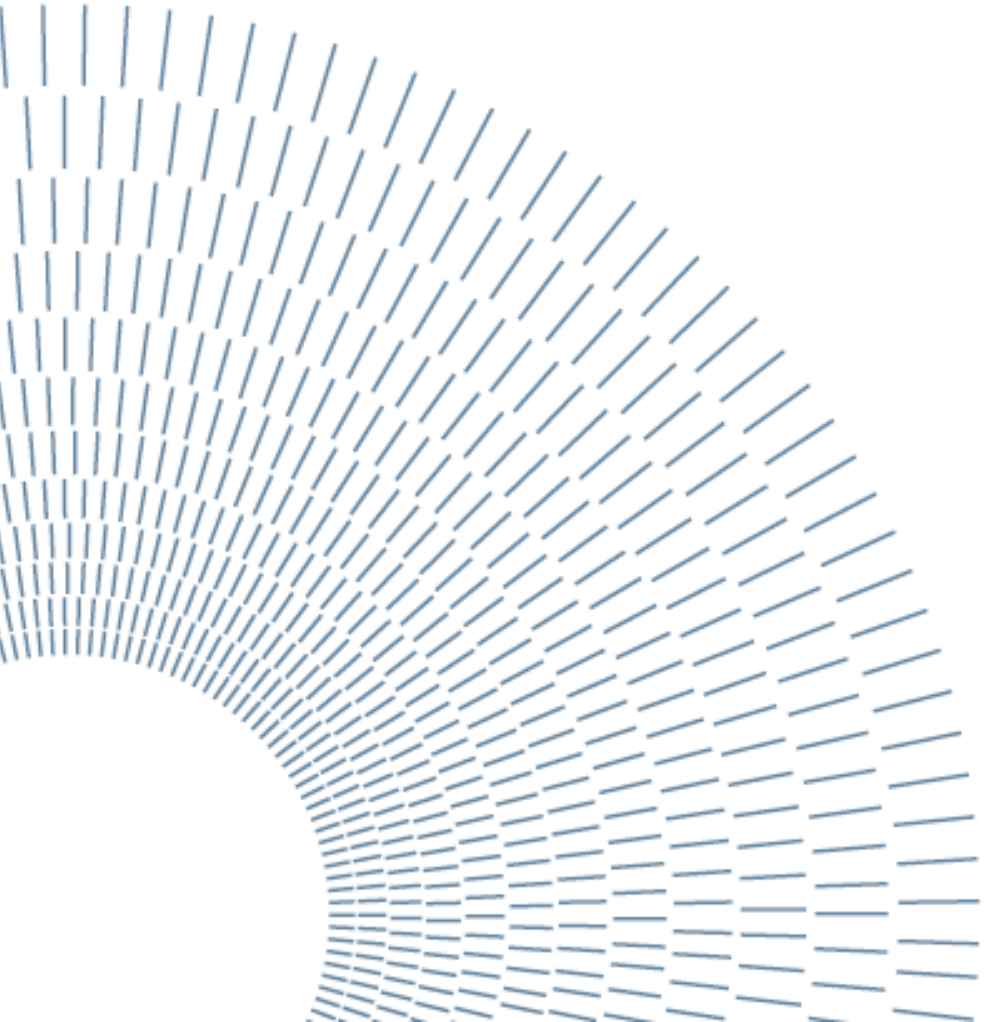
POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software Engineering 2 2016/2017
Project Power EnJoy

Design Document

V 1.1



Redacted by:

Leonardo Chiappalupi
CP 10453536

Ivan Bugli
CP 10453746

Table of contents

1. Introduction

1.1 Purpose

In the Design Document, several important aspects of the Power EnJoy service implementation will be analyzed, including a high-to-mid level architectural design and the correspondence of these design choices with the requirements analyzed in the previous paper, the Requirement Analysis and Specifications Document.

The aim of this elaborate is to show how the Power EnJoy platform will be translated from requirements to software modules, and how this process will keep important aspects of an internet based service at the center – mainly, the scalability and the flexibility for future upgrades, as well as the reliability in case of errors and the security of the different parts.

1.2 Scope

The Power EnJoy system architecture, as extensively examined in the RASD, will have to deal with mainly four kinds of users: unregistered ones that are willing to subscribe, registered users that can access all the main platform services, system managers in charge of the maintenance and the updates to the platform data, and operators which intervene on-site to provide assistance and move cars.

The scope of the Design Document is the set of software modules and architectural layers that interact with these four users, and provide the functionalities they need. At this point, a first very high-level subdivision of components domains may help, in order to provide a basic overview of the structure of the system to be:

- **Users management:** registration, login, sign out, etc.
- **Power EnJoy Car-sharing service:** cars reservation, unlock, ride, parking and lock; sharing a ride across different users, bonuses, surcharges, charges and payments; assistance requests.
- **System management:** parking areas management, assistance requests early feedback, fleet management, requests operators forwarding, etc.
- **Operators helper service:** out-of-order or low-battery cars operations management, on-site assistance management.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

- **API Server:** the whole system that erogates the services required by the various clients.
- **Back-end:** see *API Server.*
- **Client:** any device/user that accesses the service.
- **Front-end:** the logic layer on the clients side. Relies on the *Back-end* implementation.
- **Platform/Service/etc.:** Power EnJoy systems as a whole.
- **Car applet:** the software running on the car.

Other definitions and abbreviations carry on from the previous document. See section 1.4: *Glossary* and its subsections for further details.

1.3.2 Acronyms

- **JEE:** Java Enterprise Edition
- **RASD:** Requirement Analysis and Specification Document
- **DD:** Design Document
- **DB:** Database
- **DBMS:** Database Management System
- **UI:** User Interface
- **ER:** Entity Relationship
- **UX:** User Experience
- **AWS:** Amazon Web Services
- **API:** Application Programming Interface
- **JDBC:** Java DataBase Connectivity
- **JSON:** JavaScript Object Notation
- **WP:** Windows Phone
- **HTTP:** HyperText Transfer Protocol
- **TLS:** Transport Layer Security
- **REST:** REpresentational State Transfer

- **MVC:** Model-View-Controller
- **EC2:** (Amazon) Elastic Compute Cloud
- **GSM:** Global System for Mobile communications
- **GPS:** Global Positioning System

1.4 Reference documents

The main reference document of the present DD will be the Requirement Analysis and Specification Document, provided and redacted as a prerequisite of the present paper and so, the main source of assumptions for this DD.

1.5 Document structure

After this quick **Introduction**, the document is going to focus on the actual design part: in a first section, **Architectural Design**, the system and the proposed architecture will be shown from different perspectives, such as Components, Interfaces, Runtime layout etc., together with a breeder analysis on other design decisions.

Algorithm Design will consequentially follow, exploiting the operate of the main algorithm that allow Power EnJoy to work.

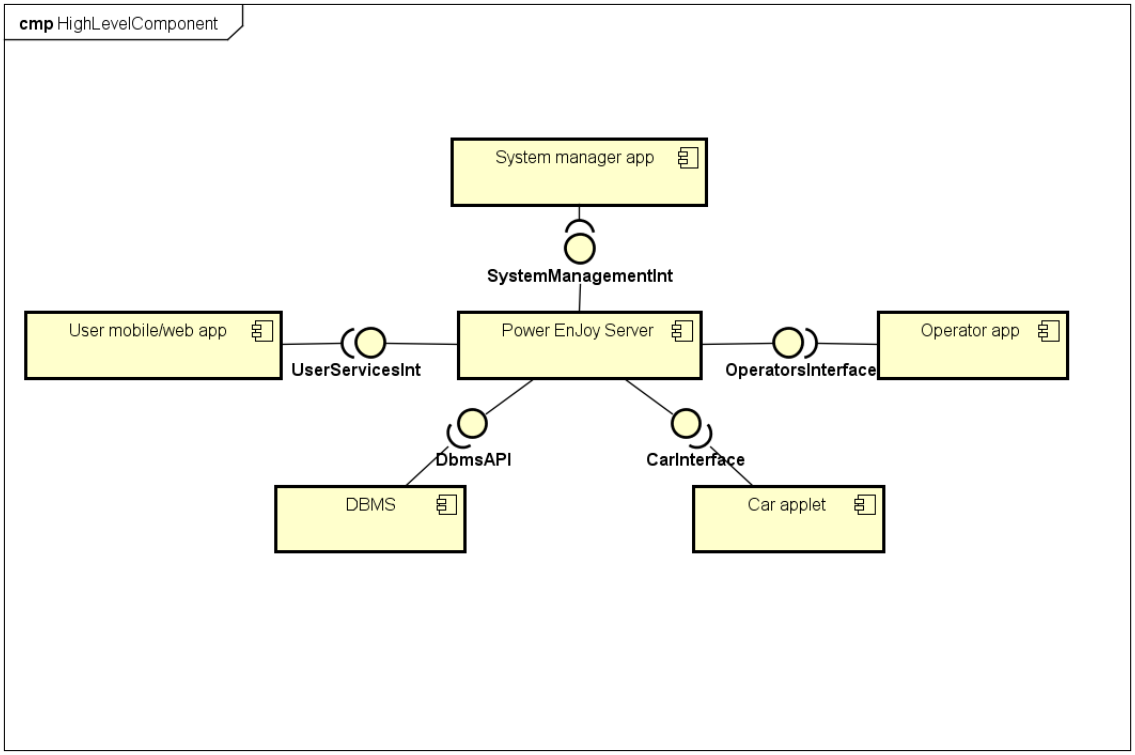
To refine the work already outlined in the RASD then, the **User Interface Design** section will introduce some details upon the UI specifications.

Finally, an extensive analysis on how all the previously analyzed requirements are covered by the proposed design will be supplied, in the **Requirements Traceability** part.

2. Architectural Design

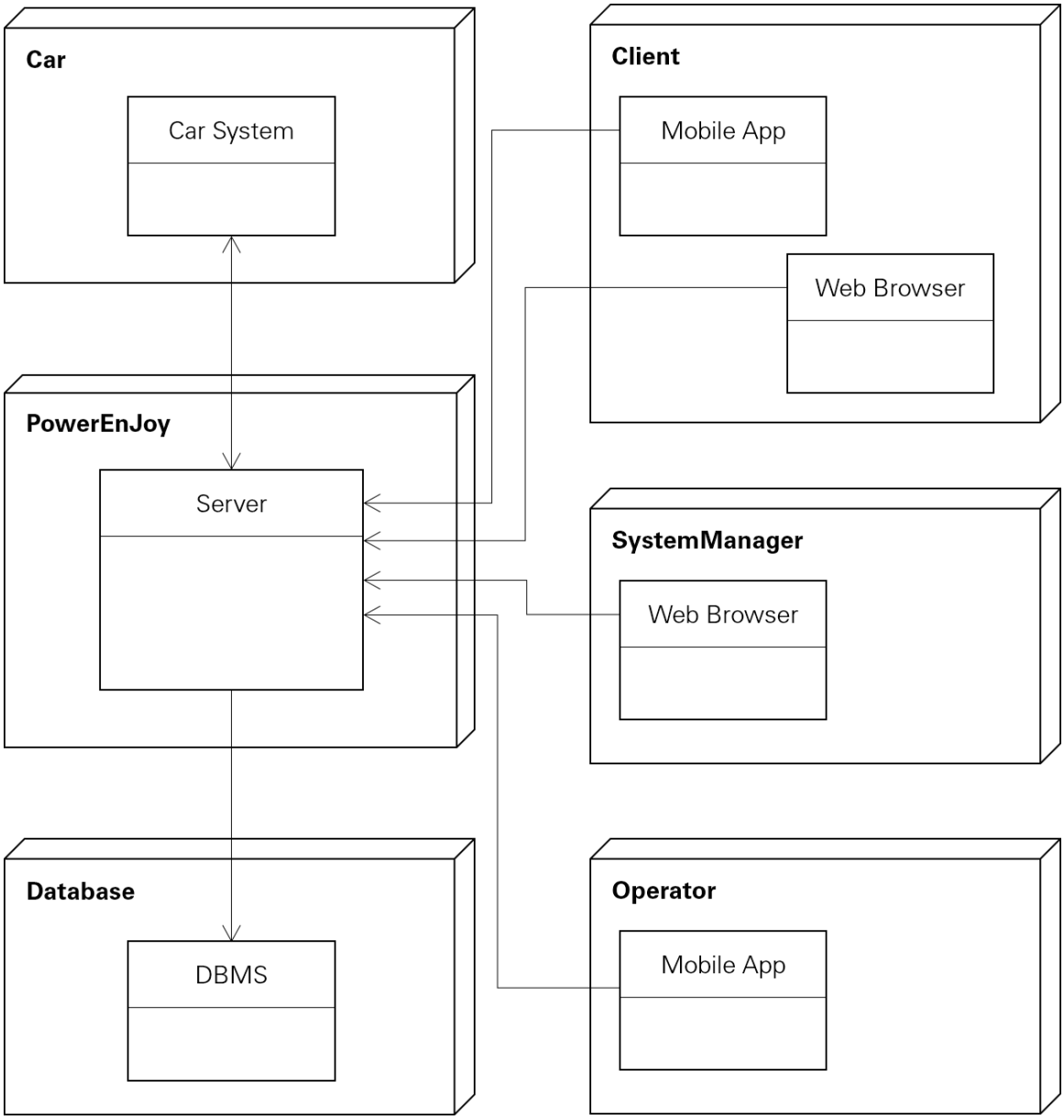
2.1 Overview

Starting by the basic "blocks" of the system defined in the *Scope* section, it is now possible to introduce a first High Level design, that can be useful to understand the "core" components that will unfold during the analysis. First, we shall start by proposing a rough high-level component view:



The core component of the system to develop is the Power EnJoy main server: note that at this stage, we call 'server' whatever software and hardware layer we will need to build our platform. As we can see from the picture, this main central block is responsible of managing all the requests from the different *end points*, as well as using and storing persistence data through the main database.

The following diagram, instead, shows how those components map on more concrete hardware counterparts. This is not a formal representation, but we think it can be convenient, at this stage, too.



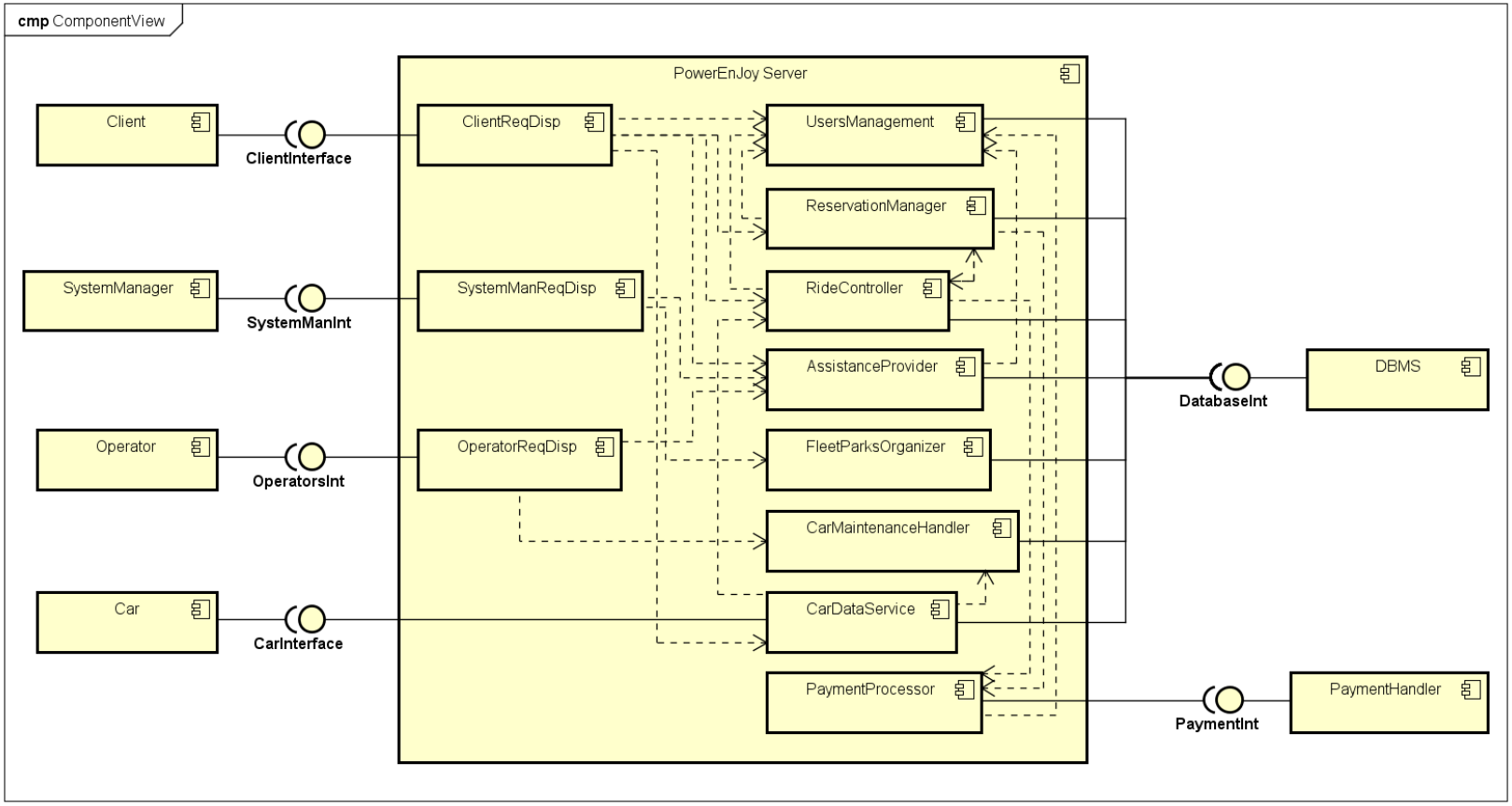
Please also refer to the picture shown in section 3.2.3: *Architectural outline* from the Requirement Analysis and Specification Document, as well as to section 1.6: *Proposed System*, to better understand the starting points that guided the transition from the RASD to the present Design Document.

2.2 Component View

2.2.1 System components

Now that a first overall design has been set, it is possible to dive into a more detailed and possibly complete component design.

The following diagram represents the proposed structure for the platform, from a software point of view. How the following components will map onto hardware tiers will be discussed in the *Deployment View*.



Except for cars, all the other external components that interface with the system (the *Client*, i.e. the users, the *System Manager(s)* and the *Operator(s)*) will communicate with three dedicated components (*ClientReqDisp, SystemManReqDisp, OperatorsReqDisp*), that route the different requests to the backend components responsible of the actual operations. These three *Request Dispatchers* process and forward requests and responses to and from the following nested components:

- **UsersManagement:** implements all the necessary method for users registration and login, authorizations and personal information management. Allows to disable (insolvent) users, or to re-enable them, and to retrieve additional user information for assistance.
- **ReservationManager:** handles reservation of cars, 1-hour timeouts, fees, etc. Interacts with *RideController* to assist the transition between reservations and rides. It's also used to register shared rides and to provide car and parkings positions map to users.
- **RideController:** manages the entire journey session from start to end. After the transition from *ReservationManager* during car unlock, provides real-time data to the car and computes ride time and charges. Processes extras and car locking, ride termination, etc. Issues payments to the PaymentProcessor.
- **AssistanceProvider:** processess assistance requests from users, provides contact and forwarding features to System Managers, exposes relevant data to Operators and handles their operations.
- **FleetParkOrganizer:** used to manage the platform resources, cars and parking areas. Offers functionalities that enable System Managers to manage the service fleet and add/edit/remove parking areas.
- **CarMaintenanceHandler:** exclusively implemented to manage low-battery and out-of-order cars. Allow Operators to list these cars and take charge of them. Permits recharged cars to move from disabled to enabled for reservations.
- **PaymentProcessor:** bridge to abstract from the actual payment handler, that may vary. Allows to make transactions for a ride, to apply fees, and is in charge of disabling users if payments fail.

Finally, the **CarDataService** component manages the communication with the car, allowing it to report its status and position, asking it to lock or unlock, and forwarding data from *RideController* to show price and time on the on-board display.

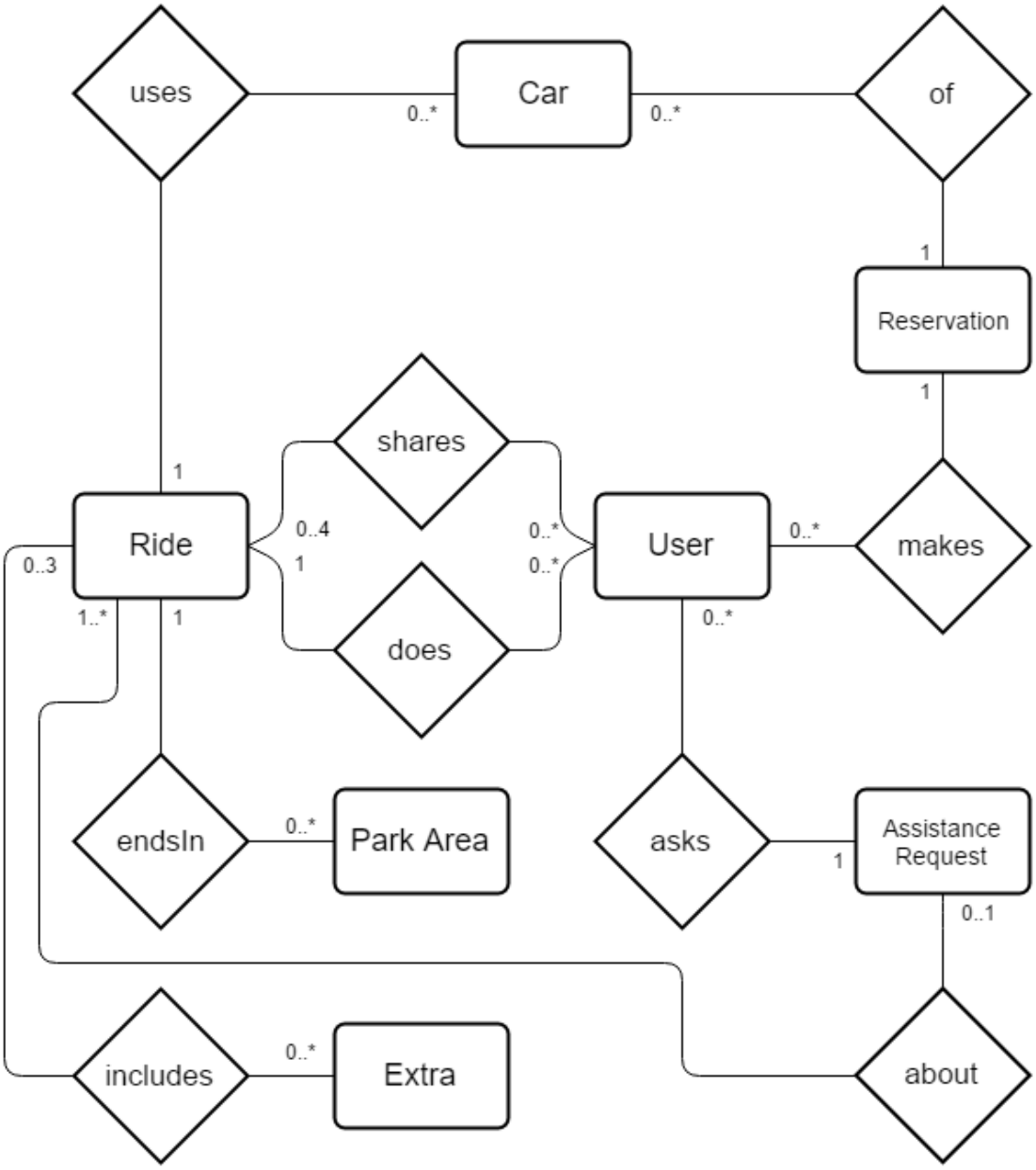
For more information about the different tasks of the component and their interfaces, see Section 2.5: *Component Interfaces*.

2.2.2 Persistance design

The system database must be created ex-novo, so it's design in terms of entities, relations and tables is a part of the design process.

The following ER (Entity Relationship) diagram shows the proposed structure for the data warehouse. Attributes are not explicitly reported for the sake of clearness and simplicity: they can be easily extracted by the logic design that comes after the diagram.

ER Diagram:



Logic design:

User (id, username, password, name, lastName, license, paymentToken, enabled)

Ride (id, user, date, timeStart, timeEnd, car, price, parkAreaEnd)

Reservation (id, user, date, time, car)

Car (name, plate, status, batteryLevel)

ParkArea (name, bounds, type)

Extra (name, type, percentage, policies)

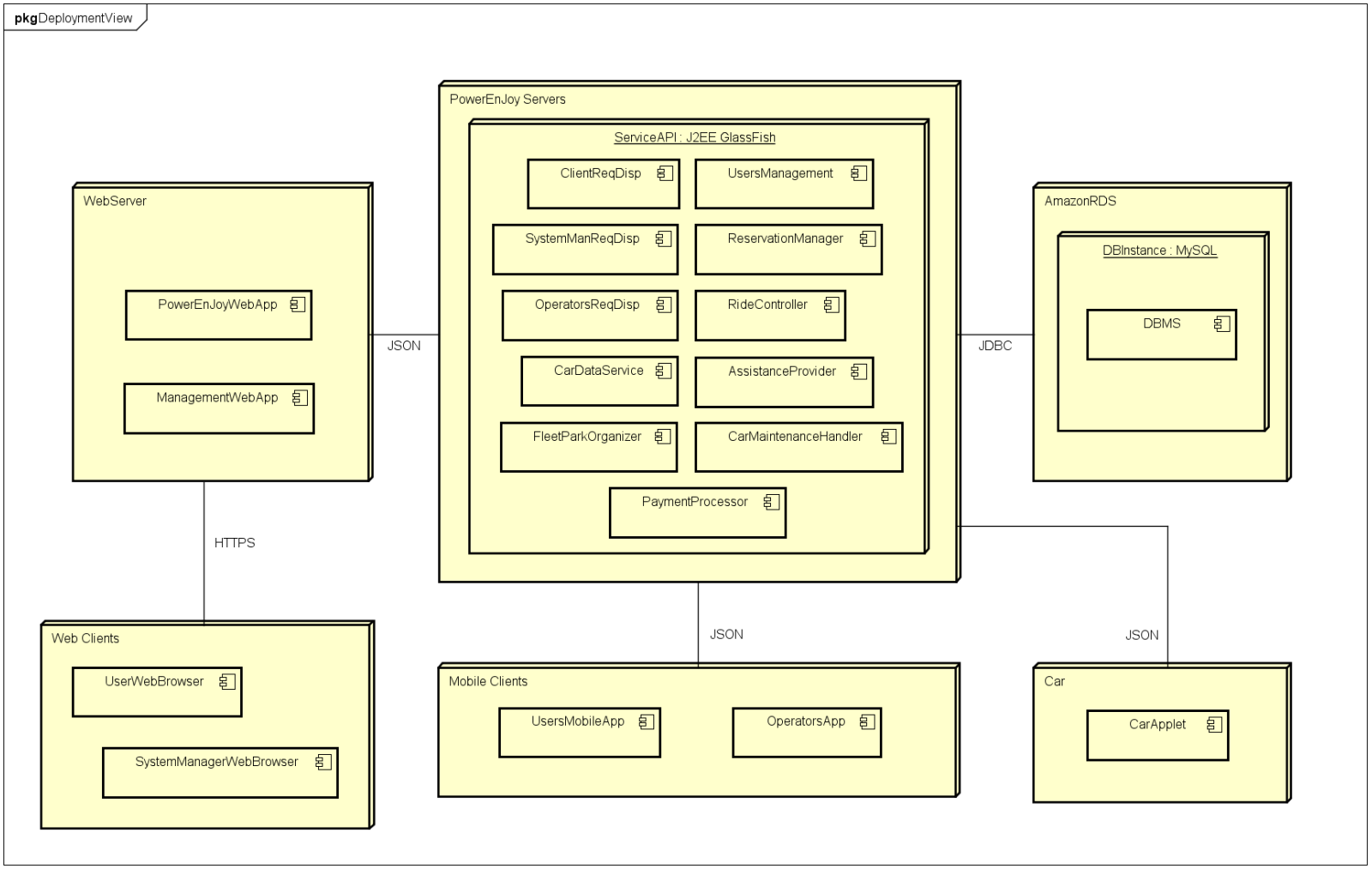
AssistanceRequest (id, user, date, time, relatedRide)

RidesExtras (id, rideId, extraName)

RidesSharedUsers (id, rideId, sharedUser)

2.3 Deployment View

The analyzed software platform will be deployed on the different systems by following this architecture scheme:



PowerEnjoy Servers will host the main API infrastructure, whose instances will contain all the components analyzed in the previous sub-section. This node will interface with the database hosting service, in the proposed design AmazonRDS. This way, the most popular database engine available, MySQL, will be available directly through JDBC, implemented inside our Java backend.

The API service will mainly use the JSON protocol to send and accept requests: hence, JSON will be used for communicating both with the WebServer and with the mobile clients.

The former will run two web applications: the **PowerEnJoyWebApp**, deployed to provide end users services through their browsers, and the **ManagementWebApp**, that will assolve the same function but for System Managers.

The latters (mobile clients) also include two different kinds of modules, here represented as components: **UsersMobileApp**, modeling the entire variety of mobile OS applications (iOS, Android, WP), and **OperatorsApp**, that models the application that operators will use to access their services. The **CarApplet** that runs on cars communicates with the API server the same way.

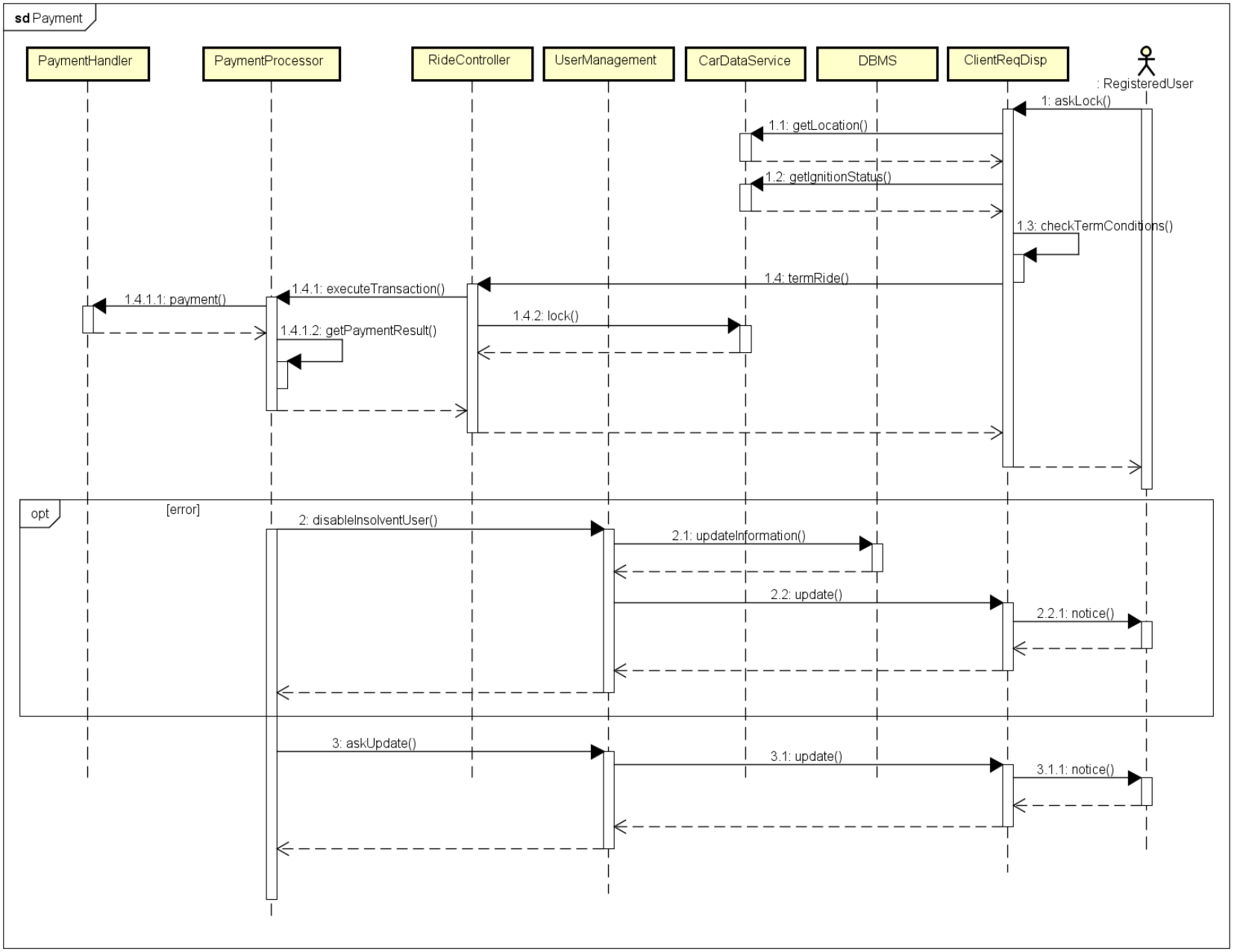
2.4 Runtime View

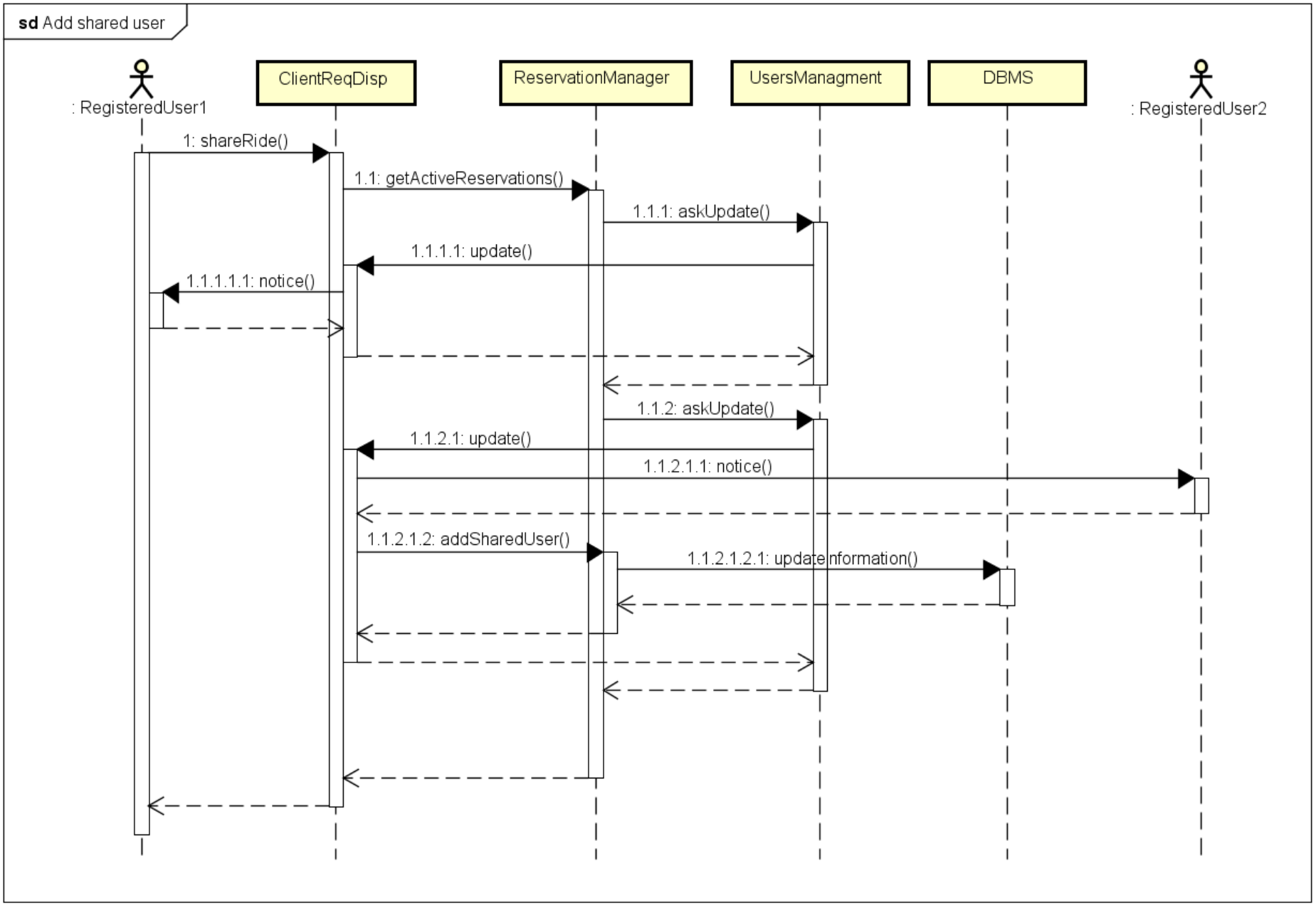
In section 6.1.2 from the *Requirement Analysis and Specification Document* (UML Models > Use Cases > Use Cases Descriptions) we already provided a wide variety of Sequence Diagrams that summarize how the different actions will be preformed.

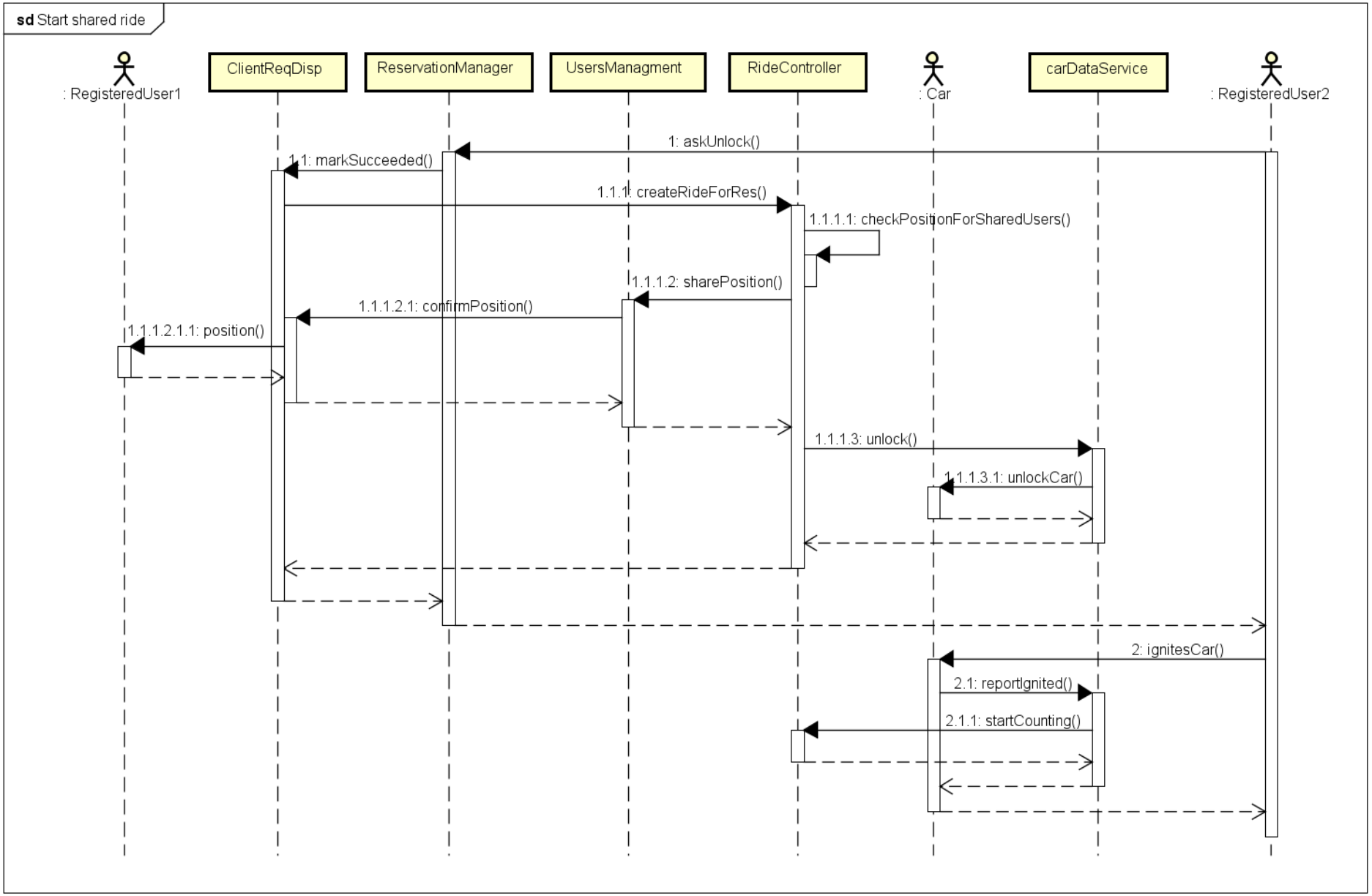
Nonetheless, since in this document we are introducing a more detailed structure of the components used in the system, we will provide a revised version of some of those diagrams featuring the components introduced previously.

Here we wanted to focus our attention on the main flow of events representing the interactions between the previously defined components, hence alternative flows and exceptions will be quite completely ignored in the following graphs if not absolutely necessary. They are however specified, as said, in section 6.1.2 of the RASD document.

Moreover it's important to notice that all the methods used in the diagrams, apart the ones from actors to intefaces and from interfaces to external handlers or to the DBMS, are interface methods which are defined in detail in section 2.5 of this document.


sd Payment
PaymentHandler
PaymentProcessor
RideController
UserManagement
CarDataService
DBMS
ClientReqDisp
: RegisteredUser
1: askLock()
1.1: getLocation()
1.2: getIgnitionStatus()
1.3: checkTermConditions()
1.4: termRide()
1.4.1: executeTransaction()
1.4.1.1: payment()
1.4.1.2: getPaymentResult()
1.4.2: lock()
opt
[error]
2: disableInsolventUser()
2.1: updateInformation()
2.2: update()
2.2.1: notice()
3: askUpdate()
3.1: update()
3.1.1: notice()


sd Add shared user
: RegisteredUser1
ClientReqDisp
ReservationManager
UsersManagment
DBMS
: RegisteredUser2
1: shareRide()
1.1: getActiveReservations()
1.1.1: askUpdate()
1.1.1.1: update()
1.1.1.1.1: notice()
1.1.2: askUpdate()
1.1.2.1: update()
1.1.2.1.1: notice()
1.1.2.1.2: addSharedUser()
1.1.2.1.2.1: updateInformation()


sd Start shared ride
: RegisteredUser1
ClientReqDisp
ReservationManager
UsersManagment
RideController
: Car
carDataService
: RegisteredUser2
1: askUnlock()
1.1: markSucceeded()
1.1.1: createRideForRes()
1.1.1.1: checkPositionForSharedUsers()
1.1.1.2: sharePosition()
1.1.1.2.1: confirmPosition()
1.1.1.2.1.1: position()
1.1.1.3: unlock()
1.1.1.3.1: unlockCar()
2: ignitesCar()
2.1: reportIgnited()
2.1.1: startCounting()

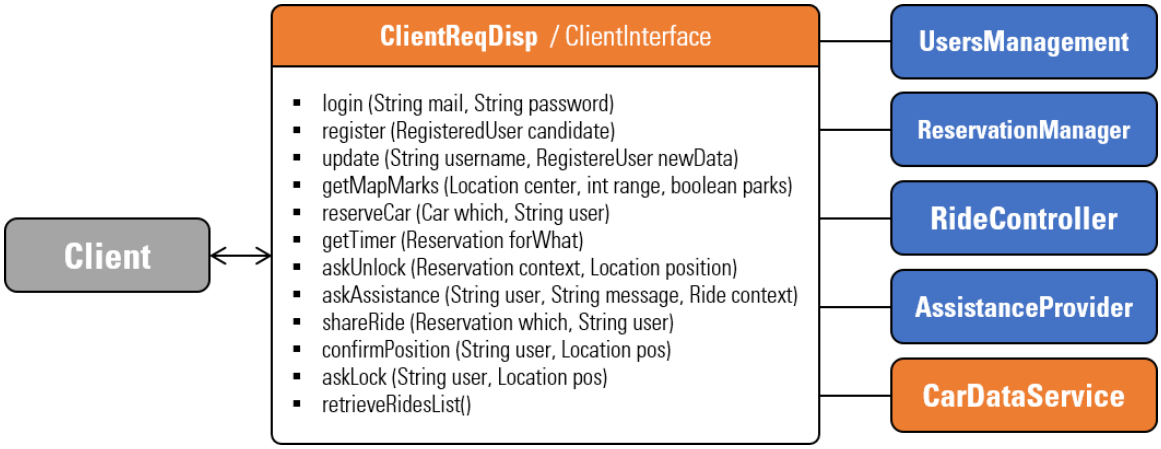
2.5 Component Interfaces

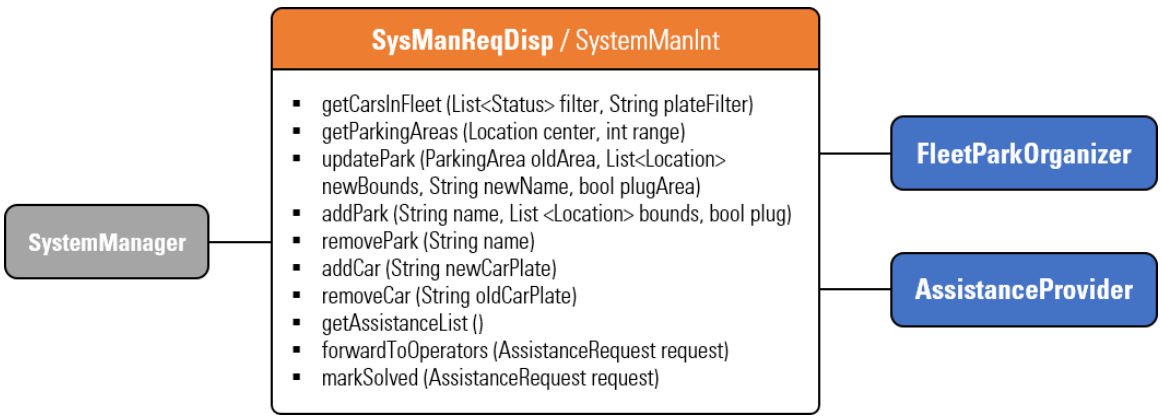
The interfaces that the various components share in this system are crucial to the actual running of the platform. In the following diagrams, a reasonably extensive list of the calls available between the components will be presented, in order to specify how the different modules interact in practice.

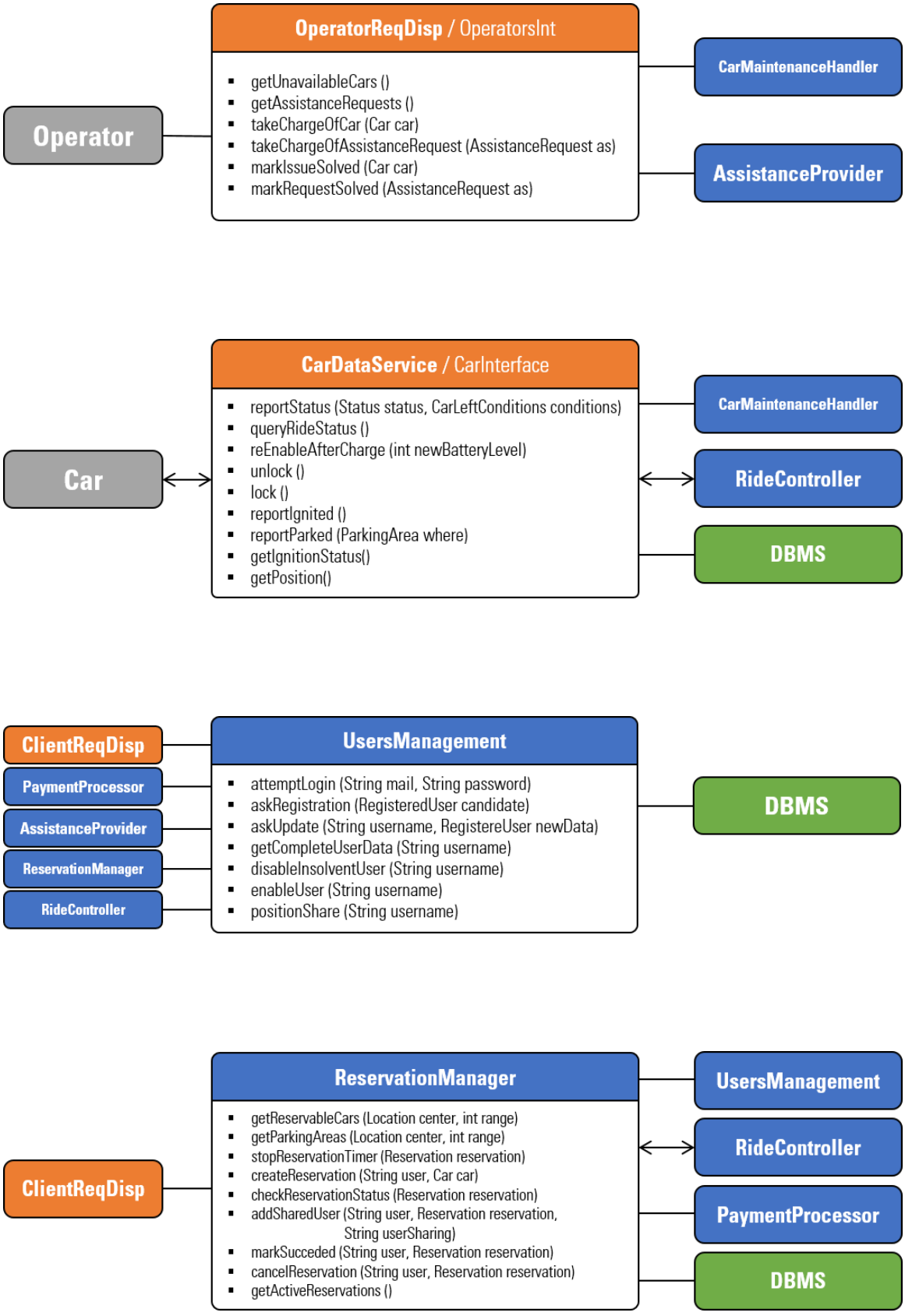
Session-related components with external interfaces will be marked in **ORANGE**;
internal (stateless) components with internal interfaces will be markes in **BLUE**;
external components that use the interfaces will be marked in **GREY**;
the DBMS component, modeling the database service, will be marked in **GREEN**.

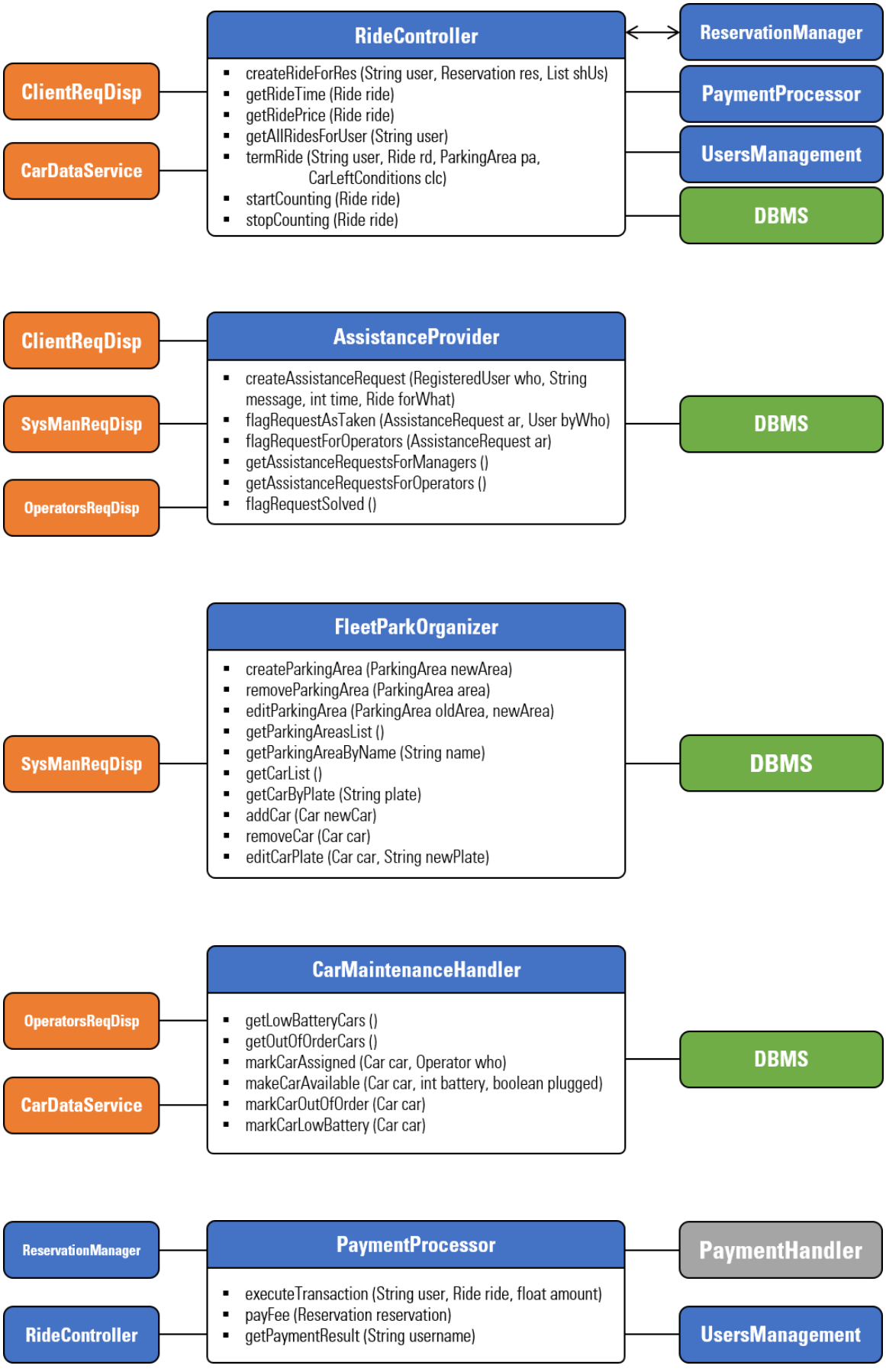
No arrows: the left component uses the right one or its interface
Both arrows: both components use each other/each other interface

For the classes used in these diagrams please refer to the *Requirement Analysis and Specification Document*, Section 6.2: *Class Diagram*.






OperatorReqDisp / OperatorsInt
getUnavailableCars ()
getAssistanceRequests ()
takeChargeOfCar (Car car)
takeChargeOfAssistanceRequest (AssistanceRequest as)
markIssueSolved (Car car)
markRequestSolved (AssistanceRequest as)
Operator
CarMaintenanceHandler
AssistanceProvider
CarDataService / CarInterface
reportStatus (Status status, CarLeftConditions conditions)
queryRideStatus ()
reEnableAfterCharge (int newBatteryLevel)
unlock ()
lock ()
reportIgnited ()
reportParked (ParkingArea where)
getIgnitionStatus()
getPosition()
Car
CarMaintenanceHandler
RideController
DBMS
ClientReqDisp
PaymentProcessor
AssistanceProvider
ReservationManager
RideController
UsersManagement
attemptLogin (String mail, String password)
askRegistration (RegisteredUser candidate)
askUpdate (String username, RegistereUser newData)
getCompleteUserData (String username)
disableInsolventUser (String username)
enableUser (String username)
positionShare (String username)
DBMS
ReservationManager
getReservableCars (Location center, int range)
getParkingAreas (Location center, int range)
stopReservationTimer (Reservation reservation)
createReservation (String user, Car car)
checkReservationStatus (Reservation reservation)
addSharedUser (String user, Reservation reservation, String userSharing)
markSucceded (String user, Reservation reservation)
cancelReservation (String user, Reservation reservation)
getActiveReservations ()
ClientReqDisp
UsersManagement
RideController
PaymentProcessor
DBMS


RideController
createRideForRes (String user, Reservation res, List shUs)
getRideTime (Ride ride)
getRidePrice (Ride ride)
getAllRidesForUser (String user)
termRide (String user, Ride rd, ParkingArea pa, CarLeftConditions clc)
startCounting (Ride ride)
stopCounting (Ride ride)
ClientReqDisp
CarDataService
ReservationManager
PaymentProcessor
UsersManagement
DBMS
AssistanceProvider
createAssistanceRequest (RegisteredUser who, String message, int time, Ride forWhat)
flagRequestAsTaken (AssistanceRequest ar, User byWho)
flagRequestForOperators (AssistanceRequest ar)
getAssistanceRequestsForManagers ()
getAssistanceRequestsForOperators ()
flagRequestSolved ()
ClientReqDisp
SysManReqDisp
OperatorsReqDisp
DBMS
FleetParkOrganizer
createParkingArea (ParkingArea newArea)
removeParkingArea (ParkingArea area)
editParkingArea (ParkingArea oldArea, newArea)
getParkingAreasList ()
getParkingAreaByName (String name)
getCarList ()
getCarByPlate (String plate)
addCar (Car newCar)
removeCar (Car car)
editCarPlate (Car car, String newPlate)
SysManReqDisp
DBMS
CarMaintenanceHandler
getLowBatteryCars ()
getOutOfOrderCars ()
markCarAssigned (Car car, Operator who)
makeCarAvailable (Car car, int battery, boolean plugged)
markCarOutOfOrder (Car car)
markCarLowBattery (Car car)
OperatorsReqDisp
CarDataService
DBMS
PaymentProcessor
executeTransaction (String user, Ride ride, float amount)
payFee (Reservation reservation)
getPaymentResult (String username)
ReservationManager
RideController
PaymentHandler
UsersManagement

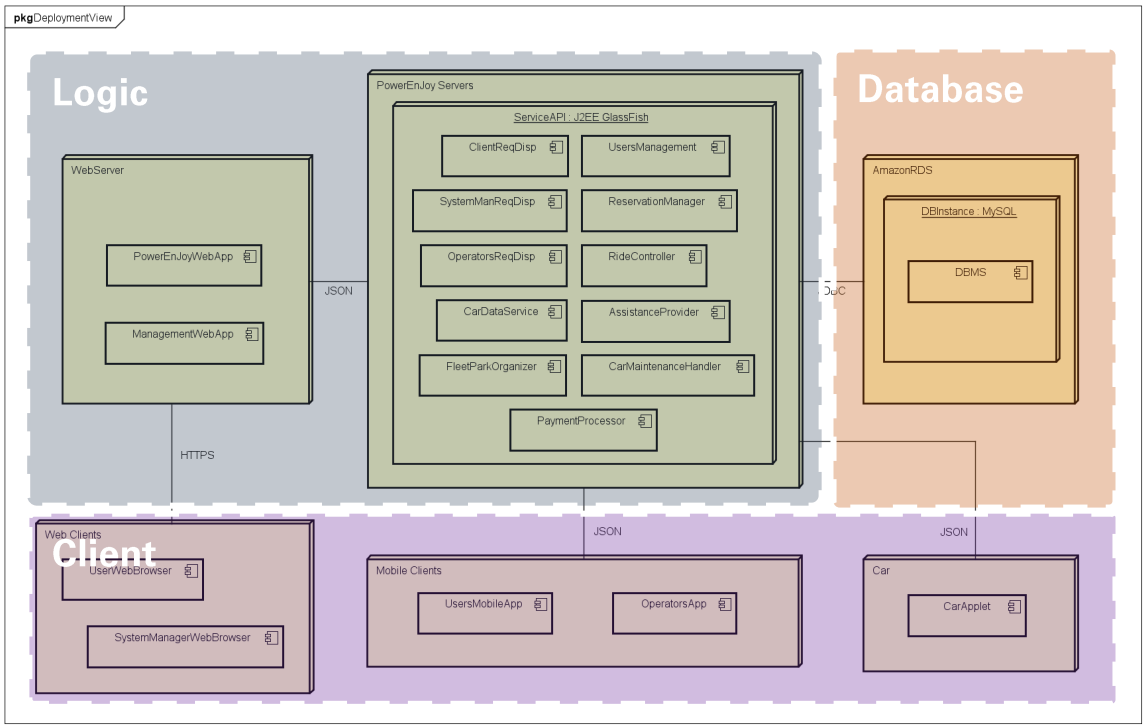
2.6 Selected architectural styles and patterns

2.6.1 Tiers

The Power EnJoy platform will be divided into 3 different tiers, each of which providing a different part in the overall running of the system:

- **Database:** provides database access and persistant data storage
- **Logic:** provides the back-end logic for all the operations. This includes both the required infrastructure for the server logic as well as the other additional layers (e.g. the web interface one) used to access the service
- **Client:** provides front-end user access to system features, indipendently from the user category (this also includes cars)

It is possible to map the tiers subdivision onto the previously examined deployment diagram:



2.6.2 Protocols

The system will use various protocols to make the different components communicate: an overview on the most important ones will now be presented.

JDBC: JAVA DATABASE CONNECTIVITY

Used to connect the logic (written in Java) with the database. Relies upon MySQL 3.x/4.0 in this very case. It is flexible enough to allow the connection to remote databases, as in our case the DB is stored using Amazon RDS (see following sections) and so it is only accessible using remote interfaces.

RESTFUL API (USING JSON)

Used by clients (browsers, apps, cars) to interact with the logic, using JSON as protocol to exchange the data. JSON is practical to use into mobile applications, as it is easy to read and write and very flexible. API calls are performed using authentication mechanisms (HTTPS, TLS, OAuth) and secured through custom certificates.

In the following list, some of the REST API calls are presented:

- pe/api/users
    - POST: Create user or authenticate
    - PATCH/PUT: Update user info
    - GET: Get users information
- pe/api/reservation
    - POST: Create new reservation
    - GET: Get current (sharable) reservations
- pe/api/ride
    - POST: Create ride from reservation
    - PATCH/PUT: Update ride status (unlock, lock, termination, etc.)
    - GET: Get users rides and information
- pe/api/assistance
    - POST: Create new assistance request
    - PATCH/PUT: Update assistance request status
    - GET: Get assistance requests information

- pe/api/fleet
    - POST: Add cars information
    - PATCH/PUT: Update car info
    - GET: Get cars information
- pe/api/parkings
    - POST: Add parking areas info
    - PATCH/PUT: Update parkings information
    - GET: Get parkings information
- pe/api/car
    - PATCH/PUT: Update car status and position
    - GET: Get ride information

JSON: JAVASCRIPT OBJECT NOTATION

Used as an alternative to XML as it is lighter, easier to read and transform into usable objects/data, since many flexible parsers are available for all platforms. To dive into an example of implementation, the following example is provided:

```
{
 “type” : “reservation”
 “user” : “johnive3”
 “car” : {
           “plate” : “DH667TV”
           “status” : “reserved”
           “battery” : 58
           “location” : {
                         “latitude” : 45.006
                         “longitude” : 9.524
                        }
           }
 “sharedUsers” : [
                   { “user” : “markTusk” }
                   { “user” : “victor88” }
                   { “user” : “taylor” }
                  ]
 “timeCreated” : 2016/05/11@12.55.31UTC+1
 “timeRemaining” : “38m54s”
}
```

2.6.3 Design patterns

CLIENT/SERVER

The entire platform is based upon the client/server model: all the logic and the data management is performed by the server, while clients only make requests and receive responses. This model is extremely practical in this case: allows us to make lighweight clients by moving all the logic to the server side, it is indipendent by the number of clients, allows an easy management of persistance data since it is shared among all clients, improves security by not allowing clients to manage the actual server data, and adds flexibility to upgrade or extend the platform in the future.

MVC

The entire system has been developed following the ModelViewController paradigm; hence, the presentation, the control and the model are indipendent and allow for concurrent operations. For instance, clients are the "View" of the system, while the "Controller" is the back-end logic and the "Model" is the database part. MVC will also be widely used inside some components: all clients will be developed using MVC paradigms (Android, iOS, WP applications, website), and also the car applet will be designed this way.

WRAPPER AND ADAPTERS

Since there will be the need of converting data between Java, JSON and MySQL, a number of wrappers and adapters will be used to make a seamless communication. This will allow to use the same model for every layer the application is using, accessing always the same granularity of data.

2.7 Other Design decisions

2.7.1 External services

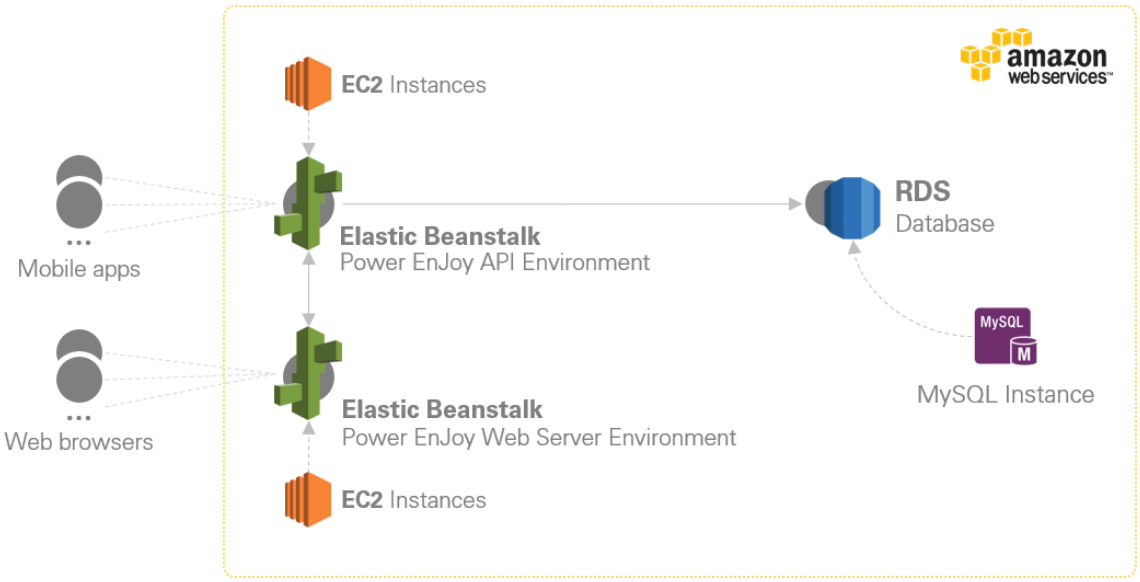
Power EnJoy will rely on some external services provided by third party companies to take advantage of their know-how to fullfill our necessities. Here are the most notable ones:

- **Google Maps:** used to provide users with map presentations, as well as to resolve coordinates to addressed and vice-versa. Google Maps also allows custom extensions to enrich maps with custom views: this will be the case with parking areas and cars finding.
- **PayPal:** the world most used online payment method will be used as the main way of performing transactions. On top of being secure, flexible and easy to implement, it also provides users a lot of different ways to pay (credit cards, bank accounts, PayPal credit, etc.).

2.7.2 Amazon AWS integration

This system can be hugely optimized by using Amazon Web Services to host and manage the entire platform. Amazon AWS uses a pay-as-you-go system so it is also convenient for the initial growth of the application.

The following proposed architecture includes all the features introduced so far, by deploying the different parts on different AWS services:



Elastic Beanstalk provides automatic load balancing by running a Java web application that scales on one or more **EC2** instances. This service will be used to host both the API server and the Web Server. **Amazon RDS** provides a fast, scalable, flexible relational database service, and Power EnJoy's database will be dropped onto a MySQL Instance running there. All these service run in protected and closed environments, so security is not an issue. The reliability and redundancy guaranteed by AWS specifications also ensure a very low fail rate and a robust system architecture. We designed the system so that using AWS is not mandatory, but recommended.

2.7.3 API server frameworks

On both the web server and the API server we are going to use J2EE with GlassFish, one of the most popular platforms to run this kind of applications. GlassFish ensures a complete documentation and support, as well as reliability and ease of implementation. Also, Amazon AWS Elastic Beanstalk fully supports GlassFish as of release 4.1.

J2EE on the other hand is the perfect choice for this kind of environment: it integrates flawlessly with plain Java, uses stateful *SessionBeans* to maintain a context for every sessions (thus, what we called *Session-releated components* in section 2.5 can be easily converted into stateful *SessionBeans*), as well as stateless and singleton components that can provide back-end features for all the sessions (think to the *Internal components,* always in section 2.5)

2.7.4 Car applet and communications

For the system as we described it so far, the cars will need to have a very simple applet that provides data reporting both to the logic back-end as well as to users through the screen located on the dashboard.

This applet will not be described into details, since it will be very simple and comparable to mobile applications that we will not describe too; however we think that some details should be defined before the actual realization, as they are crucial to the overall design.

We assume that cars are already equipped with some kind of interface that allows to easily read status values from the car (e.g. using OBDII). We leave the choice to the client so that they can find the most convenient and affordable solution. The only constraint is that the interface will have to, at least, provide a feature to unlock/lock the doors, one to signal ignition on/off, one to interface to the on-board

computer, one to use GPS and geolocalization, and one to use the cellular network.

The applet on the car will use GSM (or any network faster that that) to stream data to the servers, and vice versa. The screen controller in the car will provide a simple interface to show the various visuals: our applet will receive updates by the central systems and display them on that screen.

2.7.5 Mobile applications

It is our commitment to design a service that can run on the majority of mobile devices available; thus, we decided to provide an application for Android, iOS and Windows Phone devices, in this order of priority. The applications will be restricted to devices that provide geolocalization and internet connection over cellular network. Other detailed specifications about mobile applications can be found in the following list:

- **GOOGLE ANDROID**
  - Minimum OS version: 4.1 codename KitKat
  - Minimum screen resolution: 800x480 (pixels)
  - Sensors required: GPS, EDGE connection or better
  - Stores of distribution: Google Play Store
  - Allow for rooted devices: no
- **APPLE IOS**
  - Minimum OS version: iOS 7
  - Minimum screen resolution: none (under minimum OS)
  - Sensor required: GPS, EDGE connection or better
  - Store of distribution: Apple AppStore
  - Allow for jailbroken devices: no
- **MICROSOFT WINDOWS PHONE**
  - Minimum OS version: WP 8
  - Minimum screen resolution: 800x480 (pixels)
  - Sensors required: GPS, EDGE connection or better
  - Store of distribution: Microsoft Store
  - Allow for unlocked devices: yes

3. Algorithm Design

At the core of Power EnJoy features there are the abilities to reserve cars and use them in a ride. We therefore chose to showcase a possible implementation template for the components **ReservationManager** and **RideController**, by providing two code examples written in Java. Please note that these lines only focus on the most important methods among all those previously exposed in section 2.5: *Component Interfaces*, omitting the implementation of trivial ones (like getters, setters, etc.). The aim is to show the algorithm design for the phases of reserving a car and moving from the reservation to the ride, and then how the system processes extras and makes payments.

Even if we sticked to Java conventions, we are using **nothing more than plain Java** just to focus on the flow of events and their correlations. Please also note that we **purposely left the DB management out of the code**, by not showing how values are read at the beginning and written in the end: the focus here is not on our usage of JDBC and MySQL, but on how we use the data stored in the DB.

Aside from these caveats, the following implementation are coherent with the *Class Diagram* examined in the RASD, with the interfaces shown previously, and with the overall design of the system.

3.1 Reservation Manager

```
class ReservationManager {
    public static final int MAX_SHARED_USERS = 4;
    public static final int RESERVATION_TIMEOUT_MINUTES = 60;

    UsersManagement uMan;
    PaymentProcessor pProc;
    RideController rContr;
    Map<Reservation, Timer> feeTimers = new HashMap<Reservation, Timer>(); // Timers for reservation fees
    Map<Reservation, List<String>> sharedUsers = new HashMap<Reservation, List<String>>();
    // A simple map to link shared users with relative reservations

    public boolean createReservation (String user, Car car) {
        RegisteredUser userData = uMan.getCompleteUserData(user);
        if(userData.getActiveReservation() == null){ // If the user has not already an active res.
            Reservation reservation = new Reservation (car); // Create the reservation
            car.setStatus(Status.reserved); // Set the car status as "reserved"
            userData.setActiveReservation(reservation); // Set the created reservation as the user' active
            Timer newTimer = new Timer(true); // Create a timer for this reservation
            feeTimers.put(reservation, newTimer); // Put the timer in the map
            newTimer.scheduleAtFixedRate(new feeTimer(user, reservation), 0, 1000); // Schedule it
            return true;
        } else return false;
    }

    // Add a shared user to a reservation
    public void addSharedUser (String user, Reservation res, String sharingUser) {
        RegisteredUser userData = uMan.getCompleteUserData(user);
        if(userData.getActiveReservation().equals(res)){ // If the reservation is the user' active one
            List<String> currentSharedUsers = sharedUsers.get(res); // Get the shared users for that res.
            // If sh. users are not over the maximum and do not already feature the new user:
            if(!currentSharedUsers.contains(sharingUser) && currentSharedUsers.size() < MAX_SHARED_USERS)
                currentSharedUsers.addSharedUser(sharingUser); // Add the new shared user

        }
    }
```

```
private class feeTimer extends TimerTask {
    private int secondsRemaining = 60 * RESERVATION_TIMEOUT_MINUTES;
    private Reservation res;
    private String user;

    public feeTimer(String user, Reservation res){
        this.res = res;
        this.user = user;
    }

    @Override
    public void run(){
        secondsRemaining--; // Called every second
        if(secondsRemaining <= 0){ // If the timer reaches zero
            cancelReservation(user, res); // Cancel the reservation
            pProc.payFee(res); // Pay the fee
        }

    }
}
public void stopReservationTimer(Reservation res) {
    Timer t = feeTimers.get(res);
    if(t!=null){
        t.cancel;
        feeTimers.remove(res);
    }
}

public void cancelReservation(String user, Reservation res) {
    RegisteredUser userData = uMan.getCompleteUserData(user);
    if(userData.getActiveReservation().equals(res)){
        stopReservationTimer(res); // Stop the possibly active timer for this res.
        userData.setActiveReservation(null); // Remove the res. from the user' active
        res.getCar().setStatus(Status.reservable); // Reset the car status
    }
}

// Called to move from a reservation to the ride
public boolean markSucceded(String user, Reservation res) {
    stopReservationTimer(res); // Stop the relative timer
    RegisteredUser userData = uMan.getCompleteUserData(user);
    userData.setActiveReservation(null);
    userData.getSuccededReservations().add(res); // Move this res. to user' succeded ones
    // Delegate the control to RideController by issuing the creation of the new ride
    return rContr.createRideForRes(user, res, getSharedUsersCompleteList(sharedUsers.get(res)));

}

// Get a list of RegisteredUsers by a list of their usernames
private List<RegisteredUser> getSharedUsersCompleteList(List<String> usernames) {
    List<RegisteredUser> usersData = new ArrayList<RegisteredUsers>();
    for(String username : usernames)
        userData.add(uMan.getCompleteUserData(username));
    return usersData;
}
```

3.2 Ride Controller

```
class RideController {
    public static final float CHARGE_PER_MINUTE = 0.25f;
    public static final int SHARED_USERS_BONUS = 2;
    public static final int BATTERY_LEVEL_BONUS = 50;
    public static final int SURCHARGE_BATTERY_LEVEL = 20;
    public static final int MAX_DISTANCE_FOR_SHARING = 100; // In meters


    PaymentProcessor pProc;
    Map<Ride, Timer> timers = new HashMap<Ride, Timer>(); // A timer for every ride
    UsersManagement uMan;

    // This gets called from ReservationManager at markSucceded()
    public boolean createRideForRes (String user, Reservation res, List<RegisteredUser> sharedUsers) {
        // Check the position of every registered shared user and keep only those whose distance from
        // the car is less than MAX_DISTANCE_FOR_SHARING
        List<RegisteredUsers> actualSharedUsers = checkPositionForSharedUsers(sharedUsers, res.getCar());
        Ride activeRide = new Ride(res.getCar(), res, actualSharedUsers); // Create Ride object
        List<Ride> userRides = uMan.getCompleteUserData(user).getRides();
        userRides.add(activeRide); // Adds the new ride as last of user' rides
        CarDataService carInt = res.getCar().getCarDataService();

        if(carInt != null)
            boolean unlocked = carInt.unlock(); // Unlocks the car
            if (unlocked)
                return true;
            else
                return false;
        else return false;
        }

    private List<RegisteredUser> checkPositionForSharedUsers(List<RegisteredUsers> candidates, Car c) {
        List<RegisteredUser> finalSharingUsers = new ArrayList<RegisteredUser>();
        for(RegisteredUser u : candidates) { // For every user candidate for sharing
            Location actualLoc = uMan.positionShare(u.getUsername());
            // If the distance to the car is lower than MAX_DISTANCE_FOR_SHARING
            if(Location.distance(c.getLocation(), actualLoc) < MAX_DISTANCE_FOR_SHARING)
                finalSharingUsers.add(u); // Add them to the returned list
        }
        return finalSharingUsers;
    }

    public void startCounting (Ride ride) {
        if(!timers.containsKey(ride)) {
            Timer timer = new Timer(true);
            timer.scheduleAtFixedRate (new TimeUpdater(ride), 0, 60000); //Schedules time update every minute
            timers.put(ride, timer);
        } else {
            timers.get(ride).scheduleAtFixedRate (new TimeUpdater(ride), 0, 60000);
        }

    }
    private class TimeUpdater extends TimerTask {
        Ride ride;

        public TimeUpdater (Ride ride) {
            this.ride = ride;
            }

        @Override
        public void run(){
            ride.setTime(ride.getTime() + 1); //Updates the time count in the system
            ride.getCar().getCarDataService().updateTime(ride.getTime()); //Sends updated count to the car
        }
    }

    public void stopCounting (Ride ride) {
        if(timers.containsKey(ride)) {
            timers.get(ride).cancel();
            timers.remove(ride);
        }
    }
```

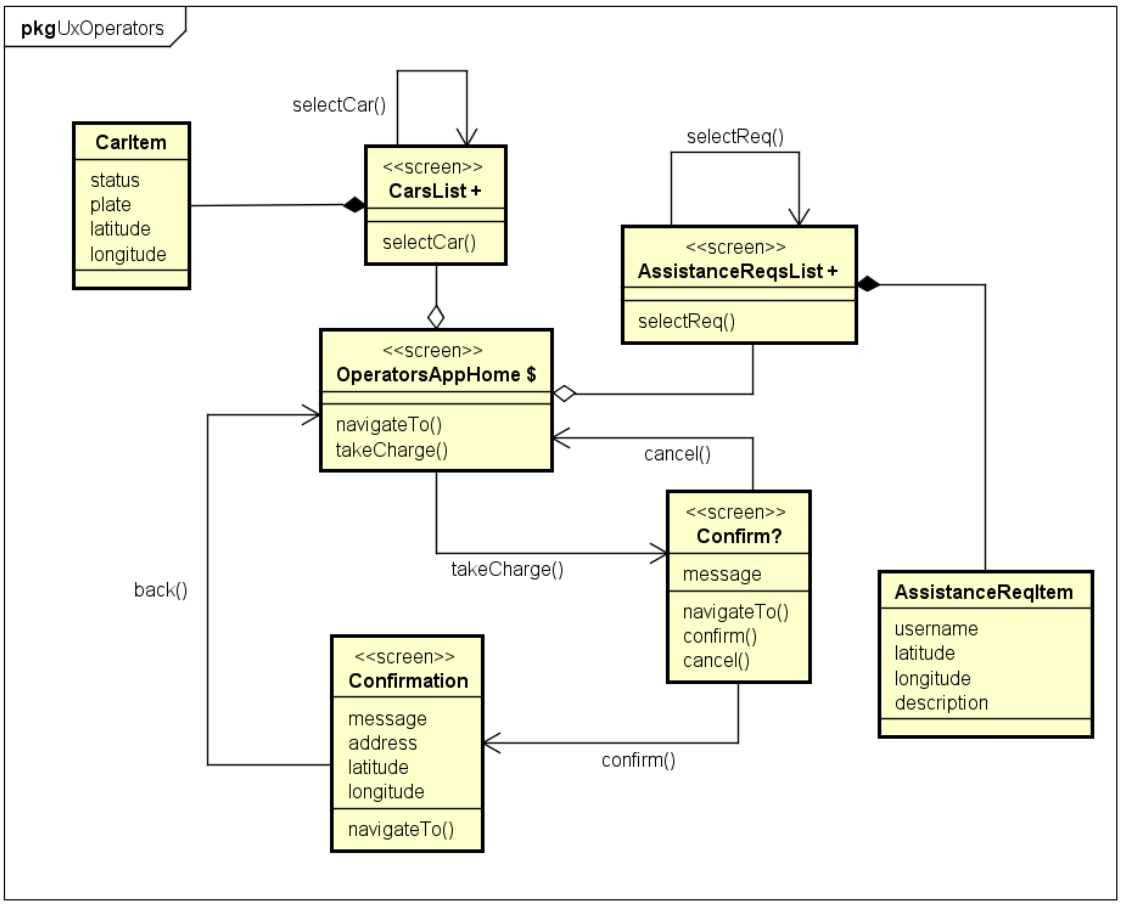
```
//Invocated by ClientReqDisp that must have already checked the termination conditions
public boolean termRide (String user, Ride rd, ParkingArea pa, CarLeftConditions clc) {
    RegisteredUser owner = uMan.getCompleteUserData(user);
    lastUserRide = owner.getRides().get(owner.getRides().size - 1);
    if (lastUserRide.equals(rd)) {
        lastUserRide.setPark(pa);
        lastUserRide.setCarLeftConditions(clc);
        computeTempPrice(lastUserRide); //Calculates price without extras
        processExtras(lastUserRide, pa, clc); //Adds extras
        //Delegate payment to PaymentProcessor
        bool paymentOk = pProc.executeTransaction(username, lastUserRide, lastUserRide.getTotalCost());
        lastUserRide.setPaymentOk(paymentOk);
        boolean locked = lastUserRide.getCar().getCarDataService().lock(); // Lock the car
        if(locked)
            return true;
        else return false;
    } else return false;
}
private void computeTempPrice(Ride ride) {
    float totalPrice = ride.getTime() * CHARGE_PER_MINUTE;
    ride.setTotalCost(totalPrice);
}

private void processExtras(Ride ride, ParkingArea pa, CarLeftConditions clc) {
    List<Extra> extras = new ArrayList<Extra>();

    //Extras added from the least to the most convenient
    checkSharingDiscountEligibility(extras, ride);
    checkBatteryLevelDiscountEligibility(extras, clc);
    checkPlugDiscount(extras, pa, clc);
    checkCarStatusPenalty(extras, clc);

    for(Extra e: extras)
        applyExtra(ride, e);

    ride.setExtras(extras);
}

private void checkSharingDiscountEligibility(List<Extra> extras, Ride ride) {
    if(ride.getSharedUsers().size() > SHARED_USERS_BONUS)
        extras.add(new SharingDiscount());
}

private void checkBatteryLevelDiscountEligibility(List<Extra> extras, CarLeftConditions clc) {
    if(clc.getBatteryLeft() > BATTERY_LEVEL_BONUS)
        extras.add(new BatteryLevelDiscount());
}

private void checkPlugDiscount(List<Extra> extras, ParkingArea pa, CarLeftConditions clc) {
    if(pa.isPlug() && clc.isPlugged() == true)
        extras.add(new PlugDiscount());
}

private void checkCarStatusPenalty(List<Extra> extras, CarLeftConditions clc) {
    if(clc.isDistantFromPlug() && clc.getBatteryLeft() < SURCHARGE_BATTERY_LEVEL)
        extras.add(new CarStatusPenalty());
}

private void applyExtra(Ride ride, Extra e) {
    float diff = ride.getTotalCost * e.getPercentage() / 100f;
    if(e.getExtraType() == surcharge)
        ride.setTotalCost(ride.getTotalCost() + diff);
    else if (e.getExtraType() == discount)
        ride.setTotalCost(ride.getTotalCost() - diff);
}
```

4. User Interface Design

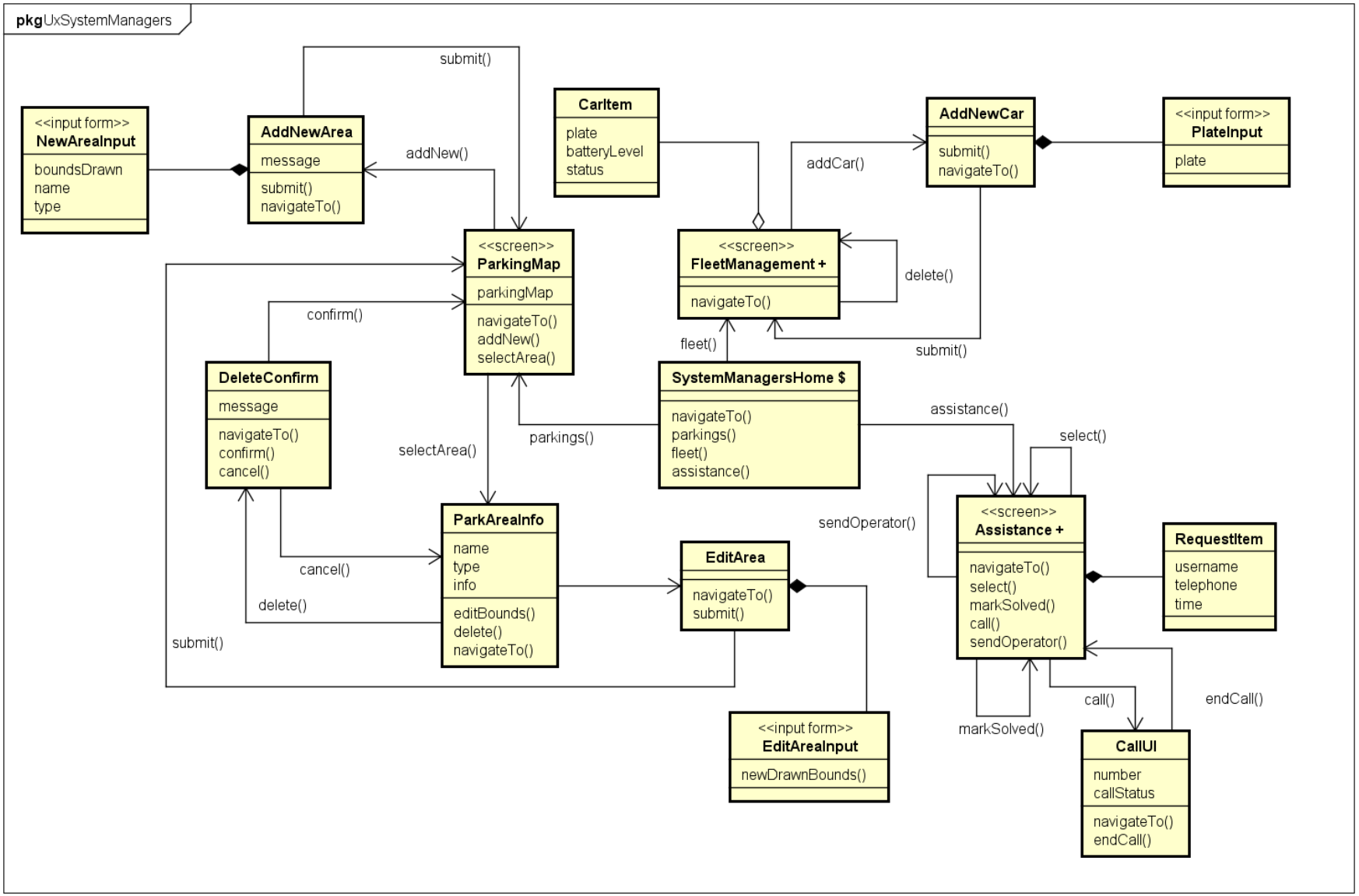
In section 3.2.1 of the *Requirement Analysis and Specification Document* (Requirements > Non-functional requirements > User Interface) we already provided plenty of user interfaces mockups that show the structure of the GUI for users, system managers and operators.

Nonetheless, we will now present the underlying structure of those interfaces, by providing UX diagrams for all the three scenarios. These diagrams also include details on screens and features that were not included in the mockups.

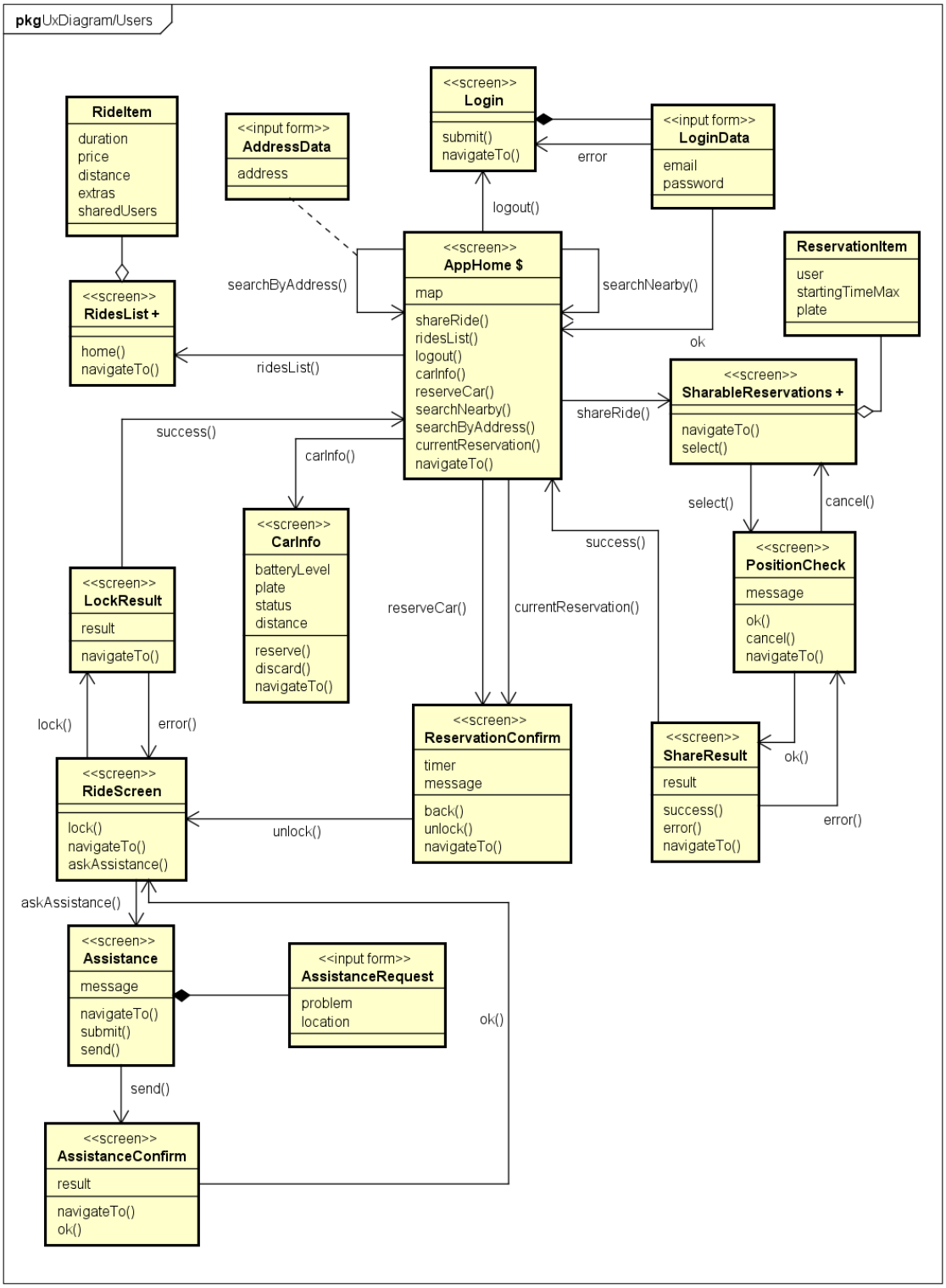
4.1 Operators application



4.2 System managers web application



4.3 Operators application



5. Requirements Traceability

The commitment of this document is to identify a suitable design that covers the totality of the goals – and relative requirements – examined in the previous documents. The following list shows which parts or components of this design will fullfill every goal.

CG1 Users Account Management

- ClientReqDisp
- UserManagement

CG2 Car Finding

- ClientReqDisp
- ReservationManager

CG3 Car Reservation

- ClientReqDisp
- ReservationManager

CG4 Car Reservation Reset & Fee

- ClientReqDisp
- ReservationManager
- PaymentProcessor

CG5 Car unlock/activation

- ClientReqDisp
- ReservationManager
- RideController
- CarDataService

CG6 Charges

- RideController
- PaymentProcessor

CG7 Charges termination and lock

- ClientReqDisp
- RideController

CG8 Safe Areas Management

- SysManReqDisp
- FleetParkOrganizer

CG9 Assistance and maintenance

- SysManReqDisp
- OperatorReqDisp
- AssistanceProvider
- FleetParkOrganizer
- CarMaintenanceHandler

AG1 Discount for ride sharing

- ClientReqDisp
- ReservationManager
- RideController

AG2 Discount for battery saving

- RideController
- CarDataService

AG3 Discount for plugging

- RideController
- CarDataService

AG4 Additional charge for compensation

- RideController
- CarDataService

6. Appendices

6.1 Tools used

- Microsoft Office Word 2016 *https://products.office.com/it-it/home*
  For redacting, reviewing, layout and graphic design of this document
- Astah Professional *http://astah.net/editions/professional*
  For component and deployment diagrams, UX diagrams
- Draw.io *http://draw.io*
  For creating the ER diagram
- Microsoft Office PowerPoint 2016 *https://products.office.com/it-it/home*
  For creating the high level structure diagram, Amazon AWS scheme, interfaces diagrams
- Notepad++ *https://notepad-plus-plus.org*
  For editing Java code for algorithms templates

6.2 Hours of work

The present document required almost the same time for both the curators (Leonardo Chiappalupi, Ivan Bugli).

The total time spent apiece on the creation of the paper is: ~35 Hours.

6.3 References

- **AMAZON WEB SERVICES:** http://aws.amazon.com
- **POWER ENJOY: REQUIREMENT ANALYSIS AND SPECIFICATION DOCUMENT – CHIAPPALUPI L., BUGLI I.** Politecnico di Milano, 2016
- **PAYPAL API:** https://developer.paypal.com
- **GOOGLE MAPS API:** https://developers.google.com/maps
- **ANDROID VERSIONS:** https://www.android.com/history
- **IOS VERSIONS:** https://developer.apple.com
- **WINDOWS PHONE VERSIONS:** https://msdn.microsoft.com/it-it/library/windows/apps/hh202996(v=vs.105).aspx
- **JSON:** www.json.org

6.4 Updates

6.4.1 Version 1.1

- Fixed sequence diagrams
- Fixed some inconsistencies in the interface section
- Fixed some inconsistencies in the component diagram
- Fixed some layouts issues