POLITECNICO MILANO 1863

myTaxiService

INTEGRATION TEST PLAN DOCUMENT

Paolo Antonini 858242
Andrea Corneo 849793

version 1 – 21st January 2016

Contents

1
Introduction

1.1 Purpose and scope

This document describes the plans for testing the integration between the components of myTaxiService system, which were presented in the previous *Design document*.

As such, a good familiarity of the past documents[1], namely the *Requirement analysis and specification document* (RASD) and the already mentioned *Design document* (DD), is needed. As a consequence, here we are going through the scope of myTaxiService project very rapidly.

[1] See section 1.3 for references to these documents.

Milan city council has recently committed to improving public transport services, and taxi service in particular. myTaxiService is a comprehensive system which allows a simplification of the service, as well as the possibility for customers of requesting and reserving taxi rides, even through mobile applications. Taxi drivers will be obviously supported in their activity with a mobile application as well, letting them receive the requests for taxi rides.

1.2 Definitions, acronyms, and abbreviations

Please refer to the corresponding section in the RASD for the definitions of words used in the document. Some technical expressions and abbreviations may be used in the following, but definitions are given when necessary.

1.3 References

In order to get information about the functionalities of the system, its architecture and design, please refer to the following documents:

- *Assignments 1 and 2*, section 2, parts I and II; 13th October 2015. This is to be regarded as the high level project description. Available at: `https://beep.metid.polimi.it/documents/3343933/d5865f65-6d37-484e-b0fa-04fcfe42216d`.
- *Requirement analysis and specification document*; 6th November 2015. Available at: `https://github.com/Cordaz/SE2_AntoniniCorneo/raw/master/Deliveries/1_RASD.pdf`.

- *Design document*; 4th December 2015. Available at: `https://github.com/Cordaz/SE2_AntoniniCorneo/raw/master/Deliveries/2_DD.pdf`.

1.4 Overview of the document

This document is structured as follows.

The strategies to be used and the components to be tested are identified in chapter 2, whereas the detailed description of the tests to be performed is provided in chapter 3. Finally, in chapter 4 we provide some additional information to support testers' work: in particular program stubs and test data required for each integration step is presented in section 4.1, whereas in section 4.2 we present all tools and test equipment needed to accomplish the integration.

1.5 Revision history

- 13th January 2016 creation of the document;
- 21st January 2016 first complete release (v. 1).

2
Integration strategy

2.1 Entry criteria

The first obvious condition which has to be satisfied before starting the integration test process is that the modules to be integrated are fully developed. Moreover, in order to provide a reliable background, most of the functions shall have already been unit tested. In detail, we state that at least 75 % of each component to be integrated should have been unit tested[1].

It is also important that some mock data are created, in order to let the components work. Please, refer to section 4.1 for further details.

[1] By this non-mandatory figure we are requesting that all the non-trivial methods are tested (i.e., we find it useless and frustrating to impose the testing of, for example, basic getters and setters methods).

2.2 Elements to be integrated

Integration testing deals with components. That is why in figure 2.1 we are showing the component diagram of myTaxiService system, already presented in section 2.3 of the *Design document*, to which the reader should refer for a detailed explanation.

This testing plan covers most of the system, but some components are excluded. For instance, `UserLogin` and `UserRegistration` are left out, as they cope with security protocols and integration mechanisms, which would need a comprehensive document on their own. `JavaServerFaces` and `Java Persistence API` are left out as well, for similar reasons.

2.3 Integration testing strategy

As of the strategy to adopt in order to complete the integration test, we select the top-down approach, which requires the highest-level modules (in terms of required functionalities and inclusions) be test and integrated first. This way a verified input is going to be provided to the following component or subsystem to be integrated.

This incremental approach, focused on the architecture of the system, may require some additional work, since stubs need to be created extensively, but nevertheless should be easy to understand and extend, and should arguably guarantee the quality of the

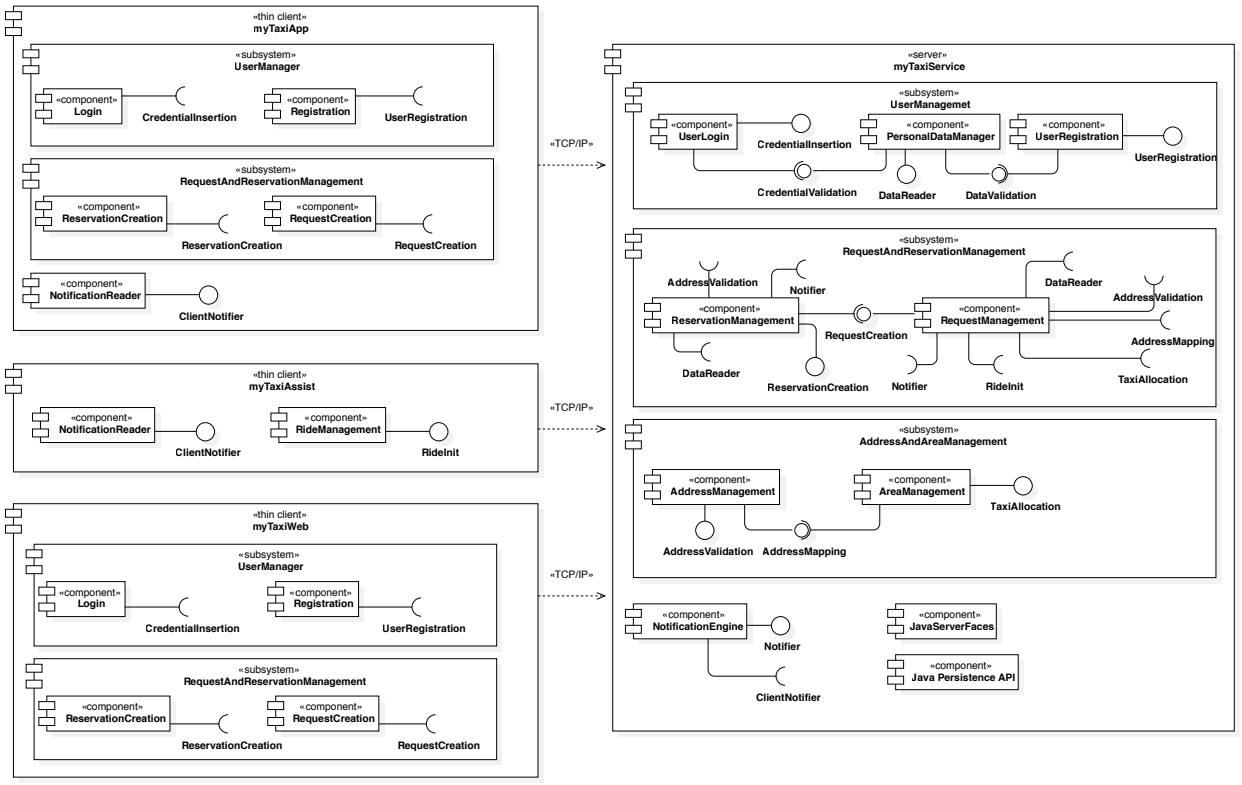

Figure 2.1: Component diagram.

software, high-level logic and data flow being tested early in the process.

At the end of the integration testing process, the system can be assumed as fully functional. As such, after undergoing a thorough system testing[2], which shall include also careful performance assessments, can be released to users.

[2] Please notice that the planning of this kind of process is outside the scope of this document.

2.4 *Integration sequence*

In the following figure 2.2 we present the actual sequence of integration to be followed, and reference details are provided in table 2.1.

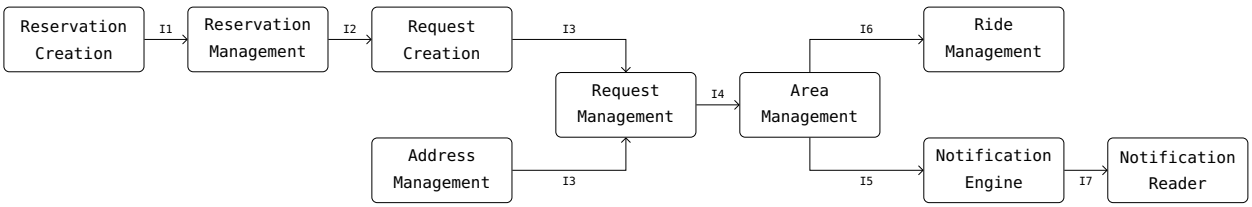

Figure 2.2: Sequence of integration.

ID	Integration test	Reference
I1	`ReservationCreation` → `ReservationManagement`	section 3.1.1
I2	`ReservationManagement` → `RequestCreation`	section 3.1.2
I3	`RequestCreation`, `AddressManagement` → `RequestManagement`	section 3.1.3
I4	`RequestManagement` → `AreaManagement`	section 3.1.4
I5	`AreaManagement` → `NotificationEngine`	section 3.1.5
I6	`AreaManagement` → `RideManagement`	section 3.1.6
I7	`Notification Engine` → `NotificationReader`	section 3.1.7

Table 2.1: Integration sequence reference details.

3
Test description

3.1 Test case specification

In this section we are going to detail each step of the integration process identified in section 2.4. We identify in particular the requested input and the expected output.

3.1.1 Integration test case `I1`

Test case identifier	`I1T1`
Test item(s)	`ReservationCreation` → `ReservationManagement`
Input specification	Typical reservation data
Output specification	Check if the reservation is properly created and data are valid
Environmental needs	Client devices: smartphone (myTaxiApp) and PC (myTaxiWeb)

3.1.2 Integration test case `I2`

Test case identifier	`I2T1`
Test item(s)	`ReservationManagement` → `RequestCreation`
Input specification	Typical reservation data
Output specification	Check if the request is properly created with reservation output
Environmental needs	`I1` succeeded

3.1.3 Integration test case `I3`

Test case identifier	`I3T1`
Test item(s)	`RequestCreation` → `RequestManagement`
Input specification	Typical request data
Output specification	Check if the request is correctly handled by `RequestManagement`
Environmental needs	`I2` succeeded

Test case identifier	`I3T2`
Test item(s)	`AddressManagement` → `RequestManagement`
Input specification	Typical request data
Output specification	Check if addresses are correctly mapped
Environmental needs	`I3T1` succeeded

3.1.4 *Integration test case* `I4`

Test case identifier	`I4T1`
Test item(s)	`RequestManagement` → `AreaManagement`
Input specification	Typical request data
Output specification	Check the taxi allocation (correct area, taxi dequeuing, ...)
Environmental needs	`I3` succeeded

3.1.5 *Integration test case* `I5`

Test case identifier	`I5T1`
Test item(s)	`AreaManagement` → `NotificationEngine`
Input specification	Typical request data
Output specification	Check if notification are correctly created (proper data, ...)
Environmental needs	`I4` succeeded

3.1.6 *Integration test case* `I6`

Test case identifier	`I6T1`
Test item(s)	`AreaManagement` → `RideManagement`
Input specification	Typical customer's withdrawal address data
Output specification	Check if the data are in the correct format for the driver (myTaxiAssist)
Environmental needs	`I5` succeeded

3.1.7 *Integration test case* I7

Test case identifier	I7T1
Test item(s)	NotificationEngine → NotificationReader
Input specification	Create notification message
Output specification	Check if the communication protocol works properly and the correct format of the notification
Environmental needs	I6 succeeded

Test case identifier	I7T2
Test item(s)	NotificationEngine → NotificationReader
Input specification	Create notification message
Output specification	Check if the SMS is sent
Environmental needs	I6 succeeded

3.2 *Test procedures*

During the integration and testing process, particular attention should be paid on the following procedures, because they are specifically aimed at checking that the system provides the functionalities it is written to accomplish.

3.2.1 *Integration test procedure* TP1

Procedure identifier	TP1
Purpose	This procedure verifies the correct creation of a request
Procedure steps	Execute I3

3.2.2 *Integration test procedure* TP2

Procedure identifier	TP2
Purpose	This procedure verifies the correct creation of a request, starting from a reservation
Procedure steps	Execute I3 after I1-I2

3.2.3 *Integration test procedure* TP3

Procedure identifier	TP3
Purpose	This procedure verifies the correct allocation of a taxi
Procedure steps	Execute I6 after I3-I4

3.2.4 *Integration test procedure* TP4

Procedure identifier	TP4
Purpose	This procedure verifies the correct notification of the ride start
Procedure steps	Execute I7 after I3-I6

4
Supporting information

4.1 Program stubs and test data

There are no particular requirements about mock data to perform the tests. According to the top-down strategy we adopted, every integration shall be performed on the basis of the thoroughly tested previous one. As a consequence, the data created to perform each integration may be exploited by the following one.

We only suggest to perform tests with Milan map being already loaded, in order to identify possible issues with the mapping of areas.

As of stubs, as it was already mentioned, they are crucial in the planned process. They can be easily identified by analysing the procedure outlined in section 2.3 and detailed in the following chapter. As of their creation and (software-based) management, refer to the following section 4.2.

4.2 Test equipment and tools

In order to support the integration phase and perform an effective integration testing, we suggest the following software tools. The documentation of the tools is referenced in section 1.3.

Mockito thanks to Mockito, developers and testers will be able to generate stubs, which are a crucial part of the process we planned.
Reference: `http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html`.

Arquillian this tool being specifically designed to manage the test of Java Beans and the integration between components and module, we suggest developers to use it.
Reference: `http://arquillian.org/guides/`.

JUnit even though JUnit is designed mostly to perform unit testing, nevertheless it is a valid framework to support integration testing, along with the other tools.
Reference: `http://junit.org/`.

That said, this section is not mandatory, nor comprehensive. We believe that manual testing may sometimes be the fastest, smartest and easiest way to test software. However, automated and supporting tools do exist, and developers had better make use of them, bearing in mind that this project can grow far beyond what was stated in the previous specification and design documents.

Appendix

Changes in the structure

Given the architecture of our system, we decided not to subdivide section 2.4 ("Integration sequence"), unlike it was suggested. We rearranged chapters 4 and 5 of the suggested structure into two sections of a single chapter, since logically related.

Also, we slightly changed some section titles for the sake of clarity and conciseness.

Hours of work

The writing of this document took the following amount of time:

Paolo Antonini 6 hours.

Andrea Corneo 4 hours.