Integration Test Plan Document

Version 1.0

Luca Santini	808710
Riccardo Remigio	874939

Index

1. Introduction

1.2 Purpose and Scope

This document specifies all the necessary information about the integration test plan for PowerEnJoy.
The purpose of integration testing is to verify functional and non functional requirements placed on RASD and Design Document, by testing the interaction between the different modules composing the whole software.

1.3 List of definitions and abbreviations

RASD: Requirements and Specifications Document

DD: Design Document

API: Application Programming Interface

DBMS: Database Management System

1.4 List of reference documents

- Assignments AA 2016-2017.pdf
- Verification and validation, part I.pdf
- Verification and validation, part II.pdf
- PowerEnJoy RASD
- PowerEnJoy DD
- Integration testing example document.pdf
- Mockito, Junit and Arquillian documentation

2. Integration strategy

2.1 Entry criteria

Before the beginning of the integration test phase, there are some other things that have to be necessarily completed.
For example, the RASD and the Design Document must have been released and part of the code, that covers the components that should be integrated, must already be written.
When will begin the integration testing there will still be some incomplete components that will be necessarily completed before finishing the integration testing.
Before starting the integration test all the unit tests must have almost completed and they will be continued during the integration test.

We do not write the precise percentage of completion, but we can say that, using a bottom-up approach, the first components to be tested are: Data Manager and Payment Manager.
Then Profile Manager, Ride Manager and Vehicle manager.
Finally, Map Manager and Reservation Manager, and for last Request Manager that is called only by the web browser.

To be able to proceed smoothly we need to follow this pattern during the writing of code, so completing first the components to be tested first.
In the following chapters will be clearer the integration methods and the criteria we will use

2.2 Elements to be integrated

In this chapter we list all the components that have to be integrated in the testing phase, following the architecture written in the design document.

Based on the component view of the design document, it’s possible to identify the interaction between the components of the application server, which is the main high level component of our architecture.

These components are:

- Request manager
- Ride manager
- Reserve manager
- Profile manager
- Payment manager
- Data manager
- Vehicle manager
- Map manager

We will test also the integration between the components above and the external interfaces given by outer services.

2.3 Integration testing strategy

We decided to use a bottom-up strategy to choose the order of the components to be integrated. This decision was taken as it is a simple solution to effectively apply to our system that is formed by few and small components.
Using this strategy, we will not need to use stubs but we need to use drivers for testing the components.

2.4 Sequence of component/Function integration

This section is dedicated to the detailed description of the order of integration of the components that have been previously mentioned. Using the bottom-up approach we will use the drivers to make calls from the higher levels even when these are not yet integrated.

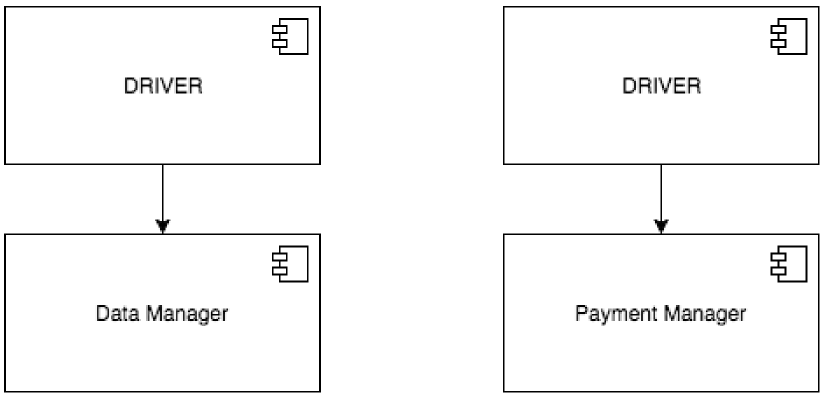
2.4.1 Software integration sequence

To decide the order of the components to be tested first, we rely on the component view of the design document, respecting the bottom-up method.
In this way we follow the possible interaction between the components as shown in the diagram quoted above.
First of all, we integrate the components that are used by multiple components, and which interact with external services.

These components are:

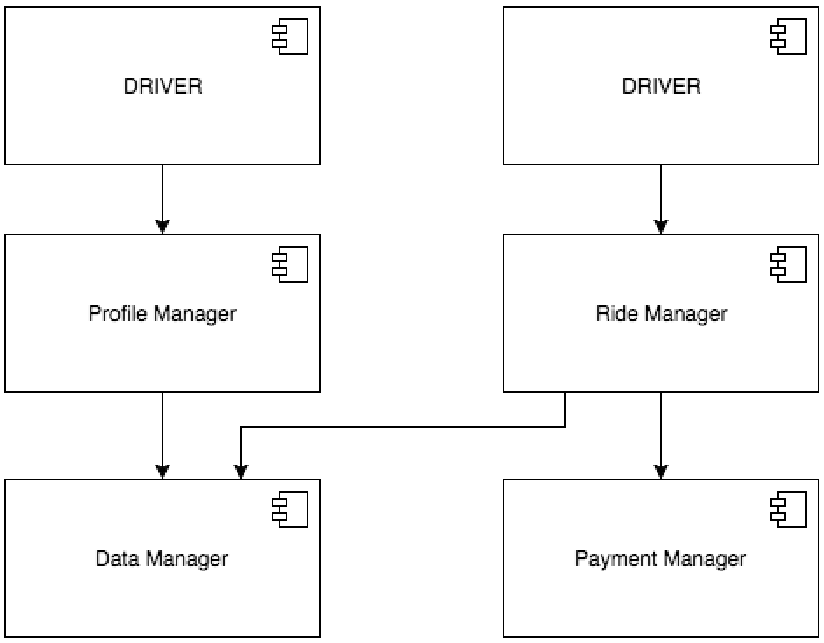
-Data manager, which permits the access to data to the other components.

-Payment manger, which manage all the operation involved with payment.



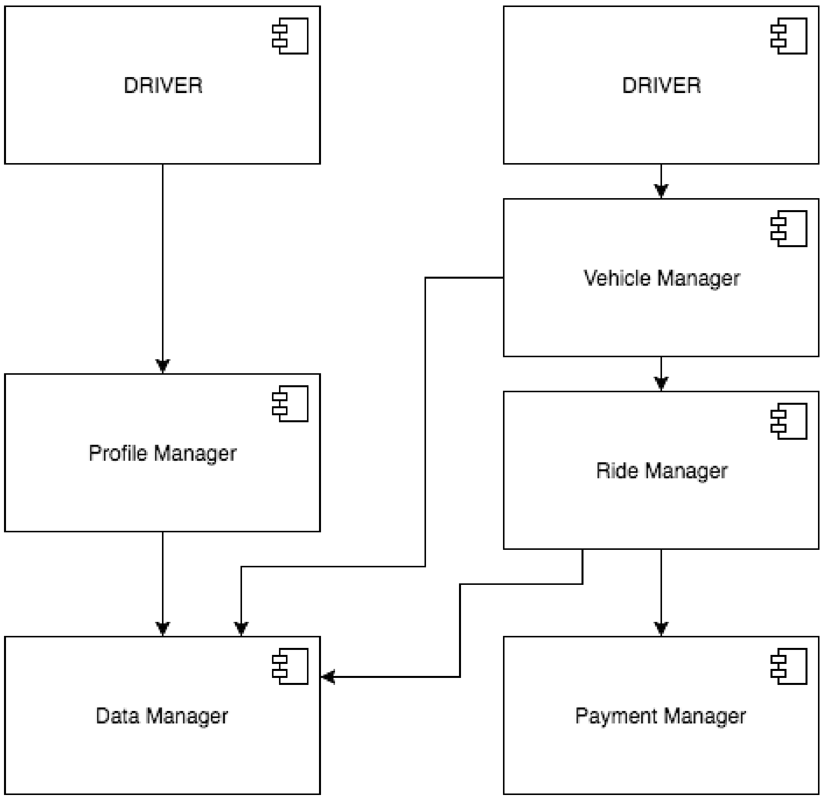
In the second step we develop other drivers that will be used to test:

-Profile manager
-Ride manager



In the next step we develop another driver to test:

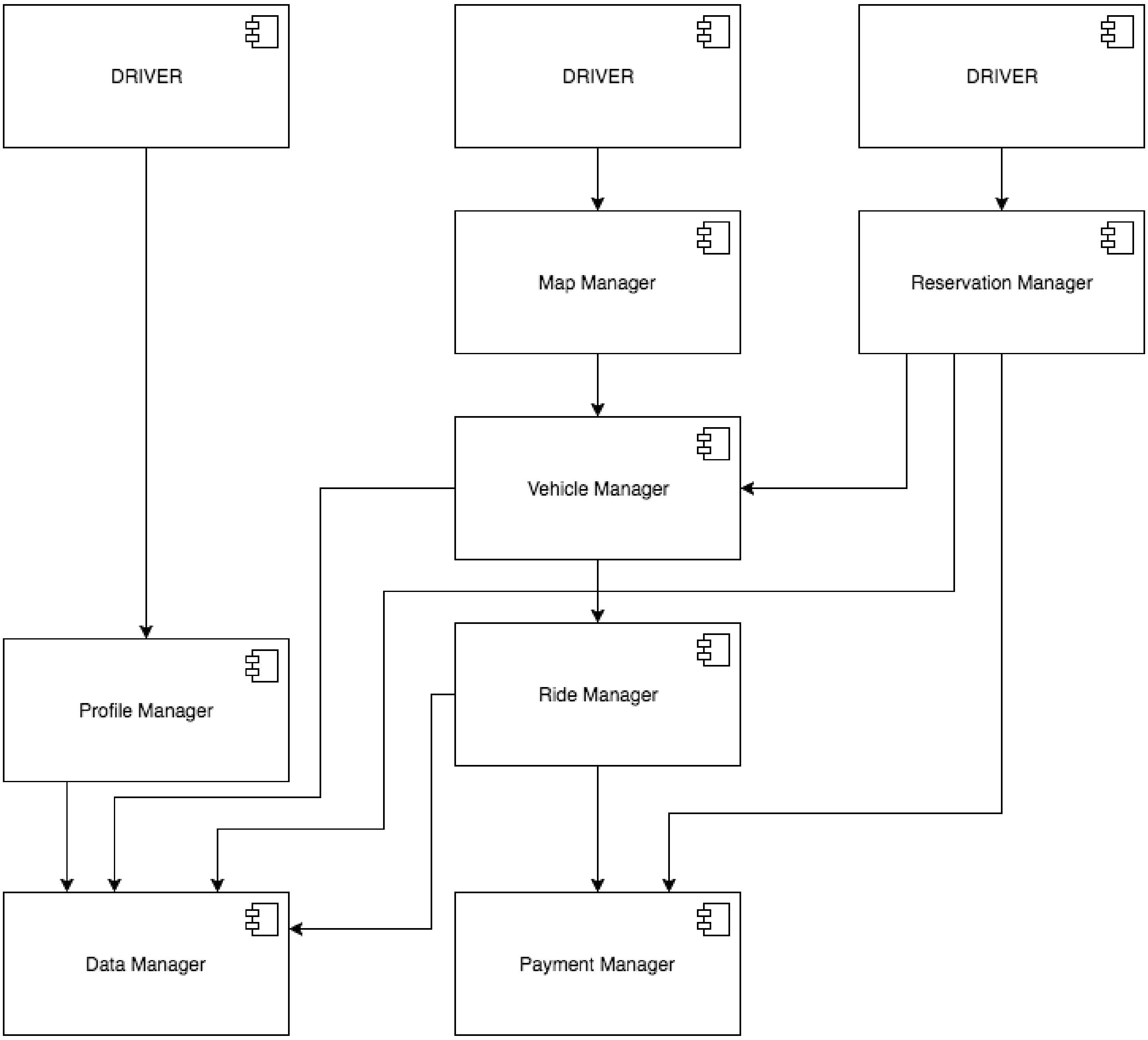
-Vehicle Manager



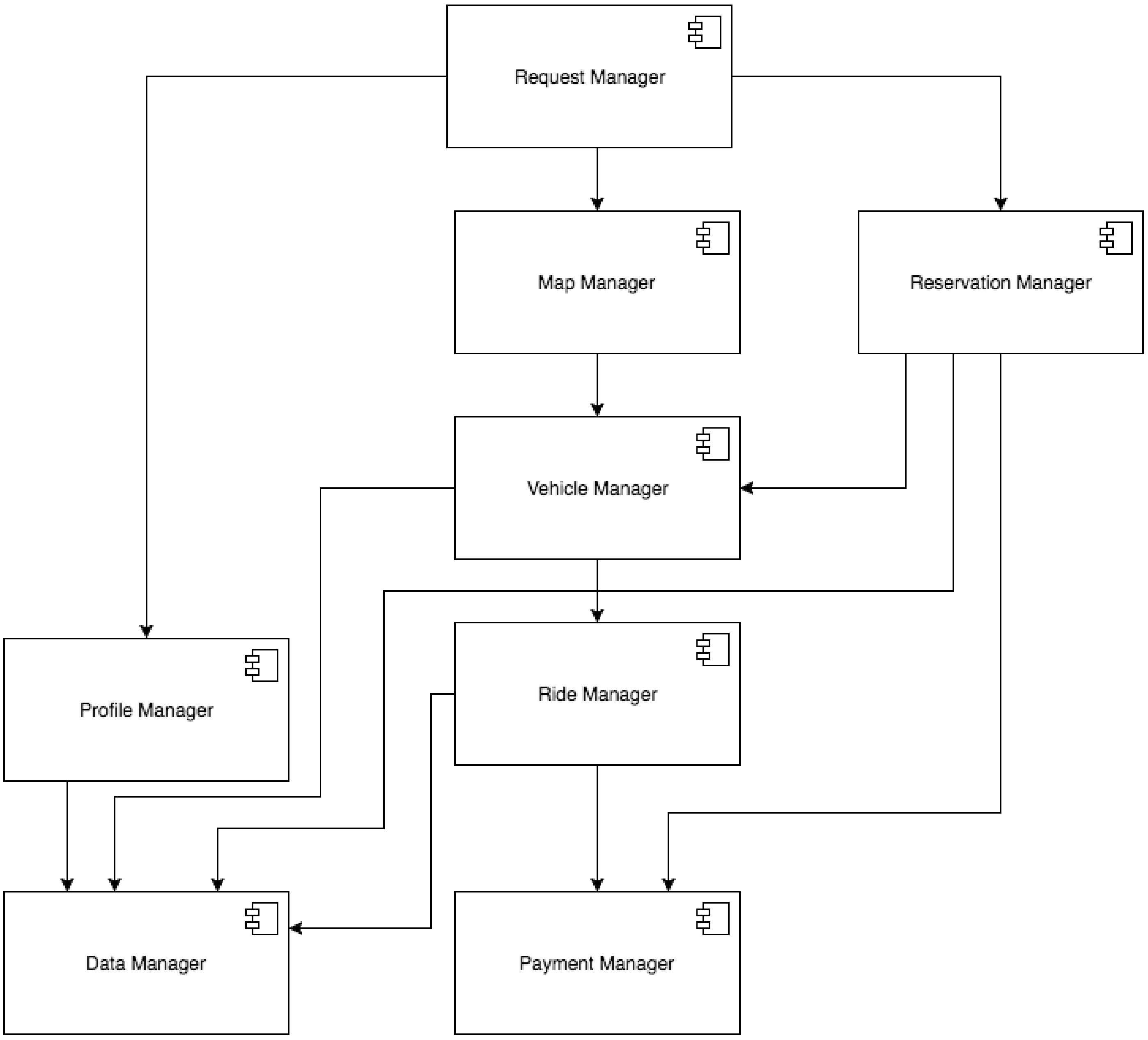
In the next step we integrate, using two new drivers,

-Map Manager

-Reservation Manager



The last one is the request manager, because it is called only by the browser of the client through an interface.
We can now delete all the drivers and we have the whole system integrated.



3. Individual Steps and Test Description

In this chapter we describe the test that must be executed on each component, so to arrive at the integration level shown in the chapter before. We list some methods involved in the interactions between components and for each of them we describe the effect for some critical inputs.

Driver -> Data Manager:

checkPassword(email,psw)	
Input	**Effect**
An email in the database with the corresponding password	Returns the object user, corresponding to this email
An email in the database with an incorrect password	An InvalidPasswordException is raised
An email not in the database	An InvalidEmailException is raised
A null parameter	An NullPointerException is raised

readVehicleInfo()	
Input	**Effect**
-	If there are vehicles in the database this method return a list with all the vehicles, if the database is empty an emptyDatabaseException is raised

newUser(form)	
Input	Effect
Null parameter	NullArgumentException is raised
Invalid data in the form	Returns false
Correct data	The system creates a password, then sends it by email to the user and the method returns true
Incomplete data in the form	Returns false

findVehicleInfo(vehicle)	
Input	Effect
Null parameter	NullArgumentException is raised
A vehicle with an id in the database	Returns the vehicle in the database
A vehicle with an id not in the database	Returns the vehicle in the database

createReservation(user,vehicle)	
Input	Effect
A user and a vehicle in the database	Creates a new reservation object in the database and returns true
A user not in the database	IllegalArguementException is raised
A vehicle not in the database	IllegalArguementException is raised
A null parameter	NullPointerException is raised

readReservation(user)	
Input	**Effect**
A user in the database	Returns the reservation associated to the user
A user not in the database	IllegalArguementException is raised
A user in the database without a reservation	IllegalArguementException is raised
A null parameter	NullPointerException is raised

newRide(ride)	
Input	**Effect**
A ride with valid attributes	Creates a new ride object in the database with the correct attributes
A ride with wrong attributes	IllegalArguementException is raised
A null parameter	NullPointerException is raised

updatePosition(vehicle,position)	
Input	**Effect**
A vehicle in the database and a valid position	Edits the vehicle position in the database
A vehicle not in the database	IllegalArguementException is raised
A null parameter	NullPointerException is raised
An invalid position	IllegalArguementException is raised

findPaymentInformation(user)	
Input	**Effect**
A user in the database	Returns an object paymentInfo associated with the user
A user not in database	IllegalArguementException is raised
A null parameter	NullPointerException is raised

findAllChargingStation()	
Input	**Effect**
-	Returns the list of the charging station in the database

updateState(state,vehicle)	
Input	**Effect**
A valid state and a vehicle in the database	Edits the vehicle state stored in the database
A null parameter	NullPointerException is raised
An invalid state	IllegalArguementException is raised
A vehicle not in database	IllegalArguementException is raised

Driver -> Payment Manager

chargeUser(accreditedMoney, paymentInfo)	
Input	**Effect**
Null parameters	NullArgumentException is raised
Negative value of accreditedMoney	IllegalArgumentException is raised
User hasn’t enough money in his payment system	The system must put the user in the blacklist and the method returns false
Wrong paymentInfo	IllegalArgumentException is raised
User has enough money and correct paymentInfo	Return true

Driver -> Profile Manager

profileAccess(email, psw)	
Input	**Effect**
Null parameters	NullArgumentException is raised
An email in the database with the corresponding password	Returns the object user, corresponding to this email
An email in the database with an incorrect password	An invalidPasswordException is raised
An email not in the database	An invalidEmailException is raised

createProfile(form)	
Input	**Effect**
Null parameters	NullArgumentException is raised
Invalid data in the form	Return false
Correct data	The system creates a password, sends it by email to the user and the method returns true
Incomplete data in the form	Return false

changeVehicleState(state, vehicle)	
Input	**Effect**
Null parameters	NullArgumentException is raised
Invalid state	IllegalArgumentException is raised
The vehicle does not exist	NullPointerException is raised
Correct parameters	The vehicle in the parameters must change the state in the database and the method returns true
The vehicle is in used	Return false

Driver -> Ride Manager

createRide(user, vehicle)	
Input	**Effect**
Null parameters	NullArgumentException is raised
Invalid user	IllegalArgumentException is raised
Invalid vehicle	IllegalArgumentException is raised
Correct parameters	The system creates a new ride in the database, the vehicle changes its state in “in use” and the method returns true
The vehicle is in a state that is not “reserved”	IllegalArgumentException is raised

endRide(vehicle)	
Input	**Effect**
Null parameters	NullArgumentException is raised
The vehicle does not exist	IllegalArgumentException is raised
The vehicle’s state is not “in use”	IllegalStateException
Correct parameter	The system changes the state of the vehicle, locks the car, updates his position in the database and returns true

calculateFee(ride)	
Input	**Effect**
Null parameter	NullArgumentException is raised
The ride does not exist	IllegalArgumentException is raised
Ride with some bonuses or ride without bonuses	The methods must calculate the correct amount of money to charge to the user

Driver -> Vehicle Manager

findAllVehicles()	
Input	**Effect**
-	Returns a list with all the vehicles in the database

unlockVehicle(vehicle,user)	
Input	**Effect**
A valid state and a vehicle in the database	Edits the vehicle state stored in the database
A null parameter	NullPointerException is raised
An invalid state	IllegalArguementException is raised
A vehicle not in database	IllegalArguementException is raised

lockCar(vehicle)	
Input	**Effect**
A valid vehicle in the database	Locks the car's doors and returns true
A null parameter	NullPointerException is raised
A vehicle not in database	IllegalArguementException is raised
A vehicle with state in use, in the database	Returns false

Driver -> Map Manager

generateMap(user)	
Input	**Effect**
Null parameters	NullArgumentException is raised
The user does not exist	IllegalArgumentException is raised
Correct parameter	The method returns a map with the vehicles

Driver -> Reservation Manager

reserveVehicle(vehicle)	
Input	**Effect**
Null parameter	NullArgumentException is raised
The vehicle does not exist	IllegalArgumentException is raised
The vehicle is not in state “available”	IllegalStateException is raised
Correct parameter	The system creates a reservation in the database and the method returns true

findReservation(user)	
Input	**Effect**
Null parameters	NullArgumentException is raised
The user does not exist	IllegalArgumentException is raised
Correct parameter	The method returns the reservation of the user

Driver -> Request Manager

loginRequest(email, psw)	
Input	**Effect**
Null parameters	NullArgumentException is raised
An email in the database with the corresponding password	Load the web page for the user
An email in the database with an incorrect password	Sends an error message to the user
An email not in the database	Sends an error message to the user

registrationRequest(form)	
Input	**Effect**
Null parameters	NullArgumentException is raised
Invalid data in the form	Sends an error message to the user
Correct data	The system creates a password, sends it by email to the user and the method sends a confirmation message to the user
Incomplete data in the form	Sends an error message to the user

reservationRequest(vehicle)	
Input	**Effect**
Null parameter	NullArgumentException is raised
The vehicle does not exist	IllegalArgumentException is raised
The vehicle is not in state “available”	IllegalStateException is raised
Correct parameter	The system creates a reservation in the database and the method sends a confirmation message to the user

showReservation(user)	
Input	**Effect**
Null parameter	NullArgumentException is raised
The user does not exist	IllegalArgumentException is raised
Correct parameter	The method returns the reservation of the user

changeStateRequest(state, vehicle)	
Input	**Effect**
Null parameters	NullArgumentException is raised
Invalid state	IllegalArgumentException is raised
The vehicle does not exist	NullPointerException is raised
Correct parameters	The vehicle in the parameters must change the state in the database and the method sends a confirmation message to the technician
The vehicle is in used	The methods sends an error message to the technician

4. Tools and Test Equipment Required

We decide to use the following tools for executing the integration testing phase:

Mockito: is an open source testing framework for java.
We use it mainly because it provides a framework for interaction test.
Junit: is a unit testing framework for java. We have used it for the unit tests, and we will use it also for the integration test. It is useful to perform state testing, it asserts properties on an object.
Arquillian: is a test framework that can be used to perform testing inside a remote or embedded container, or deploy an archive to a container so the test can interact as a remote client. Arquillian integrates with other testing frameworks like Junit and mockito.

These testing tools together allows to test all the functional requirements.
For the non functional requirements we will use Apache JMeter, it tests the performance on static and dynamic environments of the system. We will use it especially in the system test, so in the next testing phase.

5. Program stubs and Test Data Required

5.1 Program stubs

Because of the use of the bottom-up method for the integration test, we don't have the need of stubs but only of driver that simulates the calls to the methods of the components under testing. For each driver we specify which methods it calls and in which component.
In this chapter we describe the drivers used, which component they substitute and in which components they will call the methods.

-Profile manager driver: this driver substitutes the profile manager component which calls methods on data manager component [updateState(state, vehicle), checkPassword(email, psw), newUser(form)].

-Ride manager driver: this driver substitutes the ride manager component which calls method on data manager [newRide(user, vehicle), findPaymentInformation(user), findChargingStation(), findVehicleInformation(vehicle)] and payment manager [chargeUser(accreditedMoney, payInf)] components.

-Vehicle manager driver: this driver substitutes the vehicle manager component which calls method on ride manager [createRide(user, vehicle)] and data manager [updatePosition(vehicle, position), readVehiclesInfo()] components.

-Map manager driver: this driver substitutes the map manager component which calls method on vehicle manager component [findAllVehicles()].

-Reservation manager driver: this driver substitutes the reservation manager component which calls methods on vehicle manager [unlockVehicle(vehicle, user)], data manager [readReservation(user), findVehicleInfo(vehicle), createReserv(user, vehicle)] and payment manager components.

-Request manager driver: this driver substitutes the request manager component which calls methods on profile manager [changeVehicleState(state, vehicle), profileAccess(email, psw), createProfile(form)] , map manager [generateMap(user)], reservation manager [findReservation(user), reserveVehicle(vehicle)] component.

We decided also to implement some stubs to simulate the behaviour of the external components such payment system, vehicle system and map service. In this way we can test, for example, the charge of the user or the information retrieved by the vehicles.

5.2 Test data required

To execute correctly the integration test, we need some data in the database to make possible the calls, by the drivers, to the methods with the input specified in the chapter 3.
So we need:

- a list of vehicles with the respective information
- users and technicians to simulates the behaviour of the client through a driver
- rides since we can’t create a ride through the system because it is created when a vehicle is ignited
- charging stations to calculate the correct fee
- safe area to check if the vehicles are in the right position at the end of the rides

6. Effort Spent

These are approximatively the time we spent to write this document

Luca Santini: 25 hours

Riccardo Remigio: 25 hours