("source", token);
customerParams.put("description", "Mario Rossi");
Customer customer = Customer.create(customerParams);

// Charge the Customer

Map<String, Object> chargeParams = new HashMap<String, Object>();
chargeParams.put("amount", 1000); // Amount in cents
chargeParams.put("currency", "eur");
chargeParams.put("customer", customer.getId());
Charge.create(chargeParams);
```

- **MapsService: Google Maps**
  We have chosen to use Google Maps APIs to deal with maps visualization; geolocalization on a map; path search; distance calculation between GPS positions. We have made this choice since:
  - Google Maps APIs are the most popular maps API.
  - Google Maps APIs are easy to use: the APIs provided are wide and very well documented.
  - Google Maps APIs are portable, since Google has released them for all the most used software platforms.
  - Google Maps APIs provide a very high availability.

- **VehicleInterface**
  The VehicleInterface will be used by the Car Application in order to interact with the hardware and sensors of the car. Through this interface the Car Application will be able to get informations from the car and also to actuate commands on the car's hardware. The cars' provider from which our customer has bought the cars will provide us a library which will contain the above mentioned interface and all the code necessary for that interface to accomplish its tasks.
  This library can be included in the Car Application source code during the development.
  An example of the VehicleInterface is the following:
  - `getSoC();`
  - `getCurrent();`
  - `getBatteryStatus();`
  - `getVoltage();`

- `getDoorsStatus();`
- `getTrunkDoorStatus();`
- `getDirection();`
- `getSpeed();`
- `getFaults();`
- `getSpecificFaultInformations(fault);`
- `getPassengersNumber();`
- `execCommand(command, params[]);`

3 Algorithm Design

3.1 Money Saving Option

The money saving option is a feature offered by the PowerEnjoy system to its users: its purpose is to guarantee a uniform distribution of the cars in the city area, considering both the destination of the registered users and the availability of power plugs in the safe areas. As stated in the RASD document, the safe areas are distributed uniformly in the city area (RASD, 2.5.2, D17), only the safe areas known as "power safe area" are provided with power plugs (RASD, 1.6) and every parking slot in a power safe area has a dedicated and working power plug (RASD, 2.5.2, D14): the following algorithm, then, considers only the power safe areas as possible destinations for the registered users, evaluating both the distance from the original destination of the user and the occupied spots in the power safe areas. The algorithm focuses on offering the closest power safe area to the user's original destination while realizing a uniform distribution of the cars in the city, so that the saving money option can be more gladly used by the users of PowerEnjoy.

The algorithm is based on two arrays defined as **spots** and **distance**, further on called **s** and **d** for short. Notice that the cells of the two arrays corresponds to the ordered set of power safe areas saved in the system: so the first cell of array **s** and **d** corresponds to the first power safe area saved in the system and so on. The algorithm also uses a list of integers **q**. The dimension **n** of both the arrays is equal to the number of power safe areas saved in the system.

Steps of the algorithm:

- initialize **q** as an empty list;
- update **s** by checking how many spots are occupied in each power safe area and inserting the number in the corresponding cell of **s**;
- update **d** by getting the distance (in meters) of each power safe area from the user's destination, using the Google Maps API;
- calculates the average **a** of the occupied spots by summing the values of the array **s** and dividing the sum by **n**;
- executes the following cycle:
  - if **q** has all **n** indices, the algorithm stops and returns the index **i** of the minimum value in the array **d** (that's when the algorithm starts from a situation of uniform distribution and offers to the user the closest power safe area);
  - finds the minimum value in the array **d** without considering the cells whose index is saved in the list **q** and save the corresponding index **i**;

– check if the cell at index **i** of the array **s** is less than **a** and if the associated power safe are in the system has at least one available spot; if the occupied spots are strictly less than the average **a** and if there is at least one spot available, the iteration stops and the algorithm returns the index **i**, corresponding to the power safe area where the system will send the user.
Otherwise the index **i** is saved in the list **q** and the iteration restarts.

This algorithm provides a way to guarantee a uniform distribution of cars in the city and privileges the minimum possible distance of the selected power safe area from the destination of the user, in order to avoid a situation in which the user is sent to a power safe area too distant from his destination.
Both from space and time complexity view this algorithm offers a linear performance, in the order of O(**n**).

3.1.1 Money Saving Option: example of execution

Array **s**:

3	2	4	5	1

Array **d**:

25	100	300	250	600

Array available_spots:

2	3	2	0	6

(notice that this array is not actually created by the algorithm, since these informations about the available free spots can be retrieved directly from the power safe areas entities in the system; it is represented for better clearness on how the algorithm works)
List **q**: <empty> ← step 1), 2), 3), 4)
'0' ← step 5), 6), 7)

- The algorithm updates **s** and **d** and initializes **d**;
- The algorithm evaluates the average **a**: given five power safe areas **n**=5 and a is evaluated as (3+2+4+5+1)/5 = 3.
- The algorithm checks if **q** contains all the **n** indices: since the algorithm just started, **d** is still empty and the execution proceeds;
- The minimum of **d** is search by considering only the indices that are not contained in **q**: a minimum with value 25 is found and its corresponding index is saved, thus setting the value of **i** to 0;
- The value saved in the cell at index **i** of the array **s** is checked: since 3 is not less than the average **a** (even though the associate power safe area in the system has still 2 spots available), **i** is saved in the list **q** and the algorithm restarts iterating from step 3;
- The minimum of **d** is search by considering only the indices that are not contained in **q**: a minimum with value 100 is found (since 25 is ignored) and its corresponding index is saved, thus setting the value of **i** to 1;

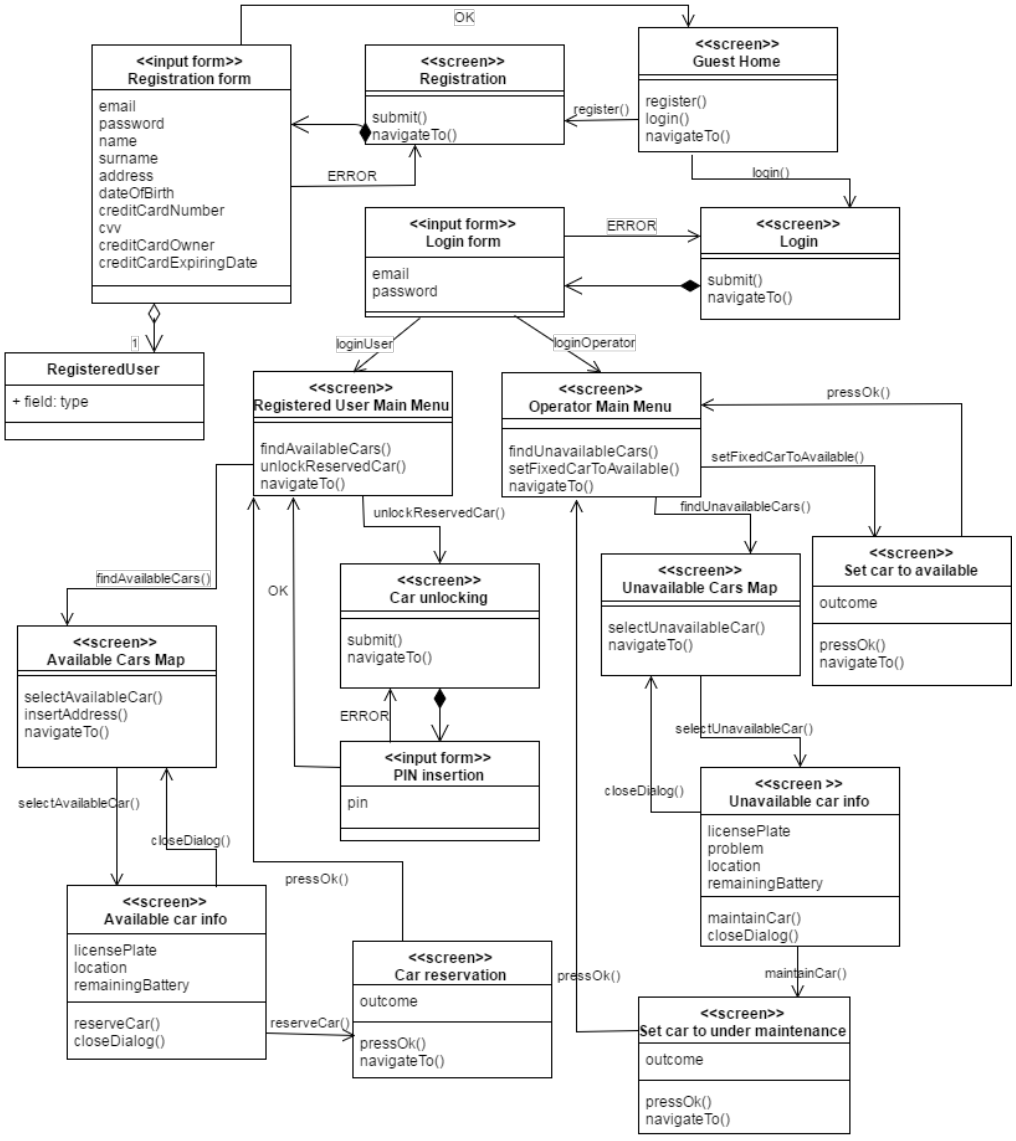
- The value saved in the cell at index **i** of the array s is checked: since 2 is less than the average **a** and since the associate power safe area in the system has still 3 spots available, the algorithm stops and returns the index **i**;

The system will then communicate to the user the location of the second power safe area saved in the system, since the algorithm has returned the second index of the arrays.

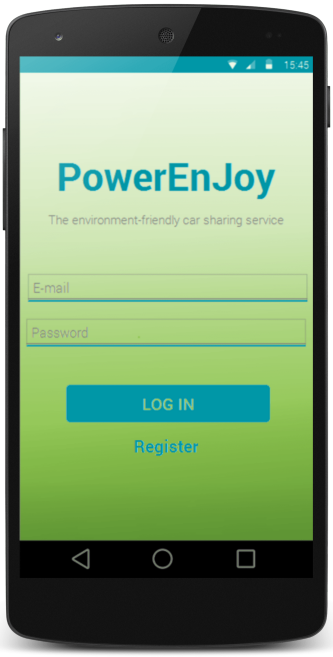
4 User Interface Design

4.1 UX Diagram

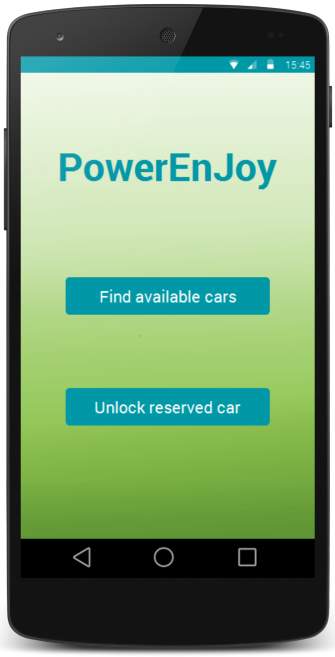
The UX Diagram shows what are the screens of the User and Operator mobile interfaces and in which way they interact with each other. Both registered users and operators interact with our system through the PowerEnJoy application only; in future releases, a web browser interaction may be added.



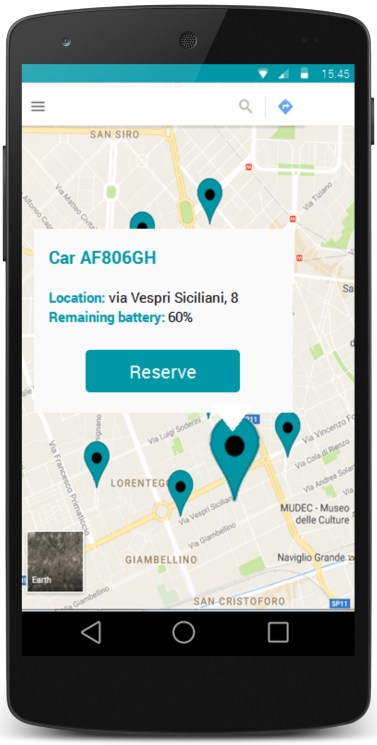
4.2 Mobile Application: User Interfaces



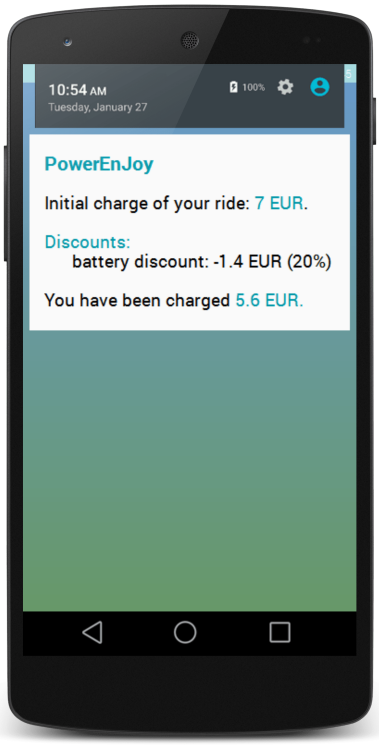
Login screen: initial screen of the mobile application for the registered user. It allows the user to log in to PowerEnJoy and to sign up if he is not registered yet.



Main user menu screen: the main menu of the mobile application for the registered user. It allows the user to go to the screen where he can find and reserve available cars, or to unlock a car that he previously reserved.

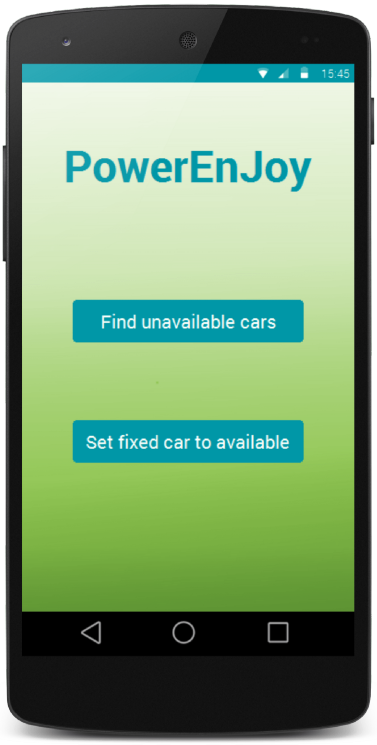


Find available cars screen: the screen through which the registered user can find available cars and reserve them. It features a map that uses Google Maps API to let the user search for an address or use his GPS position in order to center the map on that position and show all the available cars in the city. By clicking on a car, a pop up that shows all car's info and allows the user to reserve it opens.

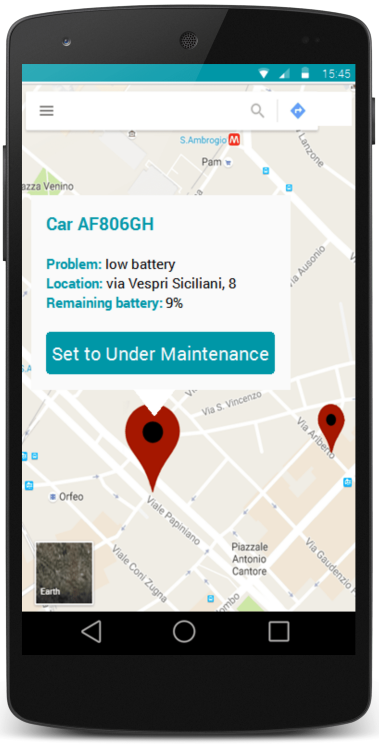


Notification email: at the end of the ride, the user receives a notification on his mobile device, where he is informed about the charge he will pay and the discount applied.

4.3 Mobile Application: Operator Interfaces

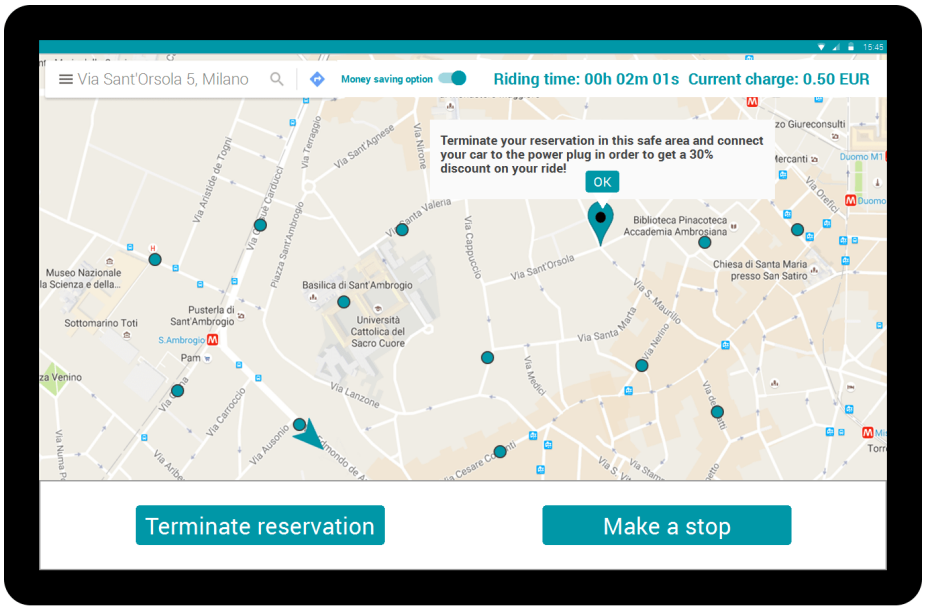


Main operator menu screen: the main menu of the mobile application for the operator. It allows the operator to go to the screen where he can find unavailable cars and set one of them as under maintenance, or to set back to available a car that he fixed.



Find unavailable cars screen: the screen through which the operator can find unavailable cars in the city area and set one of them as under maintenance. It features a map that uses Google Maps API to let the operator see all unavailable cars in the city. By clicking on a car, a pop up that shows all car's info and the problem that occurred opens.

4.4 Car Application: Screen Interface



Main car screen: The main screen in the car screen interface. It features a map that uses Google Maps API to show the car's current location and all the safe areas in the city; by clicking on one of them a pop up with all its basic information opens. The top bar contains an address bar through which the user can insert his destination; he can also enable the money saving options through a switch button, if he does so the suggested destination safe area is shown with a marker on the map. The top bar contains information about the riding time and the current charges as well. In the bar to the bottom of the screen there are two buttons to terminate the ride or make a stop.

5 Requirements Traceability

5.1 Functional Requirements

In the following table we provide the mapping between the components of the system and their related functional requirements identified in the RASD.

Component	Functional Requirements
AccountManager	R1.1.1 ; R1.1.2 ; R1.1.3 R1.1.4 ; R1.1.5 ; R1.1.6 ; R2.2.10
UserManager	R1.2.1 ; R1.2.2 ; R2.1.1
OperatorManager	R2.2.6 ; R2.2.7

ReservationManager	R1.3.1 ; R1.3.2 ; R1.3.3 R2.1.1 ; R2.1.2 ; R2.1.3 R2.1.6 ; R2.1.7 ; R2.1.8 R2.1.9 ; R2.1.10 ; R2.1.11 R2.1.12 ; R2.1.13 ; R2.1.14 R2.1.15 ; R2.1.16 ; R2.1.17 R2.2.2 ; R2.2.3 ; R2.2.4 R2.2.8 ; R2.2.9 ; R3.3 R4.1 ; R4.2.2
BillingManager	R1.1.6 ; R2.1.3 ; R2.1.4 R2.1.17 ; R3.1 ; R4.1 R4.1.1 ; R4.1.2 ; R4.2.1
CarManager	R1.2.1 ; R1.2.2 ; R1.2.3 R1.3.1 ; R1.3.3 ; R2.1 R2.2 ; R2.1.1 ; R2.1.2 R2.1.3 ; R2.1.5 ; R2.1.6 R2.1.7 ; R2.1.8 ; R2.1.10 R2.1.13 ; R2.1.14 ; R2.1.15 R2.1.16 ; R2.2.1 ; R2.2.2 R2.2.4 ; R2.2.5 ; R2.2.6 R2.2.7 ; R2.2.8 ; R2.2.9 R3.1 ; R3.2 ; R4.1.1 R4.1.2 ; R4.2.1
CarConnectionManager	R1.2.3 ; R2.1 ; R2.2 R2.1.6 ; R2.1.13 ; R2.2.1 R3.2

5.2 Non functional requirements

In this section we map each non functional requirement defined in the RASD with one or more design decisions taken in this document:

- [NF1] [NF3] [NF4]: see the CarConnectionManager component, the section 2.2.2 “Technological Viewpoint”, the section 2.8 “Other Design Decisions”.
- [NF2]: this non functional requirement can be achieved by using a proper configuration of the technology used by the Web Layer and by the Application Layer. See “System configuration” in the section 2.8 “Other Design Decisions”.
- [NF5] [NF6]: the scalability and modularity of the system provided by the distinction of 4 logical layers will help to satisfy these quality requirements, for example by using replication of components.
- [NF7]: see the section 2.2.2 “Technological Viewpoint”.
- [NF8]: see “BillingService: Stripe API” in the section 2.8 “Other Design Decisions”.

6 Software and tools used

- Git (https://github.com/) : for the version controlling of files shared between the team.
- Slack (https://slack.com/): used for group communication.
- GoogleDocs (https://www.google.it/intl/it/docs/about/): to write this document.
- Astah Professional (http://astah.net/editions/professional): to create all the UML diagrams.
- Alloy Analyzer (http://alloy.mit.edu/alloy/): to construct a model of part of our S2B and to prove its consistency.
- Lyx (http://www.lyx.org/): to format this document.
- JustInMind (https://www.justinmind.com/): to create the mockups.
- Android Device Art Generator (https://developer.android.com/distribute/tools/promote/device-art.html): to insert the mockups in an Android phone frame.

7 Effort spent

Total hours of work for the DD creation:

- Stefano Brandoli: 30 hours
- Silvia Calcaterra: 25 hours
- Samuele Conti: 25 hours