Integration Test Plan 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

Chapter 1

Introduction

1.1 Revision History

1.2 Purpose

This document represents the Integration Testing Plan Document for PowerEnJoy. The purpose of the integration test plan is to describe the necessary tests to verify that all of the components of PowerEnJoy are properly assembled. Integration testing ensures that the unit-tested modules interact correctly. This document aims to explain to the development team what to test, in which order and which tools are needed for testing.

1.3 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.

1.4 List of Definitions and Abbreviations

This section provides definitions for common terms used in the document. They are provided to help minimize ambiguity throughout the document.

- DD: Design Document
- RASD: Requirement Analysis and Specification Document

1.5 List of Reference Documents

The project description document: Assignments AA 2016-2017.pdf

- PowerEnJoy Requirement Analysis and Specification Document: RASD v1.1.pdf
- PowerEnJoy Design Document: DD.pdf
- Integration test plan example from previous year project:
  - Integration testing example document.pdf
  - Integration Test Plan Example.pdf

Chapter 2

Integration Strategy

2.1 Entry Criteria

This section concerns the prerequisites for the integration testing plan. At first the **Requirements Analysis and Specification Document** and the **Design Document** have to be drawn up.

Furthermore all classes of each component must be correctly **documented** for better understanding the overall behavior of the component that has to be integrated.

In addition to the documentation each class must be tested with **unit testing**. Unit tests give the ability to verify that functions work as expected. In fact unit tests are low-level, focusing on a small part of the software system. Each class has to be tested up to the thresholds of 80/90%.

2.2 Elements to be Integrated

IThis section identifies the components that have to be integrated. As described in the Design Document our system is mainly composed of these subsystems:

- Database Tier
- Server Tier
- Client Tier
  - Client
  - Operator
  - CarSystem

At the lowest-level of the server subsystem we have to integrate all the components that are dependent. These components are the same components described in the Design Document.

The components involved in this phase are: **DBMS Manager, AccountManager, CarManager, CityManager, ReservationManager, RideManager, OperationManager, PaymentManager, ClientHandler, OperatorHandler** and **CarHandler.**

Some low-level components depend on higher component, as in the case of the **DBMS Manager** that rely on the DBMS.

In conclusion of the integration process the higher level subsystems have to be integrated in order to obtain the full system.

2.3 Integration Testing Strategy

The integration testing process will follow a **bottom-up approach**. The bottom-up approach was chosen because of its many advantages.

Starting at the bottom of the hierarchy means that the critical components are generally tested first and therefore any errors or mistakes in these components are find out early in the process.

Component will be tested starting from those without any dependencies, then we will iteratively choose those with no dependencies.

Furthermore with a bottom-up approach no stubs are needed for the integration process, however temporarily driver are needed.

Notice that the DBMS is a commercial that have been already developed and can be used directly in the bottom-up approaches.

2.4 Sequence of Component/Function Integration

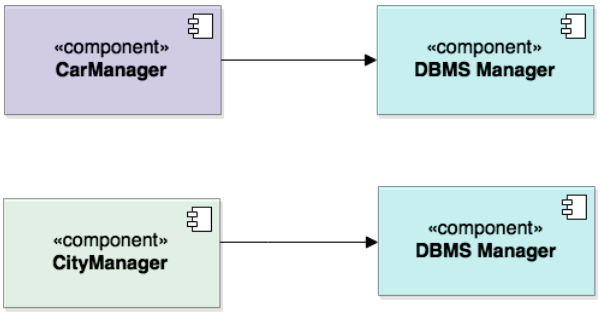
In this section we're going to describe the order of integration (and integration testing) of the various components and subsystems of PowerEnJoy.

As a notation, an arrow going from component C1 to component C2 means that C2 is necessary for C2 to function and so it must have already been implemented.

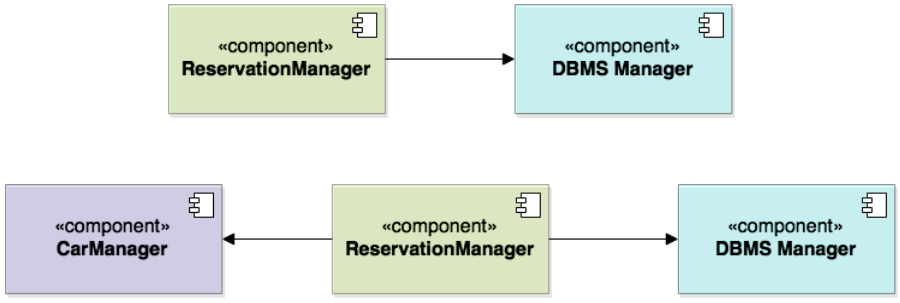
2.4.1 Software Integration Sequence

Following the already mentioned bottom-up approach, we now describe how the various subcomponents are integrated together to create higher level subsystems. We enter only in the server's subcomponents, because is our only non atomic subsystem.

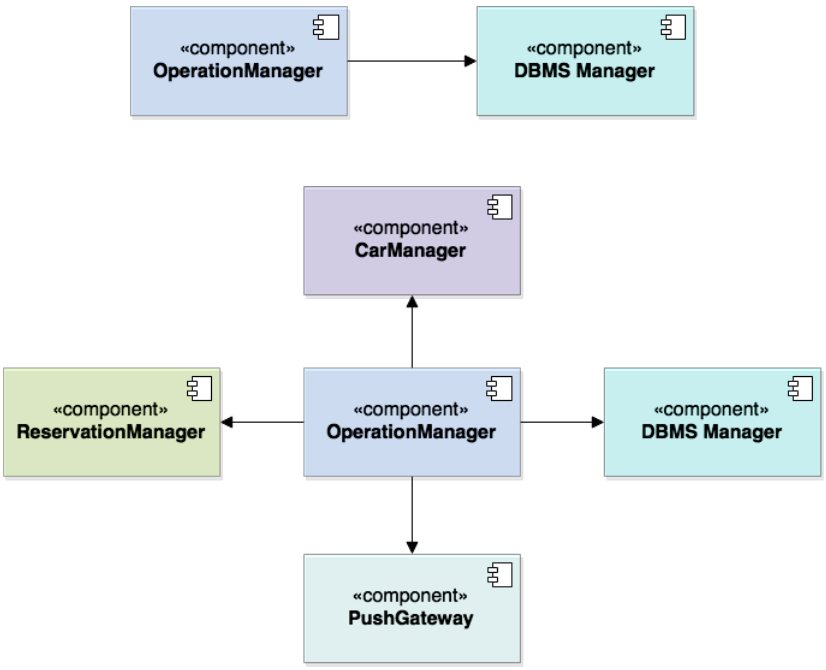
First of all we need to integrate **CarManager** subcomponent and **DBMS Manager** subcomponent, and then **CityManager** and **DBMS Manager**. We start from here because **DBMS Manager** is necessary for any other manager in the server in order to perform query to the DBMS.



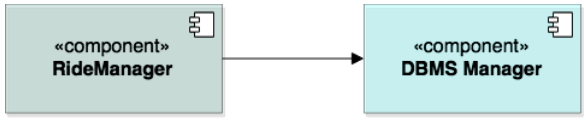
Then we proceed by integrating together the**ReservationManager** subcomponent and **DBMS Manager** as first. And then with the **ReservationManager** and **CarManager**.

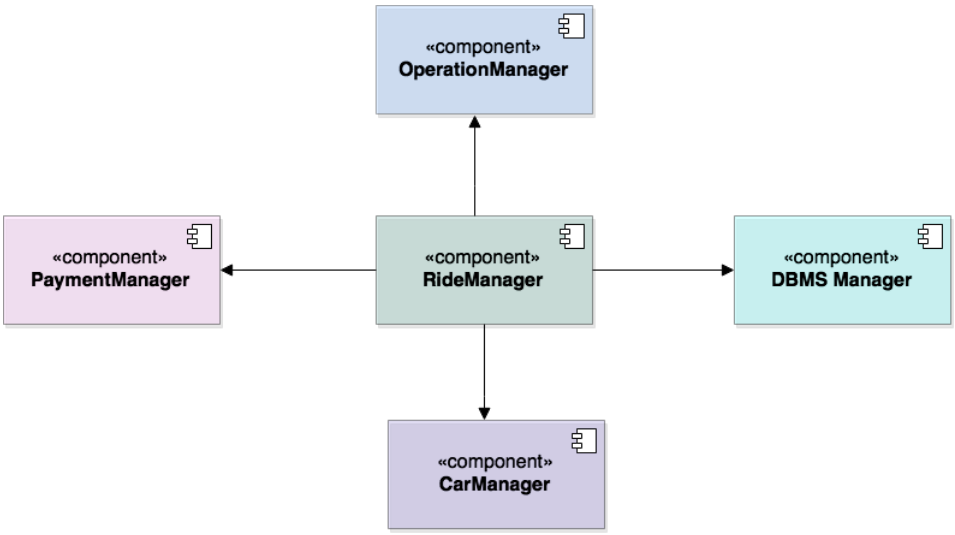


The same activity is performed between the **OperationManager** subcomponents and **DBMS Manager,** as first, and then with **CarManager, ReservationManager** and **PushGateway** subcomponents.

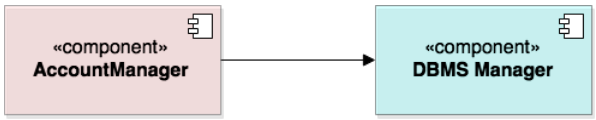


Again, the same activity as above is performed between the **RideManager** subcomponent and **DBMS Manager**, **OperationManager**, **PaymentManager** and **CarManager** subcomponents. As above the **DBMS Manager** is the first subcomponent to be integrated with the **RideManager**.

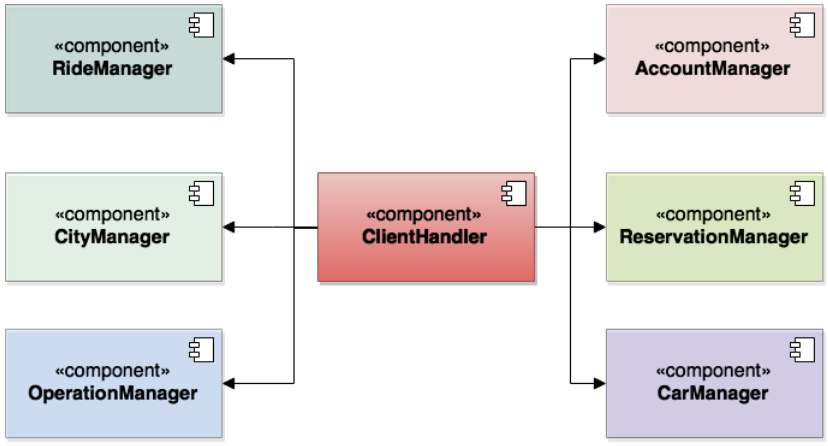




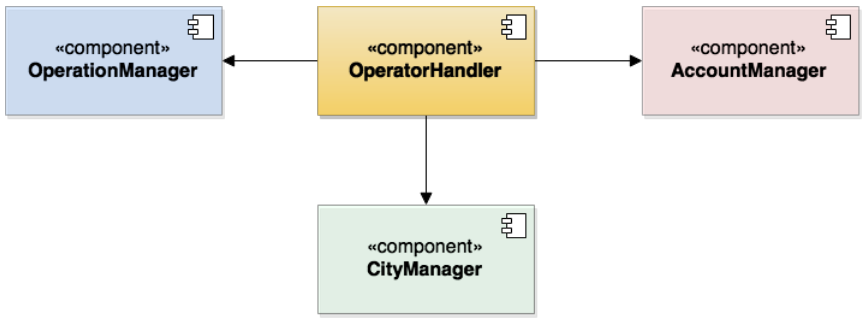
At this point we need to integrate the **AccountManager** subcomponent with the **DBMS Manager**.



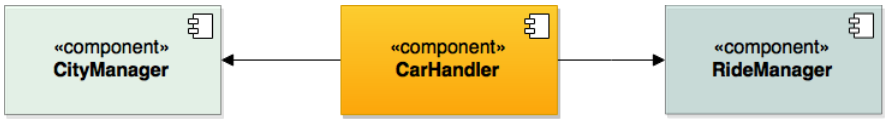
After that we can proceed by integrating together the **ClientHandler** subcomponent with the **RideManager**, **CityManager**, **OperationManager**, **AccountManager**, **ReservationManager** and **CarManager**.

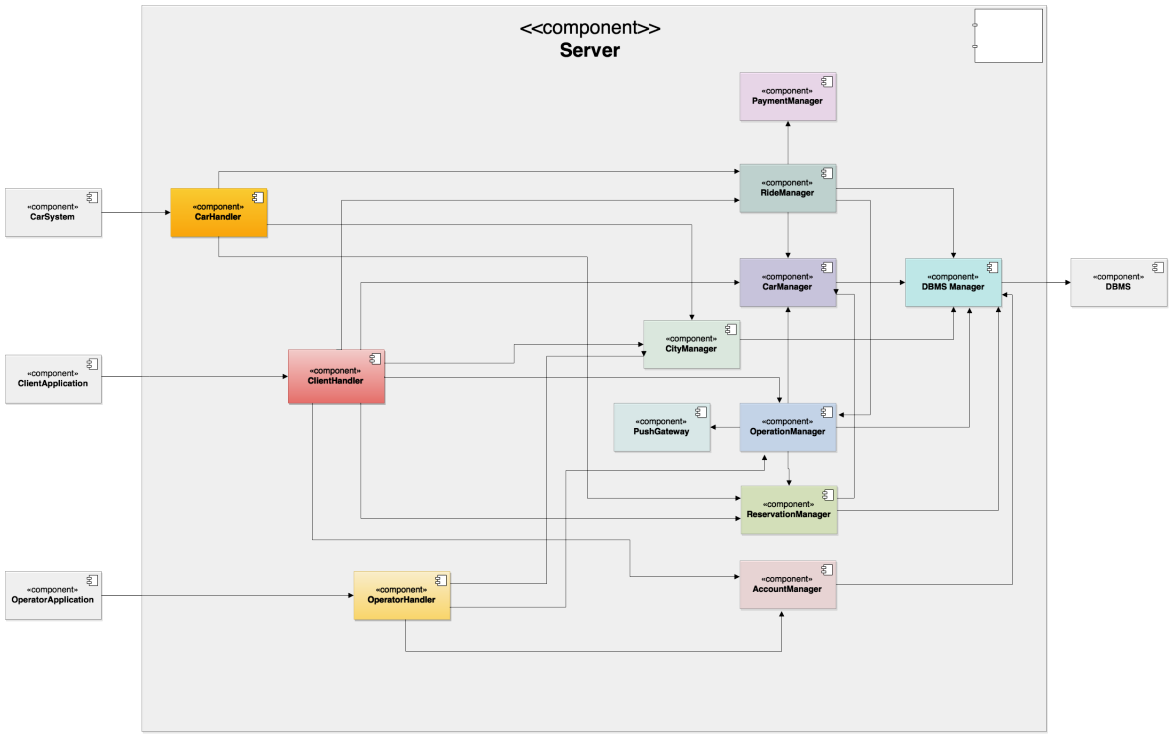


We can do the same with the **OperatorHandler** and the **CarHandler** subcomponents. For the **OperatorHandler** we need to integrate to the **OperationManager, CityManager** and **AccountManager**.



And finally we can integrate the **CarHandler** with the **CityManager** and **RideManager** subcomponents.




<<component>>
Server
«component»
PaymentManager
«component»
RideManager
«component»
CarSystem
«component»
CarHandler
«component»
CarManager
«component»
DBMS Manager
«component»
DBMS
«component»
CityManager
«component»
ClientApplication
«component»
ClientHandler
«component»
PushGateway
«component»
OperationManager
«component»
ReservationManager
«component»
AccountManager
«component»
OperatorApplication
«component»
OperatorHandler

2.4.2 Subsystem Integration Sequence

The PowerEnJoy application designed is divided in different sub-system. From the "High Level Component" (Design Document, figure 2.2 page 8) we can recognize five different sub-system:

- Operator Application
- Client Application
- Car System
- Server
- DBMS

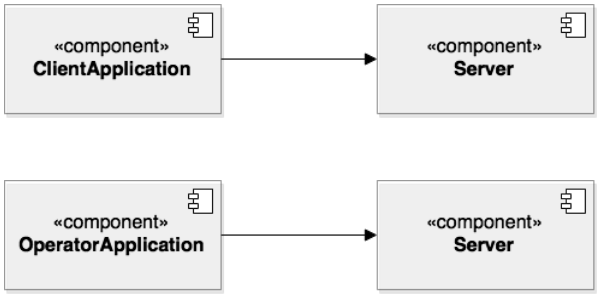
The **Operator Application** subsystem, the **Client Application** subsystem, the **Car System** subsystem and the **DBMS** subsystem are atomic subsystem, for this reason we do not enter in more details.

The central component of the system is the server, which is mainly composed of different controllers, and also of three differents view handle. More detailed description is on Design Document, at paragraph 2.3.

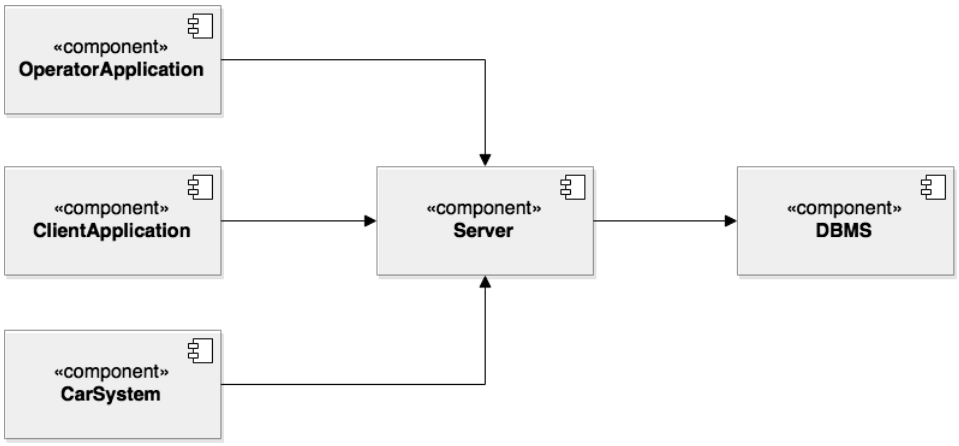
Only the server component is composed of different subcomponents, and only the components of the server subsystem need to be tested. The order of integration of the component of the server subsystem is explained in the previous paragraph.

Concerning the order of integration of subsystem, the server has to be integrated to the DBMS at first. In specific the **DBMS Manager** subcomponent of the server subsystem has to be integrated to the **DBMS**.

Once the server is fully integrated, the **Operator Application**, **Client Application** and **Car System** will be integrate with the server component. The order in which they will be integrated is not relevant, so they could be integrated with the server component in parallel.



At this point we have integrated the overall system.



Chapter 3

Individual Steps and Test Description

3.1 Server

3.1.1 Car Manager, DBMS Manager

getCar(code)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A valid code	Return the car.

getBatteryLevel(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A valid code	Return the battery level of the car passed as argument.

setAvailability(car, availability)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A valid set of parameters.	Set the availability of the car.

3.1.2 City Manager, DBMS Manager

getSafeArea()	
Input	*Effect*
Nothing	Return all the safe areas.

getPowerGridStation()	
Input	*Effect*
Nothing	Return all the power grid stations.

3.1.3 Reservation Manager, DBMS Manager

insertReservation(car, user)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A non-existing user	An InvalidArgumentValueException is raised.
A valid set of parameters.	An entry containing the reservation data is inserted in the database.

deleteReservation(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non existing reservation.	An InvalidArgumentValueException is raised.
A valid set of parameters.	An entry containing the reservation data is deleted from the database.

3.1.4 Reservation Manager, Car Manager

checkAvailability(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
An available car	Return true.
An unavailable car	Return false.

setTimerEnded(reservation, ended)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing reservation	An InvalidArgumentValueException is raised.
A valid set of parameters.	Set the attribute ended of the entry associated to the reservation at the value passed as argument.

setDeleted(reservation, ended)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing reservation	An InvalidArgumentValueException is raised.
A valid set of parameters.	Set the attribute deleted of the entry associated to the reservation at the value passed as argument.

3.1.5 Operation Manager, DBMS Manager

insertOperation(user, car, operator, operation_type)	
Input	*Effect*
A null parameter	A non-existing ride
A non-existing user	An InvalidArgumentValueException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A non-existing operator	An InvalidArgumentValueException is raised.
A valid set of parameters.	An entry containing the operation data is inserted in the database.

getLastUser(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car.	An InvalidOperationException is raised.
A valid car.	Return the last user of the car.

getUser(operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operation.	An InvalidOperationException is raised.
A valid operation.	Return the user of the operation.

getCost(operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operation.	An InvalidOperationException is raised.
A valid operation.	Return the cost of the operation.

3.1.6 Operation Manager, PushGateway

sendPushNotification(operator, operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operator	An InvalidArgumentValueException is raised.
A non-existing operation	An InvalidArgumentValueException is raised.
A set of valid parameters	The **PushGateway** sends a push notification to the operator that describes the operation that has to be done.

3.1.7 Operation Manager, Car Manager

setCarAvailable(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A valid car	The car passed as parameter is available.

3.1.8 Operation Manager, Reservation Manager

deleteReservation_request(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing reservation.	An InvalidOperationException is raised.
A valid reservation.	The car is tagged as available, the reservation is removed from the database.

3.1.9 Ride Manager, DBMS Manager

insertRide(car, user, reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A non-existing user	An InvalidArgumentValueException is raised.
A non-existing reservation	An InvalidArgumentValueException is raised.
A valid set of parameters.	An entry containing the ride data is inserted in the database.

setRideEnded(ride)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing ride	An InvalidArgumentValueException is raised.
A valid set of parameters.	Set the attribute ended of the entry associated to the ride passed as argument at true.

3.1.10 Ride Manager, Car Manager

setCarAvailable(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A valid car	The car passed as parameter is available.

3.1.11 Ride Manager, Operation Manager

reportOperation(user, car, operation_type)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A non-existing user	A non-existing user
A non-existing operation_type	An InvalidArgumentValueException is raised.
A user that has ended a ride with a low battery level.	The operation manager find the nearest operator and send him through the **PushGateway** a notification of a new operation for charging a car.
A user that has ended a ride farther than 3 km from a power grid station.	The operation manager find the nearest operator and send him through the **PushGateway** a notification of a new operation for moving a car.

3.1.12 Ride Manager, Payment Manager

totalCost(ride, user)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing ride	An InvalidArgumentValueException is raised.
A non-existing user	A non-existing user
A user that has ended a ride in a safe.	The **Payment Manager** compute the total price taking into account potential discount or fee.

3.1.13 Account Manager, DBMS Manager

insertUser(user)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An user that already exist	An InvalidArgumentValueException is raised.
A valid user	An entry containing the user data is inserted in the database.

insertOperator(operator)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An operator that already exist	An InvalidArgumentValueException is raised.
A valid user	An entry containing the operator data is inserted in the database.

checkCredentials(user, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An user that doesn't exist	An InvalidArgumentValueException is raised.
An existing user with an incorrect password.	Return false.
A valid combination of user and password	Return true.

checkCredentials(operator, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An operator that doesn't exist	An InvalidArgumentValueException is raised.
An existing operator with an incorrect password.	Return false.
A valid combination of operator and password	Return true.

3.1.14 Client Handler, Account Manager

register(user)	
Input	*Effect*
A null parameter.	A NullArgumentException is raised.
An user with one or more emtpy fields.	An InvalidArgumentValueException is raised.
An user with the same mail address of an existing user.	The registration wasn't succesfull.
A valid user	The registration was succesfull.

login(user, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An user that doesn't exist	An InvalidArgumentValueException is raised.
An existing user with an incorrect password.	The login wasn't succesfull.
A valid combination of user and password	The login was succesfull.

3.1.15 Client Handler, Car Manager

seeBatteryLevel(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A valid code	Return the battery level of the car passed as argument.

3.1.16 Client Handler, City Manager

seeSafeAreas()	
Input	*Effect*
Nothing	Return the safe areas.

seePowerGridStations()	
Input	*Effect*
Nothing	Return the power grid stations with availability greater than 0.

3.1.17 Client Handler, Reservation Manager

makeReservation(car, user)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A non-existing user	An InvalidArgumentValueException is raised.
A car that has already been reserved.	An InvalidOperationException is raised.
A valid set of parameters.	The car is tagged as unavailable, the reservation is written into the database.

deleteReservation_request(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing reservation.	An InvalidOperationException is raised.
A valid reservation.	The car is tagged as available, the reservation is removed from the database.

unlockCar(code, user)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing code.	An InvalidArgumentException is raised.
A non-existing user.	An InvalidArgumentException is raised.
A car not reserved by the user.	An InvalidOperationException is raised.
A car is reserved by the user but the timer of the reservation is ended.	An InvalidOperationException is raised.
A car is reseved by the user and the timer is not ended.	The car is unlocked and the user could start the engine of the car.

3.1.18 Client Handler, Operation Manager

reportDamage(car, reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing car	An InvalidArgumentValueException is raised.
A non-existing reservation	An InvalidArgumentValueException is raised.
A valid set of parameters.	The **Operation manager** find the nearest operator and send him through the **PushGateway** a notification of a new operation for repairing the car. The reservation is deleted from the database.

3.1.19 Operator Handler, Account Manager

login(operator, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An operator that doesn't exist	An InvalidArgumentValueException is raised.
An existing operator with an incorrect password.	The login wasn't succesfull.
A valid combination of operator and password	The login was succesfull.

3.1.20 Operation Handler, Operation Manager

operationEnded(operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operation	An InvalidArgumentValueException is raised.
A valid operation	The attribute ended of the operation is setted at true, the last user that used the car is charged of the cost of the operation.

3.1.21 Car Handler, Ride Manager

startRide(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing reservation	An InvalidArgumentValueException is raised.
A valid reservation	An entry containing the ride data is inserted in the database.

endRide(position, user)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing user	An InvalidArgumentValueException is raised.
A valid user that doesn't park the car in a safe area.	An InvalidOperationException is raised.
A valid user parks the car in a safe area with low level battery.	The user is charged of the total amount of the ride, the nearest operator is notified of an operation for charging the car.
A valid user parks the car in a safe area farther than 3 km from a power grid station.	The user is charged of the total amount of the ride, the nearest operator is notified of an operation for moving the car.
A valid user parks the car in a safe area within 3 km from a power grid station with a batery level greater than 20%.	The user is charged of the total amount of the ride and the car is tagged as available.

moneySavingOption(address)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing address	An InvalidArgumentValueException is raised.
A valid address	The **RideManager** return the nearest available power grid station.

3.1.22 Car Handler, City Manager

seeSafeAreas()	
Input	*Effect*
Nothing	Return the safe areas.

seePowerGridStations()	
Input	*Effect*
Nothing	Return the power grid stations with availability greater than 0.

Chapter 4

Tools and Test Equipment Required

In order to test our component efficiently some automated tools are needed.

JUnit: is an open source framework designed for the purpose of writing and running test. The main purpose of JUnit is unit testing, but it could be used also for testing the interaction between component. JUnit allows the developer to incrementally build test suites to measure progress and detect side effects. Furthermore it can be used in conjunction with Mockito. Mockito: is a popular mock framework which can be used in conjunction with JUnit.

Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. Mockito is useful for the generation of the stubs needed.

JMeter: is pure Java open source software used in performance testing. Performance testing means testing a web application against heavy load, multiple and concurrent user traffic. JMeter simulates a group of users sending requests to a target server, and return statistics information of target server.

Chapter 5

Program Stubs and Test Data Required

5.1 Driver

- **DBMS Manager Driver:** this module will invoke the methods exposed by the **DBMS Manager** in order to test its interaction with the **DBMS**.
- **Car Manager Driver:** this module will invoke the methods exposed by the **Car Manager** in order to test its interaction with the **DBMS Manager**.
- **City Manager Driver:** this module will invoke the methods exposed by the **City Manager** in order to test its interaction with the **DBMS Manager**.
- **Operation Manager Driver**: this module will invoke the methods exposed by the **Operation Manager** in order to test its interaction with **DBMS Manager**, **PushGateway**, **Car Manager** and with the **Reservation Manager**.
- **Ride Manager Driver:** this module will invoke the methods exposed by the **Ride Manager** in order to test its interaction with **DBMS Manager, Operation Manager, Payment Manager** and with the **Car Manager**.
- **Reservation Manager Driver:** this module will invoke the methods exposed by the **Reservation Manager** in order to test its interaction with the **DBMS Manager** and with the **Car Manager.**
- **Account Manager Driver:** this module will invoke the methods exposed by the **Account Manager** in order to test its interaction with the **DBMS Manager**.

- **Client Handler Driver:** this module will invoke the methods the methods exposed by the **Client Handler** in order to test its interaction with the **Account Manager, Reservation Manager, Car Manager, Ride Manager, City Manager** and with the **Operation Manager.**
- **Operator Handler Driver:** this module will invoke the methods exposed by the **Operation Handler** in order to test its interaction with the **Account Manager, City Manager** and with the **Operation Manager.**
- **Car Handler Driver:** this module will invoke the methods exposed by the **Car Handler** in order to test its interaction with the **Ride Manager** and with the **City Manager.**

5.2 Stub

- **Payment Gateway Stub:** this module will be used to simulate the behaviour of the external system when it is triggered by the **Payment Manager, Reservation Manager** or by the **Operation Manager.**

5.3 Test Data

In order to test the integration between the **DBMS Manager** and the **DBMS** a database that respect the configuration specified in the Design Document. The database needed in the integration test is a draft version of the final version sufficient for testing the **DBMS**.

Chapter 6

Effort Spent

For redacting and writing this document we spent 20 hours per person