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. There are also specifications about the integration strategy and the sequence of the integration of the components of the system already defined, along with an justification of the tools that are needed for the testing.
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**

- **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**: a modular part of a system with encapsulated content and whose manifestation is replaceable within its environment.

- **Subsystem**: a behavioral unit in the physical system, and hence in the model.

- **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

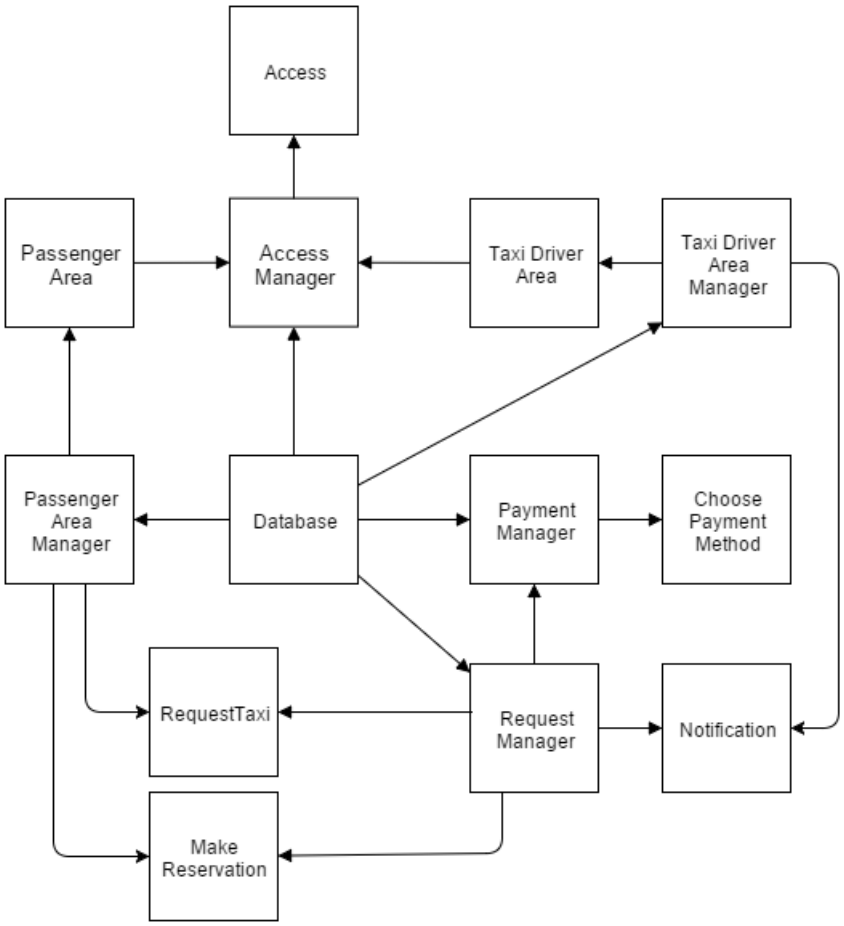
2.1 Entry Criteria

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 high prioritized bugs fixed
- 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.

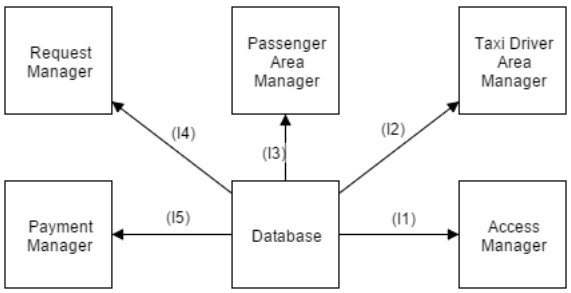
2.4 Sequence of Component/Function Integration

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.

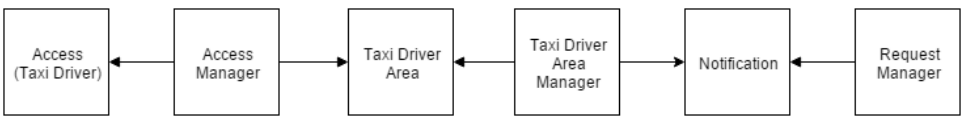
- **Integration Test of the "Manager" subsystem**

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



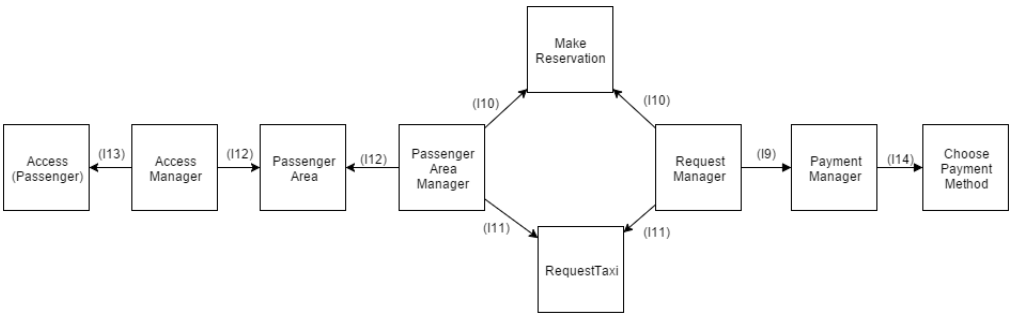
- **Integration Test of the "Taxi Driver Client" subsystem**

ID	Integration Test	Paragraph
I6T1	RequestManager → Notification	3.6
I6T2	TaxiDriverAreaManager → Notification	3.6
I7T1	TaxiDriverAreaManager → TaxiDriverArea	3.7
I7T2	AccessManager → TaxiDriverArea	3.7
I8	AccessManager → Access (Taxi Driver)	3.8



- **Integration Test of the "Passenger Client" subsystem**

ID	Integration Test	Paragraph
I9	RequestManager → PaymentManager	3.9
I10T1	RequestManager → MakeReservation	3.10
I10T2	PassengerAreaManager → MakeReservation	3.10
I11T1	RequestManager → RequestTaxi	3.11
I11T2	PassengerAreaManager → RequestTaxi	3.11
I12T1	PassengerAreaManager → PassengerArea	3.12
I12T2	AccessManager → PassengerArea	3.12
I13	AccessManager → Access (Passenger)	3.13
I14	PaymentManager → ChoosePaymentMethod	3.14



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)	Database → AccessManager
Input specification	The Database sends to the AccessManager the data that has been requested with a query, with an INSERT or with an UPDATE.
Output specification	Check if the INSERT and UPDATE have been done correctly, check if the result of the query is what is expected and check if the result is computed correctly by the AccessManager.
Environmental needs	Database data available.

3.2 Integration Test case I2

Test case identifier	I2
Test item(s)	Database → TaxiDriverAreaManager
Input specification	The Database sends to the TaxiDriverAreaManager the data that has been requested with a query, with an INSERT or with an UPDATE.
Output specification	Check if the INSERT and UPDATE have been done correctly, check if the result of the query is what is expected and check if the result is computed correctly by the TaxiDriverAreaManager.
Environmental needs	Database data available.

3.3 Integration Test case I3

Test case identifier	I3
Test item(s)	Database → PassengerAreaManager
Input specification	The Database sends to the PassengerAreaManager the data that has been requested with a query, with an INSERT or with an UPDATE.
Output specification	Check if the INSERT and UPDATE have been done correctly, check if the result of the query is what is expected and check if the result is computed correctly by the PassengerAreaManager.
Environmental needs	Database data available.

3.4 Integration Test case I4

Test case identifier	I4
Test item(s)	Database → RequestManager
Input specification	The Database sends to the RequestManager the data that has been requested with a query, with an INSERT or with an UPDATE.
Output specification	Check if the INSERT and UPDATE have been done correctly, check if the result of the query is what is expected and check if the result is computed correctly by the RequestManager.
Environmental needs	Database data available.

3.5 Integration Test case I5

Test case identifier	I5
Test item(s)	Database → PaymentManager
Input specification	The Database sends to the PaymentManager the data that has been requested with a query, with an INSERT or with an UPDATE.
Output specification	Check if the INSERT and UPDATE have been done correctly, check if the result of the query is what is expected and check if the result is computed correctly by the PaymentManager.
Environmental needs	Database data available.

3.6 Integration Test case I6

Test case identifier	I6T1
Test item(s)	RequestManager → Notification
Input specification	The RequestManager sends to the Notification component all the notifications of a specific Taxi driver.
Output specification	Check if the notifications actually belongs to the correct Taxi driver.
Environmental needs	Notification driver and I4 succeeded

Test case identifier	I6T2
Test item(s)	TaxiDriverAreaManager → Notification
Input specification	The TaxiDriverAreaManager sends an alert message to the Notification component if a notification is detected.
Output specification	Check if the message is computed correctly by the TaxiDriverAreaManager.
Environmental needs	Notification driver and I2 succeeded

3.7 Integration Test case I7

Test case identifier	I7T1
Test item(s)	TaxiDriverAreaManager → TaxiDriverArea
Input specification	The TaxiDriverAreaManager sends to the TaxiDriverArea component the profile information of the Taxi driver requested.
Output specification	Check if the correct information has been sent.
Environmental needs	TaxiDriverArea driver and I2 succeeded

Test case identifier	I7T2
Test item(s)	AccessManager → TaxiDriverArea
Input specification	The AccessManager sends to the TaxiDriverArea component the identification data of the Taxi driver logged in.
Output specification	Check if the TaxiDriverArea component has built the profile page of the correct Taxi driver logged in.
Environmental needs	TaxiDriverArea driver and I1 succeeded

3.8 Integration Test case I8

Test case identifier	I8
Test item(s)	AccessManager → Access
Input specification	The Access component sends to the AccessManager the data that the client has inserted in the registration or login input form.
Output specification	Check if the AccessManager computes the inputs correctly.
Environmental needs	Access driver and I1 succeeded

3.9 Integration Test case I9

Test case identifier	I9
Test item(s)	RequestManager → PaymentManager
Input specification	The PaymentManager sends to the RequestManager the data containing the payment choice of the passenger.
Output specification	Check if the payment method is managed correctly.
Environmental needs	I5 and I4 succeeded

3.10 Integration Test case I10

Test case identifier	I10T1
Test item(s)	RequestManager → MakeReservation
Input specification	The MakeReservation component sends the data that the passenger has inserted in the reservation form to the RequestManager.
Output specification	Check if the RequestManager component sends back the right answer, computing the input correctly.
Environmental needs	MakeReservation driver and I4 succeeded

Test case identifier	I10T2
Test item(s)	PassengerAreaManager → MakeReservation
Input specification	The PassengerAreaManager sends an alert message to the MakeReservation component when a taxi is found.
Output specification	Check if the MakeReservation component computes the message correctly.
Environmental needs	MakeReservation driver and I3 succeeded

3.11 Integration Test case I11

Test case identifier	I11T1
Test item(s)	RequestManager → RequestTaxi
Input specification	The RequestManager searches a taxi for a request and it sends the results to the RequestTaxi component.
Output specification	Check if the RequestTaxi component computes the inputs correctly.
Environmental needs	RequestTaxi driver and I4 succeeded

Test case identifier	I11T2
Test item(s)	PassengerAreaManager → RequestTaxi
Input specification	The PassengerAreaManager sends an alert message to the RequestTaxi component when a taxi is found.
Output specification	Check if the RequestTaxi component computes the message correctly.
Environmental needs	RequestTaxi driver I3 succeeded

3.12 Integration Test case I12

Test case identifier	I12T1
Test item(s)	PassengerAreaManager → PassengerArea
Input specification	The PassengerAreaManager sends to the PassengerArea component the profile information of the Passenger requested.
Output specification	Check if the correct information has been sent.
Environmental needs	PassengerArea driver and I3 succeeded

Test case identifier	I12T2
Test item(s)	AccessManager → PassengerArea
Input specification	The AccessManager sends to the PassengerArea component the identification data of the Passenger logged in.
Output specification	Check if the PassengerArea component has built the profile page of the correct Passenger logged in.
Environmental needs	PassengerArea driver and I1 succeeded

3.13 Integration Test case I13

Test case identifier	I13
Test item(s)	AccessManager → Access
Input specification	The Access component sends to the AccessManager the data that the client has inserted in the input form of the registration or the log in.
Output specification	Check if the AccessManager computes the inputs correctly.
Environmental needs	Access driver and I1 succeeded

3.14 Integration Test case I14

Test case identifier	I14
Test item(s)	PaymentManager → ChoosePaymentMethod
Input specification	The PaymentManager sends to the ChoosePaymentMethod component the data of the Passenger's credit card.
Output specification	Check if the ChoosePaymentMethod component computes the inputs correctly.
Environmental needs	ChoosePaymentMethod driver and I5 succeeded

4 Tools and Test Equipment Required

4.1 Unit Testing and JUnit

Integration testing is usually performed after unit testing has been finished. Unit testing is usually done in the first part of the development and, 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 done in order to already have a visual feedback and all the possible problems sorted out, but all the classes are tested separately, so it will be necessary to check if they integrate properly with an integration testing afterwards.
In fact, in this phase all the classes are tested with each test case independent from the others, allowing to find both bugs in the programmer's implementation and flaws or missing parts of the specification.
JUnit is a unit testing framework for the Java programming language and is the tool usually used for performing unit testing. It will be employed to write and run repeatable tests on the classes and methods of the myTaxiService application.

4.2 Mockito

In unit testing, mock objects can simulate the behavior of complex, real objects, so they are useful when a real object is impractical or impossible to incorporate into a unit test.
They are also useful, as they allow developers to focus their tests on the behavior of the system without worrying about its dependencies and having a predictable results.
A tool that can be exploited is Mockito, a simple mocking framework widely used for Java, to create unit testing mockups.
It will be used for producing mock objects out of the Database component, or out of objects that supplies non-deterministic results or whose states are difficult to reproduce.

4.3 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.

4.4 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.

5 Program Stubs and Test Data Required