POLITECNICO
MILANO 1863

Politecnico di Milano

COMPUTER SCIENCE AND ENGINEERING

Software Engineering 2

Integration Test Plan Document

PowerEnJoy

Authors:
Francesco Fabiani
Jagadesh Manivannan
Niccolò Pozzolini

Professors:
Elisabetta Di Nitto
Luca Mottola

Academic Year 2016/2017

Contents

Chapter 1

Introduction

1.1 Revision history

Version	Date	Authors	Summary
1.0	15/01/2017	Fabiani, Manivannan, Pozzolini	Initial release

Table 1.1: Changelog of this document

1.2 Purpose and scope

The Integration Test Plan Document (ITPD) is intended to indicate the necessary tests in order to verify that all the components of the previously described system are properly integrated. This process is critically important to ensure that the unit-tested modules interact correctly.

The product described is PowerEnJoy, a car-sharing service which offers to its users exclusively electric cars. It includes the common functionalities of its category: permitting to registered users to obtain the position of all the available cars, reserving one within a certain amount of time and continuously displaying the up-to-the-minute cost of the ride are just few of them. Moreover, PowerEnJoy stimulates users to behave virtuously towards the ecosystem by applying various types of discounts under specific conditions.

1.3 Definitions, acronyms, abbreviations

- *API*: Application Programming Interface
- *BCE*: Business Controller Entity

- *Car*: electric vehicle provided by the service
- *DB*: Database
- *DBMS*: Database Management System
- *DD*: Design Document
- *ER*: Entity-Relationship
- *GPS*: Global Positioning System
- *Guest* or *Guest user*: person not registered to the service
- *ITPD*: Integration Test Plan Document
- *MVC*: Model View Controller
- *OS*: Operating System, related both to desktop and mobile platforms
- *PIN*: Personal Identification Number
- *RASD*: Requirements Analysis and Specification Document
- *Registered user*: see *User*
- *REST*: REpresentational State Transfer
- *RESTful*: that follows the REST principles
- *Safe area*: set of parking spots where a user can leave a car without penalization
- *User*: person with a valid driving license registered to the service
- *UX*: User eXperience
- *W3C*: World Wide Web Consortium

1.4 Reference documents

The Integration Test Plan Document has been composed following the indications reported in the previous documents delivered for this project: the Requirements Analysis and Specification Document, describing fundamental aspects of PowerEnJoy such as domain assumptions, goals, functional and non-functional requirements, and the Design Document, which shows more accurately all the functionalities provided by focusing on the software design of the system.

With regards to the course named Software Engineering 2 and held by professors Luca Mottola and Elisabetta Di Nitto (Politecnico di Milano, a. y. 2016/17), the document conforms to the guidelines provided during the lectures and within the material of the course.

Chapter 2

Integration strategy

2.1 Entry criteria

This section shows the conditions that must be met before starting the integration in order to obtain significant results.

First, it is crucial that the RASD and DD documents are completely composed, so that a whole vision of the components of the system and their functionalities is available.

As regards the individual components, the development must go forward along with their unit testing, so that the new modules implemented do not interfere with the solidity if the system. For this reason, every component should have at least 90% of its functionalities completed before the integration with other components is tested.

Moreover, the integration process should start when the following percentages of development are achieved:

- 100% of the database and availability helper components
- at least 80% of the controller components
- at least 90% of the payment components
- at least 50% for the client application

The decision of requiring different amounts of functionalities according to the component is based on the order the integration will take place and on the time needed to accomplish the integration testing phase of each one.

2.2 Elements to be integrated

The integration tests are planned to be performed following the architectural division in layers given in the Design Document. Therefore this phase is divided in the following three steps, each for every subsystem:

Database. It represents the layer where data are stored and handled. The components to be integrated are:

- Database
- Availability Helper

Application server. It includes all the business logic and provides web services so that users can interact with this layer from both the interfaces. The components to be integrated are:

- Map Manager
- Selection Controller
- Reservation Controller
- Ride Controller
- Parking Controller
- Discount Controller
- Selection Controller
- Account Manager

Client. It includes all the implementations, that is to say the mobile App, the web app and the on-board application. These modules must be integrated individually with the layers they interface with.

2.3 Integration testing strategy

The items to be tested consist of the integration of the code modules developed for the Power Enjoy project. For testing we choose the bottom-up approach. This means that integration testing starts at the bottom level.

Using the bottom-up approach, we will start integrating together those components that do not depend on other ones to work, or that only depend on already developed components. We chose this because the top level component when built has to be tested from the bottom level components in our project, i.e. we can simply say that it has few dependencies on the bottom level components for testing.

Moreover, working bottom-up allows us to follow more carefully the development process and the developers to start performing integration testing earlier in the development process as soon as the required components have been developed, so that parallelism and efficiency are maximized.

We want to test using the real values and functionalities. The integration tests described in this documents are at the component level. The integration tests of lower level code modules are described in the corresponding components unit test.

2.4 Sequence of component/Function integration

In this section we are going to describe the order of integration (and integration testing) of the various components and subsystems of Power Enjoy. As a notation, an arrow going from component C1 to component C2 means that C1 is necessary for C2 to function and so it must have already been implemented.

2.4.1 Software integration sequence

Following the already mentioned bottom-up approach, we now describe how the various subcomponents are integrated together to create higher level subsystems.

Database Management System

The very basic elements that needs to be integrated and tested are Database and Availability Helper. We test this first because of the bottom up approach and all the other subsystems components relies in this structure.

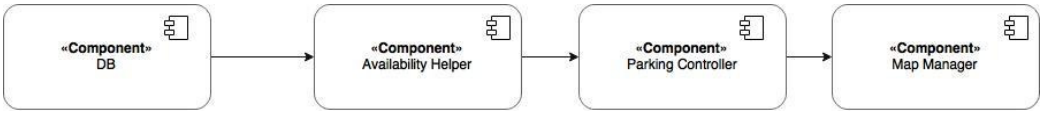
Controller System

The second step is the integration of the components of the Controller System. Most of the important operations are assumed to be executed/performed

by the components of the controller system. We proceed further by showing which components are executed or integrated in sequence.

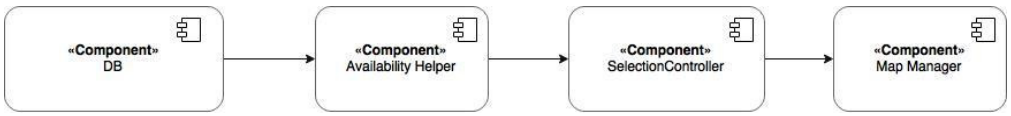
First, we proceed by integrating the Selection Controller with the database and Availability Helper.

Then the same procedure is followed by replacing the selection controller with other components of the controller system in the following sequence: Reservation Controller, Ride controller, Parking Controller, Discount Controller.



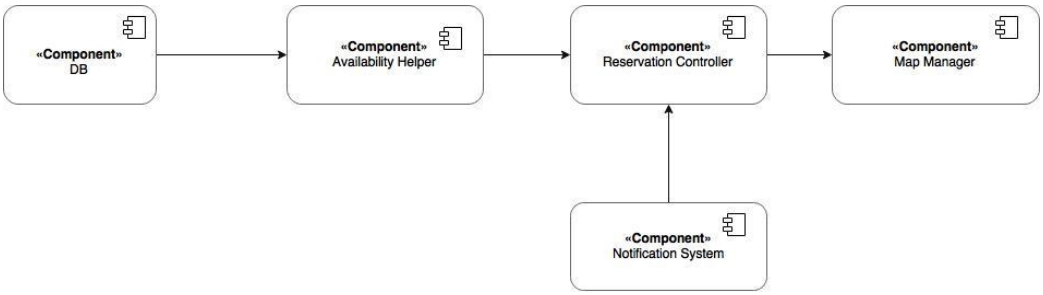
Map Manager System

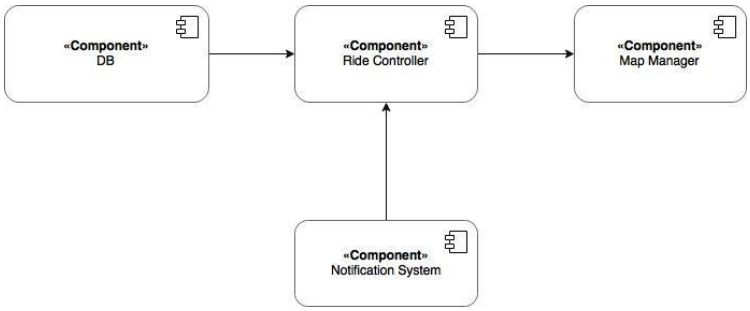
Generally the communication between the controllers happens through the Map Manager. So, the Map Manager has to be integrated with the Controllers for their communication and also for interaction of the Client with the system.

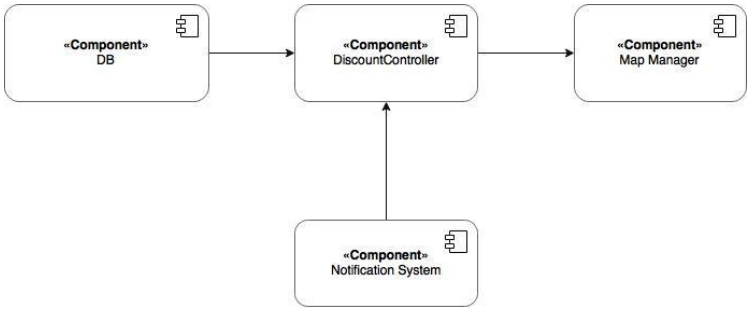


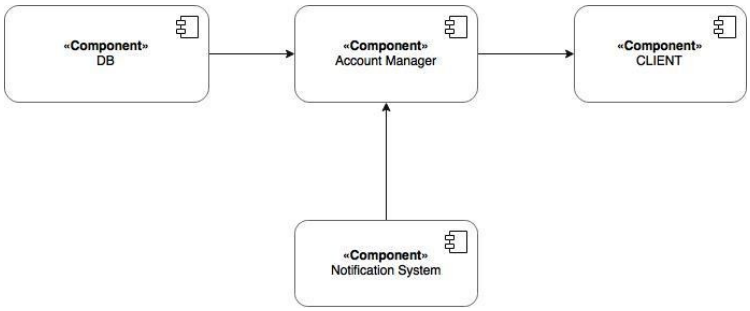
Notification System

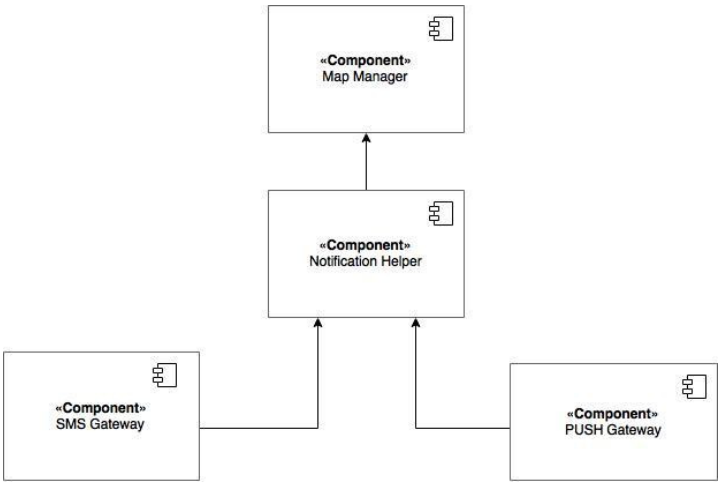
The next integration test is executed with the notification system. This notification system is developed by the third party and we just integrate and test here the functionalities. For example: after reserving a car, the user is notified with a confirmation acknowledgement.


«Component»
DB
«Component»
Availability Helper
«Component»
Reservation Controller
«Component»
Map Manager
«Component»
Notification System


«Component»
DB
«Component»
Ride Controller
«Component»
Map Manager
«Component»
Notification System

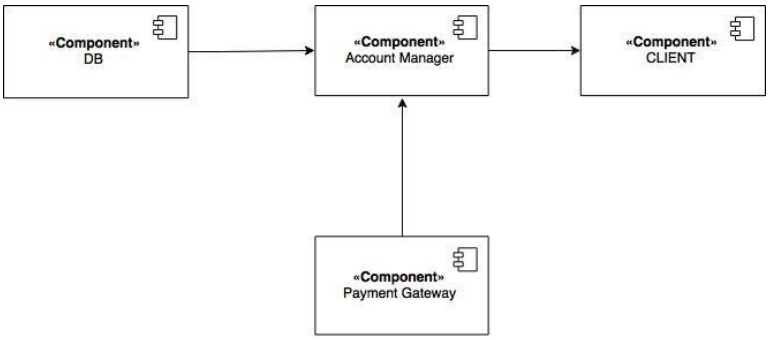

«Component»
DB
«Component»
DiscountController
«Component»
Map Manager
«Component»
Notification System


«Component»
DB
«Component»
Account Manager
«Component»
CLIENT
«Component»
Notification System



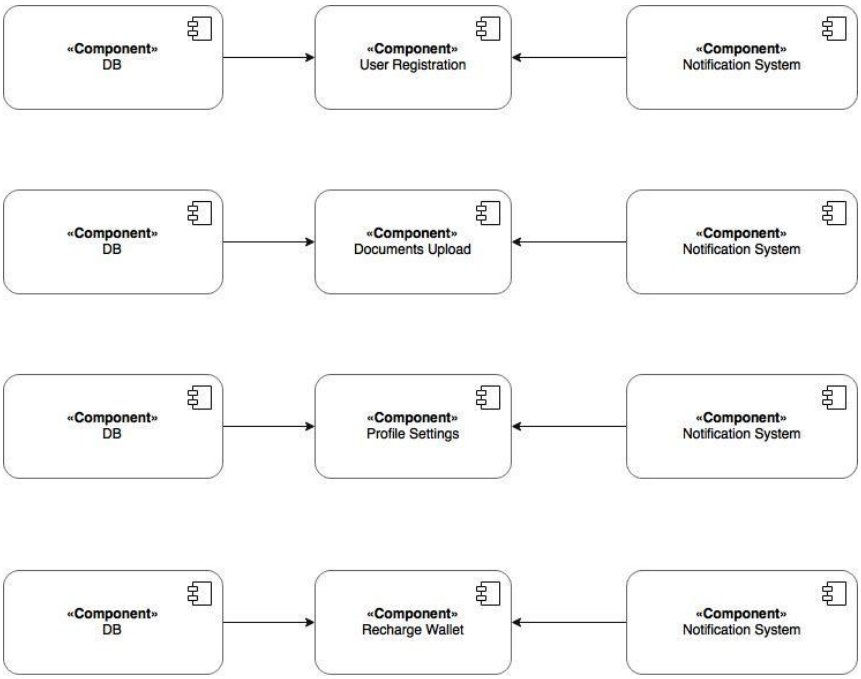
Payment System

The payment system is same like the notification system as it is developed by the third party and we just integrate and test here the functionalities.



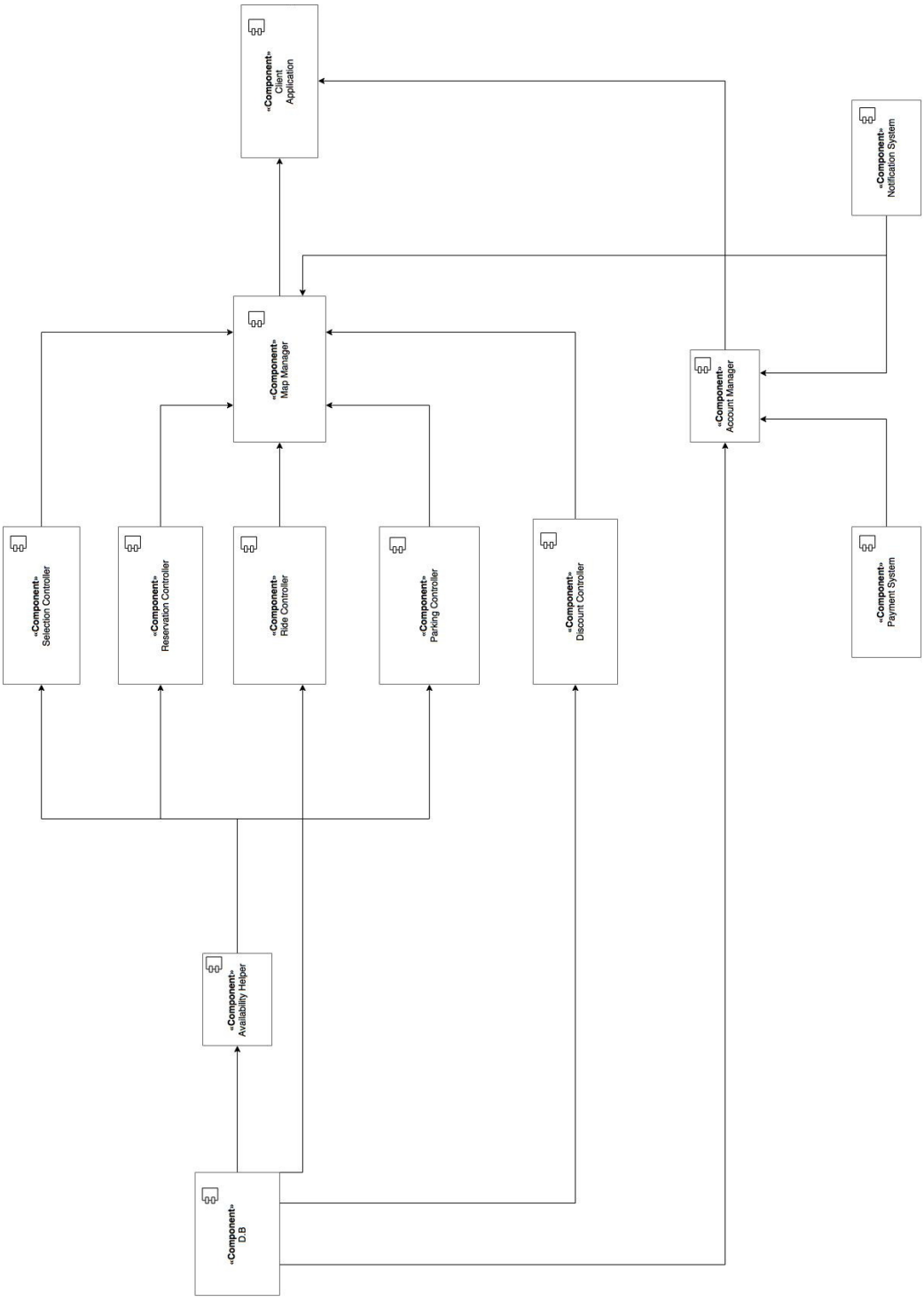
Account Manager System

After the integration of the above systems, the account manager is integrated and tested here. The account manager has information about the user details and allows him/her to edit personal informations, upload documents and other services.



2.4.2 Subsystem integration sequence

Finally test integrate and test the Client system with the overall integrated systems. The high level subsystems are integrated together and the integration architecture of the Power Enjoy service is shown here.


«Component»
Client
Application
«Component»
Notification System
«Component»
Map Manager
«Component»
Account Manager
«Component»
Selection Controller
«Component»
Reservation Controller
«Component»
Ride Controller
«Component»
Parking Controller
«Component»
Discount Controller
«Component»
Payment System
«Component»
Availability Helper
«Component»
D.B

Chapter 3

Individual steps and test description

In this chapter we are going to describe the tests that need to be executed for every couple of components which interact with each other. Since the controllers communicate through the MapManager, the Integration Testing proceeds in the same way.

3.1 Integration test case I1

Test case identifier	I1T1
Test item(s)	Database → Availability Helper
Input specification	Create typical SQL query (Database inputs)
Output specification	Check if the correct functions are called in the Availability Helper
Environmental needs	Necessary input parameters for testing

3.2 Integration test case I2

Test case identifier	I2T1
Test item(s)	Availability Helper → Selection Controller
Input specification	–
Output specification	–
Environmental needs	I1 succeeded

checkAvailability(zone)	
Input	*Effect*
A valid zone	Returns a list containing all the available cars in that zone
An invalid zone	An InvalidZoneException is generated
An null zone	A NullZoneException is generated

3.3 Integration test case I3

Test case identifier	I3T1
Test item(s)	Availability Helper → Reservation Controller
Input specification	–
Output specification	–
Environmental needs	I1 succeeded

changeTagRequest(car, tag) returns response	
Input	*Effect*
A valid car and a valid tag	If parameter tag != old(car.tag) returns positive response. Otherwise an UnconsistentChangeException is generated (in this case most of the times it happens because someone else has been quicker than the user to reserve a car)
An invalid tag	An InvalidTagException is generated

3.4 Integration test case I4

Test case identifier	I4T1
Test item(s)	Availability Helper → Parking Controller
Input specification	–
Output specification	–
Environmental needs	I1 succeeded

getSpecialParkingAreas(destination) returns SpecialParkingAreas	
Input	*Effect*
A valid destination	Returns a possibly empty SpecialParkingArea list
An invalid destination	An InvalidDestinationException is generated

3.5 Integration test case I5

Test case identifier	I5T1
Test item(s)	Selection Controller → Map Manager
Input specification	Post a SelectionController request
Output specification	Check if the request is correctly routed
Environmental needs	I1, I2 succeeded

Test case identifier	I5T2
Test item(s)	Reservation Controller → Map Manager
Input specification	Post a ReservationController request
Output specification	Check if the request is correctly routed
Environmental needs	I1, I2, I3 succeeded

placeReservation(car) returns response	
Input	*Effect*
A valid car	ReservationController will invoke Availability-Helper.changeTagRequest() and get the response R. R is then returned
A null or invalid car	A NullCarException or InvalidCarException is generated

getReservation(user) returns Reservation	
Input	*Effect*
The user logged to the client	ReservationController looks in the database for an active reservation made by the user. If found its returned, otherwise MissingReservationException is generated

Test case identifier	I5T3
Test item(s)	Ride Controller → Map Manager
Input specification	Post a RideController request
Output specification	Check if the request is correctly routed
Environmental needs	I1, I2, I3 succeeded

startRide(reservation, userLocation) returns response	
Input	*Effect*
An active reservation, checked by ReservationController .getReservation(u), and userLocation	If the user is not close to the car a NotCloseException is thrown. Otherwise the Ride, Reservation and Car objects are updated and a positive response is returned

getRide(user)	
Input	*Effect*
The user logged to the client	ReservationController looks in the database for an active ride of the user. If found its returned, otherwise MissingRideException is generated

Test case identifier	I5T4
Test item(s)	Parking Controller → Map Manager
Input specification	Post a ParkingController request
Output specification	Check if the request is correctly routed
Environmental needs	I1, I2, I3 succeeded

Test case identifier	I5T5
Test item(s)	Discount Controller → Map Manager
Input specification	Post a DiscountController request
Output specification	Check if the request is correctly routed
Environmental needs	I1, I2, I3, I4 succeeded

endRide(ride) returns response	
Input	*Effect*
A valid Ride	If the current location of the car is not a legal spot to park, an IllegalSpotException is generated, otherwise the Ride and Car objects are updated, DiscountController.getPrice(Ride) is called, the user's credit updated and a positive response returned
An invalid Ride	An InvalidRideException is generated
An ongoing Ride	A NotParkedException is generated

3.6 Integration test case I6

Test case identifier	I6T1
Test item(s)	Map Manager ← Client Application
Input specification	Create typical User/Client inputs
Output specification	Check if the correct functions are called in the Map Manager
Environmental needs	I1, I2, I3, I4, I5 succeeded

3.7 Integration test case I7

Test case identifier	I7T1
Test item(s)	SMS Gateway → Notification Helper
Input specification	Create typical SMS Gateway inputs
Output specification	Check if the SMS is delivered
Environmental needs	I1, I2, I3, I4, I5, I6 succeeded

3.8 Integration test case I8

Test case identifier	I8T1
Test item(s)	PUSH Gateway → Notification Helper
Input specification	Create typical PUSH Gateway inputs
Output specification	Check if the correct functions are called in the PUSH Gateway
Environmental needs	I1, I2, I3, I4, I5, I6 succeeded

3.9 Integration test case I9

Test case identifier	I9T1
Test item(s)	Notification Helper → Map Manager
Input specification	–
Output specification	–
Environmental needs	I1, I2, I3, I4, I5, I6, I7, I8 succeeded

3.10 Integration test case I10

Test case identifier	I10T1
Test item(s)	Database → Account Manager
Input specification	Create typical SQL query (Database inputs)
Output specification	Check if the correct functions are called in the Availability Helper
Environmental needs	–

3.11 Integration test case I11

Test case identifier	I11T1
Test item(s)	Payment Gateway → Account Manager
Input specification	Create typical Payment Gateway inputs
Output specification	Check if the payment process correctly executes
Environmental needs	I10 succeeded

3.12 Integration test case I12

Test case identifier	I12T1
Test item(s)	Account Manager ← Client Application
Input specification	–
Output specification	–
Environmental needs	I10, I11 succeeded

Chapter 4

Program stubs and test data required

As we have mentioned in the Integration Testing Strategy section, we adopted the bottom-up approach for the components integration and testing. For this reason the usage of stubs is minimized, but a few number of drivers is needed to perform the necessary method invocations of the integrated components that are going to be tested.

Below the reader can find the list of the required drivers for the integration test plan.

- *AvailabilityHelper driver*. This testing module will invoke the methods exposed by the AvailabilityHelper component in order to test its interaction with the database management system
- *Controllers driver*. This testing module will invoke the methods exposed by the Controllers (acting like MapManager) in order to test its interaction with each other. A stub acting like the Client is needed to fulfill the request/response protocol needed for client/system interaction

Chapter 5

Effort spent

5.1 Francesco Fabiani

- 20/12/2016: 1h30min
- 22/12/2016: 1h
- 23/12/2016: 2h
- 29/12/2016: 1h
- 30/12/2016: 2h
- 03/01/2017: 1h30min
- 05/01/2017: 2h30min
- 06/01/2017: 1h
- 08/01/2017: 1h
- 10/01/2017: 2h
- 12/01/2017: 1h30min
- 14/01/2017: 3h
- 15/01/2017: 2h
- 16/01/2017: 2h

5.2 Jagadesh Manivannan

- 18/12/2016: 1h
- 19/12/2016: 1h30min
- 21/12/2016: 2h
- 23/12/2016: 2h30min
- 27/12/2016: 1h30min
- 28/12/2016: 2h
- 29/12/2016: 1h30min
- 30/12/2016: 1h30min
- 02/01/2017: 2h
- 03/01/2017: 2h
- 04/01/2017: 1h30min
- 06/01/2017: 1h
- 07/01/2017: 3h
- 08/01/2017: 2h
- 10/01/2017: 1h
- 11/01/2017: 1h30min
- 13/01/2017: 2h
- 14/01/2017: 2h30min

5.3 Niccolò Pozzolini

- 20/12/2016: 1h
- 21/12/2016: 2h
- 22/12/2016: 2h
- 28/12/2016: 2h

- 29/12/2016: 1h30min
- 30/12/2016: 1h
- 02/01/2017: 1h
- 03/01/2017: 2h
- 04/01/2017: 3h
- 05/01/2017: 1h30min
- 07/01/2017: 2h
- 10/01/2017: 2h
- 11/01/2017: 1h
- 12/01/2017: 2h
- 14/01/2017: 2h30min
- 15/01/2017: 2h