POLITECNICO
MILANO 1863

Software engineering 2 Project

Integration Test Plan Document

PowerEnJoy

Perugini Alex	876359
Re Marco	873564
Scotti Vincenzo	875505

Table of Contents

1 Introduction

1.1 Purpose

This document will describe the testing plans and procedures for the PowerEnJoy application. All the tests described in the document will be based on the architecture description provided in the Design Document. This document will also be a guideline for the developers that will face the testing phase of the new software.

1.2 Scope

The scope of this document is to present the test planning activities and overall test strategy to be followed in order to ensure that the PowerEnJoy application works as expected and as requested by the company.

1.3 Glossary

1.3.1 Glossary

Charging Area/Station	Subset of safe area where is possible to plug the car to a power grid to charge its battery
Client	User of PowerEnJoy registered to the system with normal log in elements
Driver	User sitting in the car that has the control of the vehicle, can be both a client or an operator
Driving Licence	Category B licence needed to drive the cars provided by PowerEnJoy
Money Saving Option	Functionality provided through the navigation system that enlights the nearest charging area to the client destination with free plugs
Operator	Employee of the company that provides the PowerEnJoy service, he is in charge of managing the cars according to the instructions he receives from the system
Passenger	Person inside the car that is not on the driver seat
Payment Method	The payment methods are all the methods accepted by the external payment system that the system will interface with
Power Grid	Grid located at each power station, it has plugs to charge the PowerEnJoy cars

Punishment	Fee on the charge for the ride
Ride	Entire operation of driving the car from the moment the engine is ignited to the moment the car is left parked in a safe area
Reward	Discount on the charge for the ride
Safe Area	Area defined by a set of GPS positions in which is possible to park the PowerEnJoy cars
Special login elements	Email and password gave to operators to authenticate as employes during the worktime

1.3.2 Acronyms

GPS	Global Positioning System
RASD	Requirement Analysis and Specification Document
DD	Design Document
ITPD	Integration Test Plan Document
DB	Data Base
DBMS	Data Base Management System
RDBMS	Relational Data Base Management System
UX	User Experience
BCE	Boundary Control Entity
ER	Entity Relationship
JEE	Java Enterprise Edition

1.3.3 Abbreviations

App	Application, in this case refers to the software installed on smartphones

1.4 Reference Documents

1. Project description: Assignments AA 2016-2017.pdf
2. DDPowerEnJoy.pdf
3. RASDPowerEnJoy.pdf
4. Example documents:
    - Integration Test Plan Example.pdf
    - Integration testing example document.pdf

1.5 Document Structure

The document is organized as follows:

- Section 1: Introduction, provides a general description for the features regarding the testing phase of the PowerEnJoy application, in order to realize a completely working version of the software presented in the design document .
- Section 2: Integration strategy, provides the details for the whole testing procedure, including also the dependencies and sequences between components/functions integration.
- Section 3: Individual steps and test description, will present both the type of test that will be used and the expected results.
- Section 4: Tools and test equipment required, describes tools and test equipment needed to accomplish the integration, and how they are going to be used.
- Section 5: Program stubs and test data required, presents the stubs and the special test data for the derived test strategy.
- Section 6: Used tools, contains the list of the tools used to realize the DD and the scope they were used for.
- Section 7: Hours of work, report of work time for each member.

2. Integration strategy

2.1 Entry criteria

The whole integration test procedure will be done following the bottom up criteria, the integration will be possible only when a certain threshold of the platform will be available and when each component will have been correctly tested through unit testing.
In detail the subsystems identified by component diagram will represent the main division for the platform development, so for the final steps of the integration at least the 90% of each high level component need to be completed.
Following this idea, while going deeper into the components division, testing phase should start only when a certain percentage of each main component has been developed, in particular 90% of the main server components, 70% of the smartphone application and 80% of the car application.
Different completion percentages have been proposed to take into account time required to perform integration testing and also to manage any identified problem as soon as possible.

2.2 Elements to be integrated

As specified in the Design Document, the system we are developing is composed of three different levels of abstraction.
Starting from lowest one, components should be integrated on the basis of higher level functionality they provide. In particular we start with the **Server Application** component, the first components we are taking into account are **Users Information Manager**, **Reservation Manager** and **Car Manager**. These three components don't need integration among themselves because they provide different functionalities, but they must be integrated with the **Database Manager** to obtain the whole **Model** component integrated with the **Database Manager**. For the same reasons the three components mentioned before must be integrated also with the **Controller** component.
Similarly to the components of the model, the components of the **View** which are **Notification Manager**, **User View** and **Car view** don't need integration among themselves but must be integrated with the **Controller** component. These three components have also to interface with the external world, in particular we need to test the communication and functionalities when **Notifications Manager** and **User view** interact with the **App** and when **Car view** interacts with the **Car**.
Another interaction that needs to be tested is the one between the **Database Manager** and the **DBMS** and those between **Model** and **Motorization Gateway** and **Payment Gateway**.
Then we focus on the **Car**, in which the **Car Application** component is the most critical one because it must be integrated with **Display**, **Satellite Navigation** and **Sensors** and it also has to communicate with the external world, in particular with the **Server Application** component.
In the end we have to test the **App** integration, which is quite simple because the only tests needed are those between the **User App** on the smartphone side and **Notifications Manager** and **User View** on the server side.

2.3 Integration testing strategy

As stated at the beginning of this section the approach for the testing strategy will be Bottom-Up. The main reasons for this choice are, first of all, the fact that such approach permits to start testing phase even before components are not completely developed yet, but also the possibility to perform tests on the real platform and not on simple stubs.
Another reason is that we can start integrating together those components that do not depend on other components functions.
Once the procedure reaches the highest level in the component representation, the proper functioning of the main functionalities will be tested through a thread approach. This additional testing method allows to focus on the user, on the functionalities that he will be provided and on modules of different components that provide that functionalities in a concurrent way.
For what concerns DBMS, Satellite Navigation, Smartphone OS no tests will be required since they are third-party software and will be supposed to work properly for the use we are going to do.

2.4 Sequence of component/function integration

As described by the integration test strategy the procedure will deal first with Bottom Up testing and then with a final thread procedure. This section describes the chosen sequence of integration, according to the dependencies among components.

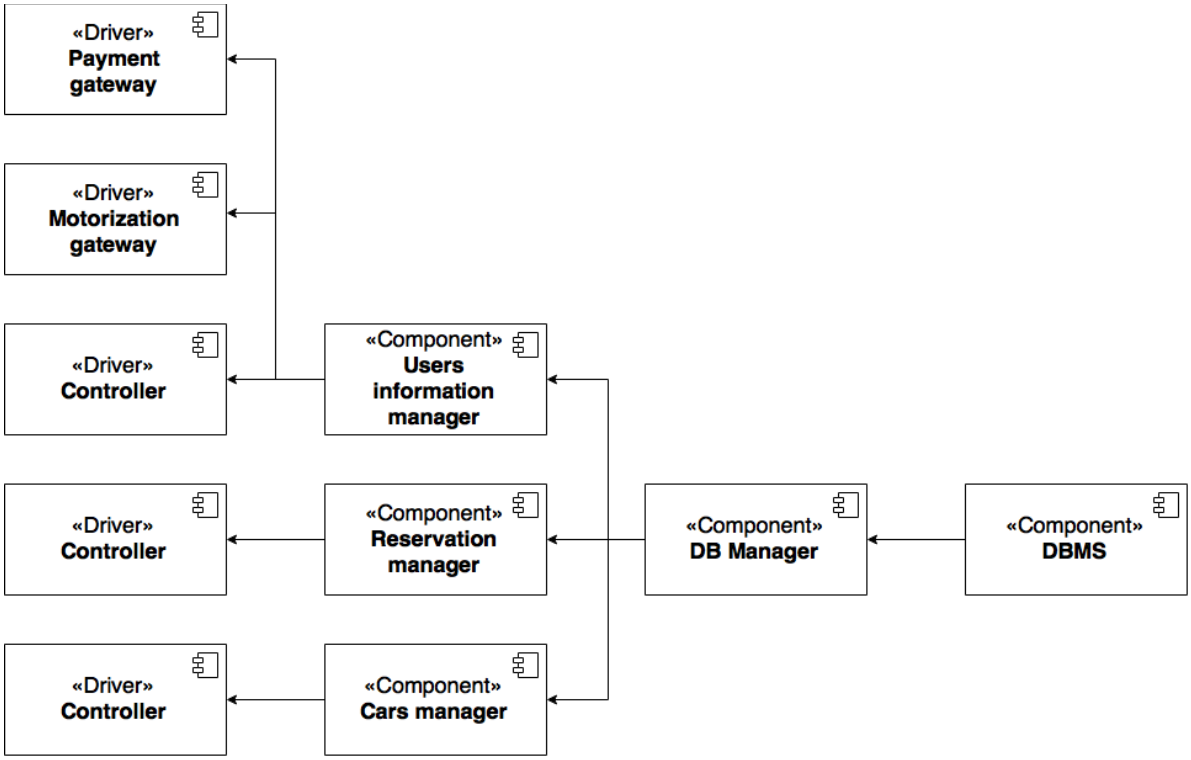
2.4.1 Bottom Up integration sequence

2.4.1.1 Database Manager

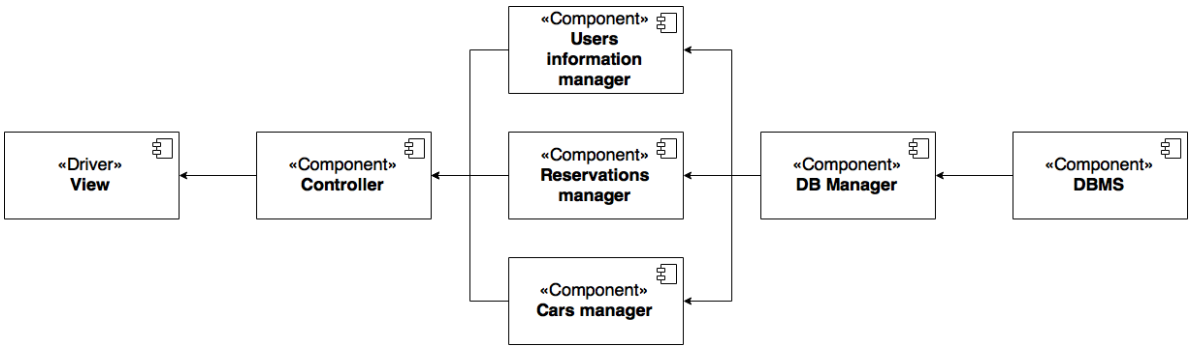
The first components that need to be integrated are **Database Manager** and **DBMS**, in order to provide access to data to the whole system. Data are fundamental to build up all the system functionalities. For this first integration a driver for the **Model** component is needed because the **Database Manager** will exchange information with it.

2.4.1.2 MVC architecture

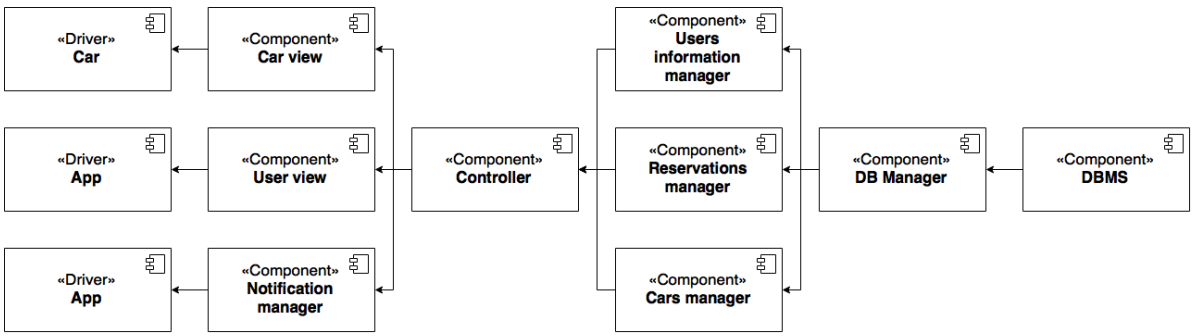
Once the **Database Manager** testing is finished is possible to go on with the MVC architecture starting from the **Model** component, each component that is part of it has to be integrated as stated above and also need to be connected to a driver for the **Controller**. Furthermore the **User Information Manager** component needs also two more drivers for the **Motorization** and **Payment** gateways that it will interface with.



The testing can now continue with **Controller** and **View**.
The first is the **Controller**, it simply manages interactions between **Model** and **View** and needs a driver only for the **View** since the **Model** has just been tested.



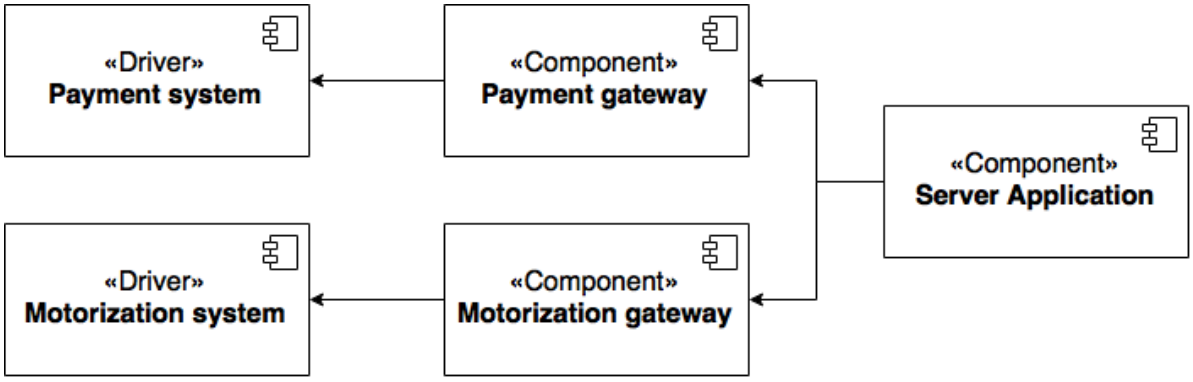
Lastly the **View** is tested, it is built up of **Notifications Manager**, **User View** and **Car View**, each one connected to the **Controller**. The first two mentioned components need a driver for the **App** because they will have to interact with it while the last one needs a driver for the **Car** component.



At the end of this section the testing of the **Server Application** component is finished and integrated with the **DBMS**. The sequence will now proceed with the components that are outside the server application and that will be then integrated with the server application itself.

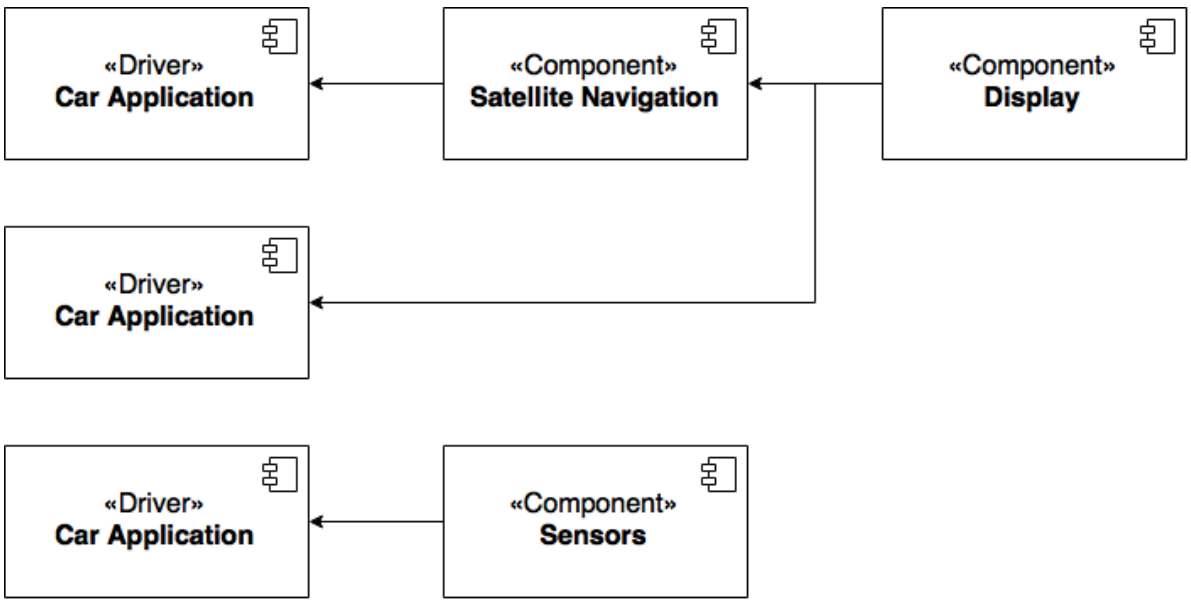
2.4.1.3 Gateways

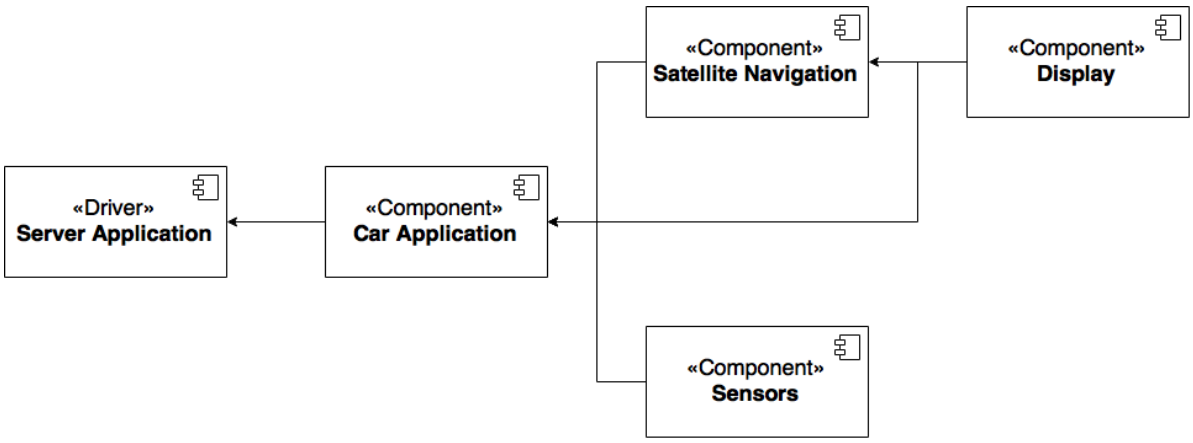
Now that the **Server Application** component is finished it's possible to test the **Motorization Gateway** and the **Payment Gateway**. They communicate with the **Server Application** and with external system so a driver for each external system is needed.



2.4.1.4 Car

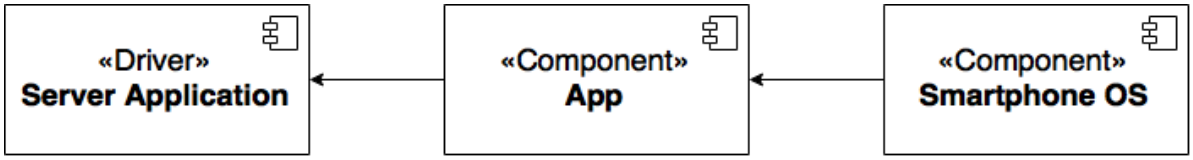
In the **Car** component the **Satellite Navigation** must be integrated with the **Display** at first, to do this integration a driver for the **Car Application** is needed. Then the **Car Application** has to be integrated with all the other components because it has to manage them and communicate with them. So the **Car Application** needs that **Sensors**, **Satellite Navigation** and **Display** are developed in order to provide its functionalities. Another integration to be tested is the one with the **Server Application** which is necessary for the **Car Application** to work properly.





2.4.1.5 App

Last we consider the **App** which has the **User App** component that must be integrated with the third-party **Smartphone OS** component and with the **Server Application** which, as in the case of the **Car Application**, is necessary to provide the requested functionalities.



2.4.1.6 Conclusion

Now the whole system is tested and integrated, the interaction between the components described in the DD is working properly in order to provide all the functionalities requested by the PowerEnJoy company and described in the RASD.

2.4.2 Thread integration sequence

At this point the components of the system have been correctly integrated with a bottom-up procedure described before, so in order to complete the test integration plan and check more precisely the behaviour of the system in different situations, we decided to add a thread integration for some important functionalities of the system. These functionalities involve a large number of component (as shown in the DD), for this reason they have been chosen for this integration testing.

2.4.2.1 Reservation

Tests for reservation will be useful to check the interaction between **App**, **Server Application** and **DBMS**. In fact all the interfaces are going to be used since the **App** is going to rely on the **Server Application** to find the available cars and to reserve them, the **Server Application** at the same time is going to use the **DBMS** for both operations, to check informations about the user and the cars. In particular **User App**, **Smartphone OS**, **User View**, **Controller**, **Reservation Manager**, **Cars Manager**, **DB Manager** and **DBMS** are the components used for this specific functionality.

2.4.2.2 Ride

The ride involves many components, testing these functionality will be useful to check the interaction between **Car**, **Server Application** and **DBMS**. At first only some components of the car (**Sensors**, **Car Application**, **Display**) are involved in order to interact with the user, then the **Car Application** needs to communicate with the **Server Application**. **Car View**, **Controller**, **Cars Manager**, **User Information Manager**, **Payment Gateway**, **DB Manager** and **DBMS** are the components used server side to accomplish this functionality.

2.4.2.3 Notification

In the same way of the ride tests, the tests for the notification system are going to use all the interfaces. First the **Server Application** will use both **Car application** and **DBMS** to identify the situation that requires the intervention of an operator, at this point the **Server Application** will identify the right operator through the **DBMS** and will send him the notification that will appear in the **App**.
During this process **Cars Manager**, **Controller**, **User Information Manager**, **DB Manager**, **DBMS**, **Notification Manager**, **Smartphone OS** and **User App** are involved to achieve the functionality.

2.4.2.4 Conclusion

This second integration testing sequence allows us to ensure that the system is correctly integrated, its main functionalities will work properly and that the most dangerous bugs were found during these procedures. It also allows to focus on the visibility of the functionality for the user and focusing on the user experience will probably led to release a better product.

3. Individual steps and test description

3.1 Bottom up integration tests

3.1.1 DB Manager

Test	
Input	**Expected Output**
New null entries (car, user, reservation...)	Null argument exception
New already existing entries (car, user, reservation...)	Already existing value exception
New entries with invalid parameters (car, user, ride…)	Illegal argument exception
New entries with formally valid parameters (car, user, ride...)	A new tuple with the desired values in the table
Delete single entry that doesn’t exists	The entry is not deleted from the DB
Delete multiple entries and some of the specified doesn’t exist	Invalid argument exception (with the list of wrong arguments) and the valid entries are deleted from the DB
Delete single entry with formally valid parameters	The entry is deleted from the DB
Delete multiple entries with formally valid parameters	All entries are deleted from the DB
Delete entry in use	Failed task exception
Update entry with invalid parameters	Invalid argument exception and no update
Update entry with null parameters	Null argument exception and no update
Update entry in use	Failed task exception
Update entry with formally valid parameters	The entry is correctly updated
Requirements	- The complete DB table developed and filled with sample data for testing - A machine running the RDBMS containing the DB table connected to internet - A machine running the server application the DB Manager - The driver for the model to test the DB Manager functionalities

3.1.2 Model

3.1.2.1 Users information manager

Test	
Input	**Expected Output**
New account with invalid values	Invalid argument exception (with invalid values) and no creation of account
Update account with invalid values	Invalid argument exception (with invalid values) and no update
Log in with wrong values	Invalid credential exception
Log in with null values	Null argument exception
Log in operator from invalid phone	Illegal access exception
Logout	Deletes session cookie
New account with valid values	An account is created with the input values
Update account with valid values	The values of the selected account are updated with the new one
Log in with valid values	Returns a session cookie
Find position when GPS is enabled	The actual coordinates registered in the DB
Find position when GPS is not enabled	Position not found exception
Find rides of a user	Returns a *List<Ride>* containing all the rides done by the specific user
Find reservations of a user	Returns a *List<Reservation>* containing all the reservations done by the specific user
Find payments of a user	Returns a *List<Payment>* containing all the payments done by the specific user
Requirements	- All the tests for the DB Manager completed - Same requirements from the DB Manager plus the Account management running in the server application - The drivers for the Server Controller, the Payment Gateway and the Motorization Gateway to test the Account management

3.1.2.2 Reservations manager

Test	
Input	**Expected Output**
Create reservation with one already active	Illegal state exception (according to RASD)
Create reservation without one already active	Reservation is created with all fields correctly filled and added to the list
Create or modify reservation with wrong car code	Invalid car code exception
Create or modify reservation with null arguments	Null pointer exception
Set reservation finished	The field “expired” of the specific object of type Reservation has to be set to **true**
Delete reservation not expired	Illegal state exception (according to RASD)
Delete reservation expired	Requested reservation is removed from the list
Requirements	- All the tests for the DB Manager completed - Same requirements from the DB Manager plus the Reservation manager running in the server application - The driver for the controller to test the Reservation manager

3.1.2.3 Cars manager

Test	
Input	**Expected Output**
Change car state with a different state, reachable from the starting one	New state of the car has to be equal to the passed one
Change car state with a different state, unreachable from the starting one	Illegal argument exception, state of the car has to remain the same as before the invocation of the method
Change car state with a null	Null pointer exception, state of the car has to remain the same as before the invocation of the method
Change car state with the same state	State of the car has to remain the same
Unlock request with correct code	The signal to open the car has to be sent

Unlock request with wrong code	Illegal argument exception with details about the error
Unlock request without reservation	Illegal state exception (according to RASD car can't be open if not reserved by the client)
Unlock request with correct position	The signal to open the car has to be sent
Unlock request with wrong position	Illegal state exception (according to RASD car can't be open if client is too far from it)
Unlock request by operator	The signal to open the car has to be sent (because for the operator car unlocking is granted)
Find reservation of a car	Returns a *Reservation* active on the specified car, if it doesn't exist a Car Is Not Reserved Exception is thrown
Requirements	- All the tests for the DB Manager completed - Same requirements from the DB Manager plus the Cars manager running in the server application - The driver for the Controller to test the Car manager

3.1.3 Controller

Test	
Input	**Expected Output**
Receive a message from view component	Forwards the information calling the right function in the model component
Receive an update from model component	Notifies the view component about the new state of the model
Requirements	- All the tests for the Server Model completed - Same requirements from the Server Model plus the Controller running in the server application - The driver for the Server View to test the Controller

3.1.4 View

3.1.4.1 User view

Test	
Input	**Expected Output**
Update from controller	Forwards to correct user application
Request from user application	Forwards to controller
Requirements	- All the tests for the Server Controller completed - Same requirements from the Controller plus the User view running in the server application - The driver for the App to test the User view

3.1.4.2 Car view

Test	
Input	**Expected Output**
Update from controller	Forwards to correct car
Request from car	Forwards to controller
Requirements	- All the tests for the Server Controller completed - Same requirements from the Controller plus the Car view running in the server application - The driver for the Car to test the Car view

3.1.4.4 Notification manager

Test	
Input	**Expected Output**
New pending notification	Forwards to correct user application
Solved notification	Forwards to controller
Requirements	- All the tests for the Server Controller completed - Same requirements from the Controller plus the Notification

	manager running in the server application - The driver for the App to test the Notification manager

3.1.5 Gateways

3.1.5.1 Motorization gateway

Test	
Input	**Expected Output**
Check null driving license	Null pointer exception
Check correct driving license	Confirmation message
Check wrong driving license	Error message
Requirements	- All the tests for the Server Application completed - A machine running the DBMS and one running the Server Application, plus the Motorization gateway running in the server machine - The driver for the Motorization System to test the Motorization gateway

3.1.5.2 Payment gateway

Test	
Input	**Expected Output**
Check null payment method	Null pointer exception
Check correct payment method	Confirmation message
Check wrong payment method	Error message
Requirements	- All the tests for the Server Application completed - A machine running the DBMS and one running the Server Application, plus the Payment gateway running in the server machine - The driver for the Payment System to test the Payment gateway

3.1.6 Satellite navigation

Test	
Input	**Expected Output**
Search valid location	Shows map near inserted location
Search invalid location	Returns error message
Set destination	Shows the calculated path to reach the destination
Requirements	- At least one car for the car sharing with the display installed - The Satellite Navigation installed in the Car computer - The connection for the satellite navigation - The driver for the Car application to test the Satellite navigation

3.1.7 Sensors

Test	
Input	**Expected Output**
“Driver is sitting in the car” request	Returns the correct answer
User opens the door	Sends the information received from outside
User closes the door	Sends the information received from outside
Check doors closed	Returns the correct answer
Check nobody inside	Returns the correct answer
User ignites engine	Sends the information received from outside
User turns off engine	Sends the information received from outside
Request of GPS position	Returns the correct answer
Requirements	- At least one car for the car sharing - The Sensors installed in the Car - The driver for the Car application to test the Sensors

3.1.8 Car application

Test	
Input	**Expected Output**
Unlock	Sends unlock request to the server application
Engine ignited	Starts charging the client and shows the charge on the display
Engine turned off	Stops charging the client
Save money request	Ask the client for a destination address
Correct destination address	Nearest stations to the given address
Wrong destination address	Invalid address exception
Requirements	- At least one car for the car sharing with the display installed - The Car application installed in the Car computer - The connection for the satellite navigation - The sensors installed - The tests for the Satellite Navigation and the Sensors completed - The driver for the Server application to test the Satellite navigation

3.1.9 App

Test	
Input	**Expected Output**
Registration with invalid values	Illegal argument exception
Registration with valid values	Operation success
Change informations with invalid values	Illegal argument exception
Change informations with valid values on Operator account	Invalid operation exception
Change information with valid values on Client account	Operation success
Log in with wrong credentials	Invalid credentials exception
Log in of Operator with right credentials from a wrong phone	Invalid operation exception

Log in with right credentials	Operation success
Search car with GPS/ by position too far from the cars area	No car found
Search car with GPS/ by position in the cars area	List<Car> sorted from the nearest to the farthest
Require movements	List<Movement> corresponding to the requesting account
Receive notification for other Operator	-
Receive notification for right Operator	Display notification alert
Reserve car	Operation result
Open car with GPS too far from the car	Invalid position exception
Open car with wrong windscreen code	Illegal argument exception
Open car with GPS next to the car	Operation success
Open car with right windscreen code	Operation success
Requirements	- At least one smartphone for each target Smartphone OS - The App installed in the smartphone - The driver for the Server application to test the App

3.2 Thread integration tests

3.2.1 App - Server Application - DBMS

Reservation	
Input	**Expected Output**
Car reservation with GPS	New reservation for the car is created
Double reservation attempt	First reservation is created, the second attempt produces an illegal action exception
Reservation expiration	Client is notified and charged, while the reservation is deleted
Car reservation with address	New reservation for the car is created
Requirements	- All the tests from the bottom up procedure previously described completed and successfully passed - At least one car with all the software and the sensors installed - A machine running the entire Server application - A machine running the DBMS - At least one smartphone with the App installed - All the components connected as described by the deployment diagram in the DD

3.2.2 Car - Server Application - DBMS

Ride	
Input	**Expected Output**
Start ride procedure	The client starts being charged and the car state is set to “in use”
Activate safe money option	The safe money option results correctly activated
Conclude ride procedure	The car is set available again and the payment procedure is correctly executed
Requirements	- All the tests from the bottom up procedure previously described completed and successfully passed - At least one car with all the software and the sensors installed

	- A machine running the entire Server application - A machine running the DBMS - At least one smartphone with the App installed - All the components connected as described by the deployment diagram in the DD

3.2.3 Car - App - Server Application - DBMS

Notification	
Input	**Expected Output**
Car left with low battery level	Nearest Operator receives notification and car state is set to maintenance
Car needs to be relocated	Nearest Operator receives notification and car state is set to maintenance
Requirements	- All the tests from the bottom up procedure previously described completed and successfully passed - At least one car with all the software and the sensors installed - A machine running the entire Server application - A machine running the DBMS - At least one smartphone with the App installed - All the components connected as described by the deployment diagram in the DD

4. Tools and test equipment required

4.1 Tools

After presenting the integration testing procedure is now necessary to discuss about which testing framework are advisable for the system presented since this point.
For server side testing we have chosen **JUnit** and **Arquillian**, the first one mostly for unit testing and the second one for integration testing.
JUnit is the most used java testing framework due to its simplicity and flexibility so it's perfect to execute unit testing on the system. Even though it is mainly used for unit testing it can also be used for some minor integration testing tasks (verify correct types, null arguments and other kind of exceptions).
Arquillian offers more functionalities devoted to integration testing, in fact it will be used to check connections and interactions between the components of the system. Arquillian is similar to JUnit, in fact one of its tests looks like a JUnit test case with some additional functions. We have selected these two frameworks also for this reason, in fact, being similar, they allow to execute all the testing phase in an easier way.
Instead for client side testing we have chosen **Appium**.
Appium is a well known testing framework for mobile devices, it is used for its flexibility in fact it works on both iOS and Android application. This feature is the most interesting one because allows to test the app in two completely different environments with only one tool, simplifying again the whole testing procedure.
For what concerns Windows Phone we have decided to use the tools included in Visual Studio since it has Windows Phone Emulator, a complete platform for developing and testing realized by Microsoft, who has realized the Windows Phone system.
Another part of the client is the car application, in this case we prefer to let the developers choose what they consider the best framework for their application since they also have chosen the technology to use to develop it.

4.2 Test equipment

In this section all the equipment needed for the testing phase will be listed.
For what concern the mobile app the following devices are required:

- At least one iOS device for each of the last three iOS releases: iOS 8, iOS 9 and iOS 10.
- At least one device for each of the last three Android releases: Android 5 Marshmallow, Android 6 Lollipop, Android 7 Nougat.
- At least one device for each of the three Windows Phone versions: Windows Phone 7 and Windows Phone 8.1 and Windows Phone 10.

These devices will be used to test the application and see how it runs in different environments and on different devices[1].
For what concern the car application we leave the decisions to the developers as done for other choices before.

1In this case we selected for each mobile operating system the last three releases but is more advisable to make a market research in order to know which are the currently still used operating systems of each company, after this analysis it could be possible for example to exclude Windows Phone 7 and include Android 5 KitKat.

Finally for the server side the most advisable thing to do is to use a cloud computing platform (for example Microsoft Azure) in which load the system and test the functionalities. The advantages of this choice are more than one. First in this way is possible to verify also the real necessary computing power needed by the system, without buying anything that can't be changed at a later time, in fact with a cloud computing platform it is possible to change configurations and functionalities that can't be changed after the purchase of a real physical server. Developers can test the whole system emulating the real one, that will be bought only if in the cloud testing platform everything is working correctly. Only after a period of successful testing on the cloud computing platform the system will be moved into a server. Since the system was working on the computing platform the software configuration of the server will be done simply replying the same environment tested on the cloud. In this way the server is going to host same Web Server, DBMS and Application Server as the cloud infrastructure used to test.

5. Program stubs and test data required

5.1 Program stubs and drivers

As stated before in this same document, the testing procedure will begin following a Bottom Up approach for the first part of the integration testing, so it will be required to have drivers for the various components at each level of the procedure.
In addition, for the Thread part of the integration we will need some stubs due to the need to emulate the presence of users and of cars. Such an integration test can't be performed in fact without their use. They do not have to implement all the specific functionalities but just few ones. In fact stub of user only have to emulate the invocation of some functionalities of the app and both stubs have to write on a log all messages received.
All the required drivers, that are listed below, will be used by both **JUnit** and **Arquillian**, even though the use of drivers is mostly required by the tests handled by **Arquillian**.

Driver	**Emulated component(s)**	**Tested subsystem(s)**
Model Driver	● Car manager ● User information manager ● Reservation manager	● DB Manager
Controller Driver	● Controller	● Car manager ● User information manager ● Reservation manager
Payment Gateway Driver	● Payment gateway	● User information manager
Motorization Gateway Driver	● Motorization gateway	● User information manager
View Driver	● Car view ● User view ● Notification manager	● Controller
App Driver	● App	● User view ● Notification manager
Car Driver	● Car	● Car view
Server Driver	● Server Application	● Car application ● App
Car Application Driver	● Car application	● Satellite navigation ● Sensors
Payment System Driver	● Payment system	● Payment gateway

Motorization System Driver	• Motorization system	• Motorization gateway

Here follows the required Stubs to conclude the second part of the integration testing phase and the respective emulated components.

Stub	**Emulated component(s)**
User Stub	• Client • Operator
Car Stub	• Car

5.2 Test data

In this section are listed and described all the test data needed to perform the test procedures specified before.

Component	**Test data**
Database Manager	• Null entries • At least one valid tuple for each table • At least one invalid argument for each table
User Information Manager	• Null objects • New and valid account values • New and invalid account values • Valid login values • Invalid login values
Reservation Manager	• Null objects • Code of the car to be reserved (valid) • Code of the car to be reserved (invalid)
Cars Manager	• Null objects • Code of the car to be modified (invalid) • Code of the car to be modified (valid)
Payment Gateway	• Null objects • Valid payment method • Invalid payment method
Motorization Gateway	• Null objects • Valid driving licence • Invalid driving license
Car Application	• Signals from sensors • Valid destination address • Invalid destination address
App	• Valid login values • Invalid login values

	• Valid registration values • Invalid registration values

6. Used tools

- Google Drive: documents sharing
- Google Docs: word processor, concurrent work platform
- GitHub: control version

7. Hours of work

Perugini Alex:

- 28/12/16 2h 30m
- 02/01/17 2h 30m
- 03/01/17 2h
- 04/01/17 1h 30m
- 05/01/17 1h 30m
- 06/01/17 3h

Re Marco:

- 28/12/16 5h 30m
- 30/12/16 1h
- 03/01/17 2h
- 04/01/17 1h
- 05/01/17 1h 30m

Scotti Vincenzo:

- 21/12/16 1h
- 28/12/16 5h
- 30/12/16 1h
- 03/01/17 2h
- 04/01/17 1h
- 05/01/17 30m