Politecnico di Milano A.A 2016/2017

Software Engineering 2 project:

PowerEnJoy

Integration Test Plan Document

(ITPD)

Alessandro Perini, Federico Saini, Ali Merd Türkçapar

Version: 1.0 Release date: 15/01/2017

Contents

Chapter 1
Introduction

1.1 Revision History

Version	Date	Description
1.0	15/01/2017	Document release. First version

1.2 Purpose and Scope

This document is the Integration Test Plan Document (ITPD) of the PowerEnJoy project.
The scope of the ITPD is to plan and describe how to accomplish the testing of component groups. It explains which components must be tested, in which order and which tools must be used. The document is a guideline for the developers that shows how the entire process must proceed.
The document is structured as follows:

- **Chapter 1 - Introduction:** a brief introduction of the document.
- **Chapter 2 - Integration Strategy:** a description of the integration testing approach and the reasons behind it. It also describes the necessary conditions needed to start the integration testing and all the components that must be tested.
- **Chapter 3 - Individual Steps and Test Description:** a description of the tests to be taken and the sequence in which they must be applied.
- **Chapter 4 - Tools and Test Equipment Required:** an overview of the tools that the tester must use to perform the integration testing.
- **Chapter 5 - Program Stubs and Test Data Required:** a list of all the program stubs used to simulate components with the related descriptions.
- **Chapter 6 - Effort Spent:** contains information about the effort spent by the group.

1.3 List of Definitions and Abbreviations

All the definitions from the previous documents remain valid. Here, only the new ones are listed.

Module: is a high level subsystem. Also used as a synonymous of subsystem. Modules are the two Applications, the Central System, the Car, the PGS and the Bank. Central System is also divided in *submodules*: BackOffice, Interface Server and AppServer.

1.4 List of Reference Documents

All the following documents are available at the GitHub project directory: https://github.com/AlessandroPerini/PowerEnJoy

[1] PowerEnJoy specification document: *"ASSIGNMENTS AA 2016-2017.PDF"*.

[2] Requirements Analysis & Specification Document - *"PowerEnJoy RASD";*
Alessandro Perini, Federico Saini, Ali M. Türkçapar.

[3] Design Document - *"PowerEnJoy DD";*
Alessandro Perini, Federico Saini, Ali M. Türkçapar.

Chapter 2
Integration Strategy

This chapter will describe the integration strategy. In the section 2.1 the prerequisites for the integration test phase will be presented. The section 2.2 is dedicated to the required components that will be integrated in the system to execute some tests. Finally, in the sections 2.3 and 2.4 the strategy used to test the integrations will be discussed.

2.1 Entry Criteria

In order to start the integration test phase, some criteria must be respected:

- **RASD and DD must be complete and correct.**

  This is required in order to have a description of the single components, a clear picture of their interactions and a brief description of the functionalities that they perform.

- **Interfaces between modules must be completely coded.**

  To ensure the correct communication between modules, the related interfaces must be completely coded before starting the integration testing. If this is not possible, the interfaces must be completely available to start the integration among modules (phase C). In this way, using the bottom-up integration testing approach, the modules yet to be coded will be simulated and the right communication among them will be ensured by the coded interfaces. This interfaces are:

    - AppServer API
    - Interface API
    - BO Connector
    - Car Connector

  We also assume that the already available APIs work properly. They are:

    - GMaps API
    - Bank API
    - GPS Driver
    - JDBC

- **Components must be unit tested (at least the one to be integrated).**

  Before starting any phase of the integration testing, all the components that will be integrated must be unit tested in order to ensure the right working. Unit testing must be done firstly in a structural way, where each method is tested in white-box, and then in a functional way, where all the functionalities of the class or classes of the component are tested in black-box. In this way is ensured the correct working of every component.
  Assuming that the available APIs listed above were already been tested, they should not be unit tested. Only a functional test with the components that use them is needed.

- **Database must be structured and must work properly.**

  To start the integration phase, the database must be structured. Every table with every attribute must be created even without containing all the data. The minimum amount of data needed to perform the integration phase is described in the last chapter at paragraph 5.2. Also the correct working of the database must be ensured. It must be verified during the unit testing of the DB Communication Manager component. This means that, when the DB Communication Manager is unit tested, also the query to the database must be tested. This is useful to check either the correct working of the component and to check the correct working of the physical database.

2.2 Elements to be Integrated

Here are listed all the components of the system that need to be integrated.

- App-Server
  - All the internal components
  - AppServer API

- Interface Server
  - All the internal components
  - Interface API

- BackOffice
  - All the internal components
  - BO Connector
  - Bank API

- Car
  - OnBoard Unit
  - Touchscreen Stub
  - Communication Port Stub
  - Car Connector

- Applications
    - All the internal components
    - Google Maps
    - GPS Stub

2.3 Integration Testing Strategy

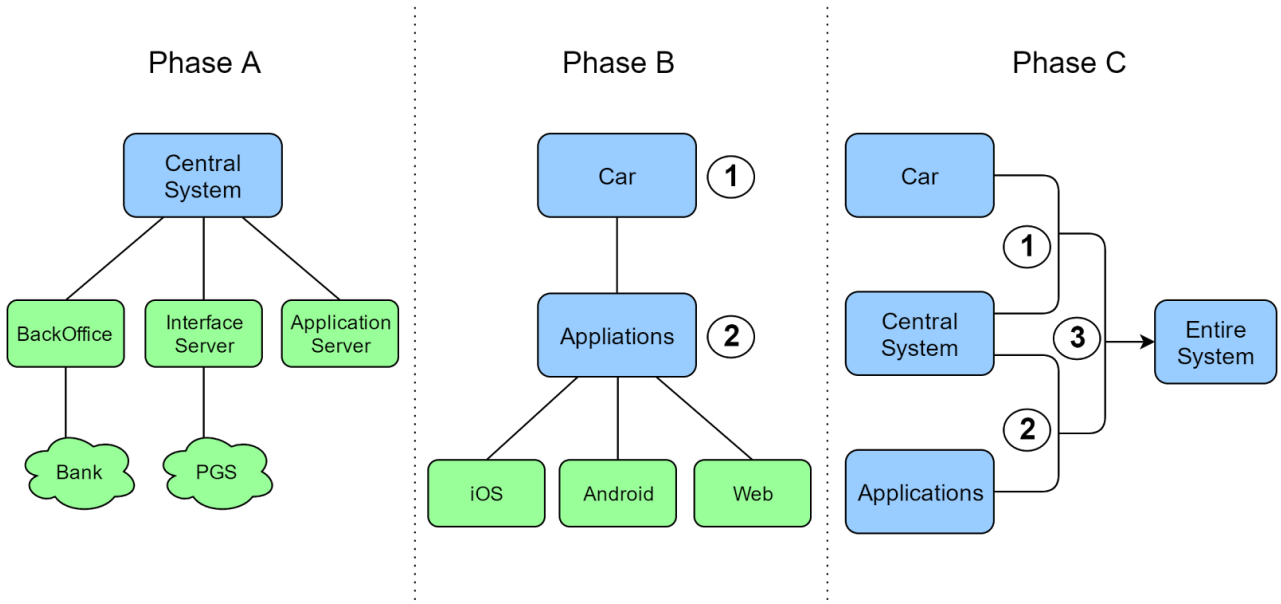
As written in the Design Document [3], our system is divided in three high level modules, the Central System, the Car and the Applications. Each of them is then divided into other smaller components. Our plan is to first test the Central System, then the Car and eventually the Applications. Central System is the most important part of the PowerEnJoy service. All the modules communicate with each other thanks to the Central System, so we need to ensure firstly the correct working of it. Once this module is tested, the integration phase can move to the Car module. All its components will be integrated obtaining the whole Car module. Car will then be integrated with the Central System. The procedure is then repeated for the Applications. This is a critical-module-first approach, where the most important components or modules are tested first.

Parallel to that, the bottom-up approach will be used. This is because the major modules are divided in smaller components. For example, the Central System is made by servers and a BackOffice and each of them is made up of components. It is important to decide a strategy that defines the sequence of integration on which the components and the modules are tested. This strategy is defined in the following paragraph 2.4. In the bottom-up approach the integration testing starts from the lower level components simulating the behaviour of the higher ones. We are taking this choice because of two major reasons. Firstly, because we can perform the testing on “real” components. It means that the component that has to be tested is the already coded one that will be part of the system. In this way we don’t need to simulate the component, and its behaviour is the one that it will have in the real word usage. In this way, more precise indications about how the system will behave, once completed, are given. Secondly the testing phase can start as soon as the needed components are coded. In this way we can start the integration testing phase earlier, minimizing the time needed.

Once a module is fully tested, it can be integrated to the others using the already coded interfaces. They must be available because it is an entry condition for the starting of the integration test phase, as defined in the paragraph 2.1. To summarize, our plan is to use both a bottom-up approach, to choose the way components of a module must be integrated, and a critical-module-first approach, to choose the sequence on which modules are integrated.

2.4 Sequence of Component/Function Integration

As specified in the Design Document [3], the system is built on the interactions of high-level subsystems, called modules, each one is further obtained by the combination of several lower-level components, simply called components. Because of this software architecture, the integration phase will involve the integration of components at two different levels of abstraction.



Phase A

Every submodule (BackOffice and servers) must be tested first. Once they are all tested, they can be integrated together to complete the Central System. This entire phase is described in paragraph 2.4.1.1. Bank and PGS are integrated in this phase and not the phase C where all modules are integrated. This because their interactions with the Central System are very basic (few functionality) and fast to test.

Phase B

The first module to be tested must be the Car. This is because it is an important module and its integration with the Central System can start as soon as possible. So, once it is tested, the last phase can start parallel to the testing of the Applications.

Phase C

As said before, the first step can start as soon as the Car is tested. Step 2 must wait the completion of the phase B. Once both Car and Applications are singularly integrated with the Central System, all the system can be integrated together to complete the whole integration test phase.

2.4.1 Software Integration Sequence

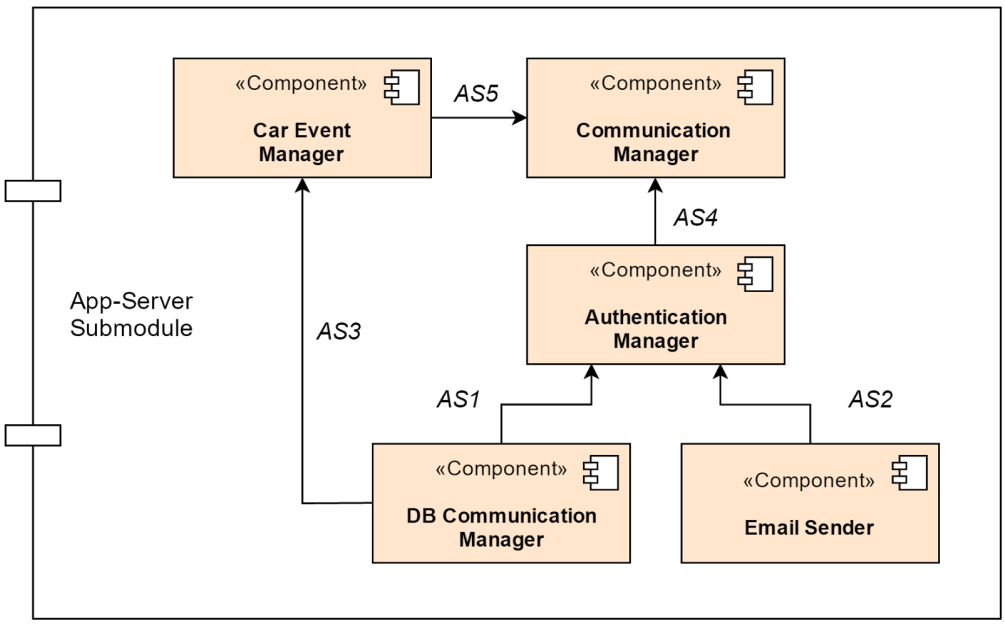
In this paragraph it is described how the various components are integrated together to create each module.

The enumeration in the names of the test cases are not binding to the sequence on which they must be integrated. This because for every integration test, only the two interested components are required. There aren't any other component dependences. This except for the phase C of the integration in which every module must be already integrated.

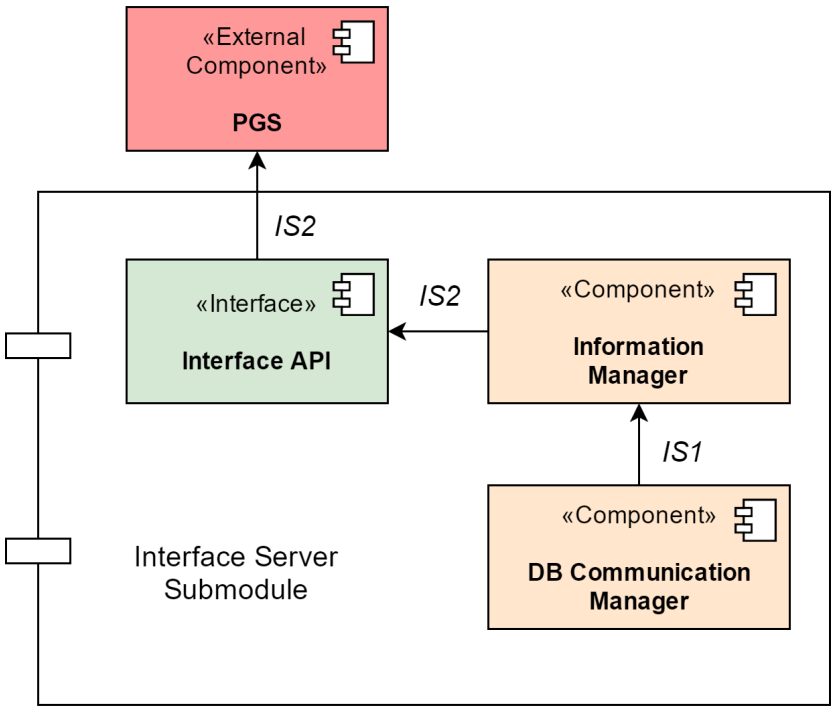
2.4.1.1 Central System

First, every submodule of the Central System is tested, then all of them are integrated together to form the Central System and complete the Phase A.

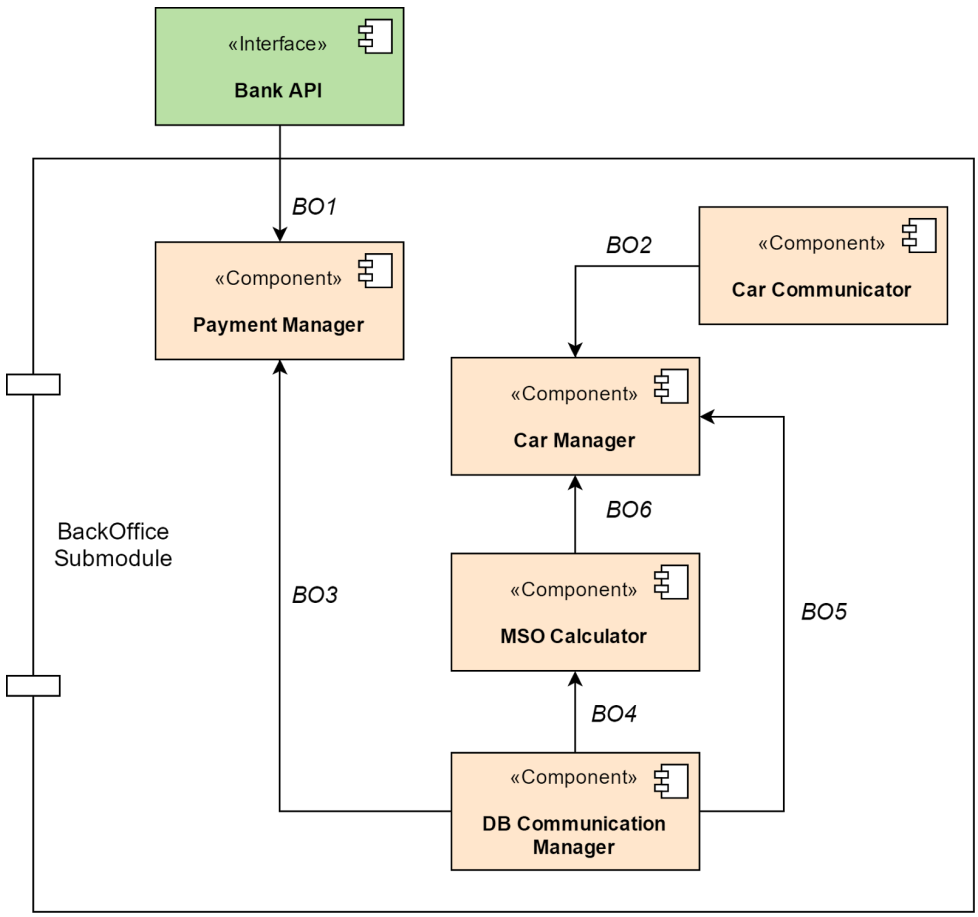
App-Server



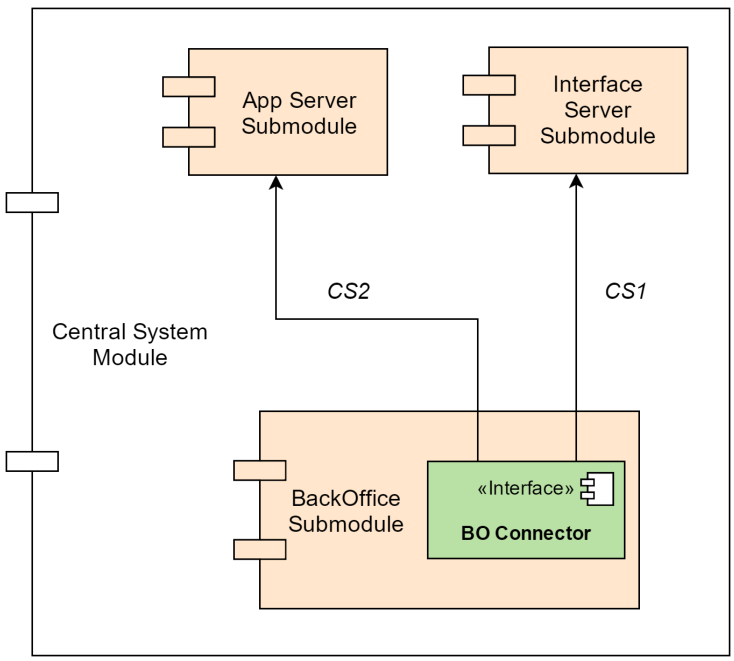
Interface Server



BackOffice

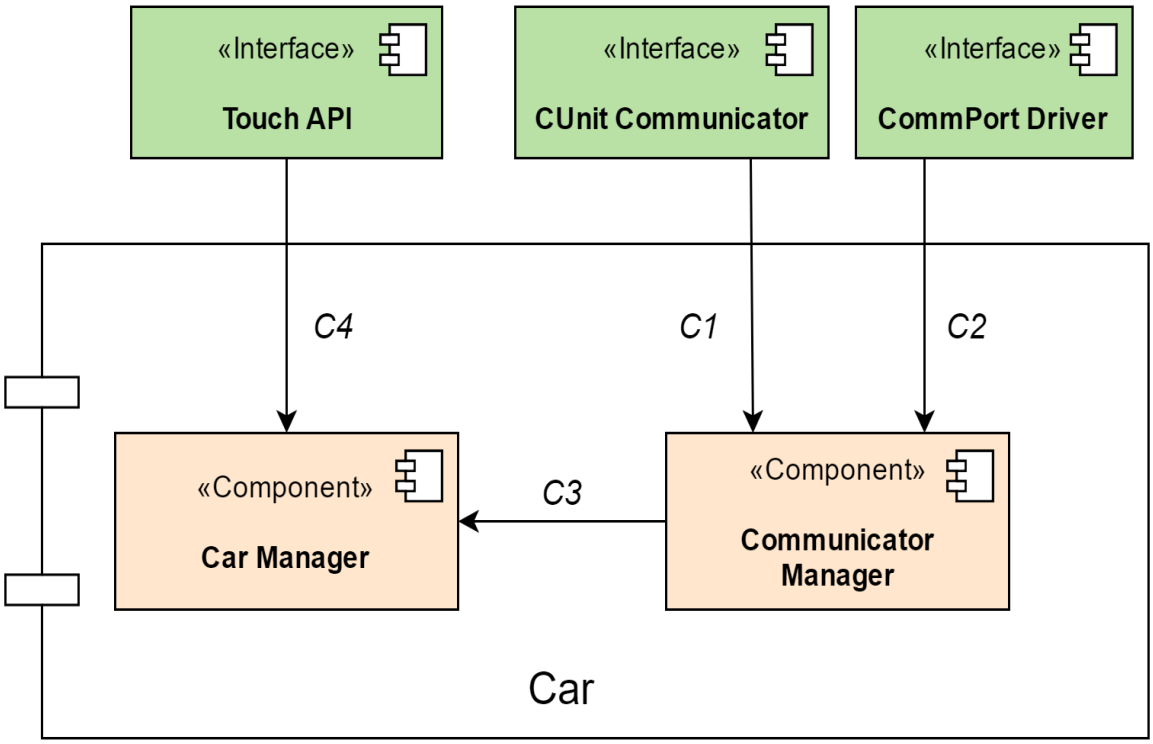


Integration of submodules of the Central System



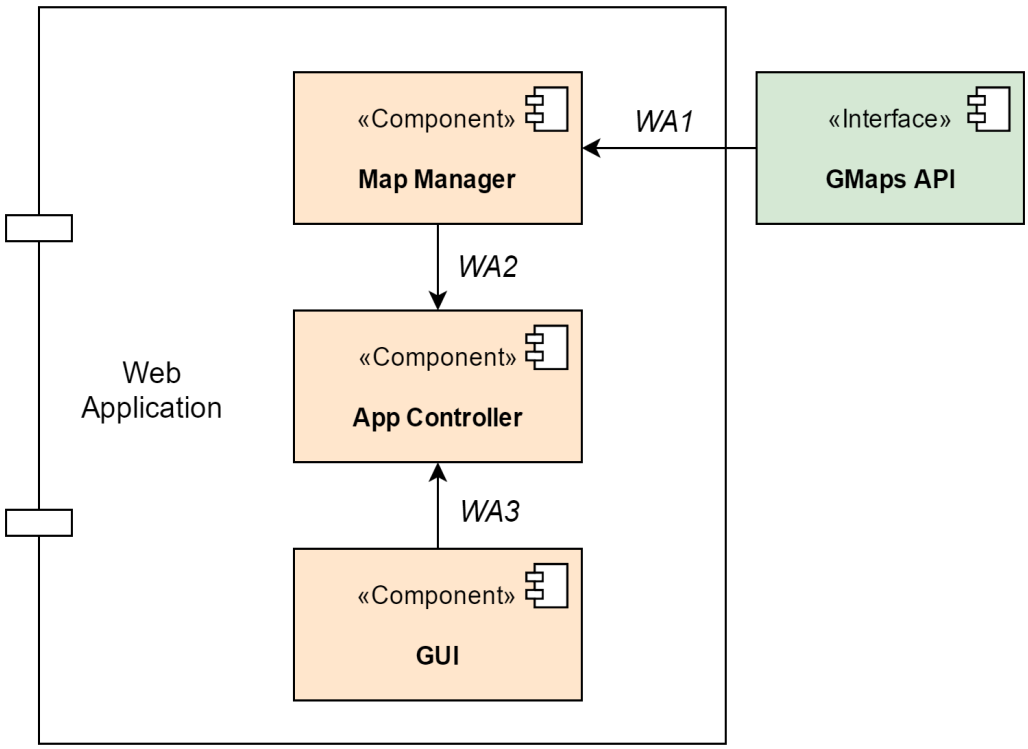
2.4.1.2 Car

After the Central System is integrated, the Phase A is completed. Now the Phase B can start. Car must be the first module to be tested.
After the Car is tested, there are two options: test the applications or integrate the Central System with the Car and start the Phase C. Of course without completing the integration of the applications, only the first step of the Phase C can be performed.

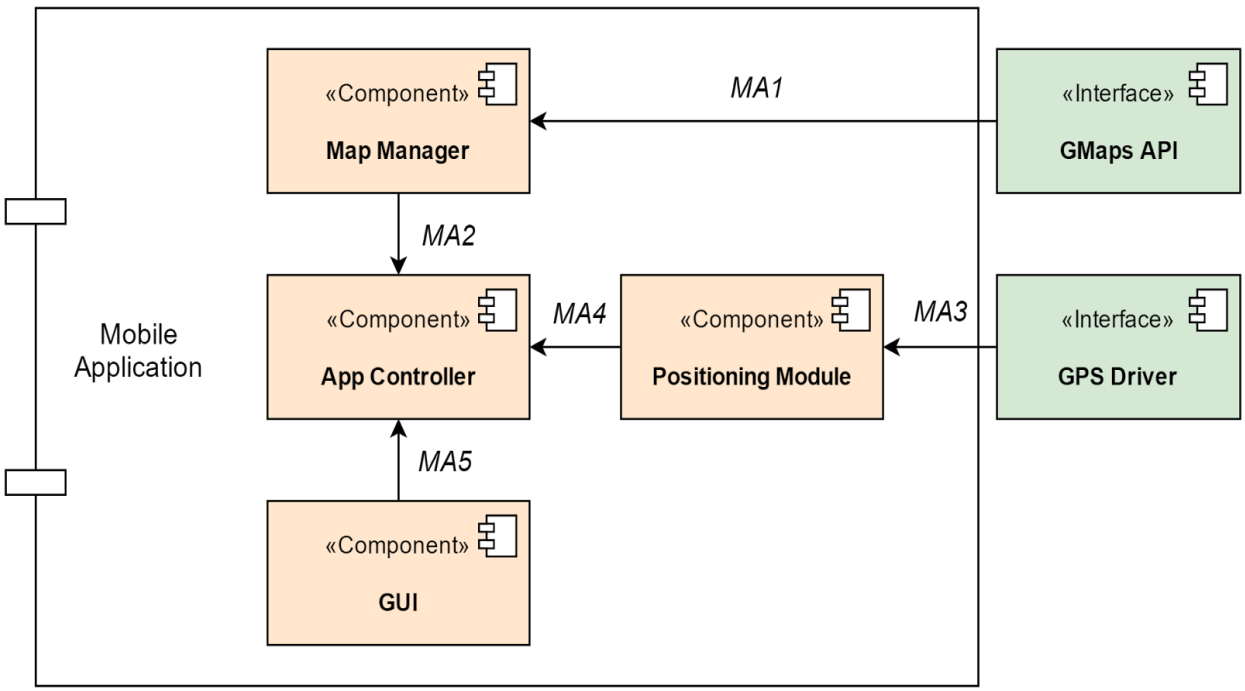


2.4.1.3 Applications

Web Application

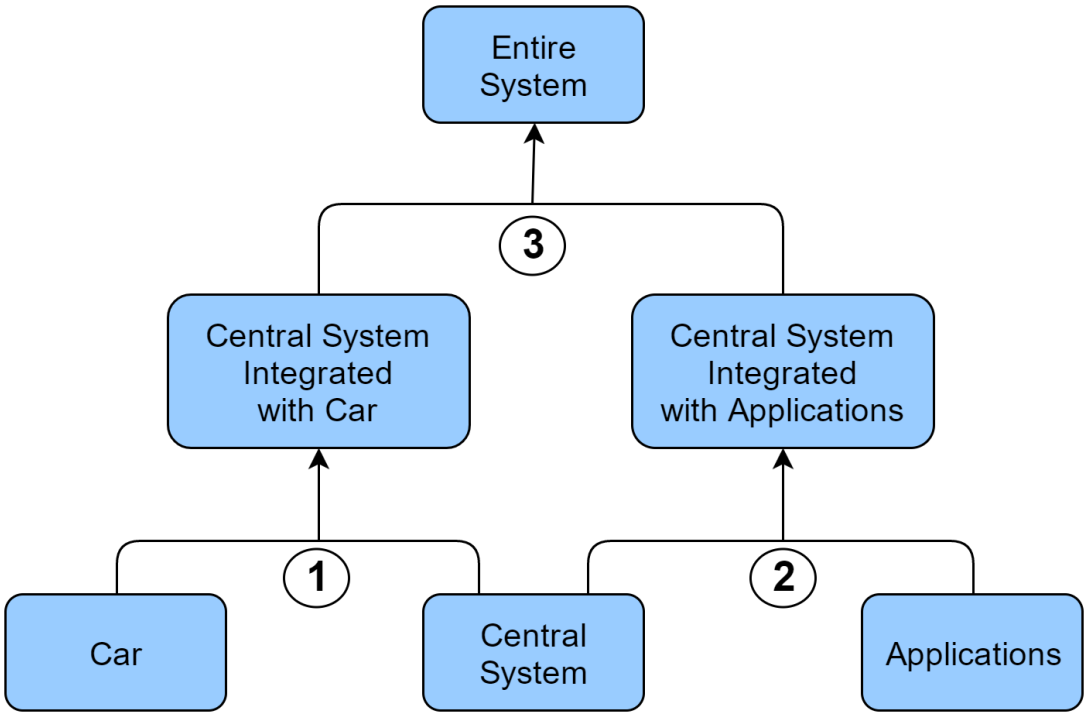


Mobile Applications (iOS and Android)

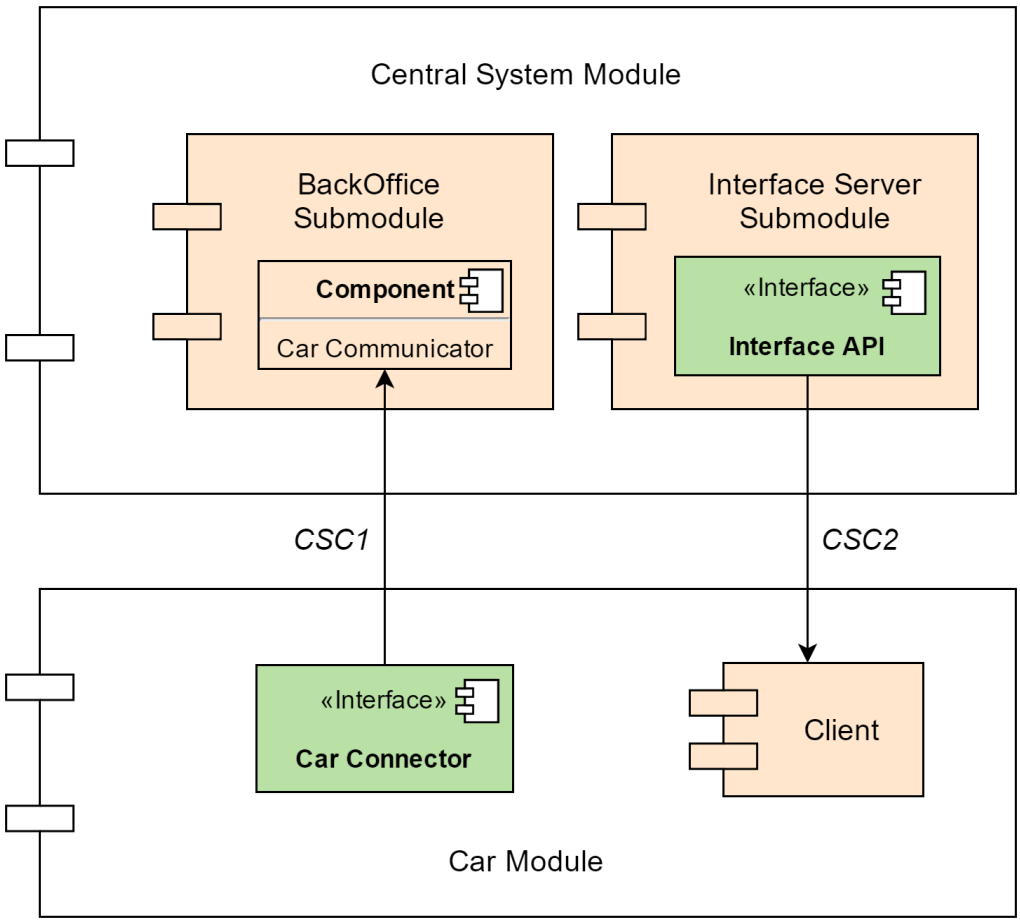


2.4.2 Subsystem Integration Sequence

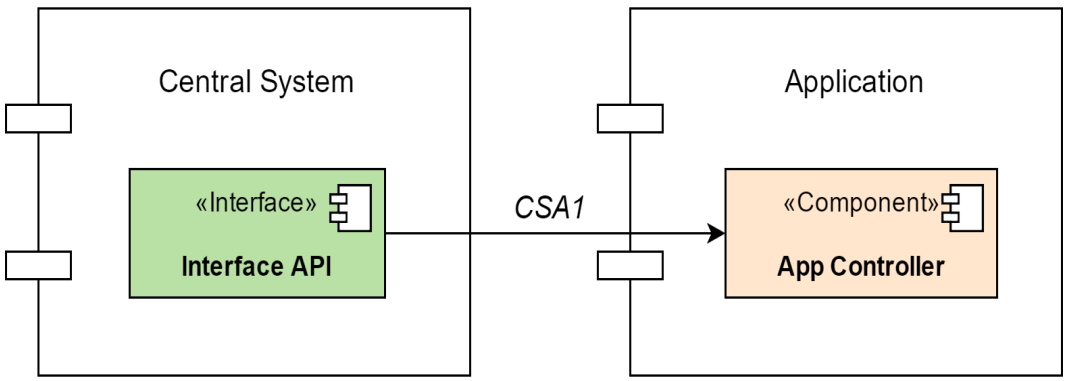
In this paragraph it is provided a general overview on how the different high-level subsystems (modules) are integrated together to create the entire PowerEnJoy infrastructure.
The first thing to do, following the critical-module-first approach as defined before, is to test the Central System. Secondly Car must be tested and then integrated with the Central System. Then the Applications can be tested and integrated with the Central System. At the end, the system must be tested together simulating a race. So the sequence must be the one described in the diagram below.



2.4.2.1 Integrate Central System with Car

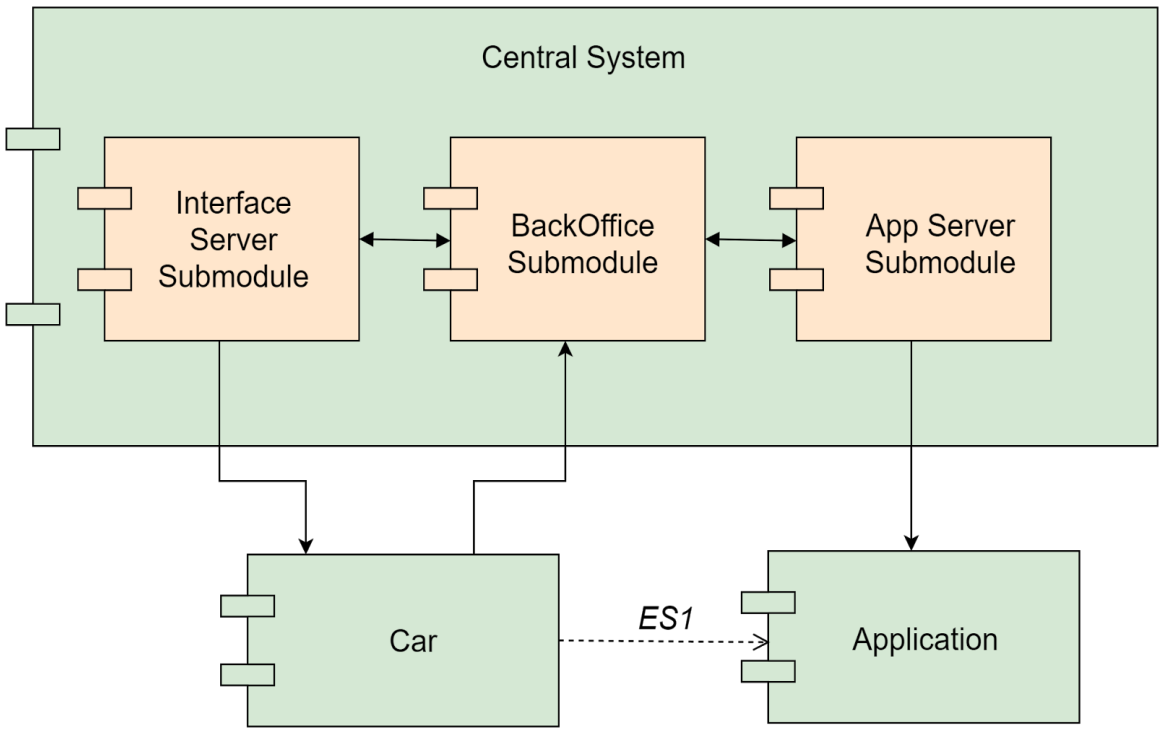


2.4.2.2 Integrate Central System with Applications



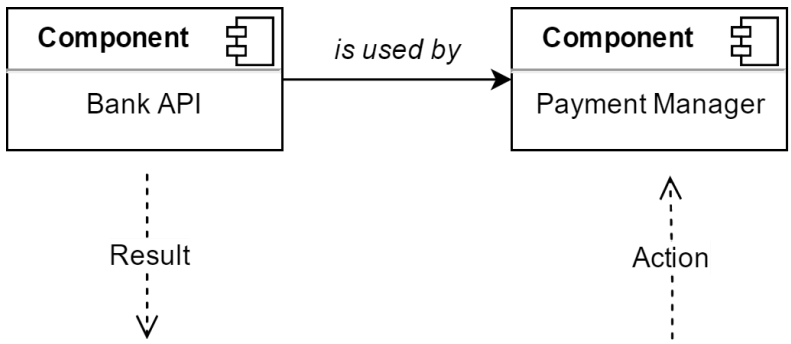
2.4.2.3 Integrate the entire system

The only missing integration is the one between Application and Car, so the only performed test case is this. Then the entire system is created and it is ready for future testings.



Chapter 3
Individual Steps and Test Description

As notation, the arrow between components, *Component 1* → *Component 2*, means that *Component 1* is used by *Component 2.* In the following example Bank API is used by Payment Manager. This notation is very useful in diagrams because it makes easy to construct the sequence of integration. Below are also shown the Action and Result used in the test cases.



The following test cases are intended to be the minimum set of operations that the tester performs in order to test the components. Any additional test can be performed to give a higher level of assurance that the system works as expected.

During the integration of the components, the other part of the system has not already been tested, so the actions that must be performed on each component to test it are low level actions. This means that, for the test of components, we don't specify a user type action (like "User click the unlock button") that then will cause an indirect use of that component. Instead, we will indicate an action (like "The method to send the unlock signal to the car is called") directly to that component like calling a specific method.

3.1 Central System

3.1.1 BackOffice

Test Case Identifier	BO1
Test Components	Bank API → Payment Manager
Action	1. Using the Payment manager verifies the validity of a valid user payment information. 2. Using the Payment manager try to verify the validity of an invalid user payment information. 3. Try to perform a transaction.
Result	1. The Bank API indicates that the information are correct. 2. The Bank API indicates that the information are not correct. 3. Money is transferred to the PowerEnjoy bank account.
Environmental Needs	Valid and invalid payment information are needed.

Test Case Identifier	BO2
Test Components	Car Communicator → Car Manager
Action	1. Ask the Car Manager to unlock a car. 2. Ask the Car Manager to send the result of the MSO Calculator to a car.
Result	3. Car Communicator sends an unlock signal to the car. 4. Car Communicator sends the information to the car.
Environmental Needs	-

Test Case Identifier	BO3
Test Components	DB Com. Manager → Payment Manager
Action	1. Ask to the Payment Manager to check into the DB if there are some unpaid races.
Result	1. The DB Com. Manager performs a query on the DB and returns the result to the Payment Manager.
Environmental Needs	At least one unpaid race.

Test Case Identifier	BO4
Test Components	DB Com. Manager → MSO Calculator
Action	1. Ask to the MSO Calculator to calculate the best station to park.
Result	1. The DB Com. Manager performs a query on the DB in order to get the state of each station and the result is returned back to the MSO Calculator.
Environmental Needs	At least one PGS with available plugs.

Test Case Identifier	BO5
Test Components	DB Com. Manager → Car Manager
Action	1. Simulate the end of a race on the Car Manager.
Result	1. The DB Com. Manager writes the race's finish into the DB.
Environmental Needs	The related race in the database.

3.1.2 Interface Server

Test Case Identifier	IS1
Test Components	DB Com. Manager → Information Manager
Action	1. The Information Manager receives the status of a car. 2. The Information Manager receives information from a PGS
Result	1. The DB Com. Manager performs the query to upload it. 2. The DB Com. Manager performs the query to upload it.
Environmental Needs	The related car and PGS on the database.

Test Case Identifier	IS2
Test Components	Information Manager → Interface API
Action	1. Simulate the PGS data sending to the Interface Server through the API. 2. Simulate the Car status sending to the Interface Server through the API.
Result	1. The Information Manager receives correctly the information. 2. The Information Manager receives correctly the information.
Environmental Needs	-

3.1.3 Application Server

Test Case Identifier	AS1
Test Components	DB Com. Manager → Authentication Manager
Action	1. The Authentication Manager asks to the DB Com. Manager to insert a new user into the DB.
Result	1. DB Com. Manager performs the query to insert the user information.
Environmental Needs	Information of a user.

Test Case Identifier	AS2
Test Components	Email sender → Authentication Manager
Action	1. The Authentication Manager asks to the Email sender to send an email to the user.
Result	1. The Email Sender generates a random password and sends the email to the user.
Environmental Needs	A user with a valid email address.

Test Case Identifier	AS3
Test Components	DB Com. Manager → Car Event Manager
Action	1. The Car Event Manager simulates a reservation of a car and ask to the DB Com. Manager to write the data in the DB.
Result	1. The DB Com. Manager receives the information and performs a query to the DB to write the related race information.
Environmental Needs	Information of a race.

Test Case Identifier	AS4
Test Components	Authentication Manager → Com. Manager
Action	1. Com. Manager simulates the login of a user.
Result	1. The Authentication Manager communicates to the DB Com. Manager to perform a query to check the validity of the data.
Environmental Needs	-

Test Case Identifier	AS5
Test Components	Car Event manager → Com. Manager
Action	1. Com. Manager simulates the car reservation from a user. 2. Com. Manager simulates the car unlock signal from a user.
Result	1. Car Event Manager sends a request to the DB Com. Manager to control the permission and possibly forward the reservation to the BackOffice in order to create the related process. 2. Car Event Manager forwards the request to the BackOffice.
Environmental Needs	-

3.1.4 Entire Central System

Test Case Identifier	CS1
Test Components	BO connector → Interface Server submodule
Action	1. Interface Server tries to forward the destination address of a race to the BO Connector. 2. Interface Server tries to forward the number of passenger of a race to the BO Connector.
Result	1. The BO Connector receives the information correctly. 2. The BO Connector receives the information correctly.
Environmental Needs	BO1,..,BO6,IS1,IS2,AS1,..,AS5 successfully executed.

Test Case Identifier	CS2
Test Components	BO Connector → App Server Submodule
Action	1. The App Server Submodule forwards the reservation of a car.
Result	1. The BO Connector receives the information correctly.
Environmental Needs	BO1,..,BO6,IS1,IS2,AS1,..,AS5 successfully executed.

3.2 Car

3.2.1 OnBoard Unit

Test Case Identifier	C1
Test Components	CUnit Communicator → Communicator Manager
Action	1. Communicator Manager sends a signal to the CUnit Com. to indicates that the user can't restart the engine.
Result	1. CUnit Communicator receives the signal and block the ignition of the engine.
Environmental Needs	CS1,CS2 successfully executed.

Test Case Identifier	C2
Test Components	CommPort Driver → Communicator Manager
Action	1. Communication Port Stub simulates signals coming from the sensors.
Result	1. The Communicator Manager receives the signals.
Environmental Needs	CS1,CS2 successfully executed. Communication Port Stub needed.

Test Case Identifier	C3
Test Components	Communicator Manager → Car Manager
Action	1. Car Manager locks the doors.
Result	1. Communicator Manager sends the request to lock the doors to the CUnit Communicator.
Environmental Needs	CS1,CS2 successfully executed.

Test Case Identifier	C4
Test Components	Touch API → Car Manager
Action	1. The Car Manager sends information to be shown in the touchscreen. 2. Input are submitted using the touchscreen stub.
Result	1. The Touch API displays it on the touchscreen. 2. Car Manager receives it.
Environmental Needs	CS1,CS2 successfully executed. Touchscreen stub needed.

3.3 Applications

3.3.1 Mobile Applications (iOS + Android)

Test Case Identifier	MA1
Test Components	GMaps API → Map Manager
Action	1. Ask the Map Manager to show a portion of the city by passing an address.
Result	1. GMaps API searches the address and gives back the related map.
Environmental Needs	C1,C2 successfully executed.

Test Case Identifier	MA2
Test Components	Map Manager → App Controller
Action	1. Ask the App Controller for available cars.
Result	1. The Map Manager send two requests: one to the GMaps API to have the city map and the other to the AppServer to have a list of the available cars with the related positions.
Environmental Needs	C1,C2 successfully executed.

Test Case Identifier	MA3
Test Components	GPS Driver → Positioning Module
Action	1. Ask to the Positioning Module to locate the position of the current user.
Result	1. GPS Driver gets the coordinates from the GPS Stub and gives it back to the Positioning Module.
Environmental Needs	C1,C2 successfully executed. GPS Stub needed.

Test Case Identifier	MA4
Test Components	Positioning Module → App Controller
Action	1. Ask to the App Controller to locate the position of the current user.
Result	1. The Positioning Module receives the request and forward it to the GPS using the GPS Driver.
Environmental Needs	C1,C2 successfully executed.

Test Case Identifier	MA5
Test Components	GUI → App Controller
Action	1. Simulate the user app browsing.
Result	1. GUI shows the pages through the display.
Environmental Needs	C1,C2 successfully executed.

3.3.2 Web Application

Test Case Identifier	WA1
Test Components	GMaps API → Map Manager
Action	1. Ask the Map Manager to show a portion of the city by passing an address.
Result	1. GMaps API searches the address and gives back the related map.
Environmental Needs	C1,C2 successfully executed.

Test Case Identifier	WA2
Test Components	Map Manager → App Controller
Action	1. Ask the App Controller for available cars.
Result	1. The Map Manager send two requests: one to the GMaps API to have the city map and the other to the AppServer to have a list of the available cars with the related positions.
Environmental Needs	C1,C2 successfully executed.

Test Case Identifier	WA3
Test Components	GUI → App Controller
Action	1. Simulate the user app browsing.
Result	1. GUI shows the pages through the display.
Environmental Needs	C1,C2 successfully executed.

3.4 Integration Between Modules

3.4.1 Central System - Car

Test Case Identifier	CSC1
Test Components	Car Module → Central System Module
Action	1. On the Central System, try to unlock the car having the right permissions. 2. On the Central System, try to unlock the car without having the right permissions.
Result	1. The car is unlocked. 2. The car remains locked.
Environmental Needs	C1,..,C3 successfully executed.

Test Case Identifier	CSC2
Test Components	Central System Module → Car Module
Action	1. Car sends his status. 2. Report an issue using the touchscreen. 3. Car asks for the best station to park (MSO). 4. Car sends the number of passengers.
Result	1. The Central System correctly receives the status. 2. The Central System correctly receives the issue. 3. The Central System calculates it and gives back the result. 4. The Central System correctly receives the number of passengers.
Environmental Needs	C1, ..,C3 successfully executed.

3.4.2 Central System - Applications

Test Case Identifier	CSA1
Test Components	Central System → Application
Action	1. Perform a login with correct email and password. 2. On the Application, search the available cars. 3. Reserve a car. 4. Report an issue.
Result	1. Central System checks the information and replays that the login has been successfully executed. 2. Central System gives back the list of available cars. 3. Central System reserves that car and replay to the Application that the reservation is active. 4. Central System correctly receives the issue.
Environmental Needs	CSC1, CSC2 successfully executed

3.4.3 Entire System

Test Case Identifier	ES1
Test Components	Car → Central System → Application
Action	1. When you are close to the car (simulate it using the GPS Stub), try to unlock it.
Result	1. The car doors are now unlocked.
Environmental Needs	CSC1, CSC2, CSA1 successfully executed

Chapter 4
Tools and Test Equipment Required

4.1 Tools

To perform the integration test phase two frameworks should be used.

The first tool is Mockito, an open source testing framework for Java that simulate the behaviour of the real components in controlled ways. It allows testers to create mock objects, so it is useful to design and implement the program stubs which are defined in the following chapter 5. It controls if the components communicate correctly one with each other and if the expected results are produced. It also controls if methods return correct objects, if exceptions are raised in a correct way and in general it searches all the possible problems that could happen during the interaction between components. The unit testing of every component can be performed either using Mockito or with JUnit.

The second, Arquillan, is a framework needed to perform the integration testing between containers. In particular, we are going to use Arquillian to check if the interaction between the component and its environment. It is useful to perform testing using embedded or remote containers.

Another useful tool can be Apache JMeter. It is an Apache project that can be used as a load testing tool for analysing and measuring the performance of the system. Measures can be performed during a normal or an intense execution of the system. We don't consider JMeter a needed tool because this part of system testing is not competence of this document. Despite that, it can be a useful way to check performance requirements defined in the RASD [2].

4.2 Equipment

Since the system is made up of different types of subsystems, also the testing activities must be performed in different software environments and may require hardware devices. The integration between subsystems, the phase C of the integration testing, must be performed on the real hardware components. This is because of some reasons. Firstly, because testing the interactions on real components ensures a more realistic use of the system. For example, the integration between the Central System and the Car is more useful if the real OnBoard Unit and touchscreen are used. Also for the integration between Central System and Applications, devices with the installed application are required. So, even if the environments can be simulated, the last integration phase must be done on real components. This approach is also used because the integration of subsystems is assumed to be taken in a quite final phase of the entire project, and so, it can be assumed that the hardware components, like the ones in the car, should be already available.

Mobile Applications
The integration of the different mobile applications must be done with application installed on the devices even if the device can be emulated. In this way more realistic behaviours are ensured.

Web Application
The web application will be tested using the most common browsers on a normal computer.

Car
For the integration with the Central System, the car software must be installed on the OnBoard Unit hardware component. Also the touchscreen must be available.

Central System
Even the Central System must run on the real hardware for the integration phase.

So, the minimum set of devices and software needed to complete the integration process consists:

- an iOS smartphone.
- an Android smartphone.
- a Windows Phone smartphone.
- a computer with the major browsers (Google Chrome, Mozilla, Safari, Edge/Internet Explorer).
- the set of hardware components of the car (OnBoard Unit, touchscreen).
- the set of hardware components of the Central System (Servers and BackOffice).

Chapter 5
Program Stubs and Test Data Required

5.1 Program Stubs and Drivers

Here are defined the stubs used to simulate the behaviour of the not already developed components.

Car

- *Touchscreen Stub*: simulates the inputs from the touchscreen, miming the decisions that a user can take.
- *Communicator Port Stub*: simulates the signals coming from the sensors of the car.

Application

- *GPS Stub*: simulates the positioning when the real GPS is not available. For example, during the unit and component tests, the application is not yet installed on the smartphone but it is running on the developer's IDE. Moreover, the simulation of the GPS coordinates allows the tester to change his position without physically moving.

5.2 Test Data

The minimum entry condition to perform integration testing is to have an already developed and structured database. Although, to integrate some components we need some data in it. The minimum amount of data required to compute those testings is listed here, but of course the better option is to have a better filled database with coherent data.
The last column of the table indicates where the related data are needed. Sometimes data must be in the database, other times data must be available to the tester in order to use it during the integration.

Test Case ID	Required Data	Needed in/by
BO1	One valid and one not valid payment information.	Database
BO3	At least one unpaid race.	Database
BO4	At least one PGS with available plugs.	Database
BO5	A race.	Database
IS1	A car and a PGS.	Database
AS1	Information about a user.	The tester
AS2	User with valid email address.	The tester
AS3	Information about a race.	The tester

Chapter 6
Effort Spent

Date	Description	Perini	Saini	Türkçapar
20/12/2016	Document structure and descriptions.	1h	1h	
23/12/2016	Integration Strategy	2h		
28/12/2016	Completed Chapter 2	2h		
05/01/2017	Team Work	3h	3h	
06/01/2017	Integration Diagrams	2h		
10/01/2017	Team Work	3h	3h	
11/01/2017	Chapter 4	1h		
12/01/2017	Team Work	3h	3h	3h
13/01/2017	Team Work	2h		2h
14/01/2017	Last Adjustments	3h	1h	2h
15/01/2017	Release	1h		