POLITECNICO DI MILANO

Department of Electronic, Information and Biomedical Engineering

Master Degree course in Computer Science Engineering

Project of Software Engineering 2

PowerEnJoy

Integration Test Plan Document

(ITPD)

Version 1.0 (15 January 2017)

Reference professor: Prof. Elisabetta Di Nitto

Authors:

Davide Anghileri matr. 879194

Antonio Paladini matr. 879118

Revision History

Name	Date	Reason for Changes	Version
First complete version	15/01		1.0
First version, small changes	17/01	Modified percentages of degree of completion for some component before to start the integration testing. Modified formatting.	1.1

Contents

1 Introduction

1.1 Purpose

The purpose of ITPD (**Integration Test Plan Document**) is to give a detailed description of the plan for the integration test of the PowerEnJoy application.
The main goal of this activity is to detect and delete all possible errors and bugs among the interfacing between the different components.
Also, it's aimed to exercise this interaction between modules and check the **compatibility** against functional, performance and reliability requirements. The preferred approach for integration testing is **black-box** since single modules are supposed to be already tested isolated.
The Integration Test Plan Document is intended to be the main reference for this process and it is mainly addressed to the integration test team.

1.2 Scope

This document regards the PowerEnJoy project whose requirements and design phase is described in the correspondent reference documents written below.
This document should be read alongside the Design Document of PowerEnJoy.

1.3 Definitions, Acronyms and Abbreviations

1.3.1 Definitions

- Other definitions are in the RASD and DD Document.

1.3.2 Acronyms

- DD: Design Document
- RASD: requirements analysis and specifications document
- ITPD: Integration Test Plan Document
- CDI: Context and Dependency Injection

1.3.3 Abbreviations

- PEJ = PowerEnJoy
- SE = Standard Edition

1.4 Reference Documents

- RASD produced before 2.0
- DD produced before 2.0
- Assignments AA 2016-2017
- Spingrid ITP version 0.1.0
- Software Engineering 2 course slides

1.5 Document structure

This document is composed of five sections and an appendix.

- The first section, this one, is intended to define the goal of the ITPD, and the resources used to draw up this document.
- The second section constitutes the core of the test plan. This section is devoted to the description of the integration test strategy: the preconditions required to start the integration test will be presented, the main rationales behind the chosen strategy will be discussed; the elements to be integrated will be listed with reference to the ones presented in the DD.
- The third section contains the definition of the test sets and test cases. Each test will be presented with reference to the sequence explained in the second section, together with the hypothesis about the initial state of the system and the expected results.
- The fourth section contains a description of the tools required in order to run the test that we've defined.
- The fifth section is a description of the drivers and stubs required in our system in order to execute the integration test.
- The sixth section is a list of tools used to redact this document and the hours of work.

2 Integration Strategy

This section is devoted to the explanation of the main choices concerning the integration **testing strategy** that we decided to use.

2.1 Entry Criteria

Before starting to describe our integration test plan in details, we are going to illustrate in this paragraph which are the **criteria** that must be met before the integration testing of specific elements may begin; if one of them is not verified it can compromise or make even impossible the entire process.
We have identified the following entry criteria:

- The Requirements Analysis Specifications Document (RASD) and the Design Document (DD) must have been fully written.
- Unit testing for every module/component is complete.
- All high prioritized bugs are fixed and closed.

Furthermore, the integration test plan can start only when the following estimated **percentage of completion** of every **component** with respect to its functionalities is achieved:

- At least 90% for the User Manager subsystem
- At least 80% for the Maintainer Manager and the Administrator Manager subsystems
- At least 50% for the User, Maintainer and Administrator applications

It should be noted that these percentages refer to the status of the project at the beginning of the integration testing phase and they do not represent the minimum **completion percentage** necessary

to consider a component for integration ready, which must be at least **90%**.

We decided to have different percentages for different components to early start our integration test with the completed components and meanwhile end up the other components in parallel with the integration phase.

2.2 Elements to be integrated

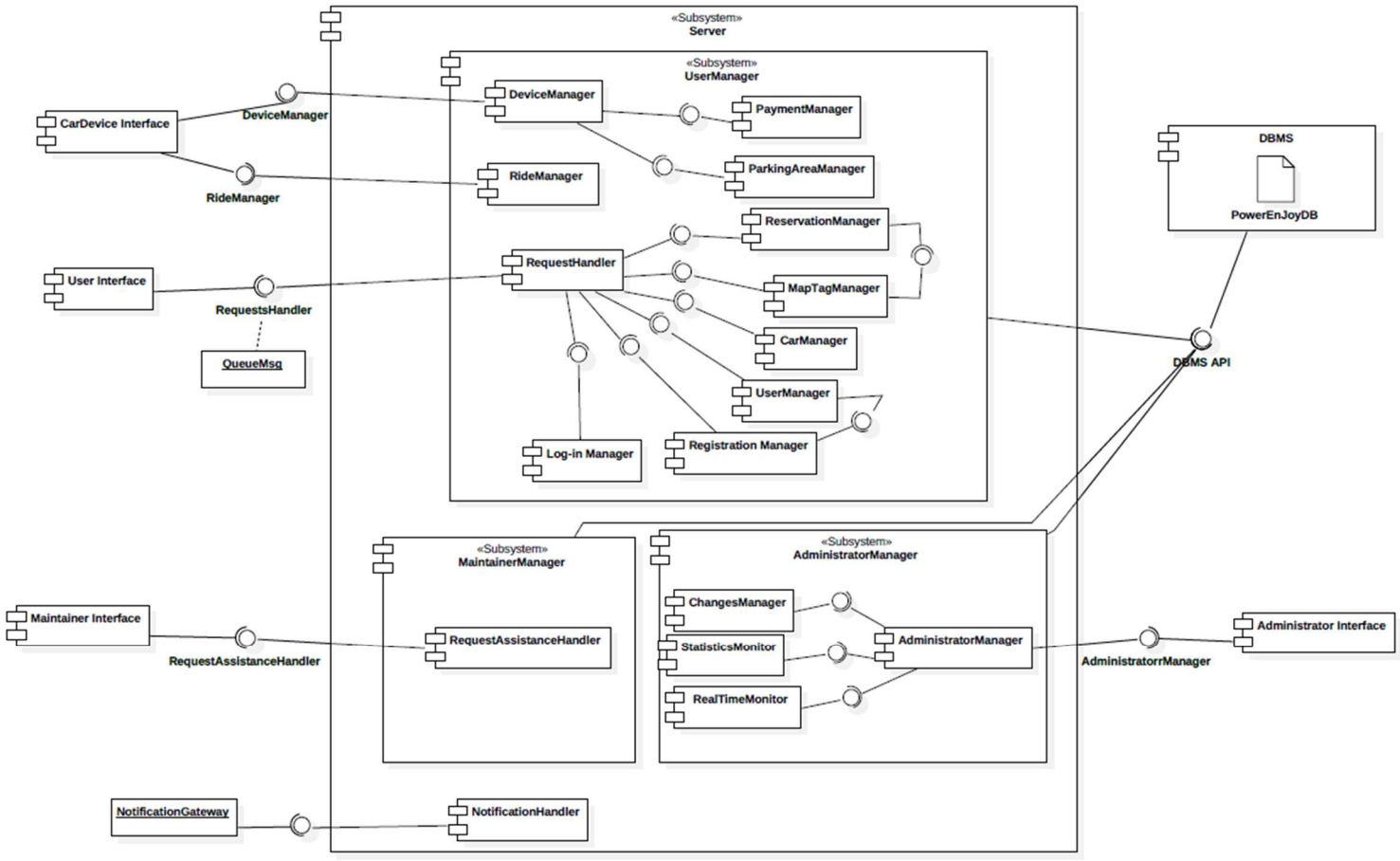


Figure 1

In this paragraph, we want to describe which are the element that need to be integrated in our system.

As you can see in the *Figure 1* our system is composed by different components and subsystems that represent the architecture in **two levels** of abstraction.

This structure is visible only in the Server side of our application because almost all the business logic of our application is in the Server while Client side we have only different types of interfaces.
At the lowest level, we'll integrate those components that are needed to build the various subsystems of the application, that in our case are:

- The RideManager, PaymentManagr, RequestHandler, ParkingAreaMAnager, Device Manager, ReservationManager, MapTagManager, CarManager, UserManager, RegistrationManager and Log-in Manager for the **UserManager Subsystem**
- The ChangesManagr, RealTimeMonitor, StatisticsMonitor and AdministratorManager for the **AdministratorManager Subsystem**

Then we proceed with the **high-level** integration part of our system that involves the integration of these subsystems with the DBMS component, the various interface components and the NotificationHandler.
It should be noted that **NotificationHandler** and **DBMS** are commercial components that have already been developed and can thus be immediately used in our integration test.

2.3 Integration testing strategy

Considered the described design architecture of PowerEnJoy application a two-level approach of integration testing should be suitable.
In particular, the integration phase will be realized at:

- *component level* : each component will be integrated and tested against every dependent component in the contest of the subsystem to which it belongs;
- *subsystems level* : once each subsystem is entirely integrated, all of them will be integrated and tested.

We decided to use this approach of integration because it can give us the following **advantages**:

- The integration at component level for each subsystem can be performed in **parallel** reducing the project overall time
- At the end of the first integration level (component level) the correctness and functionality of each subsystem is guaranteed.
- Different integration strategy can be selected for the different subsystems to be integrated.
- Following the integration test in the same way of the developing of the system and reducing of the complexity of the integration test thanks to the modularity of our application.

Considered the structure and characteristics of our system we think a proper solution is to adopt the **Bottom-up** strategy both for the component and the subsystem level because we only have 2 subsystems that once they are individually tested they need to be integrated with other single components.
The components in the lowest layer of the hierarchy are tested individually, then components belonging to the layer above are integrated and tested until the root of the hierarchy is reached.
If the upper module is not developed, a specific component, called **driver**, is used to simulate this module, initializing all the needed parameters and non-local variables.
The main advantages of this approach are:

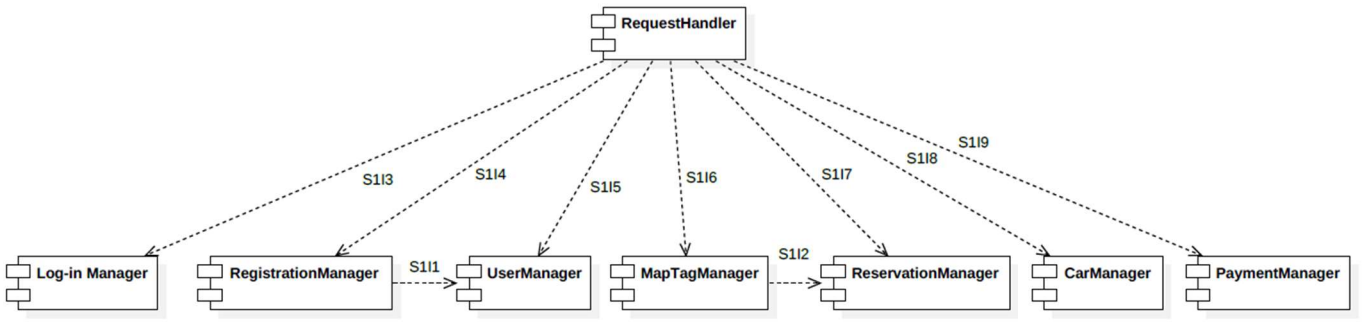
- May begin as soon as any leaf-level component is ready
- Work may proceed in parallel
- Although this pattern reduces stubbing, stubs may still be needed to break a cycle or simulate exceptions

2.4 Sequence of component integration

The components and subsystems will be integrated according to the bottom-up approach.
As we have already explained in the previous paragraph, the bottom-up means that first we test the components in the lowest level of hierarchy and then we proceed upwards until the root.
At the component level, we will perform the integration of all the components which compose the UserManager Subsystem and the AdministratorManager Subsystem.

2.4.1 UserManager Subsystem

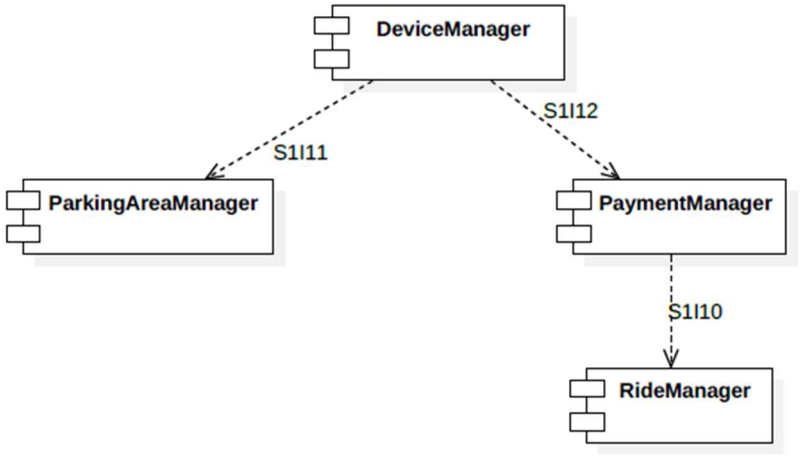
User Manager subsystem is composed by two areas, one which performs the interaction with PEJ mobile application, and the other with the devices on board of each car.
In the following graph we see how the Request Handler Component is integrated with the other components. As we are using a **bottom up** strategy of testing, the Request Handler will be substituted by a *driver* which takes care of calling different methods in the other components.



1. RequestHandler → RegistrationManager
2. RequestHandler → Log-in Manager
3. RequestHandler → UserManager
4. RequestHandler → MapTagManager
5. RequestHandler → ReservationManager

6. RequestHandler → CarManager
7. RequestHandler → PaymentManager
8. RegistrationManager →UserManager
9. MapTagManager → ReservationManager

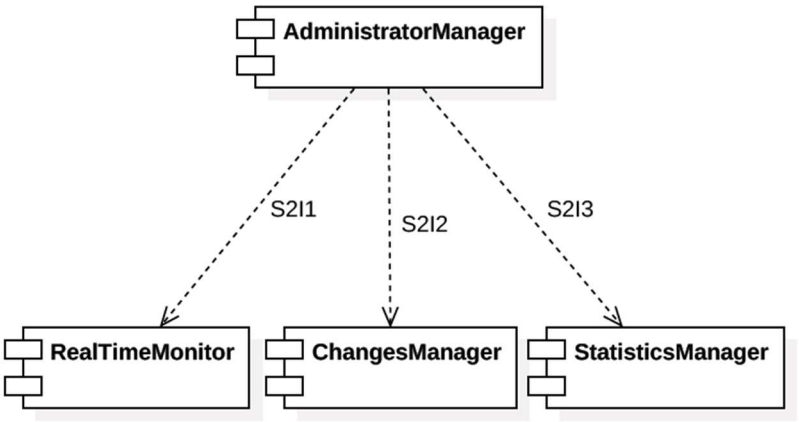
Device Manager and RideManager are the two components which interacts with the device on board of each car. RideManager is integrated with PaymentManager, which at this point has been already developed. Then ParkingAreaManager and PaymentManager are integrated with DeviceManager, which is temporary substituted by a *driver*.



10. PaymentManager → RideManager
11. DeviceManager → ParkingAreaManager
12. DeviceManager → PaymentManager

2.4.2 AdministratorManager Subsystem

In the Administrator Manager Subsystem, we perform the integration test creating a temporary *driver* which substitutes the Administrator Manager. We will use this driver in order to see if correct methods are called in the lower level components.



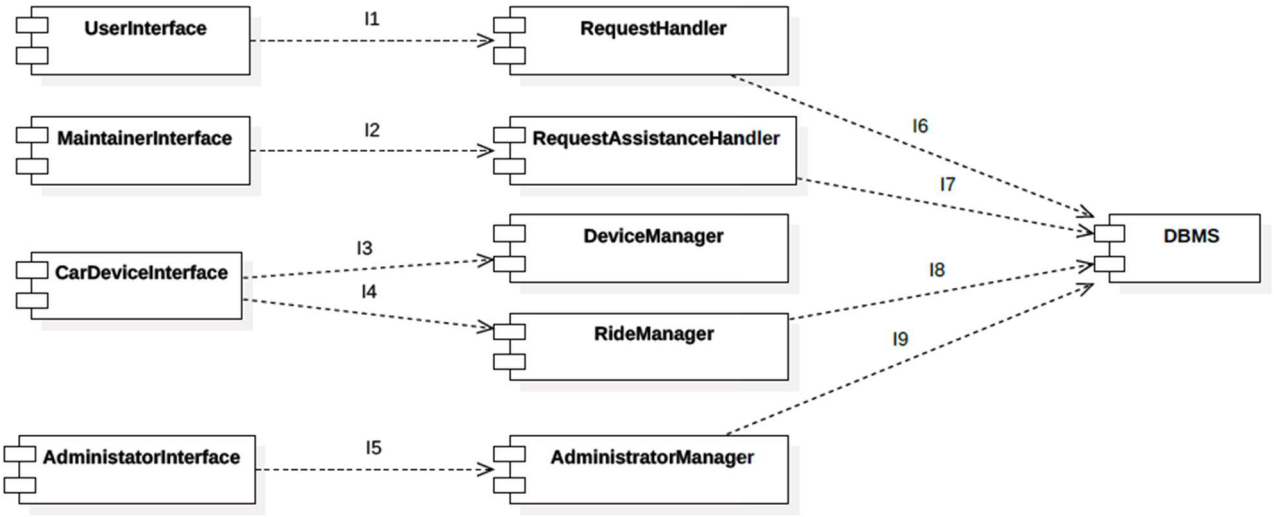
1. AdministratorManager → RealTimeMonitor
2. AdministratorManager → ChangesManager
3. AdministratorManager → StatisticsManager

2.4.3 High level integration

Now that the integration at the lower level is completed we can continue with the high level integration.

Business tier, composed by UserManager subsystem, AdministratorManager subsystem and Maintainer Manager Subsystem needs to be integrated with the client side of the application and with DBMS.

This integration is performed in the following way:



1. UserInterface → RequestHandler
2. MaintainerInterface → RequestAssistanceHandler
3. CarDeviceInterface → DeviceManager
4. CarDeviceInterface → RideManager
5. AdministratorInterface → AdministratorManager
6. RequestHandler → DBMS
7. RequestAssistanceHandler → DBMS
8. RideManager → DBMS
9. AdministratorManager → DBMS

3 Individual Steps and Test Description

In this chapter, we are going to describe the integration steps and the tests that we have defined in the previous chapter.
As for the integration at different levels of abstraction, according to the bottom-up approach, we have decided to define a test case for each integration between two components.
We describe what are the requirements needed to perform valid tests, from the component at the bottom of our component view, to the ones on the top levels.
Obviously, as the top components have not been ready yet, we need some drivers that simulate the connection with the other components needed to the test.

3.1 Subsystem 1 (UserManager Subsystem)

Test case ID	S1I1
Test Items	RegistrationManager → UserManager
Input Specification	Create a typical RegistrationManager input
Output Specification	Check if correct methods are called in UserManager
Environmental needs	RegistrationManager driver, DBMS

Test case ID	S1I2
Test Items	MapTagManager → ReservationManager
Input Specification	Create a typical MapTagManager input
Output Specification	Check if correct methods are called in ReservationManager
Environmental needs	MapTagManager driver, DBMS

Test case ID	S1I3
Test Items	RequestHandler → RegistrationManager
Input Specification	Create an input request of registration for the RequestHandler
Output Specification	Check if the correct methods are called in the Registration Manager
Environmental needs	RequestHandler Subsystem driver, DBMS

Test case ID	S1I4
Test Items	RequestHandler → Log-in Manager
Input Specification	User's username and password, received by the Request Handler.
Output Specification	Check if the correct methods are called in the Log-in Manager
Environmental needs	Request Handler Driver, DBMS

Test case ID	S1I5
Test Items	RequestHandler → UserManager
Input Specification	Create a typical request of personal data modification for RequestHandler
Output Specification	Check if the correct methods are called in the UserManager
Environmental needs	UserManager driver, DBMS

Test case ID	S1I6
Test Items	RequestHandler → MapTagManager
Input Specification	Starting position on the map is given to the RequestHandler,
Output Specification	Check if the correct methods are called in the MapTagManager in order to find cars around the input position.
Environmental needs	RequestHandler driver, DBMS

Test case ID	S1I7
Test Items	RequestHandler → ReservationManager
Input Specification	Create an input request of reservation for the RequestHandler
Output Specification	Check if the correct methods are called in the ReservationManager
Environmental needs	RequestHandler driver, DBMS

Test case ID	S1I8
Test Items	RequestHandler → CarManager
Input Specification	Create a typical input request of locking or unlocking a Car for the RequestHandler
Output Specification	Check if the correct methods are called in the CarManager
Environmental needs	RequestHandler driver

Test case ID	S1I9
Test Items	RequestHandler → PaymentManager
Input Specification	Create a typical PaymentManager input
Output Specification	Check if correct methods are called in RideManager
Environmental needs	Payment Manager Driver, DBMS

Test case ID	S1I10
Test Items	PaymentManager → RideManager
Input Specification	Create a typical PaymentManager input
Output Specification	Check if correct methods are called in RideManager
Environmental needs	PaymentManager driver, DBMS

Test case ID	S1I11
Test Items	DeviceManager → ParkingAreaManager
Input Specification	Create a typical DeviceManager input
Output Specification	Check if correct methods are called in ParkingAreaManager
Environmental needs	DeviceManager driver, DBMS

Test case ID	S1I12
Test Items	DeviceManager→ PaymentManager
Input Specification	Create a typical DeviceManager input
Output Specification	Check if correct methods are called in ParkingAreaManager
Environmental needs	DeviceManager driver, DBMS

3.2 Subsystem 2 (AdministratorManager Subsystem)

Test case ID	S2I1
Test Items	AdministratorManager → RealTimeMonitor
Input Specification	Create a typical AdministratorManager input
Output Specification	Check if the correct methods are called in the RealTimeMonitor
Environmental needs	AdministratorManager Driver, DBMS

Test case ID	S2I2
Test Items	AdministratorManager → ChangesManager
Input Specification	Create a typical AdministratorManager input
Output Specification	Check if the correct methods are called in the ChangesManager
Environmental needs	AdministratorManager Driver, DBMS

Test case ID	S2I3
Test Items	AdministratorManager → StatisticsManager
Input Specification	Create a typical AdministratorManager input
Output Specification	Check if the correct methods are called in the StatisticsManager.
Environmental needs	AdministratorManager Driver, DBMS

3.3 High Level Integration

Test case ID	I1
Test Items	UserInterface → RequestHandler
Input Specification	Create a typical UserInterface input
Output Specification	Check if the correct methods are called in the RequestHandler
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I2
Test Items	MaintainerInterface → RequestAssistanceHandler
Input Specification	Create a typical MaintainerInterface Application input
Output Specification	Check if the correct methods are called in the RequestAssistanceHandler
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I3
Test Items	CarDeviceInterface → DeviceManager
Input Specification	Create a typical CarDeviceInterface input
Output Specification	Check if the correct methods are called in the DeviceManager
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I4
Test Items	CarDeviceInterface → DeviceManager
Input Specification	Create a typical CarDeviceInterface input
Output Specification	Check if the correct methods are called in the RideManager
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I5
Test Items	AdministratorInterface → AdministratorManager
Input Specification	Create a typical AdministratorInterface input
Output Specification	Check if the correct methods are called in the AdministratoManager
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I6
Test Items	RequestHandler → DBMS
Input Specification	Create a typical RequestHandler input
Output Specification	Check if the correct methods are called in the DBMS
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I7
Test Items	RequestAssistanceHandler → DBMS
Input Specification	Create a typical input for RequestAssistanceHandler
Output Specification	Check if the correct methods are called in the DBMS
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I8
Test Items	RideManager→ DBMS
Input Specification	Create a typical RideManager input
Output Specification	Check if the correct methods are called in the DBMS
Environmental needs	Lower level integration succeeded, DBMS

Test case ID	I9
Test Items	AdministratorManager → DBMS
Input Specification	Create a typical AdministratorManager input
Output Specification	Check if the correct methods are called in the DBMS
Environmental needs	Lower level integration succeeded, DBMS

4 Tools required

Until now, we have developed our project thinking of Java EE as programming language for the implementation.
For this reason, we have decided to use *Arquillian*, the **integration testing framework** for Java EE, in order to test our components interactions and interfaces.
Arquillian is a testing framework, developed at JBoss.org, that provides a simple test harness that developers can use to produce a broad range of integration tests for their Java applications (most likely enterprise applications). This tool allows us to understand if the component works right and if the interaction with the database is properly working. It 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.
The big advantage of Arquillian is the way it facilitates the testing of Java enterprise applications in the environment that they will eventually run in. This method tends to be better than introducing mock objects that won't always behave like the real ones.
Another advantage of Arquillian is its ability to create "**micro deployments**" around your test. These are sub-sections of your application logic.
The fluent API provided by the **ShrinkWrap** project makes this technique possible. Micro deployments let you conduct testing on low level integrations and the ShrinkWrap API gives you fine-grained control over which resources are available to be tested.
Arquillian combines a unit testing framework (JUnit or TestNG), ShrinkWrap, and one or more supported target containers (Java EE container, servlet container, Java SE CDI environment, etc)
to provide a simple, flexible and pluggable integration testing environment. Beside Arquillian functionalities JUnit features (like assertions, annotations...) can be still exploited to write ad-hoc integration testing procedures.

5 Program stub and test data required

During the integration testing, if there are any uncompleted functionalities or components, we can replace them with temporary dummy programs which are called "stubs" or **"drivers".**

Stubs are used in the top-down integration testing approach, when the top module is ready to be tested, but the sub-modules are not ready yet.

On the other hand, drivers are the "calling" programs and are the ones used in the bottom-up integration testing approach. A driver is dummy code, used when the sub-modules are ready but the main module is not ready yet.

As already explained before, we adopted a **bottom-up approach** and we have used some drivers to successfully complete the integration.

In particular, in our integration process, we needed the RequestHandler driver, RegistrationManager driver, RequestHandler Subsystem driver, UserManager driver, MapTagManager driver, DeviceManager driver and the AdministratorManager driver, as already reported in the environmental needs row of the test cases table in Chapter 3.

6 References

6.1 Used Tools

To draw up this document we've used the following tools:

- Microsoft Office World 2016: to redact and format this document
- StarUML: to create Component diagram and the various component diagrams for the integration test.
- Paint: to create and adapt the images of this document
- GitHub: to save and control the version of the document

6.2 Hours of work

Date	Antonio's hours	Davide's hours
2016/12/27	4h	1.30h
2016/12/28	3h	--
2016/12/29	5h	5h
2017/01/03	6h	6h
2017/01/04	--	2.30h
2017/01/08	3h	3h
2017/01/09	3h	3h
2017/01/15	2h	2h
Total	**26h**	**23h**

Hours for review:

Date	Antonio's hours	Davide's hours
Total		