Design document

POLITECNICO
MILANO 1863

Figure 1: Logo Politecnico di Milano

PowerEnjoy

Figure 2: Logo PowerEnjoy

- Maria Chiara Zaccardi
- Nicola Sosio
- Riccardo Redaelli

15/01/2017

Contents

1

Introduction

1.1 Purpose

The purpose of this document is to give more technical details than the RASD about PowerEnjoy system.
This document is addressed to developers and aims to identify:

- The high level architecture
- The design patterns
- The main components and their interfaces provided
- The runtime behavior

1.2 Scope

The service PowerEnjoy is based both on mobile application and web application and has three different targets of people:

- Visitor
- User
- Operator

The system allows user to reserve a car via mobile or web app, using GPS or inserting an address. Furthermore the mobile app allows operators to know the operation that they have to do through a notification. As soon as operation has been done, the operator via mobile app could report the completed operation. Visitor has to sign up for the service and then login as user, while operators already have credentials assigned by PowerEnjoy. The system offers also a money saving option that users can enable inside the car and the mobile app provides information about the power grid station where to leave the car to get a discount.

1.3 Definitions, Acronyms, Abbreviations

- RASD: Requirements Analysis and Specification Document
- DD: Design Document
- UX: User experience design
- BCE: Boundary-Controller-Entity
- API: Application Programming Interface
- Client: application program that establish a connection in order to send 'request'
- Server: application program that accept connections in order to receive 'request' and to send specific 'response'
- TCP: Transmission Control Protocol is a standard that defines how to establish and maintain a network conversation via which application programs can exchange data
- IP: Internet Protocol is the method or protocol by which data is sent from one computer to another on the internet.
- GUI: Graphical User Interface is a user interface that includes graphical elements, such as windows, icons and buttons.

1.4 Reference Documents

- RASD produced before v1.1
- Specification Document: Assignments AA 2016-2017.pdf
- Example from last year: Sample Design Deliverable Discussed on Nov. 2.pdf
- Slides of lecture lesson from Beep:
  - Design Part I.pdf
  - Design Part II.pdf
  - Architecture and Design in Practice.pdf
  - Examples of architectures.pdf
  - Reasoning on design through an example.pdf
- "Relocation Strategies and Algorithms for Free-Floating Car Sharing Systems", Simone Weikl, Klaus Bogenberger , IEEE Intelligent Transportation Systems Magazine, Volume: 5, 2013

1.5 Document Structure

- **Introduction:** This section aim to explain the scope and the purpose of this document. Why it's important and which parts are covered that are not covered by RASD document.
- **Architecture Design**:
  - *Overview*: explanation of which architectural choice are been taken. How many tiers there are, and how they are divided.
  - *High level components and their interaction*: Global view of the application component and how they communicate.
  - *Component View*: This section enter in a highest level of deeper. Component of the application are more detail.
  - *Deploying view*: This section shows the components that must be deployed to have
  - *Runtime view*: This section shows sequence diagram of some of the most important activities that user can do.
  - *Component interfaces*: This section shows the interfaces between the components
  - *Selected architectural styles and patterns*: This section shows the architectural choices taken during the creation of the application
  - *Other design decision:* In this section we talk about other decisions taken.
- **Algorithm Design**: This section shows how are implemented some functionality that can be critical.
- **User Interface Design**: Ux and BCE diagrams
- **Requirements Traceability**: Describes how decision taken in the RASD are linked to design elements, and through which components are satisfied

2

Architectural design

2.1 Overview

PowerEnJoy has a three tier architecture. Three tiers are: client, server and database. To each tier corresponds a logical layer that offers functionality to the user.

The system provide two different user interfaces: one from the web app and one from the mobile application. Users can log theirself from both interfaces, while operators have their ad hoc application. Operators uses mobile phone which are provided from the company and they log in from the mobile phone.

In the server there is the application logic, because of that the client is a thin-client.

Figure 2.1: Tiers

2.2 High level components and their interaction

The high level component architecture is composed of five different components. The server component receives several types of request from:

- users
- visitors
- operators
- car systems

These components interact with the server through an interface for each different component.

Users and visitors communicate with the server by sending requests from his/her mobile application or web application. In addition users communicate through the CarSystem during the ride.

While the communication between server and operator start from the server that sends notification of operation through a PushGateway, and operators reply with an operation report from their mobile application as soon as the operation ends.

Furthermore the server communicates with the DataBase to extract data needed.

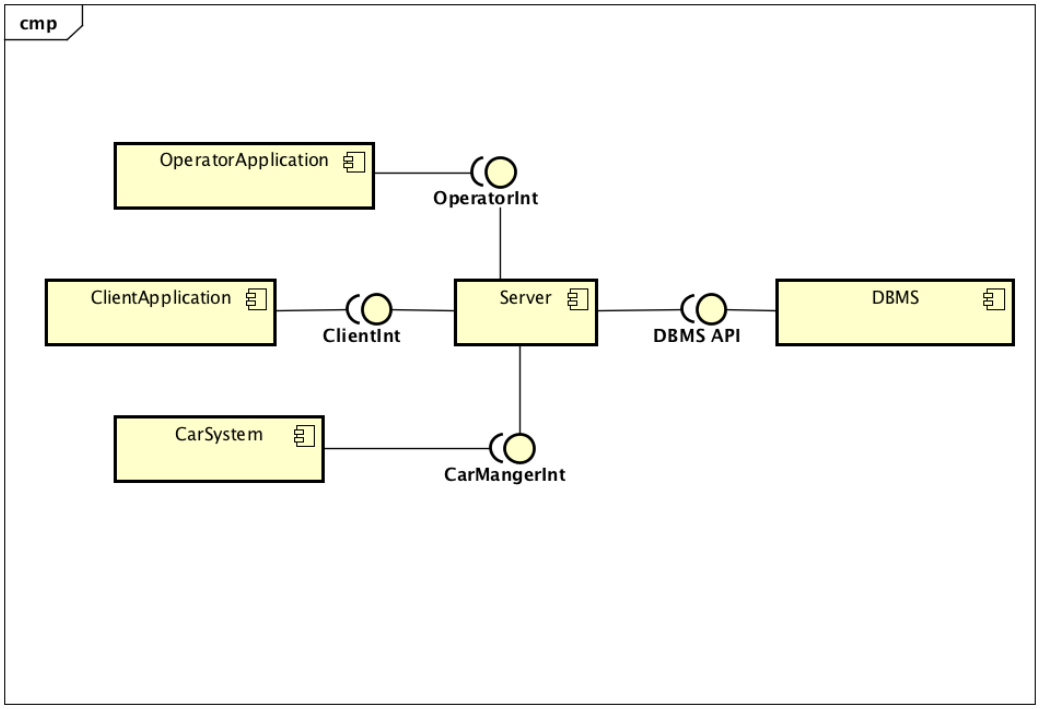


Figure 2.2: High level components

2.3 Component view

As described above the server is the central component of the system, so we decide to describe with a more detailed description this component.
The server is mainly composed of different controllers which cover the functionalities provided through interfaces to the high level component.
The three different view handle the respective request and route them to the right controller.

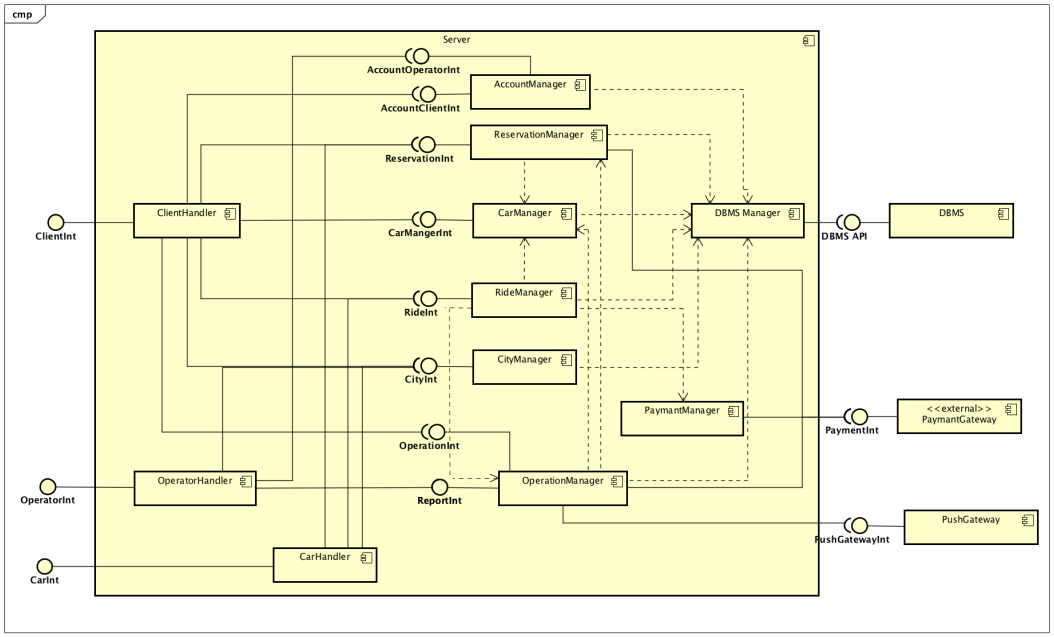


Figure 2.3: Component view

- **AccountManager**: manage login of client and operator and registration of new client
- **CarManager**: manage car state
- **CityManager**: manage power grid stations and safe areas
- **ReservationManager**: manage reservations
- **RideManager**: manage rides
- **OperationManager**: manage damage report and operation of operators
- **PushGateway**: manage notification to operators
- **PaymentManager**: manage computation of total cost of a ride included possible fees or discounts.

2.4 Database structure

The following diagram show the database of PowerEnjoy, it contains the data that the application need to know and modify.

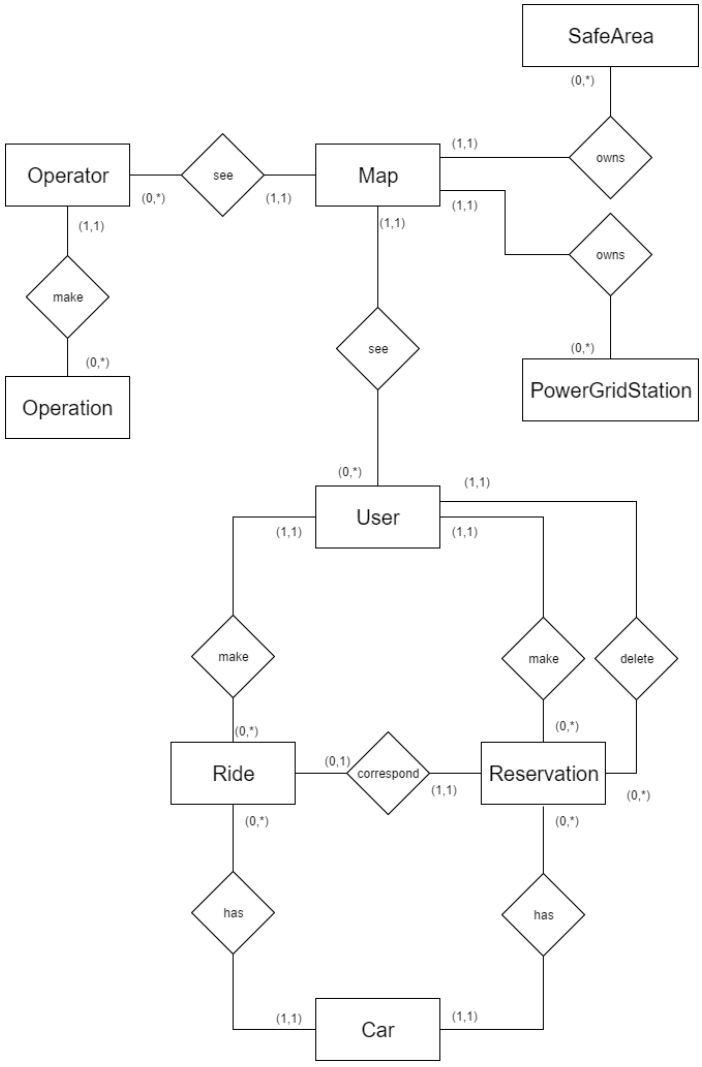


Figure 2.4: Database structure

2.5 Deploying view

This section shows how the system is composed in terms of hardware and software, and how components communicate each other. In the first tier, user can access to the service through mobile phone or personal computer. PowerEnJoy provide to each operator a smartphone, where the Operator App is preinstalled and they access from there. Computer needs a Web Server to run the Web Browser, so they are in communication. Furthermore all devices of the first tier need to communicate with the PowerEnJoy Server, in which there is the application logic. PowerEnJoy Server communicates with the database Server in order to collect data and have access to the database. Devices communicate among them using TCP/IP protocol.

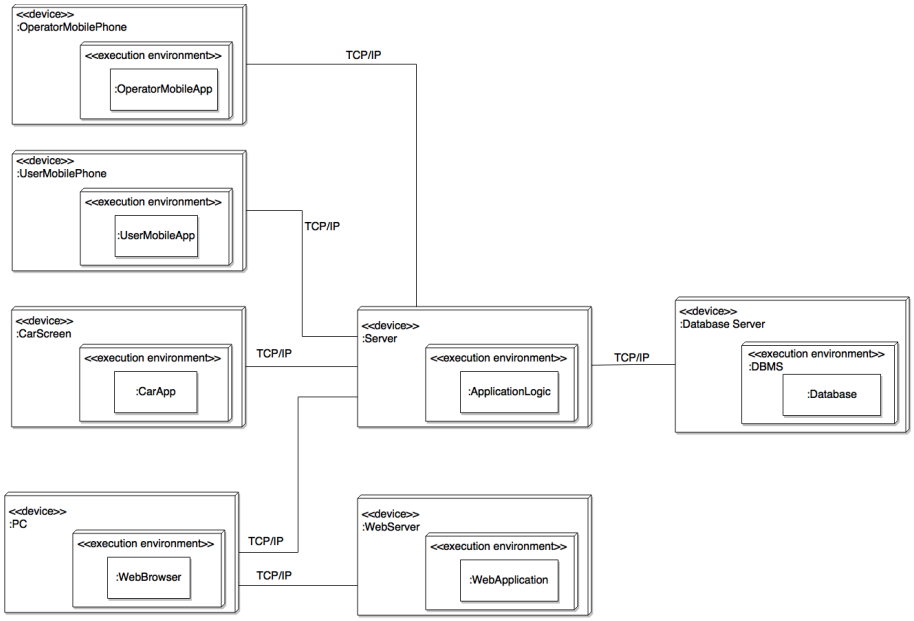
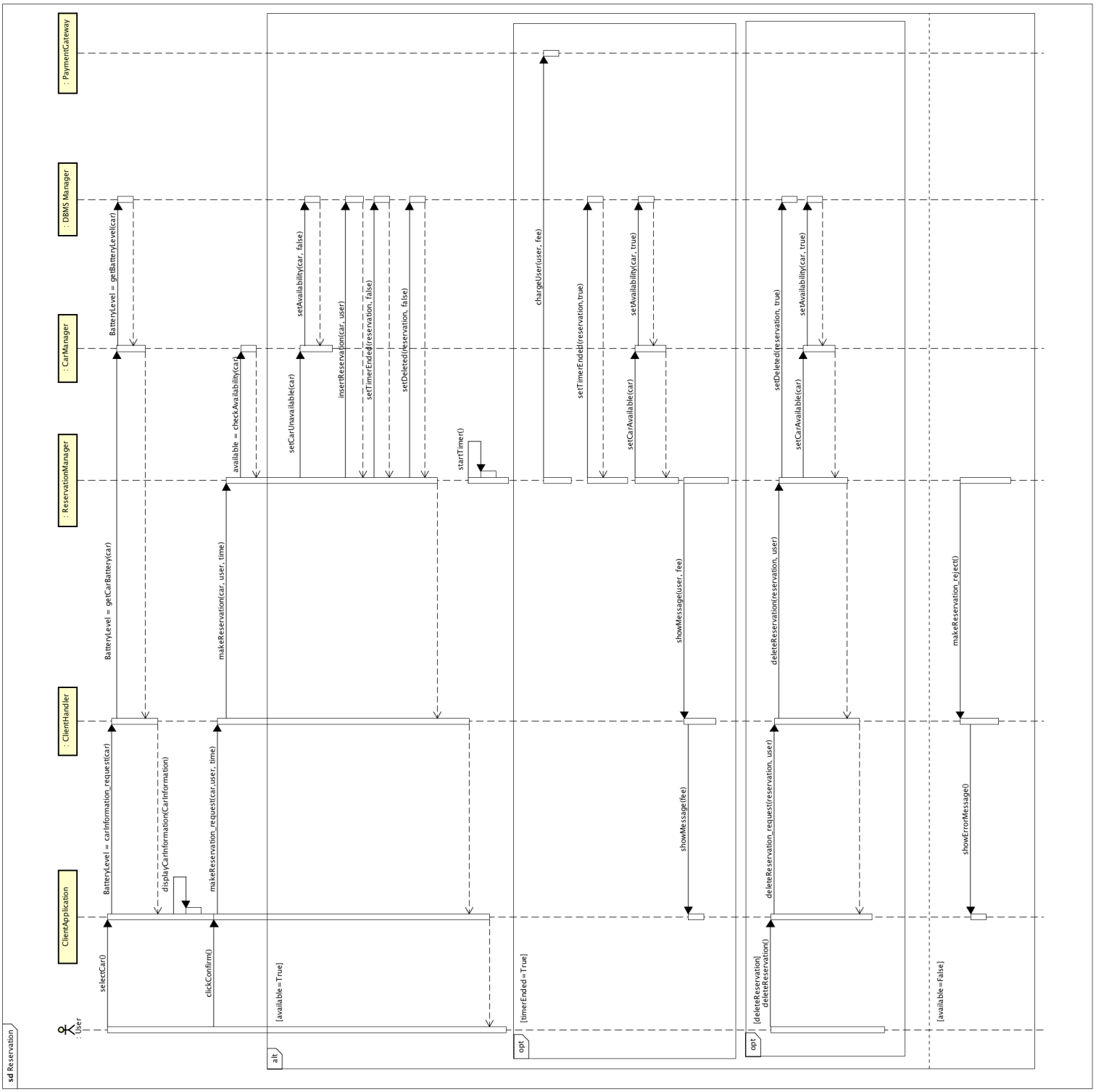


Figure 2.5: Deployment view

2.6 Runtime view

2.6.1 Reservation

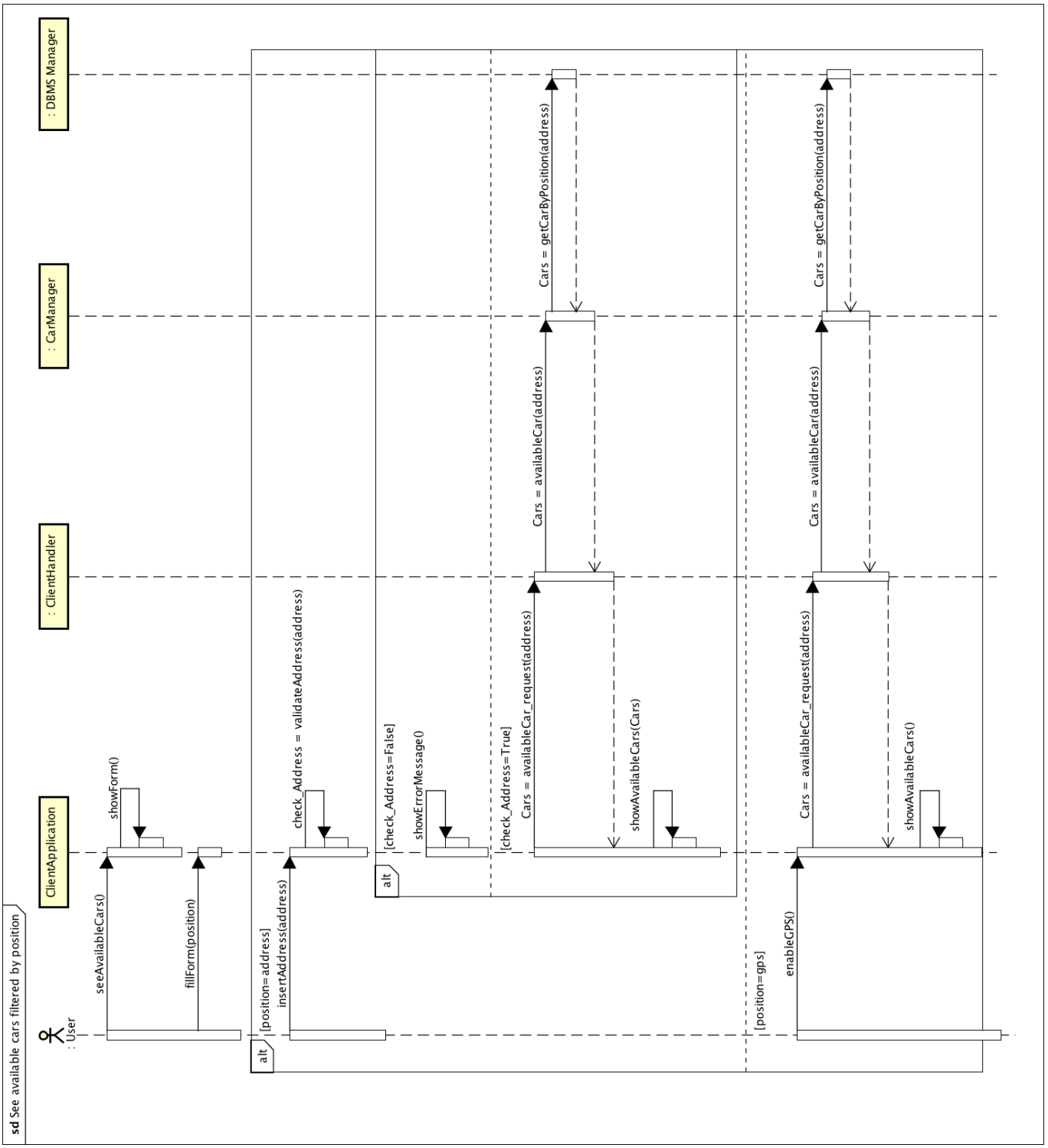
This sequence diagram describes how user can reserve an available car. First of all, user has to select a car from the available car that system shows him. Car informations request is sent to CarManager, from ClientApplication through ClientHandler, and CarManager retrieves informations from the DataBase. If car information satisfy user expectation, then user confirm the reservation: ClientApplication sends the request to the ClientHandler that transfer the request at the ReservationManager and CarManager and at the end DBMS set car as unavailable. After that, reservation manager send a request to the DBMS for create a new tuple of reservation in the DBMS and start the timer of 1 hour.


sd Reservation
: User
: ClientApplication
: ClientHandler
: ReservationManager
: CarManager
: DBMS Manager
: PaymentGateway
selectCar()
BatteryLevel = carInformation_request(car)
BatteryLevel = getBatteryLevel(car)
BatteryLevel = getCarBattery(car)
displayCarInformation(CarInformation)
clickConfirm()
makeReservation_request(car,user, time)
available = checkAvailability(car)
makeReservation(car, user, time)
alt
[available=True]
setAvailability(car, false)
setCarUnavailable(car)
insertReservation(car, user)
setTimerEnded(reservation, false)
setDeleted(reservation, false)
startTimer()
opt
[timerEnded=True]
chargeUser(user, fee)
setTimerEnded(reservation,true)
setAvailability(car, true)
setCarAvailable(car)
showMessage(user, fee)
showMessage(fee)
opt
[deleteReservation]
deleteReservation()
deleteReservation_request(reservation, user)
deleteReservation(reservation, user)
setDeleted(reservation, true)
setAvailability(car, true)
setCarAvailable(car)
[available=False]
makeReservation_reject()
showErrorMessage()

2.6.2 See available car

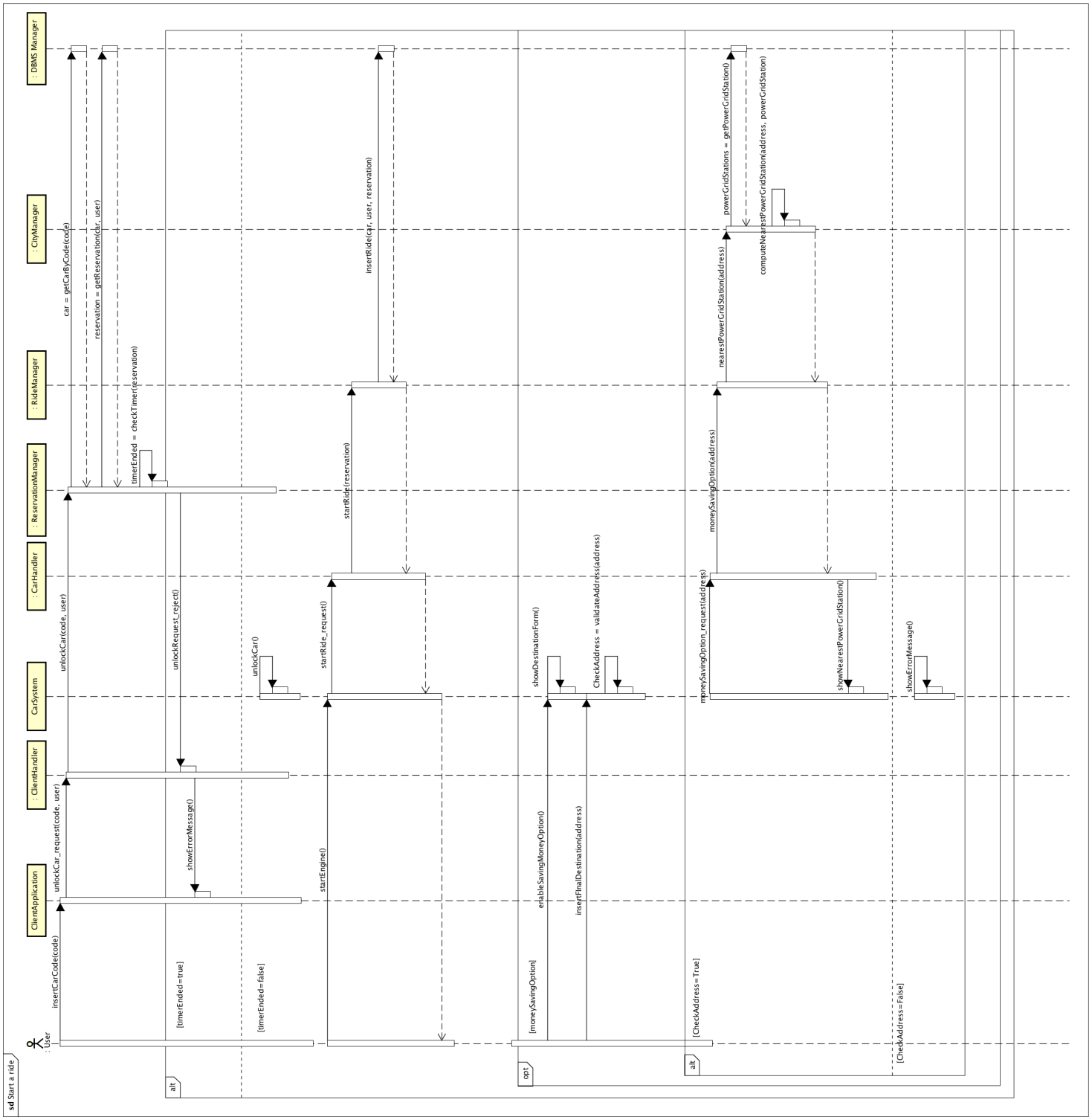
This sequence diagram shows how user can see available cars near him. When user choose to see available cars, user application shows him a form, in which the user has to choose if finds car by inserting an address or by enabling gps.

- If user choose to **insert an address**, ClientApplication checks if the inserted address is a correct one. If the user has inserted a wrong address, the ClientApplication shows him an error message. Otherwise ClientApplication forwards the request to the DBMS, passing through ClientHandler and CarManager. User application shows to the user the available cars near him.
- If user chooses to **enable GPS**, ClientApplication forwards the request to DBMS, passing through ClientApplication and CarManager. ClientApplication shows to the user the available cars near him.


sd See available cars filtered by position
: User
ClientApplication
: ClientHandler
: CarManager
: DBMS Manager
seeAvailableCars()
showForm()
fillForm(position)
alt
[position=address]
insertAddress(address)
check_Address = validateAddress(address)
alt
[check_Address=False]
showErrorMessage()
[check_Address=True]
Cars = availableCar_request(address)
Cars = availableCar(address)
Cars = getCarByPosition(address)
showAvailableCars(Cars)
[position=gps]
enableGPS()
Cars = availableCar_request(address)
Cars = availableCar(address)
Cars = getCarByPosition(address)
showAvailableCars()

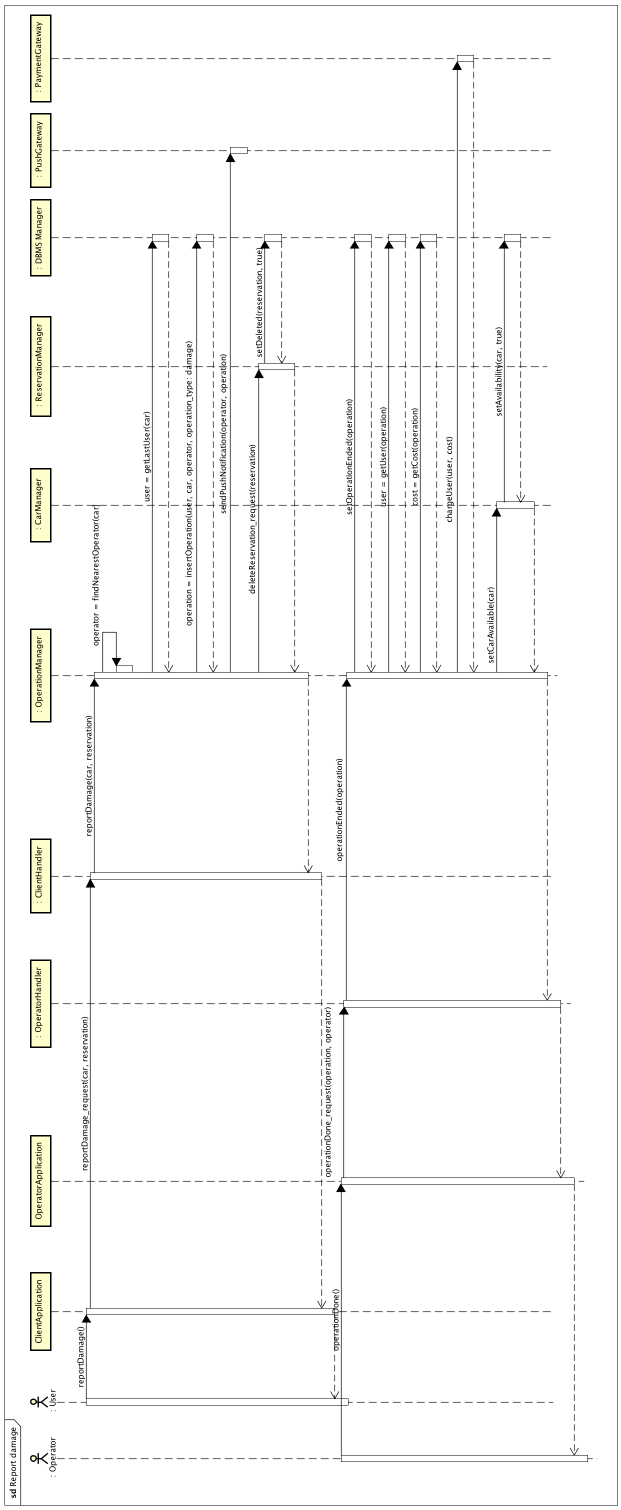
2.6.3 Start a ride

This sequence diagram shows how user starts a ride. A ride starts when the user inserts the car code in the app, the ClientApplication sends a request to unlock a car to the ClientHandler and ClientHandler sends a validate request to ReservationManager, at the end ReservationManager sends the request to the DBMS which return the reservation. If the user insert a wrong car code or the reservation doesn't exist, ClientApplication shows an error message. Otherwise the ReservationManager check if the timer is ended or not. If the timer is ended the ReservationManager reject the request of unlocking the car and the ClientApplication shows an error message. Otherwise the ReservationManager unlocks the car and as soon as the user start the engine the ride starts. Once the ride is started, the ReservationManager through the DBMS creates the ride tuple.


sd Start a ride
: User
ClientApplication
: ClientHandler
CarSystem
: CarHandler
: ReservationManager
: RideManager
: CityManager
: DBMS Manager
insertCarCode(code)
unlockCar_request(code, user)
unlockCar(code, user)
car = getCarByCode(code)
reservation = getReservation(car, user)
timerEnded = checkTimer(reservation)
alt
[timerEnded=true]
showErrorMessage()
unlockRequest_reject()
[timerEnded=false]
unlockCar()
startEngine()
startRide_request()
startRide(reservation)
insertRide(car, user, reservation)
opt
[moneySavingOption]
etableSavingMoneyOption()
insertFinalDestination(address)
showDestinationForm()
CheckAddress = validateAddress(address)
alt
[CheckAddress=True]
moneySavingOption_requestAddress()
moneySavingOption(address)
nearestPowerGridStation(address)
powerGridStations = getPowerGridStation()
computeNearestPowerGridStation(address, powerGridStation)
showNearestPowerGridStation()
[CheckAddress=False]
showErrorMessage()

2.6.4 Report a damage

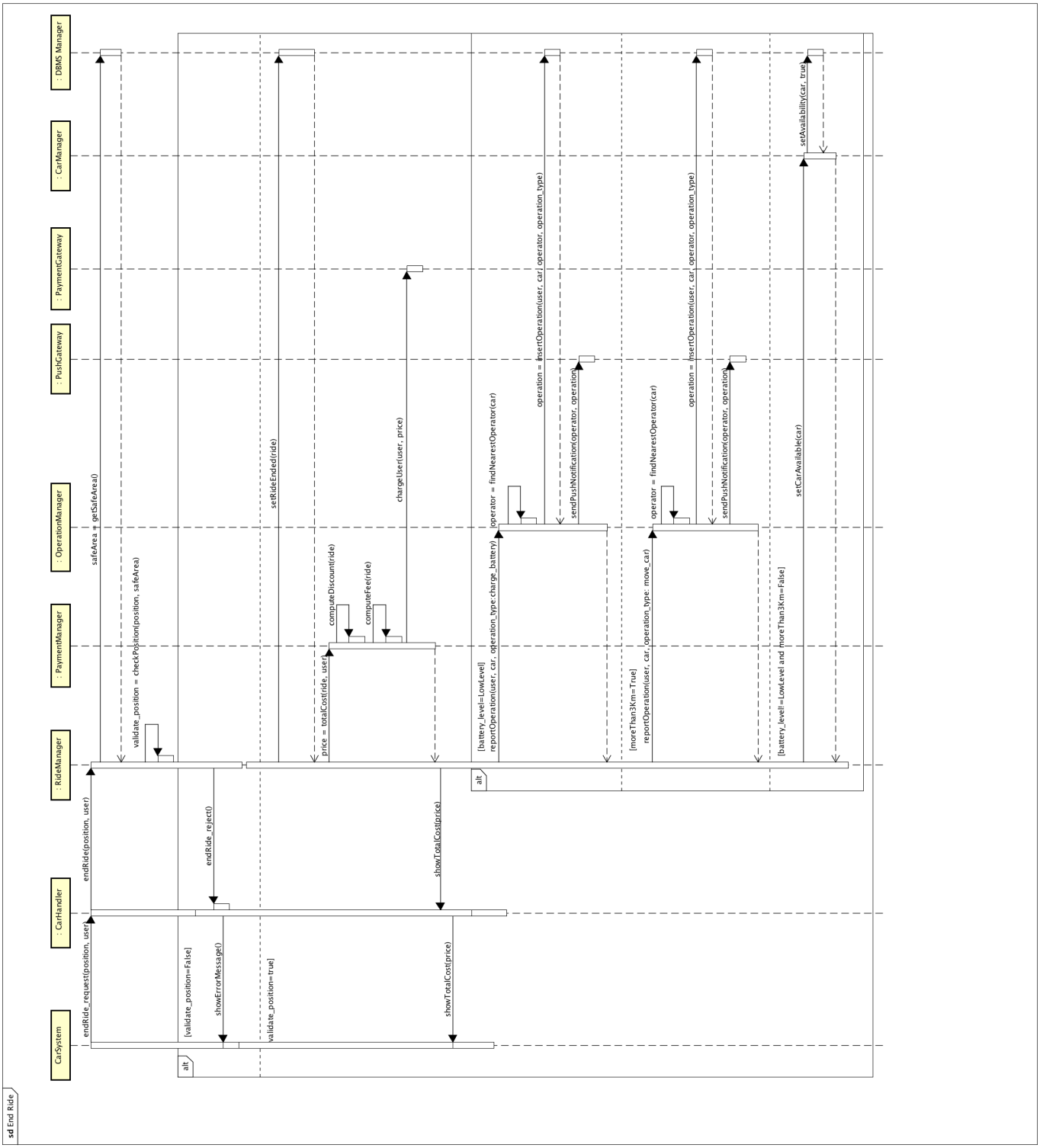
This sequence diagram describes how user can report a damage of a car. He/She has to click on the 'report damage' button and ClientApplication sends the request to ClienHandler. ClientHandler sends a request for create an operation to the OperationManager. As soon as the OperationManager receives the request finds the nearest operator to the damaged car. Then OperationManager sends a request to the DBMS, which create an operation tuple and NotificationManager take care of sending a notification to the operator that the OperationManager has selected before.


sd Report damage
: Operator
: User
: ClientApplication
: OperatorApplication
: OperatorHandler
: ClientHandler
: OperationManager
: CarManager
: ReservationManager
: DBMS Manager
: PushGateway
: PaymentGateway
reportDamage()
reportDamage_request(car, reservation)
reportDamage_request(car, reservation)
reportDamage(car, reservation)
operator = findNearestOperator(car)
user = getLastUser(car)
operation = insertOperation(user, car, operator, operation_type: damage)
sendPushNotification(operator, operation)
deleteReservation_request(reservation)
setDeleted(reservation, true)
operationDone()
operationDone_request(operation, operator)
operationEnded(operation)
setOperationEnded(operation)
user = getUser(operation)
cost = getCost(operation)
chargeUser(user, cost)
setCarAvailable(car)
setAvailability(car, true)

2.6.5 End ride

This sequence diagram shows how end ride functionality works. User has to click on the 'end ride' button on the car screen and the CarSystem will send a request of ending the ride to the RideManager passing through CarHandler. RideManager, through the DBMS, checks if the car position is inside of a safearea.

- if the car is not in a safearea, CarSystem shows to the user an error message
- if the car is in the safearea, DBMS set the value of RideEnded equal to true. RideManager asks to PaymentManager to calculate the total cost of the ride. PaymentManager will send this information to the payment gateway which has to charge the user. CarSystem shows the total cost to the user.


sd End Ride
: CarSystem
: CarHandler
: RideManager
: PaymentManager
: OperationManager
: PushGateway
: PaymentGateway
: CarManager
: DBMS Manager
endRide_request(position, user)
endRide(position, user)
validate_position = checkPosition(position, safeArea)
safeArea = getSafeArea()
alt
[validate_position=False]
endRide_reject()
showErrorMessage()
[validate_position=true]
setRideEnded(ride)
computeDiscount(ride)
computeFee(ride)
price = totalCost(ride, user)
chargeUser(user, price)
showTotalCost(price)
showTotalCost(price)
alt
[battery_level=LowLevel]
operator = findNearestOperator(car)
reportOperation(user, car, operation_type:charge_battery)
operation = insertOperation(user, car, operator, operation_type)
sendPushNotification(operator, operation)
[moreThan3Km=True]
operator = findNearestOperator(car)
reportOperation(user, car, operation_type: move_car)
operation = insertOperation(user, car, operator, operation_type)
sendPushNotification(operator, operation)
[battery_level=LowLevel and moreThan3Km=False]
setCarAvailable(car)
setAvailability(car, true)

2.7 Component interfaces

Dbms Manager Interface
+ getCar(code: int)
+ getBatteryLevel(car: Car)
+ setAvailability(car: Car, availability: boolean)
+ getSafeArea()
+ getPowerGridStation()
+ insertReservation(car: Car, user: User)
+ setTimerEnded(reservation: Reservation, ended: boolean)
+ setDeleted(reservation: Reservation, deleted: boolean)
+ insertOperation(user: User, car: Car, operator: Operator, type: operation_type)
+ getLastUser(car: Car)
+ getUser(operation: Operation)
+ getCost(operation: Operation)
+ insertRide(car: Car, user: User, reservation: Reservation)
+ setRideEnded(ride: Ride)
+ insertUser(user: User)
+ insertOperator(operator: Operator)
+ checkCredentials(user: User, password: String)
+ checkCredentials(operator: Operator, password: String)

Car Manager Interface
+ availableCar(address : String)
+ setCarUnavailable(car : Car)
+ getCarInformation(car : Car)
+ setCarAvailable(car : Car)
+ checkAvailability(car : Car)
+ seeBatteryLevel(car: Car)

City Manager Interface
+ seeSafeArea()
+ seePowerGridStation()
+ nearestPowerGridStation(address : String)

Reservation Manager Interface
+ makeReservation(car : Car, user : User, time : int)
+ deleteReservation(reservation : Reservation, user : User)
+ unlockCar(code: int, user: User)

Account Manager Interface
+ login(user : User, password : String)
+ login(operator : Operator, password : String)
+ register(user: User)

Operation Manager Interface
+ reportOpeartion(user : User, car : Car, type : operation_type)
+ operationEnded(operation: Operation,)

Ride Manager Interface
+ endRide(user : User, position : Position)
+ moneySavingOption(address : String)
+ startRide(reservation : Reservation)

Push Gateway Interface
+ sentPushNotification(operator: Operator, operation: Operation)

Payment Manager Interface
+ totalCost(ride: Ride, user: User)

2.8 Selected architectural styles and patterns

Client-Server

Our application is based on a client-server communication model. User's mobile application, user's web browser, operator's mobile application and car application are clients. There is a thin-client in order to let the application run on low-resources devices. We choose for client-server communication model because:

- Data synchronization: there is only one application that manage datas
- Having one unique server application improves the maintainability of our system
- The application is independent from the number of clients connected. Servers can play different roles for different clients.
- Upgradation and scalability: changes can be made easily by just upgrading the server. Also new resources and systems can be added by making necessary changes in server
- Improves the security between clients

Three-tiers application

Our architecture is divided in three tiers:

- Thin-client
- Application Logic
- Database

Physical tiers are about where the code runs. Three tiers architecture allows to each tier to be updated and replaced independently of requirements or technology changes. Each tier corresponds to a specific layer:

- Client → Presentation Layer
- Application Logic → Application Logic Layer
- Database → Resource Management Layer

Logical layers are about the organization of the code. We choose a three-tiers architecture because of its numerous advantages, such as :

- Layers are completely distinguished
- Allows more scalability: in fact it's possible to integrate more than one server and manage more users at the same time
- Allows more portability
- Easier maintenance
- More flexibility
- More security

2.9 Other design decision

We integrate web application, mobile application and the car system with a map service, and we choose to integrate with an open-map service.

3

Algorithm design

3.1 Relocation strategies

The system that we are going to develop belongs to free-floating Car Sharing Systems. These system allow users to pick up a car and drop it off wherever the user wants. The main issue of those new systems is that vehicles sometimes could get stuck in areas of lower demand while required in zones of higher demand. There are different relocation strategies to solve this problem which could be divided into operator-based strategies and user-based strategies. Operator-based strategies are based on interventions made by an operator while in user-based strategies the relocation process is carried out by users. The money saving option tries to relocate cars through an user-based strategies encouraging users with discounts. In "Relocation Strategies and Algorithms for Free-Floating Car Sharing Systems"[1] is presented a two-step model for the relocation of vehicles within free-floating Car Sharing Systems.

[1]Simone Weikl, Klaus Bogenberger , "Relocation Strategies and Algorithms for Free-Floating Car Sharing Systems", IEEE Intelligent Transportation Systems Magazine, Volume: 5, 2013.

4

User interface design

4.1 Mockups

We have already done mockups in RASD document.

4.2 UX diagrams

We insert UX (user experience) to show how our user performs his actions and how interacts with the system

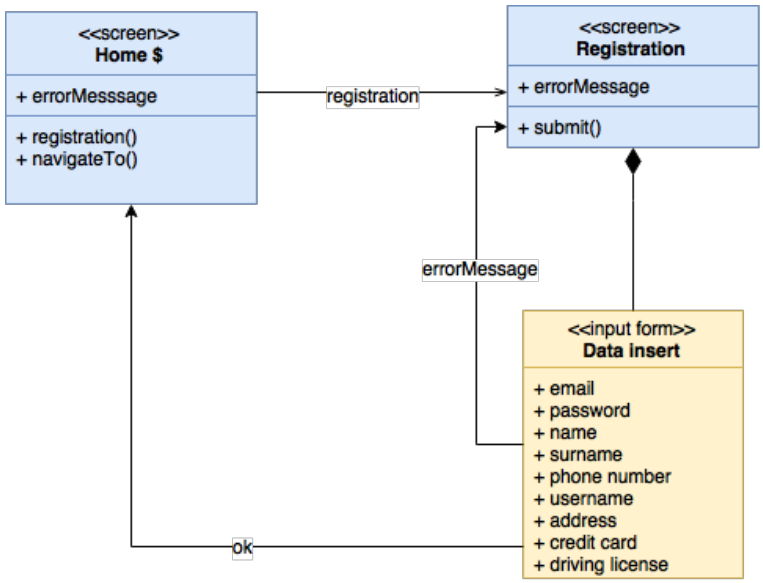


Figure 4.1: UX visitor

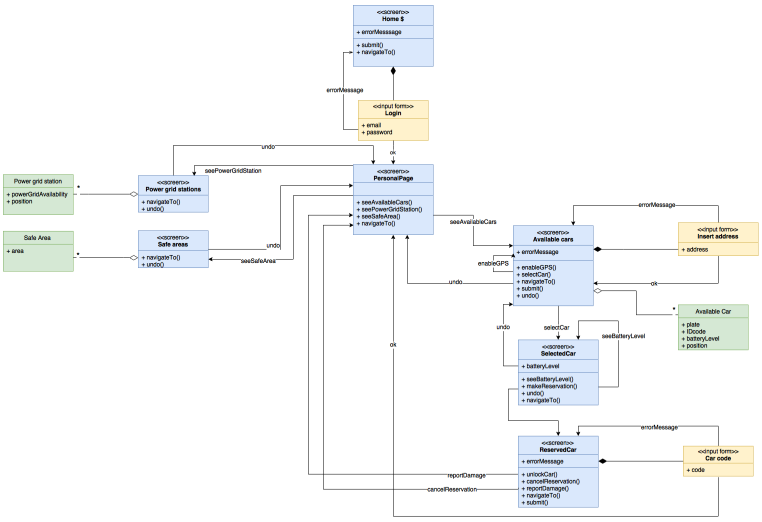


Figure 4.2: UX user

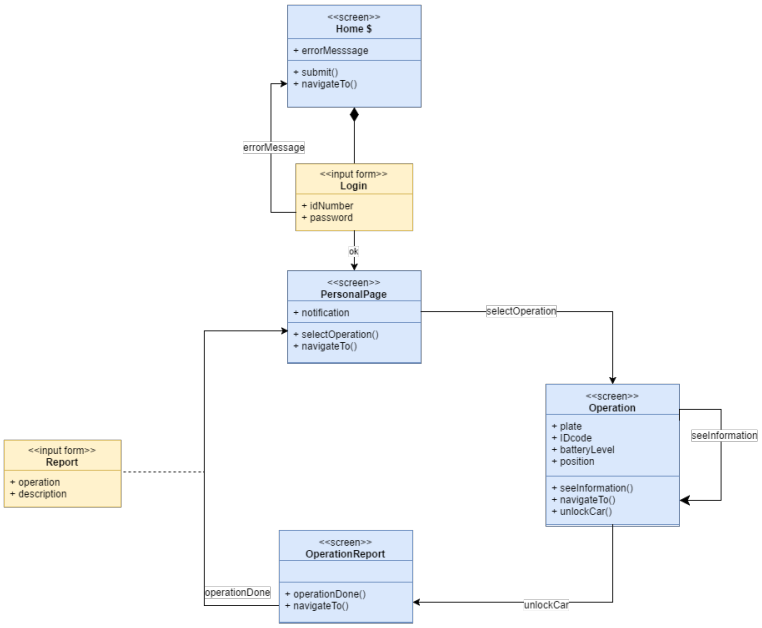


Figure 4.3: UX operator

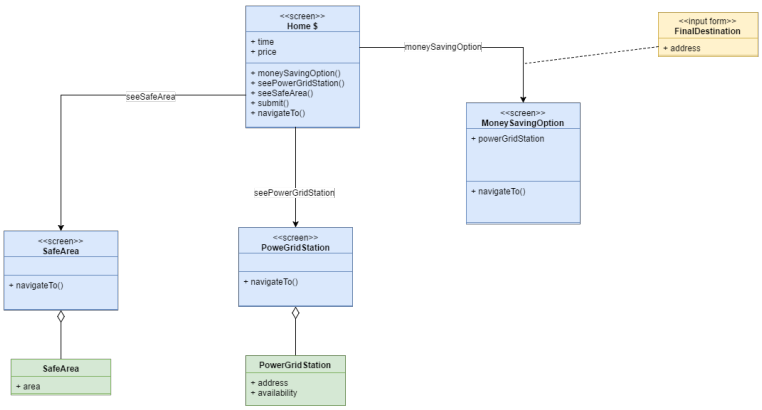


Figure 4.4: UX car

4.3 BCE diagrams

BCE diagrams shows how user and operator actions are managed internally from the system manager and how that managers interact with the database.

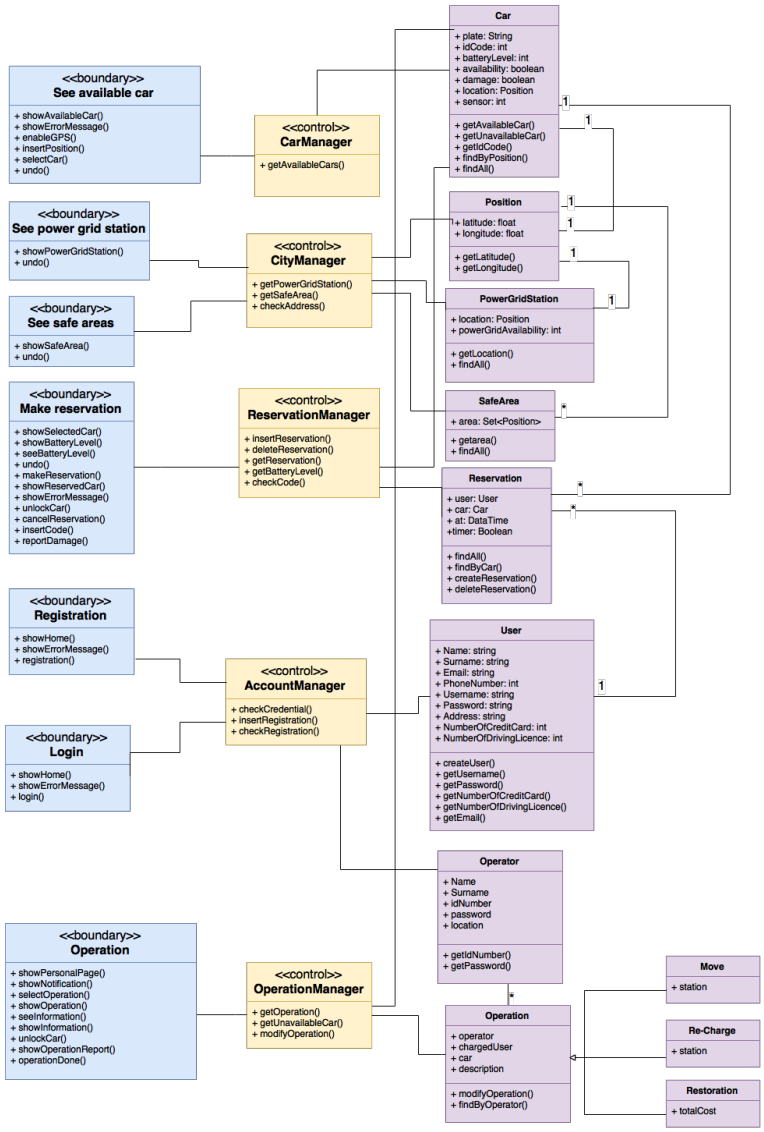


Figure 4.5: BCE user

5

Requirements traceability

The design of this project was made aiming to satisfy the requirements and goals specified in the RASD. The reader can find here the list of these requirements and goals and the design component of the application which will assure its fulfillment.

5.1 Visitor

- [G1] Allow visitors to register in the system
  - ClientApplication
  - ClientHandler
  - AccountManager
  - DBMS

5.2 User

- [G2] Allows users to login in the system
  - ClientApplication
  - ClientHandler
  - AccountManager
  - DBMS
- [G3] Users must be able to find the locations of available cars
  - ClientApplication
  - ClientHandler
  - CarManager

  - DBMS

- [G4] Allows users to request for the reservation of a car until one hour before
  - ClientApplication
  - ClientHandler
  - ReservationManager
  - CarManager
  - DBMS
- [G5] Allows users to unlock the reserved car
  - ClientApplication
  - ClientHandler
  - ReservationManager
  - DBMS
- [G6] Allows users to know the current battery level of each available cars
  - ClientApplication
  - ClientHandler
  - CarManager
  - DBMS
- [G7] Allows users to know which are the “safe area”
  - ClientApplication
  - ClientHandler
  - CityManager
  - DBMS
- [G8] Users should enable the money saving option in the car
  - ClientApplication
  - ClientHandler
  - CarSystem
  - CarHandler
  - RideManager
  - DBMS
- [G9] Users should know where the power grid stations are
  - ClientApplication

  - ClientHandler
  - CityManager
  - DBMS
- [G10] Users can report if the reserved car has damage
  - ClientApplication
  - ClientHandler
  - OperationManager
  - PushGateway
  - DBMS
- [G11] Allows users to cancel a reservation
  - ClientApplication
  - ClientHandler
  - ReservationManager
  - DBMS
- [G12] Allows users to have discount
  - RideManager
  - PaymentManager
  - DBMS
- [G13] Allows users to end the ride
  - CarSystem
  - CarHandler
  - RideManager
  - PaymentManager
  - DBMS

5.3 Operator

- [G14] Allows operators to login in the system
  - OperatorApplication
  - OperatorHandler
  - AccountManager
  - DBMS
- [G15] Allows operators to know all the informations of damaged car

  - OperatorApplication
  - OperatorHandler
  - CarManager
  - DBMS

- [G16] Allows operators to unlock cars
  - OperatorApplication
  - OperatorHandler
  - OperationManager
  - DBMS

- [G17] Operators should receive a notification for incoming request of reparation
  - OperatorApplication
  - OperatorHandler
  - OperationManager
  - PushGateway
  - DBMS

- [G18] Operators must be able to report the operation they have done
  - OperatorApplication
  - OperatorHandler
  - OperationManager
  - DBMS

- [G19] Operators should receive a notification for moving a car
  - OperatorApplication
  - OperatorHandler
  - OperationManager
  - PushGateway
  - DBMS

- [G20] Operators should receive a notification for charging a car
  - OperatorApplication
  - OperatorHandler
  - OperationManager
  - PushGateway
  - DBMS

6

References

6.1 Used tools

The tools we used to create this DD document are:

- Github: for version controller
- Lyx: to format this document
- Astah: for uml models
- Google Doc: for write the document
- Draw.io: for diagrams

7

Effort spent

For redacting and writing this document we spent 25 hours per person

8

Changelog

- v1.1
  - Sequence diagram
  - Component view diagram
  - Component interface