Integration Test Plan Document

21/01/2016

Elis Bardhi **790135**

Andrea Cavalli **841512**

Mario Dolci **773705**

1. Introduction

This document describes the Integration Test Plan (ITP) for MyTaxiService system; it contains the description of the integration tests for this project.

1.1. Purpose and scope

This document describes the decisions for testing the integration of MyTaxiService components and its purpose is to test the interfaces between the components that have already described in the Design Document.
MyTaxiService system has to be able to execute actions when it receives a set of data from users. Integration between the components of the system has to hide the general complexity in order to make the system easy to be used. Moreover, usability is also increased by offering a web-based application and website for users to access the system.

1.2. List of definitions and abbreviations

The following acronyms will be used through this document:

- RASD: Requirements Analysis and Specification Document;
- DD: Design Document;
- ITP: Integration Test Plan;
- SCS: Smartphone/Computer System;
- LS: Listener System;
- DS: Dispatcher System;
- LTS: Location-Tracking System;
- TS: Taxi System;
- DBS: Database System.

1.3. List of Reference Documents

The following documents were used in order to write this document:

- MyTaxiService RASD Document;
- MyTaxiService DD Document;
- Assignments 1 and 2 (RASD and DD);
- Assignment 4 - Integration Test Plan.

2. Integration Strategy

2.1. Entry Criteria

The criteria that must be considered before the integration testing are the following:

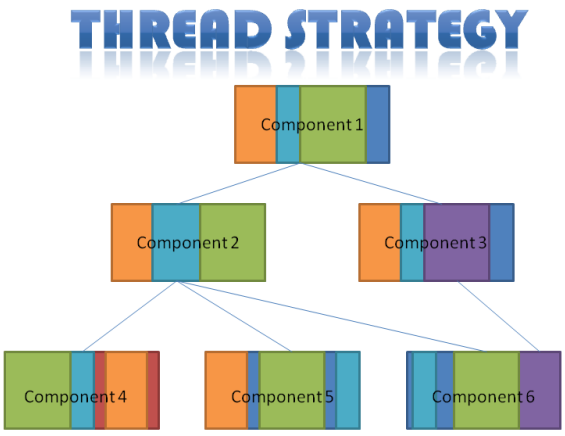
- Technical Design has already been finalized and approved;
- Environment setup is complete and stable, so all hardware components are ready and functional;
- Code development for the modules is complete;
- Code has a version control and all his changes can be tracked, compared or restored if needed;
- No known major or critical defects are pending, so all priority bugs have already been fixed and closed;
- A testing transition meeting has already been held and the developers signed off;
- Internal documentation has been updated to reflect the current state of the product;
- Project Manager approval has already been received.

2.2. Elements to be Integrated

After the satisfaction of all the entry criteria, we can proceed with the identification of the components that we have to integrate each other in order to get the whole system working. In order to do that, we refer to our Design Document (paragraph 2.2 pag.5). Since our system is very large and complex, we assume that all components that compose our subsystems (such as SCS, LS, DS, LTS, TS and DBS) are tested and integrated between each other, so the elements that have to be integrated are the subsystems.

2.3. Integration Testing Strategy

Since the project is large enough and since the architecture is a client-server architecture, we opted for a Thread Testing strategy. We have opted for this strategy because it offers a parallel development of components and a rapid integration of them.



Threads not only serve as the basis for integration, but they also tend to drive the entire software development effort from scheduling to status reporting. Each thread itself represents a microcosm of the system in that each has a documented definition and general execution path, an internal design and an associated test. Thread definition intends to communicate functional background and execution details between developers and from developers to testers. More importantly, the desired independence of threads supports incremental integration and system testing while the corresponding thread definition substantiates the results. Finally, since all system development activity progresses in relation to threads, management has an accurate method of judging the status of individual tasks, functional areas and requirements.
Keeping the goals of iterative development and software testing in mind, each thread has its own lifecycle with autonomous states and a formal process for state transitions. Individual team leaders usually decompose general requirements into groups of threads at the beginning and assign threads to developers. Developers maintain ownership of their threads and are responsible for documenting a scenario under which an integrator can verify the basic functionality, providing rudimentary definition to the thread. Following implementation and unit testing, the developer releases the corresponding software components to a daily integration build, at which point the thread enters a "testable" state. After verifying the functionality in the integration build, the developer marks the thread "ready" for an integrator who performs more extensive testing and eventually "integrates" the thread and corresponding software components into the system. At the end of each formal build, a team of key engineers in conjunction with quality assurance checks all threads against requirements as a regression test and "finalizes" those threads which pass.

2.4. Sequence of Component

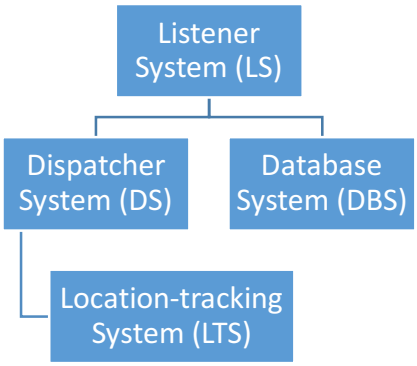
In this section the sequence of software and subsystem that must be followed in order to test the integration is explained.

2.4.1. Software Integration Sequence

Since we opted for a thread integration strategy, there does not exist a sequence of components to be integrated because the development of components is done in parallel.

2.4.2. Subsystem Integration Sequence

A valid order of subsystem integration for the server side may be structured as in the following schema.



In the end of the sequence we have to integrate the **Smartphone/Computer System (SCS)** and the **Taxi System (TS)**. SCS and TS are the last to be integrated because they represent the client side of the system, so first we have to bring up the server side.

3. Individual Steps and Test Description

Here are the main tests to do for MyTaxiService system.

3.1. Test 1

Test Item(s)	Listener System → Dispatcher System
Input specification LS	Data generated from client, for instance a request or a reservation
Output LS\ Input DS	Data to send to the Dispatcher System for example RequestQueue
Output specification DS	Notification for the client
Environmental needs	Client Stub, Taxi Stub, Database Stub

Test Item(s)	Listener System → Dispatcher System
Input specification LS	Data generated by taxi driver, for instance a request to read the RequestQueue
Output LS\ Input DS	Request sent to Dispatcher System
Output specification DS	First taxi request to send to taxi driver
Environmental needs	Client Stub, Taxi Stub, Database Stub

3.2. Test 2

Test Item(s)	Listener System → Database System
Input specification LS	Data to be saved like a new user insertion or a notification
Output LS\ Input DBS	Data incoming in input to be saved
Output specification DBS	Confirm message
Environmental needs	Client Stub, Taxi Stub

3.3. Test 3

Test Item(s)	Location tracking System ->Dispatcher System
Input specification LTS	Data incoming from the external service (GPS Satellite)
Output LS\ Input DS	List of all taxies with the correspondent GPS position and velocity
Output specification DS	N\A
Environmental needs	GPS Satellite Stub

3.4. Test 4

Test Item(s)	Smartphone/Computer System → Listener System
Input specification SCS	Form compiled data (registration)
Output SCS\ Input LS	All form compiled information
Output specification LS	Confirm message
Environmental needs	Database Stub

3.5. Test 5

Test Item(s)	Taxi System → Listener System
Input specification TS	Request for a new client
Output TS \ Input LS	Request data
Output specification LS	A package with client request information's or a message ("No clients")
Environmental needs	Database Stub

4. Tools and Test Equipment Required

Here are the tools to use in order to test the system:

- **Moquito** creates mockups in order to perform the unit testing: it allows us to abstract dependencies, to have predictable results and to check that the interaction between the testes and the mock is correct. Moquito can mock a persistent manager, an external service (such ad the LTS for location) and someone else's code;
- **Arquillian** and **JUnit** can be used to test the integration of MyTaxiService subsystems. Arquillian is a testing framework that enables developers to easily create automated integration using containers, functional and acceptance tests for Java middleware. Moreover, JUnit is a simple framework used to write repeatable tests for unit testing frameworks;
- **JMeter** can be used to test the system performances. It can be used to simulate a heavy traffic and load on a server or a network (for instance due to a lot of taxi requests), to test its strength or to analyze the system's overall performance under different load types. JMeter lets the developer to set up test plans that simulate loggin into MyTaxiService web page or app, filling out web page or app forms, clicking buttons and links; it also allows the developer to simulate the numbers of users doing an action;
- **Manual testing**: the developer plays the role of an end user and uses most of all features of MyTaxiService to ensure the correct behavior of the system. The developer has to write detailed test cases, identifying clear and concise steps to be taken with expected outcomes. In order to do this type of tests, usually a black-box technique is used (because the structure of software of MyTaxiService is unknown): test sets are based on the knowledge of some specifics of the system.

5. Program Stubs and Test Data Required

A method stub or simply a stub in software development is a piece of code used to stand in for some other programming functionality. A stub may simulate the behavior of existing code (such as a procedure on a remote machine) or be a temporary substitute for yet-to-be-developed code.
Stubs are therefore most useful in a top-down testing approach, in distributing computing as well as general software development and testing.
Since we opted for a thread strategy and since our system is distributed there will be many stubs and we can see some of them in the tests above, under “Environmental needs” rows.

6. Hours of work

Each component of the group has worked the following number of hours in order to write this document:

- Elis Bardhi: 6
- Andrea Cavalli: 5
- Mario Dolci: 5