myTaxiService

Integration **T**est **P**lan **D**ocument

A.Y. 2015/2016

Politecnico di Milano

Version 1.0

Cattaneo Michela Gaia, matr. 791685
Barlocco Mattia, matr. 792735

January 22, 2016

Contents

1 Introduction

1.1 Revision History

1.2 Purpose and Scope

1.2.1 Purpose

This document represents the Integration Test Plan Document. It aims at specifying the plan for the integration testing of myTaxiService application, ensuring that all its modules interacts properly.
This document is intended for the development team, that needs to know what it is needed to test, in which sequence it should occur and with which tools.

1.2.2 Scope

myTaxiService is an application that simplifies the access of the customers to the taxi service, managing also the taxi driver's distribution over the city and the queues of the areas. The customer can use the mobile application or the web app to request or reserve a taxi and a notification is forwarded to the designated taxi driver, who is going to answer.
This system should always be responsive and reliable, in order to keep up with all the requests of the passenger and the notifications forwarding.
This is the reason why the components that comprise the myTaxiService system are meant to be well integrated and tested.

1.3 List of Definitions and Abbreviations

- **Definitions**

- **User**: a person who requests a service from the system. It can be a visitor or a passenger.

- **Visitor**: a person who is not registered in the application.

- **Passenger**: a person who is registered in the application.

- **Taxi driver**: a taxi driver who access the application with a specific ID.

- **Request**: the request of a taxi in a certain area and position in the city made by a user.

- **Reservation**: the reservation of a taxi in a certain area, place and time that can be made only by passengers.

- **Component**:

- **Subsystem**:

- **Acronyms and abbreviations**

- **RASD**: Requirement Analysis and Specification Document

- **DD**: Design Document

- **JEE**: Java Enterprise Edition

- **JVM**: Java Virtual Machine

1.4 List of Reference Documents

- Project Description and Rules (https://github.com/MichelaCattaneo/myTaxiService/blob/master/Project%20Description%20And%20Rules.pdf)
- Requirements Analysis and Specification Document (https://github.com/MichelaCattaneo/myTaxiService/blob/master/Deliveries/RASD_1.1.pdf)
- Design Document (https://github.com/MichelaCattaneo/myTaxiService/blob/master/Deliveries/DD.pdf)
- Integration Test Plan Example (https://beep.metid.polimi.it/documents/3343933/5b3768d0-d949-4369-87e1-7a31b6943726)

2 Integration Strategy

2.1 Entry Criteria

These are the criteria that must be respected before the integration testing phase may begin:

- Requirement Analysis and Specification Document is complete and revised
- Design Document is complete and revised
- Code Inspection Document is complete and revised
- The code is complete and bugs free
- The product satisfies the requirements and the assumptions specified in the RASD
- The product satisfies the architecture and the design specified in the DD
- Test environment, test cases and test data are ready

2.2 Elements to be Integrated

These components refer to the ones specified in the Component View in chapter 2.3 of the DD. This diagram shows how these components have to be integrated and the order of integration, according to the strategy adopted.

2.3 Integration Testing Strategy

The integration testing strategy that has been chosen exploits the bottom-up approach. This strategy is the most suitable for the myTaxiService system, in fact it makes it easier to find bugs, starting from the most critical components. The bottom-up approach starts from the lowest layers of the system, testing the basic functionalities at the beginning, then moving forward the most abstract layers, such as the client interfaces modules.
In this case, it is only necessary to use drivers, in order to simulate the top layers during the testing, which are a lot easier to produce with respect to the stubs used in the top-down approach.

2.4 Sequence of Component/Function Integration

2.4.1 Software Integration Sequence

In this section the system is presented already divided in the main three subsystems: Manager, Taxi Driver Client and Passenger Client.

- **Integration Test of the "Manager" subsystem** qui metteremo l'immagine

ID	Integration Test	Paragraph
I1	RequestManager → Database	3.1
I2	PassengerAreaManager → Database	3.2
I3	TaxiDriverAreaManager → Database	3.3
I4	AccessManager → Database	3.4
I5	RequestManager → PaymentManager	3.5

- **Integration Test of the "Taxi Driver Client" subsystem** qui metteremo l'immagine

ID	Integration Test	Paragraph
I6	Notification → RequestManager	3.6
I7	TaxiDriverArea → Notification	3.7
I8	TaxiDriverArea → TaxiDriverAreaManager	3.8
I9	AccessManager → TaxiDriverArea	3.9
I10	Access → AccessManager	3.10

- **Integration Test of the "Passenger Client" subsystem** qui metteremo l'immagine

ID	Integration Test	Paragraph
I11	ChoosePaymentMethod → PaymentManager	3.11
I12	RequestManager → ChoosePaymentMethod	3.12
I13	MakeReservation → RequestManager	3.13
I14	RequestTaxi → RequestMager	3.14
I15	PassengerArea → RequestTaxi	3.15
I16	PassengerArea → MakeReservation	3.16
I17	PassengerArea → PassengerAreaManager	3.17
I18	AccessManager → PassengerArea	3.18
I19	Access → AccessManager	3.19

2.4.2 Subsystem Integration Sequence

The subsystems will be integrated in this order:

1. **Manager**: According to the strategy adopted, this is the first subsystem that is going to be integrated, as it is the lowest level of the myTaxiService system. It is necessary to test this part first, as long as this part manages the other components and it is crucial that this part does integrates perfectly with the others.

2. **Taxi Driver Client**: This is the second subsystem that will be integrated.

3. **Passenger Client**: This is the last subsystem that will be integrated.

There is no real need to test the taxi driver client before or after the passenger client, as they are both top level modules that have to be integrated in the last steps.

3 Individual Steps and Test Description

3.1 Integration Test case I1

Test case identifier	I1
Test item(s)	RequestManager → Database
Input specification	
Output specification	
Environmental needs	

4 Tools and Test Equipment Required

Integration testing is usually performed after unit testing has been done. Once all the individual units are created and tested, it is necessary to start combining these modules one by one and test their behavior as a combined unit.
Therefore, unit testing is needed to accomplish the integration, in order to already have a visual feedback and all the possible problems sorted out. A tool that can be used is **Mockito**, to create unit testing mockups useful for having dependencies identified and results predicted. For the actual tests on the parts of code, **JUnit** and **testNG** are good solution frameworks for Java programming language.
As regards the proper integration testing tools, there are some possible solutions, presented in the following sections.

4.1 Arquillian and ShrinkWrap.

Arquillian is an innovative, highly extensible and flexible testing platform for the JVM.
It enables developers to easily create automated integration, functional and acceptance tests for Java middlewares.
In fact it combines a unit testing framework (JUnit or TestNG), ShrinkWrap, and one or more supported target containers (Java EE container, servlet container, etc) to provide a simple, flexible and pluggable integration testing environment, with no need of any special configuration.
Its extensibility can be found in the different modules and extension it offers, that provide very useful functionalities for every aspect of the software system.

The Arquillian test infrastructure

Arquillian will be mainly used for:

- Managing the lifecycle of one or more containers
- Bundling the test case, dependent classes and resources into a ShrinkWrap archive
- Deploying the archive to the container
- Enriching the test case by providing dependency injection and other declarative services
- Executing the tests inside (or against) the container
- Capturing the results and returning them to the test runner for reporting

ShrinkWrap is the simplest way to create and define custom archives in Java, that encapsulate the test class and its dependent resources, powering the Arquillian deployment mechanism. In fact, the test case is dispatched to the container's environment through coordination with ShrinkWrap: Arquillian packages the ShrinkWrap-defined archive at runtime and deploys it to the target container.

4.2 Manual testing.

Manual testing can be considered a valid solution, depending on the application to be tested.
In this case a tester ensures the correct behaviors of the components, by following a test plan that allows him to find the most important use cases, without any support from tools or scripts.
This is used in alternative of the automated tests, which is more quickly, less expensive and more accurate, because in some particular scenarios it is necessary to exploit human observation, logic and experience.

4.3 Maven failsafe (?)

5 Program Stubs and Test Data Required